AWS Lambda

Developer Guide

AWS Lambda: Developer Guide

Copyright © 2016 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

What Is AWS Lambda?

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time you consume - there is no charge when your code is not running. With AWS Lambda, you can run code for virtually any type of application or backend service - all with zero administration. AWS Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, 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, Java, C# and Python).

You can use AWS Lambda to run your code in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table; to run your code in response to HTTP requests using Amazon API Gateway; or invoke your code using API calls made using AWS SDKs. With these capabilities, you can use Lambda to easily build data processing 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.

For more information about the AWS Lambda execution environment, see Lambda Execution Environment and Available Libraries (p. 157). For information about how AWS Lambda determines compute resources required to execute your code, see Compute Requirements – Lambda Function Configuration (p. 3).

When Should I Use AWS Lambda?

AWS Lambda is an ideal compute platform for many application scenarios, provided that you can write your application code in languages supported by AWS Lambda (that is, Node.js, Java, C# and Python), and run within the AWS Lambda standard runtime environment and resources provided by Lambda.

When using AWS Lambda, you are 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, or 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, and monitoring and logging your Lambda functions.

If you need to manage your own compute resources, Amazon Web Services also offers other compute services to meet your needs.

- Amazon Elastic Compute Cloud (Amazon EC2) 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.

Are You a First-time User of AWS Lambda?

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

1. **Read the product overview and watch the introductory video to understand sample use cases.** These resources are available on the AWS Lambda webpage.

   Read the "How It Works" section of this guide. This section introduces various AWS Lambda components you work with to create an end-to-end experience. For more information, see How It Works (p. 151).
2. **Review the “Lambda Functions” section of this guide.** To understand the programming model and deployment options for a Lambda function there are core concepts you should be familiar with. This section explains these concepts and provides details of how they work in different languages that you can use to author your Lambda function code. For more information, see Lambda Functions (p. 3).
3. **Try the console-based Getting Started exercise.** The exercise provides instructions for you to create and test your first Lambda function using the console. You also learn about the console provided blueprints to quickly create your Lambda functions. For more information, see Getting Started (p. 160).
4. **Read the "Building Applications with AWS Lambda" section of this guide.** This section introduces various AWS Lambda components you work with to create an end-to-end experience. For more information, see Building applications with AWS Lambda (p. 115).

Beyond the Getting Started exercise, you can explore the various use cases, each of which is provided with a tutorial that walks you through an example scenario. Depending on your application needs (for example, whether you want event driven Lambda function invocation or on-demand invocation), you can follow specific tutorials that meet your specific needs. For more information, see Use Cases (p. 175).

The following topics provide additional information about AWS Lambda:

- AWS Lambda Function Versioning and Aliases (p. 72)
- Troubleshooting and Monitoring AWS Lambda Functions with Amazon CloudWatch (p. 107)
- Best Practices for Working with AWS Lambda Functions (p. 284)
- AWS Lambda Limits (p. 285)

Lambda Functions

After you package up your custom code, including any dependencies, and upload it to AWS Lambda, you have created a *Lambda function*.

If you are new to AWS Lambda, you might ask: what type of code can I run as a Lambda function? How does AWS Lambda execute my code? How does AWS Lambda know the amount of memory and CPU requirements needed to run my Lambda code? The following sections provide an overview of how a Lambda function works.

Depending on your scenario, you can build applications where you use the AWS Lambda service to run all or parts of your application code. For more information, see Building Lambda Functions (p. 5). How It Works (p. 151) provides examples that illustrate how to create a Lambda function for specific scenarios.

The sections in this topic provide the following introductory information about Lambda functions:

Topics

Compute Requirements – Lambda Function Configuration

A Lambda function consists of code and any associated dependencies. In addition, a Lambda function also has configuration information associated with it. Initially, you specify the configuration information when you create a Lambda function. Lambda provides an API for you to update some of the configuration data. Lambda function configuration information includes the following key elements:

- **Compute resources that you need** – You only specify the amount of memory you want to allocate for your Lambda function. AWS Lambda 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 memory, your Lambda function will receive twice the CPU share than if you allocated only 128 MB.

  You can update the configuration and request additional memory in 64 MB increments. For information about relevant limits, see AWS Lambda Limits (p. 285).

To change the amount of memory your Lambda function requires, do the following:

1. Sign in to the AWS Management Console and navigate to the AWS Lambda console.
2. Choose the function whose memory size you wish to change.
3. Click the **Configuration** tab and then expand **Advanced settings**.
4. In the **Memory (MB)** list, choose your desired amount.

Optionally, you can update the memory size of your functions using the following AWS CLI command (using valid 64 MB increments):

```
$ aws lambda update-function-configuration \
   --function-name your function name  \
   --region region where your function resides \
   --memorysize memory amount \
   --profile adminuser
```

For information on setting up and using the AWS CLI, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

- **Maximum execution time (timeout)** – You pay for the AWS resources that are used to run your Lambda function. To prevent your Lambda function from running indefinitely, you specify a timeout. When the specified timeout is reached, AWS Lambda terminates your Lambda function.

- **IAM role (execution role)** – This is the role that AWS Lambda assumes when it executes the Lambda function on your behalf.

- **Handler name** – The handler refers to the method in your code where AWS Lambda begins execution. AWS Lambda passes any event information, which triggered the invocation, as a parameter to the handler method.

Invocation Types

AWS Lambda supports synchronous and asynchronous invocation of a Lambda function. You can control the invocation type only when you invoke a Lambda function (referred to as *on-demand invocation*). The following examples illustrate on-demand invocations:

- Your custom application invokes a Lambda function.

- You manually invoke a Lambda function (for example, using the AWS CLI) for testing purposes.

In both cases, you invoke your Lambda function using the Invoke (p. 358) operation, and you can specify the invocation type as synchronous or asynchronous.

However, when you are using AWS services as event sources, the invocation type is predetermined for each of these services. You don't have any control over the invocation type that these event sources use when they invoke your Lambda function. For example, Amazon S3 always invokes a Lambda function asynchronously and Amazon Cognito always invokes a Lambda function synchronously. For stream-based AWS services (Amazon Kinesis Streams and Amazon DynamoDB Streams), AWS Lambda polls the stream and invokes your Lambda function synchronously.

Introduction: Building Lambda Functions

You upload your application code in the form of one or more *Lambda functions* to AWS Lambda, a compute service, and the service can run the code on your behalf. AWS Lambda takes care of provisioning and managing the servers to run the code upon invocation.

Typically, the lifecycle for an AWS Lambda-based application includes authoring code, deploying code to AWS Lambda, and then monitoring and troubleshooting. The following are general questions that come up in each of these lifecycle phases:

- **Authoring code for your Lambda function** – What languages are supported? Is there a programming model that I need to follow? How do I package my code and dependencies for uploading to AWS Lambda? What tools are available?

- **Uploading code and creating Lambda functions** – How do I upload my code package to AWS Lambda? How do I tell AWS Lambda where to begin executing my code? How do I specify compute requirements like memory and timeout?

- **Monitoring and troubleshooting** – For my Lambda function that is in production, what metrics are available? If there are any failures, how do I get logs or troubleshoot issues?

The following sections provide introductory information and the Example section at the end provides working examples for you to explore.

Note
This topic provides an introductory overview of how you develop AWS Lambda-based applications. The How It Works (p. 151) section describes the specifics about Lambda functions, event sources, and how AWS Lambda executes your Lambda functions.

Authoring Code for Your Lambda Function

You can author your Lambda function code in the languages that are supported by AWS Lambda. For a list of supported languages, see Lambda Execution Environment and Available Libraries (p. 157). There are tools for authoring code, such as the AWS Lambda console, Eclipse IDE, and Visual Studio IDE. But the available tools and options depend on the following:

- Language you choose to write your Lambda function code.
- Libraries that you use in your code. AWS Lambda runtime provides some of the libraries and you must upload any additional libraries that you use.

The following table lists languages, and the available tools and options that you can use.

Language	Tools and Options for Authoring Code	More Info
Node.js	• AWS Lambda console	You can use the console if the languages you choose do not require

Language	Tools and Options for Authoring Code	More Info
	• Visual Studio, with IDE plug-in (see AWS Lambda Support in Visual Studio) • Your own authoring environment	compilation, the code is saved in a single file, and it does not depend on any libraries.
Java	• Eclipse, with AWS Toolkit for Eclipse (see Using AWS Lambda with the AWS Toolkit for Eclipse) • Your own authoring environment	The AWS Toolkit also creates the deployment package, which is explained in Deploying Code and Creating a Lambda Function (p. 6).
C#	• Visual Studio, with IDE plug-in (see AWS Lambda Support in Visual Studio) • .NET Core (see .NET Core installation guide) • Your own authoring environment	The AWS Toolkit also creates the deployment package, which is explained in Deploying Code and Creating a Lambda Function (p. 6).
Python	• AWS Lambda console • Your own authoring environment	You can use the console if the languages you choose do not require compilation, the code is saved in a single file, and it does not depend on any libraries.

In addition, regardless of the language you choose, there is a pattern to writing Lambda function code. For example, how you write the handler method of your Lambda function (that is, the method that AWS Lambda first calls when it begins executing the code), how you pass events to the handler, what statements you can use in your code to generate logs in CloudWatch Logs, how to interact with AWS Lambda runtime and obtain information such as the time remaining before timeout, and how to handle exceptions. The Programming Model (p. 8) section provides information for each of the supported languages.

Note
After you familiarize yourself with AWS Lambda, see the Use Cases (p. 175), which provide step-by-step instructions to help you explore the end-to-end experience.

Deploying Code and Creating a Lambda Function

To create a Lambda function, you first package your code and dependencies in a deployment package. Then, you upload the deployment package to AWS Lambda to create your Lambda function.

Topics

Creating a Deployment Package – Organizing Code and Dependencies

You must first organize your code and dependencies in certain ways and create a *deployment package*. Instructions to create a deployment package vary depending on the language you choose to author the code. For example, you can use build plugins such as Jenkins (for Node.js and Python), and Maven (for Java) to create the deployment packages. For more information, see Creating a Deployment Package (p. 56).

When you create Lambda functions using the console, the console creates the deployment package for you, and then uploads it to create your Lambda function.

Uploading a Deployment Package – Creating a Lambda Function

AWS Lambda provides the CreateFunction (p. 332) operation, which is what you use to create a Lambda function. You can use the AWS Lambda console, AWS CLI, and AWS SDKs to create a Lambda function. Internally, all of these interfaces call the `CreateFunction` operation.

In addition to providing your deployment package, you can provide configuration information when you create your Lambda function including the compute requirements of your Lambda function, the name of the handler method in your Lambda function, and the runtime, which depends on the language you chose to author your code. For more information, see Lambda Functions (p. 3).

> **Note**
> This section provides an introductory overview of developing AWS Lambda-based applications. How It Works (p. 151) describes specifics about Lambda functions, event sources, and how AWS Lambda executes your Lambda functions.

Testing a Lambda Function

If your Lambda function is designed to process events of a specific type, you can use sample event data to test your Lambda function using one of the following methods:

- Test your Lambda function in the console.
- Test your Lambda function using the AWS CLI. You can use the `Invoke` method to invoke your Lambda function and pass in sample event data.

The console provides sample event data. The same data is also provided in the Sample Events Published by Event Sources (p. 130) topic, which you can use in the AWS CLI to invoke your Lambda function.

Monitoring and Troubleshooting

After your Lambda function is in production, AWS Lambda automatically monitors functions on your behalf, reporting metrics through Amazon CloudWatch. For more information, see Accessing Amazon CloudWatch Metrics for AWS Lambda (p. 109).

To help you troubleshoot failures in a function, Lambda logs all requests handled by your function and also automatically stores logs that your code generates in Amazon CloudWatch Logs. For more information, see Accessing Amazon CloudWatch Logs for AWS Lambda (p. 110).

AWS Lambda-Based Application Examples

This guide provides several examples with step-by-step instructions. If you are new to AWS Lambda, we recommend you try the following exercises:

- Getting Started (p. 160) – The Getting Started exercise provides a console-based experience. The sample code is authored in Python. You can edit the code in the console, upload it to AWS Lambda, and test it using sample event data provided in the console.

- Use Cases (p. 175) – If you cannot author your code using the console, you must create your own deployment packages and use the AWS CLI (or SDKs) to create your Lambda function. For more information, see Authoring Code for Your Lambda Function (p. 5). Most examples in the Uses Cases section use the AWS CLI. If you are new to AWS Lambda, we recommend that you try one of these exercises.

Related Topics

The following topics provide additional information.

Programming Model (p. 8)

Creating a Deployment Package (p. 56)

AWS Lambda Function Versioning and Aliases (p. 72)

Troubleshooting and Monitoring AWS Lambda Functions with Amazon CloudWatch (p. 107)

Programming Model

You write code for your Lambda function in one of the languages AWS Lambda supports. Regardless of the language you choose, there is a common pattern to writing code for a Lambda function that includes the following core concepts:

- **Handler** – Handler is the function AWS Lambda calls to start execution of your Lambda function. You identify the handler when you create your Lambda function. When a Lambda function is invoked, AWS Lambda starts executing your code by calling the handler function. AWS Lambda passes any event data to this handler as the first parameter. Your handler should process the incoming event data and may invoke any other functions/methods in your code.

- **The context object and how it interacts with Lambda at runtime** – AWS Lambda also passes a context object to the handler function, as the second parameter. Via this context object your code can interact with AWS Lambda. For example, your code can find the execution time remaining before AWS Lambda terminates your Lambda function.

  In addition, for languages such as Node.js, there is an asynchronous platform that uses callbacks. AWS Lambda provides additional methods on this context object. You use these context object methods to tell AWS Lambda to terminate your Lambda function and optionally return values to the caller.

- **Logging** – Your Lambda function can contain logging statements. AWS Lambda writes these logs to CloudWatch Logs. Specific language statements generate log entries, depending on the language you use to author your Lambda function code.

- **Exceptions** – Your Lambda function needs to communicate the result of the function execution to AWS Lambda. Depending on the language you author your Lambda function code, there are different ways to end a request successfully or to notify AWS Lambda an error occurred during execution. If you invoke the function synchronously, then AWS Lambda forwards the result back to the client.

Note
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. Persistent state should be stored 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 language specific topics provide detail information:

- Programming Model (Node.js) (p. 9)
- Programming Model for Authoring Lambda Functions in Java (p. 21)
- Programming Model for Authoring Lambda Functions in C# (p. 46)
- Programming Model for Authoring Lambda Functions in Python (p. 39)

Programming Model (Node.js)

AWS Lambda currently supports the following Node.js runtimes:

- Node.js runtime v4.3 (runtime = nodejs4.3)
- Node.js runtime v0.10.42 (runtime = nodejs)

Important
The v0.10.42 runtime will be unavailable to create new functions beginning on December 2016, given the end-of-life announcement for this version. Use the new runtime (nodejs4.3) while creating new functions and we recommend you migrate existing functions to the nodejs4.3 runtime as soon as possible. All examples in this documentation use the new v4.3 runtime. For information about programming model differences in the v0.10.42 runtime, see Using the Earlier Node.js Runtime v0.10.42 (p. 18).

When you create a Lambda function, you specify the runtime that you want to use. For more information, see `runtime` parameter of the CreateFunction (p. 332).

The following sections explain how common programming patterns and core concepts apply when authoring Lambda function code in Node.js. The programming model described in the following sections apply to both versions, except where indicated.

Topics

Lambda Function Handler (Node.js)

At the time you create a Lambda function you specify a *handler*, a function in your code, that AWS Lambda can invoke when the service executes your code. Use the following general syntax when creating a handler function in Node.js.

```
exports.myHandler = function(event, context,) {
   ...
}
```

The callback parameter is optional and you want to return information to the caller.

```
exports.myHandler = function(event, context, callback) {
   ...

   // Use callback() and return information to the caller.
}
```

In the syntax, note the following:

- `event` – AWS Lambda uses this parameter to pass in event data to the handler.
- `context` – AWS Lambda uses this parameter to provide your handler the runtime information of the Lambda function that is executing. For more information, see The Context Object (Node.js) (p. 12).
- `callback` – You can use the optional callback to return information to the caller, otherwise return value is null. For more information, see Using the Callback Parameter (p. 11).

  Note
  The callback is supported only in the Node.js runtime v4.3. If you are using the earlier runtime v0.10.42, you need to use the context methods (done, succeed, and fail) to properly terminate the Lambda function. For information about terminating Lambda functions written for earlier runtime versions, see Using the Earlier Node.js Runtime v0.10.42 (p. 18).

- *`myHandler`* – This is the name of the function AWS Lambda invokes. You export this so it is visible to AWS Lambda. Suppose you save this code as `helloworld.js`. Then, `helloworld.myHandler` is the handler. For more information, see handler in CreateFunction (p. 332).
  - If you used the `RequestResponse` invocation type (synchronous execution), AWS Lambda returns the result of the Node.js function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the `RequestResponse` invocation type, so when you test invoke the function using the console, the console will display the return value.

    If the handler does not return anything, AWS Lambda returns null.
  - If you used the `Event` invocation type (asynchronous execution), the value is discarded.

Example

Consider the following Node.js example code.

```
exports.myHandler = function(event, context, callback) {
   console.log("value1 = " + event.key1);
   console.log("value2 = " + event.key2);
   callback(null, "some success message");
   // or
   // callback("some error type");
}
```

This example has one function, which is also the handler. In the function, the `console.log()` statements log some of the incoming event data to CloudWatch Logs. When the callback is called, the Lambda function exits only after the Node.js event loop is empty (the Node.js event loop is not the same as the event that was passed as a parameter).

Note
If you are using the earlier runtime v0.10.42, you need to use the context methods (done, succeed, and fail) to properly terminate the Lambda function. For more information, see Using the Earlier Node.js Runtime v0.10.42 (p. 18).

To upload and test this code as a Lambda function (console)

1. In the console, create a Lambda function using the following information:
   - Use the hello-world blueprint.
   - Specify **nodejs4.3** as the **runtime**.
   - In **Handler**, replace `index.handler` with `exports.myHandler`.

   For instructions to create a Lambda function using the console, see Step 2.1: Create a Hello World Lambda Function (p. 164).
2. Replace the template code with the code provided in this section and create the function.
3. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console. For instructions on how to do this, see Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167).

Using the Callback Parameter

The Node.js runtime v4.3 supports the optional `callback` parameter. You can use it to explicitly return information back to the caller. The general syntax is:

```
callback(Error error, Object result);
```

Where:

- `error` – is an optional parameter that you can use to provide results of the failed Lambda function execution. When a Lambda function succeeds, you can pass null as the first parameter.
- `result` – is an optional parameter that you can use to provide the result of a successful function execution. The result provided must be `JSON.stringify` compatible. If an error is provided, this parameter is ignored.

Note
Using the `callback` parameter is optional. If you don't use the optional `callback` parameter, the behavior is same as if you called the `callback()` without any parameters. You can specify the `callback` in your code to return information to the caller.

If you don't use `callback` in your code, AWS Lambda will call it implicitly and the return value is `null`.

When the callback is called (explicitly or implicitly), AWS Lambda continues the Lambda function invocation until the Node.js event loop is empty.

The following are example callbacks:

```
callback();     // Indicates success but no information returned to the
 caller.
callback(null); // Indicates success but no information returned to the
 caller.
```

```
callback(null, "success");  // Indicates success with information returned to
 the caller.
callback(error);    //  Indicates error with error information returned to
 the caller.
```

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

Note the following:

- Regardless of the invocation type specified at the time of the Lambda function invocation (see Invoke (p. 358)), the callback 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 synchronously (using the `RequestResponse` invocation type), the callback returns a response body as follows:
  - If `error` is null, the response body is set to the string representation of `result`.
  - If the `error` is not null, the `error` value will be populated in the response body.

Note
When the `callback(error, null)` (and `callback(error)`) is called, Lambda will log the first 256 KB of the error object. For a larger error object, AWS Lambda truncates the log and displays the text `Truncated by Lambda` next to the error object.

The Context Object (Node.js)

While a Lambda function is executing, it can interact with AWS Lambda to get useful runtime information such as:

- How much time is remaining before AWS Lambda terminates your Lambda function (timeout is one of the Lambda function configuration properties).
- The CloudWatch log group and log stream associated with the Lambda function that is executing.
- The AWS request ID returned to the client that invoked the Lambda function. You can use the request ID for any follow up inquiry with AWS support.
- If the Lambda function is invoked through AWS Mobile SDK, you can learn more about the mobile application calling the Lambda function.

AWS Lambda provides this information via the `context` object that the service passes as the second parameter to your Lambda function handler. For more information, see Lambda Function Handler (Node.js) (p. 9).

The following sections provide an example Lambda function that uses the `context` object, and then lists all of the available methods and attributes.

Example

Consider the following Node.js example. The handler receives runtime information via a `context` parameter.

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

exports.handler = function(event, context, callback) {
    //console.log('Received event:', JSON.stringify(event, null, 2));
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    console.log('remaining time =', context.getRemainingTimeInMillis());
    console.log('functionName =', context.functionName);
    console.log('AWSrequestID =', context.awsRequestId);
    console.log('logGroupName =', context.logGroupName);
    console.log('logStreamName =', context.logStreamName);
    console.log('clientContext =', context.clientContext);
    if (typeof context.identity !== 'undefined') {
        console.log('Cognito
        identity ID =', context.identity.cognitoIdentityId);
    }
    callback(null, event.key1); // Echo back the first key value
    // or
    // callback("some error type");
};
```

The handler code in this example logs some of the runtime information of the Lambda function to CloudWatch. If you invoke the function using the Lambda console, the console displays the logs in the **Log output** section. You can create a Lambda function using this code and test it using the console.

To test this code in the AWS Lambda console

1. In the console, create a Lambda function using the hello-world blueprint. In **runtime**, choose **nodejs4.3**. For instructions on how to do this, see Step 2.1: Create a Hello World Lambda Function (p. 164).
2. Test the function, and then you can also update the code to get more context information.

The Context Object Methods (Node.js)

The context object provides the following methods.

context.getRemainingTimeInMillis()

Returns the approximate remaining execution time (before timeout occurs) of the Lambda function that is currently executing. The timeout is one of the Lambda function configuration. 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 at run time.

The general syntax is:

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

The Context Object Properties (Node.js)

The `context` object provides the following property that you can update:

callbackWaitsForEmptyEventLoop
The default value is true. This property is useful only to modify the default behavior of the callback. By default, the callback will wait until the Node.js runtime event loop is empty before freezing the process and returning the results to the caller. You can set this property to false to request AWS Lambda to freeze the process soon after the `callback` is called, even if there are events in the event loop. AWS Lambda will freeze the process, any state data and the events in the Node.js event loop (any remaining events in the event loop processed when the Lambda function is called next and if AWS Lambda chooses to use the frozen process). For more information about callback, see Using the Callback Parameter (p. 11).

In addition, the `context` object provides the following properties that you can use obtain runtime information:

functionName
Name of the Lambda function that is executing.

functionVersion
The Lambda function version that is executing. If an alias is used to invoke the function, then `function_version` will be the version the alias points to.

invokedFunctionArn
The ARN used to invoke this function. It can be a function ARN or an alias ARN. An unqualified ARN executes the `$LATEST` version and aliases execute the function version it is pointing to.

memoryLimitInMB
Memory limit, in MB, you configured for the Lambda function. You set the memory limit at the time you create a Lambda function and you can change it later.

awsRequestId
AWS request ID associated with the request. This is the ID returned to the client that called the `invoke` method.

Note
If AWS Lambda retries the invocation (for example, in a situation where the Lambda function that is processing Amazon Kinesis records throws an exception), the request ID remains the same.

logGroupName
The name of the CloudWatch log group where you can find logs written by your Lambda function.

logStreamName
The name of the CloudWatch log group where you can find logs written by your Lambda function. The log stream may or may not change for each invocation of the Lambda function.

The value is null if your Lambda function is unable to create a log stream, which can happen if the execution role that grants necessary permissions to the Lambda function does not include permissions for the CloudWatch actions.

identity
Information about the Amazon Cognito identity provider when invoked through the AWS Mobile SDK. It can be null.

- **identity.cognitoIdentityId**
- **identity.cognitoIdentityPoolId**

For more information about the exact values for a specific mobile platform, see Identity Context in the *AWS Mobile SDK for iOS Developer Guide*, and Identity Context in the AWS Mobile SDK for Android Developer Guide.

clientContext
Information about the client application and device when invoked through the AWS Mobile SDK. It can be null. Using `clientContext`, you can get the following information:

- **clientContext.client.installation_id**
- **clientContext.client.app_title**
- **clientContext.client.app_version_name**
- **clientContext.client.app_version_code**
- **clientContext.client.app_package_name**
- **clientContext.Custom**

  Custom values set by the mobile client application.
- **clientContext.env.platform_version**
- **clientContext.env.platform**
- **clientContext.env.make**

- **clientContext.env.model**
- **clientContext.env.locale**

For more information about the exact values for a specific mobile platform, see Client Context in the *AWS Mobile SDK for iOS Developer Guide*, and Client Context in the *AWS Mobile SDK for Android Developer Guide*.

Logging (Node.js)

Your Lambda function can contain logging statements. AWS Lambda writes these logs to CloudWatch. If you use the Lambda console to invoke your Lambda function, the console displays the same logs.

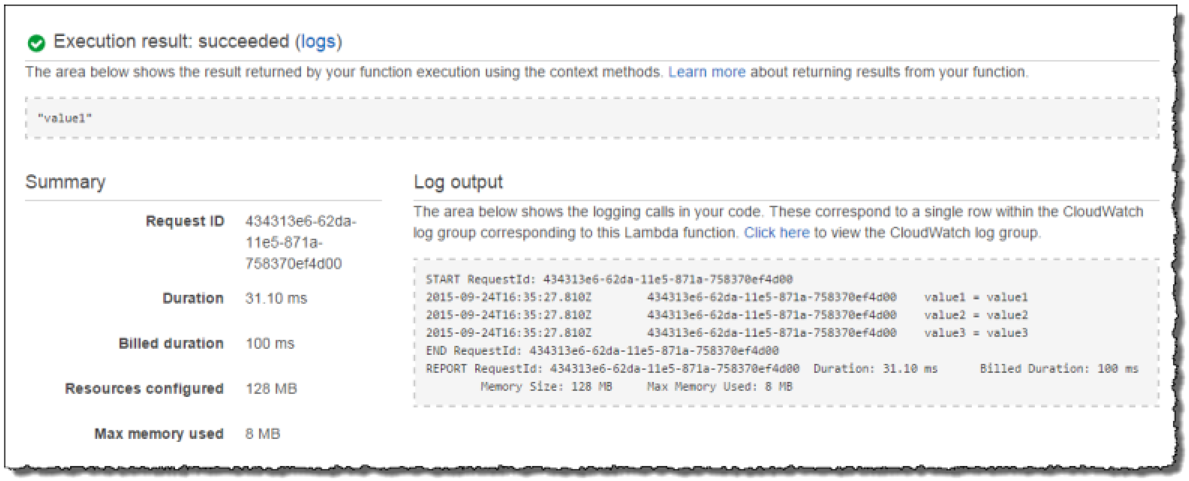
The following Node.js statements generate log entries:

- `console.log()`
- `console.error()`
- `console.warn()`
- `console.info()`

For example, consider the following Node.js code example.

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

exports.handler = function(event, context, callback) {
    //console.log('Received event:', JSON.stringify(event, null, 2));
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    callback(null, event.key1); // Echo back the first key value

};
```

The screenshot shows an example **Log output** section in Lambda console, you can also find these logs in CloudWatch. For more information, see Accessing Amazon CloudWatch Logs for AWS Lambda (p. 110).



Execution result: succeeded (logs)

The area below shows the result returned by your function execution using the context methods. Learn more about returning results from your function.

```
"value1"
```

Summary

Request ID	434313e6-62da-11e5-871a-758370ef4d00
Duration	31.10 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	8 MB

Log output

The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: 434313e6-62da-11e5-871a-758370ef4d00
2015-09-24T16:35:27.810Z	434313e6-62da-11e5-871a-758370ef4d00	value1 = value1
2015-09-24T16:35:27.810Z	434313e6-62da-11e5-871a-758370ef4d00	value2 = value2
2015-09-24T16:35:27.810Z	434313e6-62da-11e5-871a-758370ef4d00	value3 = value3
END RequestId: 434313e6-62da-11e5-871a-758370ef4d00
REPORT RequestId: 434313e6-62da-11e5-871a-758370ef4d00	Duration: 31.10 ms	Billed Duration: 100 ms
	Memory Size: 128 MB	Max Memory Used: 8 MB
```

The console uses the `RequestResponse` invocation type (synchronous invocation) when invoking the function, therefore it gets the return value (`value1`) back from AWS Lambda which the console displays.

To test the preceding Node.js code in AWS Lambda console

1. In the console, create a Lambda function using the hello-world blueprint. Make sure to select the Node.js as the **runtime**. For instructions on how to do this, see Step 2.1: Create a Hello World Lambda Function (p. 164).
2. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console. For instructions on how to do this, see Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167). You can also update the code and try other logging methods and properties discussed in this section.

For step-by-step instructions, see Getting Started (p. 160).

Finding Logs

You can find the logs that your Lambda function writes, as follows:

- **In the AWS Lambda console** – The **Log output** section in the AWS Lambda console shows the logs.
- **In the response header, when you invoke a Lambda function programmatically** – 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. AWS Lambda returns this log information in the `x-amz-log-results` header in the response. For more information, see Invoke.

  If you use AWS CLI to invoke the function, you can specify the `--log-type parameter` with value `Tail` to retrieve the same information.
- **In CloudWatch Logs** – To find your logs in CloudWatch you need to know the log group name and log stream name. You can get that information by adding the `context.logGroupName`, and `context.logStreamName` methods in your code. When you run your Lambda function, the resulting logs in the console or CLI will show you the log group name and log stream name.

Exceptions (Node.js)

If your Lambda function notifies AWS Lambda that it failed to execute properly by calling the `context.fail` method, AWS Lambda serializes the error information and returns it. If the error value passed to `context.fail` is an exception, Lambda will attempt to serialize information about the exception such as its type, error message, and stack trace. Otherwise, Lambda will attempt to convert the error object to a String. Consider the following example:

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

exports.handler = function(event, context, callback) {
    // This example code only throws error.
    var error = new Error("something is wrong");
    callback(error);

};
```

When you invoke this Lambda function, it will notify AWS Lambda that function execution completed with an error and passes error information to AWS Lambda. AWS Lambda returns the error information back to the client:

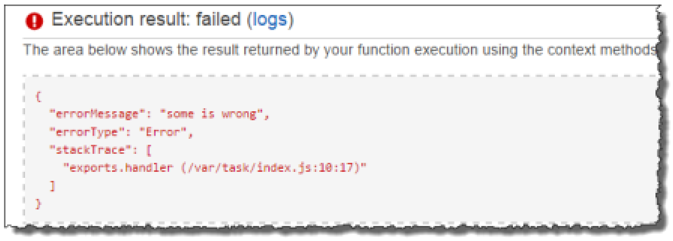
```
{
  "errorMessage": "something is wrong",
  "errorType": "Error",
  "stackTrace": [
    "exports.handler (/var/task/index.js:10:17)"
  ]
}
```

Note that the stack trace is returned as the `stackTrace` JSON array of stack trace elements.

How you get the error information back depends on the invocation type that the client specifies at the time of function invocation:

- If a client specifies the `RequestResponse` invocation type (that is, synchronous execution), it returns the result to the client that made the invoke call.

  For example, the console always use the `RequestResponse` invocation type, so the console will display the error in the **Execution result** section as shown:

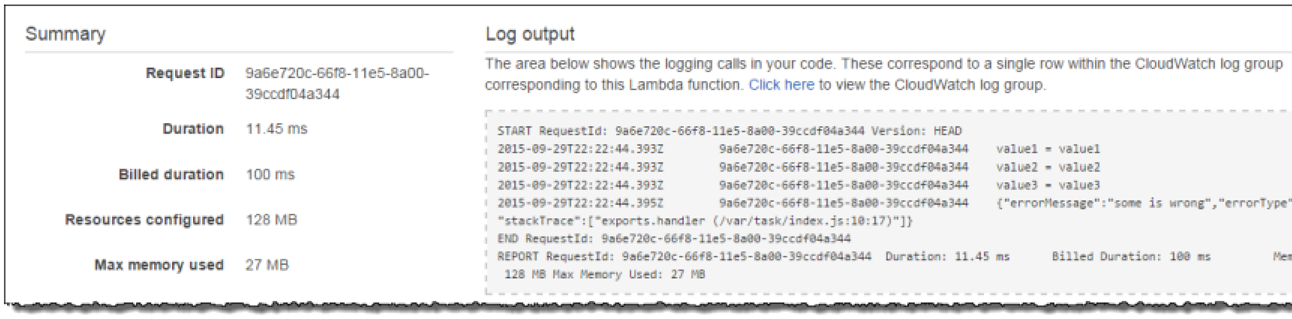


Execution result: failed (logs)

The area below shows the result returned by your function execution using the context methods

```
{
  "errorMessage": "some is wrong",
  "errorType": "Error",
  "stackTrace": [
    "exports.handler (/var/task/index.js:10:17)"
  ]
}
```

  The same information is also sent to CloudWatch and the **Log output** section shows the same logs.



Summary

Request ID	9a6e720c-66f8-11e5-8a00-39ccdf04a344
Duration	11.45 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	27 MB

Log output

The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: 9a6e720c-66f8-11e5-8a00-39ccdf04a344 Version: HEAD
2015-09-29T22:22:44.393Z	9a6e720c-66f8-11e5-8a00-39ccdf04a344	value1 = value1
2015-09-29T22:22:44.393Z	9a6e720c-66f8-11e5-8a00-39ccdf04a344	value2 = value2
2015-09-29T22:22:44.393Z	9a6e720c-66f8-11e5-8a00-39ccdf04a344	value3 = value3
2015-09-29T22:22:44.395Z	9a6e720c-66f8-11e5-8a00-39ccdf04a344	{"errorMessage":"some is wrong","errorType"
"stackTrace":["exports.handler (/var/task/index.js:10:17)"]}
END RequestId: 9a6e720c-66f8-11e5-8a00-39ccdf04a344
REPORT RequestId: 9a6e720c-66f8-11e5-8a00-39ccdf04a344  Duration: 11.45 ms      Billed Duration: 100 ms         Mem
 128 MB Max Memory Used: 27 MB
```

- If a client specifies the `Event` invocation type (that is, asynchronous execution), AWS Lambda will not return anything. Instead, it logs the error information to CloudWatch Logs. You can also see the error metrics in CloudWatch Metrics.

Depending on the event source, AWS Lambda may retry the failed Lambda function. For example, if Amazon Kinesis is the event source, AWS Lambda will retry the failed invocation until the Lambda function succeeds or the records in the stream expire. For more information on retries, see Retries on Errors (p. 154).

To test the preceding Node.js code (console)

1. In the console, create a Lambda function using the hello-world blueprint. In **runtime**, choose **Node.js** and, in **Role**, choose **Basic execution role**. For instructions on how to do this, see Step 2.1: Create a Hello World Lambda Function (p. 164).
2. Replace the template code with the code provided in this section.
3. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console. For instructions on how to do this, see Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167).

Using the Earlier Node.js Runtime v0.10.42

As of April 2016, AWS Lambda supports Node.js runtime v4.3. For information about specifying this runtime when you create your Lambda function, see the `runtime` parameter of CreateFunction (p. 332).

AWS Lambda continues to support the earlier runtime v0.10.42 (launched Nov 2014), which you can refer to using `runtime=nodejs` while creating a function. Given the upcoming end of life for this version, you will no longer be able to create new functions using this version on December 2016. Existing functions will be supported until early 2017. Use the new runtime (nodejs4.3) while creating new functions and we recommend you migrate existing functions to the nodejs4.3 runtime as soon as possible. The section highlights behavior unique to the earlier runtime v0.10.42 and how to transition your existing functions to runtime v4.3.

Topics

The Context Methods in Node.js Runtime v0.10.42

The earlier Node.js runtime v0.10.42 does not support the callback parameter for your Lambda function that runtime v4.3 supports. When using runtime v0.10.42, you use the following context object methods to properly terminate your Lambda function. The context object supports the `done()`, `succeed()`, and `fail()` methods that you can use to terminate your Lambda function. These methods are also present in runtime v4.3 for backward compatibility. For information about transitioning your code to use runtime v4.3, see Transitioning Lambda Function Code to Node.js Runtime v4.3 (p. 20).

context.succeed()

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

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

Where:

`result` – is an optional parameter and it can be used to provide the result of the function execution.

The `result` provided must be `JSON.stringify` compatible. If AWS Lambda fails to stringify or encounters another error, an unhandled exception is thrown, with the `X-Amz-Function-Error` response header set to `Unhandled`.

You can call this method without any parameters (`succeed()`) or pass a null value (`succeed(null)`).

The behavior of this method depends on the invocation type specified in the Lambda function invocation. For more information about invocation types, see Invoke (p. 358).

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

context.fail()

Indicates the Lambda function execution and all callbacks completed unsuccessfully, resulting in a handled exception. The general syntax is shown following:

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

Where:

`error` – is an optional parameter that you can use to provide the result of the Lambda function execution.

If the `error` value is non-null, the method will set the response body to the string representation of `error` and also write corresponding logs to CloudWatch. If AWS Lambda fails to stringify or encounters another error, an unhandled error occurs with the `X-Amz-Function-Error` header set to `Unhandled`.

> **Note**
> For the error from `context.fail(error)` and `context.done(error, null)`, Lambda logs the first 256 KB of the error object. For larger error objects, AWS Lambda truncates the error and displays the text: `Truncated by Lambda` next to the error object.

You can call this method without any parameters (`fail()`) or pass a null value (`fail(null)`).

context.done()

Causes the Lambda function execution to terminate.

> **Note**
> This method complements the `succeed()` and `fail()` methods by allowing the use of the "error first" callback design pattern. It provides no additional functionality.

The general syntax is:

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

Where:

- `error` – is an optional parameter that you can use to provide results of the failed Lambda function execution.
- `result` – is an optional parameter that you can use to provide the result of a successful function execution. The result provided must be `JSON.stringify` compatible. If an error is provided, this parameter is ignored.

You can call this method without any parameters (done()), or pass null (done(null)).

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

The function behavior depends on the invocation type specified at the time of the Lambda invocation. For more information about invocation types, see Invoke (p. 358).

- Regardless of the invocation type, the 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` (synchronous) invocation type, the method returns response body as follows:
  - If `error` is null, set the response body to the JSON representation of `result`. This is similar to `context.succeed()`.
  - If the `error` is not null or 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 both the `done(error, null)` and `fail(error)`, Lambda logs the first 256 KB of the error object, and for larger error object, AWS Lambda truncates the log and displays the text `Truncated by Lambda`" next to the error object.

Transitioning Lambda Function Code to Node.js Runtime v4.3

You can update your existing Lambda function to use Node.js runtime v4.3 using UpdateFunctionConfiguration (p. 387) or using the console.

Note
You might need to update your Lambda code to work in the new runtime. For a list of changes in Node.js v4.3, see API changes between v0.10 and v4 on GitHub.

If you previously created Lambda functions using Node.js runtime v0.10.42, you used one of the `context` object methods (`done()`, `succeed()`, and `fail()`) to terminate your Lambda function. In Node.js runtime v4.3, these methods are supported primarily for backward compatibility. We recommend you use the `callback` (see Using the Callback Parameter (p. 11)). The following are `callback` examples equivalent to the `context` object methods:

- The following example shows the `context.done()` method and corresponding equivalent `callback` supported in the newer runtime.

```
// Old way (Node.js runtime v0.10.42).
context.done(null, 'Success message');

// New way (Node.js runtime v4.3).
context.callbackWaitsForEmptyEventLoop = false;
callback(null, 'Success message');
```

Important
For performance reasons, AWS Lambda may reuse the same Node.js process for multiple executions of the Lambda function. If this happens, AWS Lambda freezes the Node process between execution,retaining the state information it needs to continue execution.
When the `context` methods are called, AWS Lambda freezes the Node process immediately, without waiting for the event loop associated with the process to empty. The process state and any events in the event loop are frozen. When the function is invoked again, if AWS Lambda re-uses the frozen process, the function execution continues with its same global state (for example, events that remained in the event loop will begin to get processed). However, when you use callback, AWS Lambda continues the Lambda function execution until the event loop is empty. After all events in the event loop are processed, AWS Lambda then freezes the Node process, including any state variables in the Lambda function. Therefore, if you want the same behavior as the context methods, you must set the `context` object property, `callbackWaitsForEmptyEventLoop`, to false.

- The following example shows the `context.succeed()` method and corresponding equivalent `callback` supported in the newer runtime.

```
// Old way (Node.js runtime v0.10.42).
```

```
context.succeed('Success message');

// New way (Node.js runtime v4.3).
context.callbackWaitsForEmptyEventLoop = false;
callback(null, 'Success message');
```

- The following example shows the `context.fail()` method and corresponding equivalent `callback` supported in the newer runtime.

```
// Old way (Node.js runtime v0.10.42).
context.fail('Fail object');

// New way (Node.js runtime v4.3).
context.callbackWaitsForEmptyEventLoop = false;
callback('Fail object', 'Failed result');
```

Programming Model for Authoring Lambda Functions in Java

The following sections explain how common programming patterns and core concepts apply when authoring Lambda function code in Java.

Topics

Additionally, note that AWS Lambda provides the following libraries:

- **aws-lambda-java-core** – This library provides the Context object, `RequestStreamHandler`, and the `RequestHandler` interfaces. The `Context` object (The Context Object (Java) (p. 32)) 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. 28).
- **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.
- **Custom Appender for Log4j 1.2** – You can use the custom Log4j (see Apache log4j) appender provided by AWS Lambda for logging from your lambda functions. For more information, see Logging (Java) (p. 34).

These libraries are available through the Maven Central Repository and can also be found on GitHub.

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 the 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. 28).

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. 23).
- 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. 32).

For example, consider the following Java example code.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class Hello {
    public String myHandler(int myCount, Context context) {
        return String.valueOf(myCount);
    }
}
```

In this example input is of type int and output is of type String. If you package this code and dependencies, and create your Lambda function, you specify `example.Hello::myHandler` (*`package`*.*`class`*::*`method-reference`*) as the handler.

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 an int (as in this example) or any custom data object.

For instructions to create a Lambda function using this Java code, see Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169).

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, 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. 23).
- For information about using predefined interfaces to create a handler, see Leveraging Predefined Interfaces for Creating Handler (Java) (p. 28).

  If you implement these interfaces, you can validate your handler method signature at compile time.
- If your Lambda function throws an exception, AWS Lambda records metrics in CloudWatch indicating that an error occurred. For more information, see Exceptions (Java) (p. 38).

Handler Input/Output Types (Java)

When AWS Lambda executes the Lambda function, it invokes the 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. 27).)

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**
> When you invoke a Lambda function asynchronously, any return value by your Lambda function will be ignored. Therefore you might want to set the return type to void to make this clear in your code. For more information, see Invoke (p. 358).

To test an end-to-end example, see Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169).

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.getFirstName(), 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. 25).

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 handleRequest(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 `handler` method.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class HelloPojo implements RequestHandler<RequestClass,
 ResponseClass>{

    public ResponseClass handleRequest(RequestClass request, Context context)
{
        String greetingString = String.format("Hello %s, %s.",
 request.firstName, request.lastName);
        return new ResponseClass(greetingString);
    }
}
```

In order to implement the input type, add the following code to a separate file and name it *RequestClass.java*. Place it next to the *HelloPojo.java* class in your directory structure:

```
package example;

     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() {
        }
    }
```

In order to implement the output type, add the following code to a separate file and name it *ResponseClass.java*. Place it next to the *HelloPojo.java* class in your directory structure:

```
package example;

     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() {
        }

    }
```

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 files, 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. For more information, see Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169). 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 handler 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.

Note
The input payload must be valid JSON but the output stream does not carry such a restriction. Any bytes are supported.

```
package example;

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 static void handler(InputStream inputStream, OutputStream
 outputStream, 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. For more information, see Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169). 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 the handler value.

Leveraging Predefined Interfaces for Creating Handler (Java)

You can use one of the predefined interfaces provided by the AWS Lambda Java core library (`aws-lambda-java-core`) to create your Lambda function handler, as an alternative to writing your own handler method with an arbitrary name and parameters. For more information about handlers, see (see Lambda Function Handler (Java) (p. 21)).

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 `RequestHandler` 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 Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169)). 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 the 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 Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169)). 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 the 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.
- `getFunctionVersion()`: The Lambda function version that is executing. If an alias is used to invoke the function, then `getFunctionVersion` will be the version the alias points to.
- `getInvokedFunctionArn()`: The ARN used to invoke this function. It can be function ARN or alias ARN. An unqualified ARN executes the `$LATEST` version and aliases execute the function version it is pointing to.
- `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. 34).

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.getRemainingTimeInMillis());
       System.out.println("CloudWatch log stream name: " +
 context.getLogStreamName());
       System.out.println("CloudWatch log group name: " +
 context.getLogGroupName());
}
```

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
 outputStream, Context context) {

        int letter;
        try {
            while((letter = inputStream.read()) != -1)
            {
                outputStream.write(Character.toUpperCase(letter));
            }
            Thread.sleep(3000); // Intentional delay for testing the
 getRemainingTimeInMillis() 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.getRemainingTimeInMillis());
        System.out.println("CloudWatch log stream name: " +
 context.getLogStreamName());
        System.out.println("CloudWatch log group name: " +
 context.getLogGroupName());
    }
}
```

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. For more information, see Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169). 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)

Your Lambda function can contain logging statements. AWS Lambda writes these logs to CloudWatch. We recommend you use one of the following to write logs.

- Custom Appender for Log4j™ 1.2

  AWS Lambda supports Log4j 1.2 by providing a custom appender. You can use the custom Log4j (see Apache log4j) appender provided by Lambda for logging from your lambda functions. Every call to Log4j methods, such as `log.debug()` or `log.error()`, will result in a CloudWatch Logs event. The custom appender is called `LambdaAppender` and must be used in the `log4j.properties` file. You must include the `aws-lambda-java-log4j` artifact (`artifactId:aws-lambda-java-log4j`) in the deployment package (.jar file). For an example, see Example 1: Writing Logs Using Log4J (Java) (p. 35).

  > **Note**
  > Currently, AWS Lambda supports Log4j 1.2 version.
- LambdaLogger.log()

  Each call to `LambdaLogger.log()` results in a CloudWatch Logs event, provided the event size is within the allowed limits. For information about CloudWatch logs limits, see CloudWatch Logs Limits in the *Amazon CloudWatch User Guide*. For an example, see Example 2: Writing Logs Using LambdaLogger (Java) (p. 36).

In addition, you can also use the following statements in your Lambda function code to generate log entries:

- System.out()
- System.err()

However, note that 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 log entry has multiple lines of output, AWS Lambda attempts to parse them using line breaks to identify separate events. For example, the following logs the two words ("Hello" and "world") as two separate events:

```
System.out.println("Hello \n world");
```

How to Find Logs

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 `RequestResponse` (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. 358). 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`.

Logging Examples (Java)

This section provides examples of using Custom Appender for Log4j and the `LambdaLogger` objects for logging information.

Example 1: Writing Logs Using Log4J (Java)

The following Java code example writes logs using both the System methods and Log4j to illustrate how they differ when AWS Lambda logs information to CloudWatch.

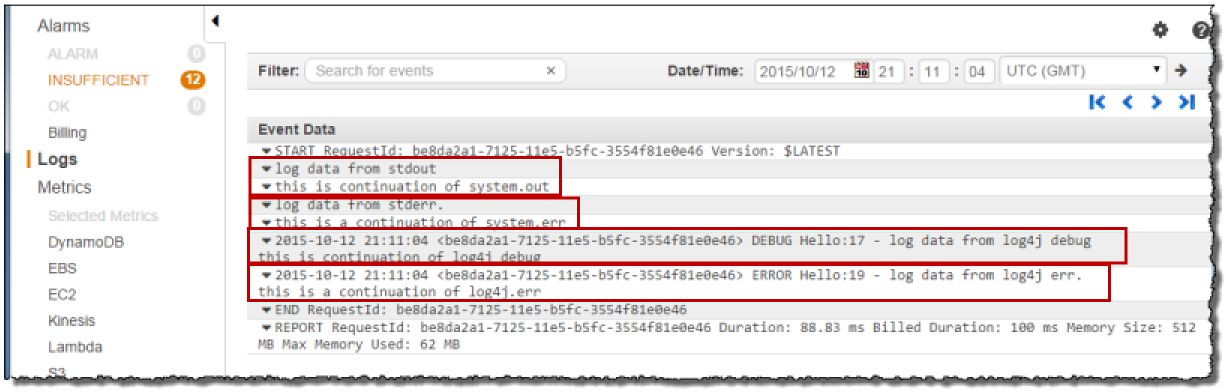
```
package example;

import com.amazonaws.services.lambda.runtime.Context;

import org.apache.logging.log4j.Logger;

public class Hello {
    // Initialize the Log4j logger.
    static final Logger log = Logger.getLogger(Hello.class);

    public String myHandler(String name, Context context) {
        // System.out: One log statement but with a line break (AWS Lambda
 writes two events to CloudWatch).
        System.out.println("log data from stdout \n this is continuation of
 system.out");

       // System.err: One log statement but with a line break (AWS Lambda
 writes two events to CloudWatch).
        System.err.println("log data from stderr. \n this is a continuation
 of system.err");

        // Use log4j to log the same thing as above and AWS Lambda will log
 only one event in CloudWatch.
        log.debug("log data from log4j debug \n this is continuation of log4j
 debug");

        log.error("log data from log4j err. \n this is a continuation of
 log4j.err");

        // 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.getLogStreamName());
    }
}
```

The example uses the following log4j.properties file (*`project-dir`*/src/main/resources/ directory).

```
log = .
log4j.rootLogger = DEBUG, LAMBDA
```

```
#Define the LAMBDA appender
log4j.appender.LAMBDA=com.amazonaws.services.lambda.runtime.log4j.LambdaAppender
log4j.appender.LAMBDA.layout=org.apache.log4j.PatternLayout
log4j.appender.LAMBDA.layout.conversionPattern=%d{yyyy-MM-dd HH:mm:ss} <
%X{AWSRequestId}> %-5p %c{1}:%L - %m%n
```

The following is sample of log entries in CloudWatch Logs.



```
START RequestId: be8da2a1-7125-11e5-b5fc-3554f81e0e46 Version: $LATEST
log data from stdout
this is continuation of system.out
log data from stderr.
this is a continuation of system.err
2015-10-12 21:11:04 <be8da2a1-7125-11e5-b5fc-3554f81e0e46> DEBUG Hello:17 - log data from log4j debug
this is continuation of log4j debug
2015-10-12 21:11:04 <be8da2a1-7125-11e5-b5fc-3554f81e0e46> ERROR Hello:19 - log data from log4j err.
this is a continuation of log4j.err
END RequestId: be8da2a1-7125-11e5-b5fc-3554f81e0e46
REPORT RequestId: be8da2a1-7125-11e5-b5fc-3554f81e0e46 Duration: 88.83 ms Billed Duration: 100 ms Memory Size: 512
MB Max Memory Used: 62 MB
```

Note:

- AWS Lambda parses the log string in each of the `System.out.println()` and `System.err.println()` statements logs as two separate events (note the two down arrows in the screenshot) because of the line break.
- The Log4j methods (`log.debug()` and `log.error()`) produce one CloudWatch event.
- AWS Lambda runtime adds the `AWSRequestId` in the MDC (see Class MDC). To get this value in the log as shown, we added `%X{AWSRequestId}` in the conversion pattern in the `log4.properties` file.

You can do the following to test the code:

- Using the code, create a deployment package. In your project, don't forget to add the `log4j.properties` files in the *project-dir*/src/main/resources/ directory.
- Upload the deployment package to AWS Lambda to create your Lambda function.
- To test your Lambda function use a string ("this is a test") as sample event. The handler code receives the sample event but does nothing with it. It only shows how to write logs.

Follow the instructions provided in the Getting Started. For more information, see Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169). Note the following differences:

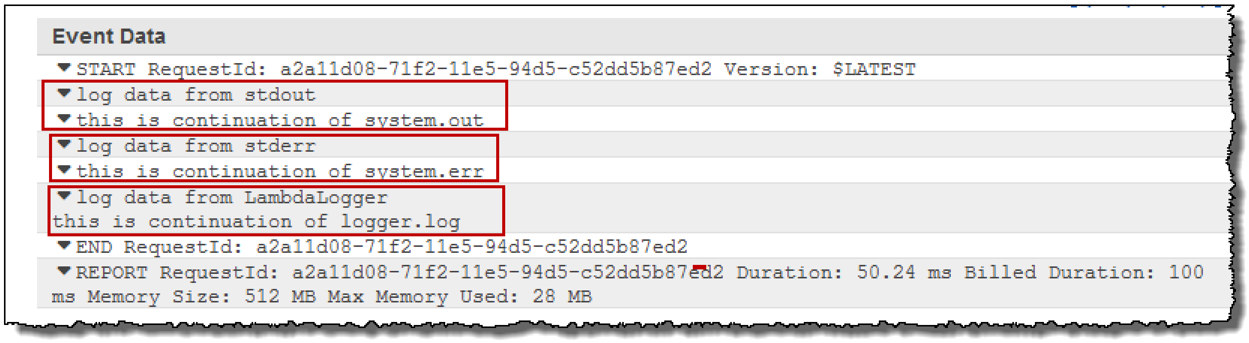
- When you create a deployment package, don't forget the `aws-lambda-java-log4j` library dependency.
- When you create the Lambda function, specify `example.Hello::myHandler` (*package*.*class*::*method*) as the handler value.

Example 2: Writing Logs Using LambdaLogger (Java)

The following Java code example writes logs using both the System methods and the `LambdaLogger` object to illustrate how they differ when AWS Lambda logs information to CloudWatch.

```
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) {

         // System.out: One log statement but with a line break (AWS Lambda
 writes two events to CloudWatch).
        System.out.println("log data from stdout \n this is continuation of
 system.out");

        // System.err: One log statement but with a line break (AWS Lambda
 writes two events to CloudWatch).
        System.err.println("log data from stderr \n this is continuation of
 system.err");

        LambdaLogger logger = context.getLogger();
        // Write log to CloudWatch using LambdaLogger.
        logger.log("log data from LambdaLogger \n this is continuation of
 logger.log");

        // 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.getLogStreamName());
    }
}
```

The following is sample of log entries in CloudWatch Logs.



```
Event Data
▼START RequestId: a2a11d08-71f2-11e5-94d5-c52dd5b87ed2 Version: $LATEST
▼log data from stdout
▼this is continuation of system.out
▼log data from stderr
▼this is continuation of system.err
▼log data from LambdaLogger
this is continuation of logger.log
▼END RequestId: a2a11d08-71f2-11e5-94d5-c52dd5b87ed2
▼REPORT RequestId: a2a11d08-71f2-11e5-94d5-c52dd5b87ed2 Duration: 50.24 ms Billed Duration: 100
ms Memory Size: 512 MB Max Memory Used: 28 MB
```

Note:

- AWS Lambda parses the log string in each of the `System.out.println()` and `System.err.println()` statements logs as two separate events (note the two down arrows in the screenshot) because of the line break.
- The `LambdaLogger.log()` produce one CloudWatch event.

You can do the following to test the code:

- Using the code, create a deployment package.

- Upload the deployment package to AWS Lambda to create your Lambda function.
- To test your Lambda function use a string ("this is a test") as sample event. The handler code receives the sample event but does nothing with it. It only shows how to write logs.

Follow the instructions provided in the Getting Started. For more information, see Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169). 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 an exception, AWS Lambda recognizes the failure and serializes the exception information into JSON and returns it. Following is an example error message:

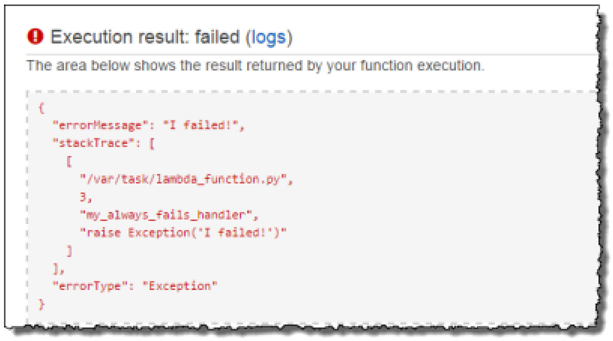
```
{
  "errorMessage": "Name John Doe is invalid. Exception occurred...",
  "errorType": "java.lang.Exception",
  "stackTrace": [
    "example.Hello.handler(Hello.java:9)",
    "sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)",

 "sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)",

 "sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.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.



Execution result: failed (logs)

The area below shows the result returned by your function execution.

```
{
  "errorMessage": "I failed!",
  "stackTrace": [
    [
      "/var/task/lambda_function.py",
      3,
      "my_always_fails_handler",
      "raise Exception('I failed!')"
    ]
  ],
  "errorType": "Exception"
}
```

- `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 retries the failed function until the Lambda function succeeds or the records in the stream expire.

Programming Model for Authoring Lambda Functions in Python

The following sections explain how common programming patterns and core concepts apply when authoring Lambda function code in Python.

Topics

Lambda Function Handler (Python)

At the time you create a Lambda function, you specify a *handler*, which is a function in your code, that AWS Lambda can invoke when the service executes your code. Use the following general syntax structure when creating a handler function in Python.

```
def handler_name(event, context):
    ...
    return some_value
```

In the syntax, note the following:

- `event` – AWS Lambda uses this parameter to pass in event data to the handler. This parameter is usually of the Python `dict` type. It can also be `list`, `str`, `int`, `float`, or `NoneType` type.
- `context` – AWS Lambda uses this parameter to provide runtime information to your handler. This parameter is of the `LambdaContext` type.
- Optionally, the handler can return a value. What happens to the returned value depends on the invocation type you use when invoking the Lambda function:
  - If you use the `RequestResponse` invocation type (synchronous execution), AWS Lambda returns the result of the Python function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the `RequestResponse` invocation type, so when you invoke the function using the console, the console will display the returned value.

    If the handler does not return anything, AWS Lambda returns null.
  - If you use the `Event` invocation type (asynchronous execution), the value is discarded.

For example, consider the following Python example code.

```
def my_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'],
                                    event['last_name'])
    return {
        'message' : message
    }
```

This example has one function called `my_handler`. The function returns a message containing data from the event it received as input.

To upload and test this code as a Lambda function

1. Save this file (for example, as `hello_python.py`).
2. Package the file and any dependencies into a .zip file. When creating the zip, include only the code and its dependencies, not the containing folder.

   For instructions, see Creating a Deployment Package (Python) (p. 69).
3. Upload the .zip file using either the console or AWS CLI to create a Lambda function. You specify the function name in the Python code to be used as the handler when you create a Lambda function. For instructions to create a Lambda function using the console, see Step 2.1: Create a Hello World Lambda Function (p. 164). In this example, the handler is `hello_python.my_handler` (*`file-name.function-name`*). Note that the Getting Started (p. 160) uses a blueprint that provides sample code for a Lambda function. In this case you already have a deployment package. Therefore, in the configure function step you choose to upload a zip.

   The following `create-function` AWS CLI command creates a Lambda function. Among other parameters, it specifies the `--handler` parameter to specify the handler name.

   ```
   aws lambda create-function \
   --region us-west-2 \
   --function-name HelloPython \
   --zip-file fileb://deployment-package.zip \
   --role arn:aws:iam::account-id:role/lambda_basic_execution  \
   --handler hello_python.my_handler \
   --runtime python2.7 \
   --timeout 15 \
   --memory-size 512
   ```

The Context Object (Python)

Topics

While a Lambda function is executing, it can interact with the AWS Lambda service to get useful runtime information such as:

- How much time is remaining before AWS Lambda terminates your Lambda function (timeout is one of the Lambda function configuration properties).
- The CloudWatch log group and log stream associated with the Lambda function that is executing.
- The AWS request ID returned to the client that invoked the Lambda function. You can use the request ID for any follow up inquiry with AWS support.

- If the Lambda function is invoked through AWS Mobile SDK, you can learn more about the mobile application calling the Lambda function.

AWS Lambda provides this information via the `context` object that the service passes as the second parameter to your Lambda function handler. For more information, see Lambda Function Handler (Python) (p. 39).

The following sections provide an example Lambda function that uses the `context` object, and then lists all of the available methods and attributes.

Example

Consider the following Python example. It has one function that is also the handler. The handler receives runtime information via the `context` object passed as parameter.

```
from __future__ import print_function

import time
def get_my_log_stream(event, context):
    print("Log stream name:", context.log_stream_name)
    print("Log group name:",  context.log_group_name)
    print("Request ID:",context.aws_request_id)
    print("Mem. limits(MB):", context.memory_limit_in_mb)
    # Code will execute quickly, so we add a 1 second intentional delay so
 you can see that in time remaining value.
    time.sleep(1)
    print("Time remaining (MS):", context.get_remaining_time_in_millis())
```

The handler code in this example simply prints some of the runtime information. Each print statement creates a log entry in CloudWatch. If you invoke the function using the Lambda console, the console displays the logs. The `from __future__` statement enables you to write code that is compatible with Python 2 or 3.

To test this code in the AWS Lambda console

1. In the console, create a Lambda function using the hello-world blueprint. In **runtime**, choose **Python 2.7**. In **Handler**, replace `lambda_function.lambda_handler` with `lambda_function.get_my_log_stream`. For instructions on how to do this, see Step 2.1: Create a Hello World Lambda Function (p. 164).
2. Test the function, and then you can also update the code to get more context information.

The following sections provide a list of available `context` object methods and attributes that you can use to get runtime information of your Lambda function.

The Context Object Methods (Python)

The context object provides the following methods:

get_remaining_time_in_millis()
Returns the remaining execution time, in milliseconds, until AWS Lambda terminates the function.

The Context Object Attributes (Python)

The context object provides the following attributes:

function_name
Name of the Lambda function that is executing.

function_version
The Lambda function version that is executing. If an alias is used to invoke the function, then `function_version` will be the version the alias points to.

invoked_function_arn
The ARN used to invoke this function. It can be function ARN or alias ARN. An unqualified ARN executes the `$LATEST` version and aliases execute the function version it is pointing to.

memory_limit_in_mb
Memory limit, in MB, you configured for the Lambda function. You set the memory limit at the time you create a Lambda function and you can change it later.

aws_request_id
AWS request ID associated with the request. This is the ID returned to the client that called the `invoke` method.

> **Note**
> If AWS Lambda retries the invocation (for example, in a situation where the Lambda function that is processing Amazon Kinesis records throws an exception), the request ID remains the same.

log_group_name
The name of the CloudWatch log group where you can find logs written by your Lambda function.

log_stream_name
The name of the CloudWatch log stream where you can find logs written by your Lambda function. The log stream may or may not change for each invocation of the Lambda function.

The value is null if your Lambda function is unable to create a log stream, which can happen if the execution role that grants necessary permissions to the Lambda function does not include permissions for the CloudWatch Logs actions.

identity
Information about the Amazon Cognito identity provider when invoked through the AWS Mobile SDK. It can be null.

- **identity.cognito_identity_id**
- **identity.cognito_identity_pool_id**

client_context
Information about the client application and device when invoked through the AWS Mobile SDK. It can be null.

- **client_context.client.installation_id**
- **client_context.client.app_title**
- **client_context.client.app_version_name**
- **client_context.client.app_version_code**
- **client_context.client.app_package_name**
- **client_context.custom**

  A `dict` of custom values set by the mobile client application.
- **client_context.env**

  A `dict` of environment information provided by the AWS Mobile SDK.

Logging (Python)

Your Lambda function can contain logging statements. AWS Lambda writes these logs to CloudWatch. If you use the Lambda console to invoke your Lambda function, the console displays the same logs.

The following Python statements generate log entries:

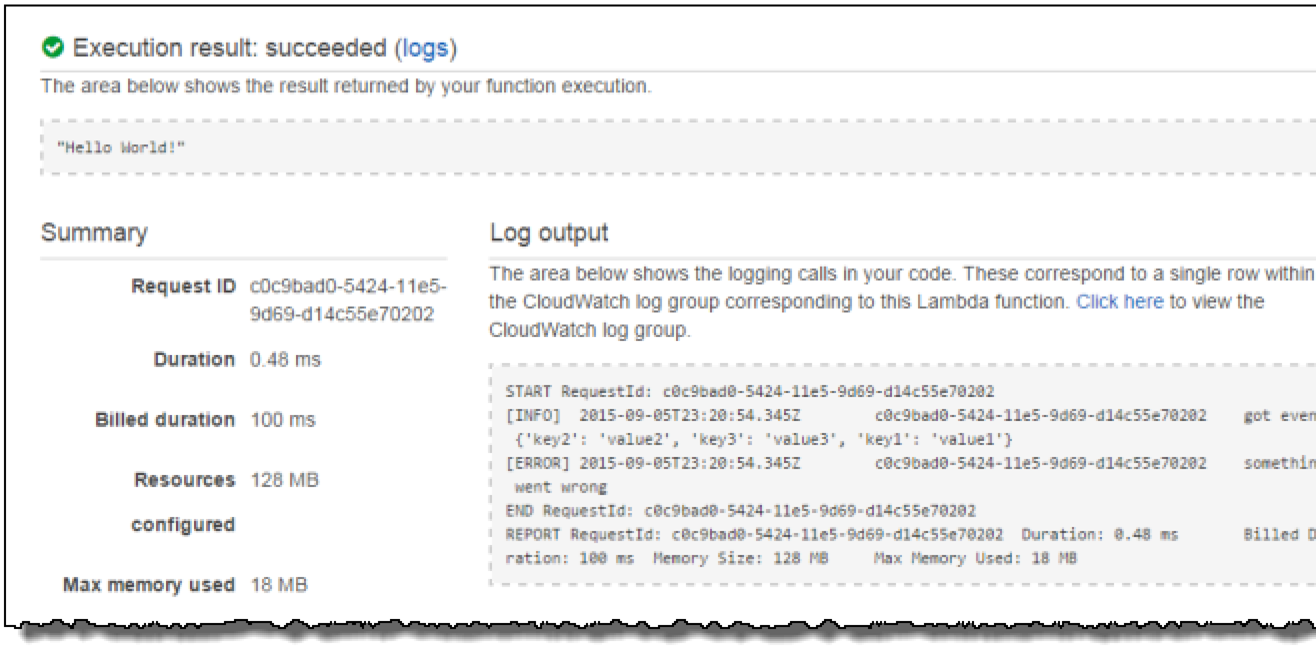
- `print` statements.
- `Logger` functions in the `logging` module (for example, `logging.Logger.info` and `logging.Logger.error`).

Both `print` and `logging.*` functions write logs to CloudWatch Logs but the `logging.*` functions write additional information to each log entry, such as time stamp and log level.

For example, consider the following Python code example.

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
def my_logging_handler(event, context):
    logger.info('got event{}'.format(event))
    logger.error('something went wrong')
    return 'Hello World!'
```

Because the code example uses the `logging` module to write message to the logs, you also get some additional information in the log such as the time stamp and the log levels. The log level identifies the type of log, such as `[INFO]`, `[ERROR]`, and `[DEBUG]`, as shown:



Execution result: succeeded (logs)

The area below shows the result returned by your function execution.

```
"Hello World!"
```

Summary

Request ID	c0c9bad0-5424-11e5-9d69-d14c55e70202
Duration	0.48 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	18 MB

Log output

The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

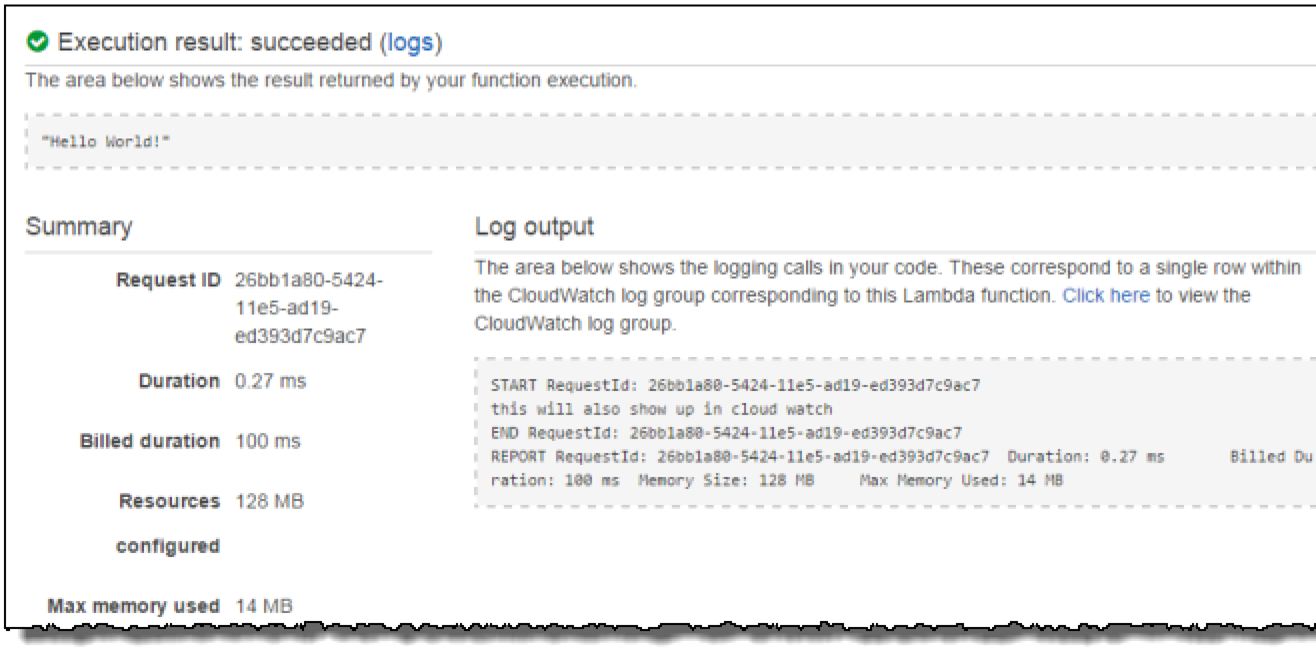
```
START RequestId: c0c9bad0-5424-11e5-9d69-d14c55e70202
[INFO]  2015-09-05T23:20:54.345Z        c0c9bad0-5424-11e5-9d69-d14c55e70202    got even
 {'key2': 'value2', 'key3': 'value3', 'key1': 'value1'}
[ERROR] 2015-09-05T23:20:54.345Z        c0c9bad0-5424-11e5-9d69-d14c55e70202    somethin
 went wrong
END RequestId: c0c9bad0-5424-11e5-9d69-d14c55e70202
REPORT RequestId: c0c9bad0-5424-11e5-9d69-d14c55e70202  Duration: 0.48 ms       Billed D
ration: 100 ms  Memory Size: 128 MB     Max Memory Used: 18 MB
```

The screen shot shows an example **Log output** section in the Lambda console; you can also find these logs in CloudWatch. For more information, see Accessing Amazon CloudWatch Logs for AWS Lambda (p. 110).

Instead of using the `logging` module, you can use the `print` statements in your code as shown in the following Python example:

```
from __future__ import print_function
def my_other_logging_handler(event, context):
    print('this will also show up in cloud watch')
    return 'Hello World!'
```

In this case only the text passed to the print method is sent to CloudWatch. The log entries will not have additional information that the `logging.*` function returns. The `from __future__` statement enables you to write code that is compatible with Python 2 or 3.



Execution result: succeeded (logs)

The area below shows the result returned by your function execution.

```
"Hello World!"
```

Summary

Request ID	26bb1a80-5424-11e5-ad19-ed393d7c9ac7
Duration	0.27 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	14 MB

Log output

The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: 26bb1a80-5424-11e5-ad19-ed393d7c9ac7
this will also show up in cloud watch
END RequestId: 26bb1a80-5424-11e5-ad19-ed393d7c9ac7
REPORT RequestId: 26bb1a80-5424-11e5-ad19-ed393d7c9ac7  Duration: 0.27 ms      Billed Du
ration: 100 ms   Memory Size: 128 MB     Max Memory Used: 14 MB
```

The console uses the `RequestResponse` invocation type (synchronous invocation) when invoking the function. And therefore it gets the return value ("Hello world!") back from AWS Lambda which the console displays.

To test the preceding Python code (console)

1. In the console, create a Lambda function using the hello-world-python blueprint. In **runtime**, choose **Python 2.7**. In **Handler**, replace `lambda_function.lambda_handler` with `lambda_function.my_other_logging_handler` and in **Role**, choose **Basic execution role**. You also replace the code provided by the blueprint by the code in this section. For step-by-step instructions to create a Lambda function using the console, see Step 2.1: Create a Hello World Lambda Function (p. 164).
2. Replace the template code with the code provided in this section.
3. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console.

Finding Logs

You can find the logs that your Lambda function writes, as follows:

- **In the AWS Lambda console** – The **Log output** section in AWS Lambda console shows the logs.
- **In the response header, when you invoke a Lambda function programmatically** – 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. AWS Lambda returns this log information in the `x-amz-log-results` header in the response. For more information, see Invoke (p. 358).

  If you use AWS CLI to invoke the function, you can specify the `--log-type parameter` with value `Tail` to retrieve the same information.
- **In CloudWatch Logs** – To find your logs in CloudWatch you need to know the log group name and log stream name. You can use the `context.logGroupName`, and `context.logStreamName`

properties in your code to get this information. When you run your Lambda function, the resulting logs in the console or CLI will show you the log group name and log stream name.

Exceptions (Python)

If your Lambda function raises an exception, AWS Lambda recognizes the failure and serializes the exception information into JSON and returns it. Consider the following example:

```
def always_failed_handler(event, context):
    raise Exception('I failed!')
```

When you invoke this Lambda function, it will raise an exception and AWS Lambda returns the following error message:

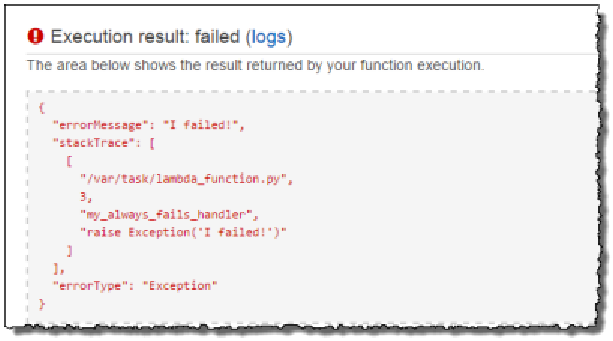
```
{
  "errorMessage": "I failed!",
  "stackTrace": [
    [
      "/var/task/lambda_function.py",
      3,
      "my_always_fails_handler",
      "raise Exception('I failed!')"
    ]
  ],
  "errorType": "Exception"
}
```

Note that the stack trace is returned as the `stackTrace` JSON array of stack trace elements.

How you get the error information back depends on the invocation type that the client specifies at the time of function invocation:

- If a client specifies the `RequestResponse` invocation type (that is, synchronous execution), it returns the result to the client that made the invoke call.

  For example, the console always use the `RequestResponse` invocation type, so the console will display the error in the **Execution result** section as shown:

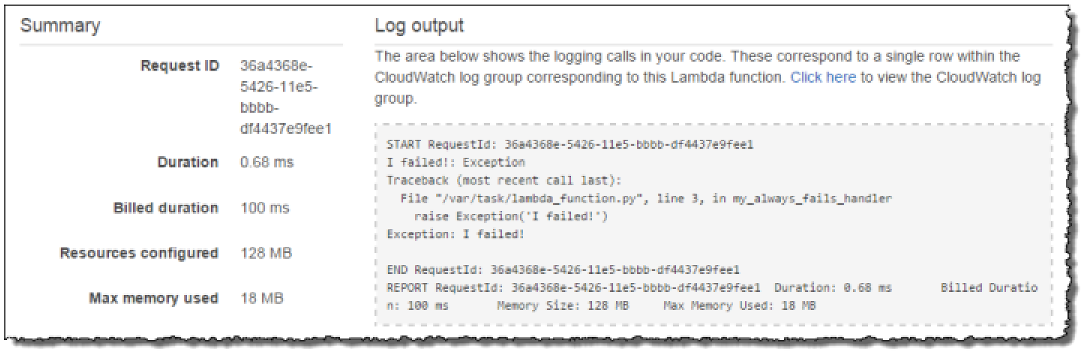


Execution result: failed (logs)

The area below shows the result returned by your function execution.

```
{
  "errorMessage": "I failed!",
  "stackTrace": [
    [
      "/var/task/lambda_function.py",
      3,
      "my_always_fails_handler",
      "raise Exception('I failed!')"
    ]
  ],
  "errorType": "Exception"
}
```

  The same information is also sent to CloudWatch and the **Log output** section shows the same logs.



Summary

Request ID	36a4368e-5426-11e5-bbbb-df4437e9fee1
Duration	0.68 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	18 MB

Log output

The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: 36a4368e-5426-11e5-bbbb-df4437e9fee1
I failed!: Exception
Traceback (most recent call last):
  File "/var/task/lambda_function.py", line 3, in my_always_fails_handler
    raise Exception('I failed!')
Exception: I failed!

END RequestId: 36a4368e-5426-11e5-bbbb-df4437e9fee1
REPORT RequestId: 36a4368e-5426-11e5-bbbb-df4437e9fee1  Duration: 0.68 ms       Billed Duratio
n: 100 ms       Memory Size: 128 MB     Max Memory Used: 18 MB
```

- If a client specifies the `Event` invocation type (that is, asynchronous execution), AWS Lambda will not return anything. Instead, it logs the error information to CloudWatch Logs. You can also see the error metrics in CloudWatch Metrics.

Depending on the event source, AWS Lambda may retry the failed Lambda function. For example, if Amazon Kinesis is the event source, AWS Lambda will retry the failed invocation until the Lambda function succeeds or the records in the stream expire.

To test the preceding Python code (console)

1. In the console, create a Lambda function using the hello-world blueprint. In **runtime**, choose **Python 2.7**. In **Handler**, replace `lambda_function.lambda_handler` with `lambda_function.always_failed_handler`. For instructions on how to do this, see Step 2.1: Create a Hello World Lambda Function (p. 164).
2. Replace the template code with the code provided in this section.
3. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console.

Programming Model for Authoring Lambda Functions in C#

The following sections explain how common programming patterns and core concepts apply when authoring Lambda function code in C#.

Topics

Additionally, note that AWS Lambda provides the following:

- **Amazon.Lambda.Core** – This library provides a static Lambda logger, serialization interfaces and a context object. The `Context` object (The Context Object (C#) (p. 51)) provides runtime information about your Lambda function.
- **Amazon.Lambda.Serialization.Json** – This an implementation of the serialization interface in **Amazon.Lambda.Core**.
- **Amazon.Lambda.Logging.AspNetCore** – This provides a library for logging from ASP.NET.
- Event objects (POCOs) for several AWS services, including:

- **Amazon.Lambda.APIGatewayEvents**
- **Amazon.Lambda.CognitoEvents**
- **Amazon.Lambda.ConfigEvents**
- **Amazon.Lambda.DynamoDBEvents**
- **Amazon.Lambda.KinesisEvents**
- **Amazon.Lambda.S3Events**
- **Amazon.Lambda.SNSEvents**

These packages are available at Nuget Packages.

Lambda Function Handler (C#)

When you create a Lambda function, you specify a handler that AWS Lambda can invoke when the service executes the function on your behalf.

You define a Lambda function handler as an instance or static method in a class. If you want access to the Lambda context object, it is available by defining a method parameter of type *ILambdaContext.*.

```
returnType handler-name(inputType input, ILambdaContext context) {
   ...
}
```

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.
- *`returnType`* – 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 `returnType` 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.

Handling Streams

Only the `System.IO.Stream` type is supported as an input parameter by default.

For example, consider the following C# example code.

```
using System.IO;
{
  namespace Example

  public class Hello
  {
    public Stream MyHandler(Stream stream)
    {
       //function logic
    }
}
```

In the example C# 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 `Stream` (as in this example) or any custom data object. The output is of type `Stream`.

Handling Standard Data Types

All other types, as listed below, require you to specify a serializer.

- Primitive .NET types (such as string or int).
- Collections and maps - IList, IEnumerable, IList<T>, Array, IDictionary, IDictionary<TKey, TValue>
- POCO types (Plain old CLR objects)
- Predefined AWS event types
- For asynchronous invocations the return-type will be ignored by Lambda. The return type may be set to void in such cases.
- If you are using .NET asynchronous programming, the return type can be Task and Task<T> types and use `async` and `await` keywords. For more information, see Using Async in C# Functions with AWS Lambda (p. 50).

Unless your function input and output parameters are of type `System.IO.Stream`, you will need to serialize them. AWS Lambda provides a default serializer that can be applied at the assembly or method level of your application, or you can define your own by implementing the `ILambdaSerializer` interface provided by the `Amazon.Lambda.Core` library. For more information, see Creating a Deployment Package (C#) (p. 57).

To add the default serializer attribute to a method, first add a dependency on `Amazon.Lambda.Serialization.Json` in your `project.json` file.

```
{
    "version": "1.0.0-*",
    "dependencies":{
        "Microsoft.NETCore.App": {
            "type": "platform",
            "version": "1.0.1"
        },
        "Amazon.Lambda.Serialization.Json": "1.0.0"
    },
    "frameworks": {
        "netcoreapp1.0": {
            "imports": "dnxcore50"
        }
    }
}
```

The example below illustrates the flexibility you can leverage by specifying the default Json.NET serializer on one method and another of your choosing on a different method:

```
public class ProductService{

 [LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]
    public Product DescribeProduct(DescribeProductRequest request)
    {
      return catalogService.DescribeProduct(request.Id);
    }

   [LambdaSerializer(typeof(MyJsonSerializer))]
   public Customer DescribeCustomer(DescribeCustomerRequest request)
```

```
    {
       return customerService.DescribeCustomer(request.Id);
    }
}
```

Handler Signatures

When creating Lambda functions, you have to provide a handler string that tells AWS Lambda where to look for the code to invoke. In C#, the format is:

`ASSEMBLY::TYPE::METHOD` where:

- *`ASSEMBLY`* is the name of the .NET assembly file for your application. When using the .NET Core CLI to build your application, if you haven't set the assembly name using the `buildOptions.outputName` setting in project.json, the *`ASSEMBLY`* name will be the name of the folder that contains your project.json file. For more information, see .NET Core CLI (p. 57). In this case, let's assume the folder name is `HelloWorldApp`.
- *`TYPE`* is the full name of the handler type, which consists of the *`Namespace`* and the *`ClassName`*. In this case `Example.Hello`.
- *`METHOD`* is name of the function handler, in this case `MyHandler`.

Ultimately, the signature will be of this format: *`Assembly::Namespace.ClassName::MethodName`*

Again, consider the following example:

```
using System.IO;
{
  namespace Example

  public class Hello
  {
    public Stream MyHandler(Stream stream)
    {
       //function logic
    }
}
```

The handler string would be: `HelloWorldApp::Example.Hello::MyHandler`

For instructions to create a Lambda function using this C# code, see Step 2.4: (Optional) Create a Lambda Function Authored in C# (p. 170).

> **Important**
> If the method specified in your handler string is overloaded, you must provide the exact signature of the method Lambda should invoke. AWS Lambda will reject an otherwise valid signature if the resolution would require selecting among multiple (overloaded) signatures.

Lambda Function Handler Restrictions

Note that there are some restrictions on the handler signature

- It may not be `unsafe` and use pointer types in the handler signature, though `unsafe` context can be used inside the handler method and its dependencies. For more information, see unsafe (C# Reference).
- It may not pass a variable number of parameters using the `params` keyword, or use `ArgIterator` as an input or return parameter which is used to support variable number of parameters.
- The handler may not be a generic method (e.g. IList<T> Sort<T>(IList<T> input)).
- Async handlers with signature `async void` are not supported.

Using Async in C# Functions with AWS Lambda

If you know your Lambda function will require a long-running process, such as uploading large files to Amazon S3 or reading a large stream of records from DynamoDB, you can take advantage of the async/await pattern. By creating a handler with this signature, Lambda will execute the function synchronously and wait a maximum of 5 minutes for execution to complete before returning or timing out. For example:

```
public async Task<Response> ProcessS3ImageResizeAsync(SimpleS3Event input)
{
   var response = await client.DoAsyncWork(input);
   return response;
}
```

If you use this pattern, there are some considerations you must take into account:

- AWS Lambda will not support `async void` methods.
- If you create an async Lambda function without implementing the `await` operator, .NET will issue a compiler warning and you will observe unexpected behavior. For example, some async actions will execute while others won't. Or some async actions won't complete before the function execution is complete.

  ```
  public async Task ProcessS3ImageResizeAsync(SimpleS3Event event) // Compiler
   warning
  {
      client.DoAsyncWork(input);
  }
  ```

- Your Lambda function can include multiple async calls, which can be invoked in parallel. You can use the `Task.WaitAll` and `Task.WaitAny` methods to work with multiple tasks. To use the `Task.WaitAll` method, you pass a list of the operations as an array to the method. Note that in the example below, if you neglect to include any operation to the array, that call may return before its operation completes.

  ```
  public async Task SaveAsync(Profile profile)
  {
    var s3Save = s3.SaveImage(profile.image);
    var ddbSave = ddb.SaveAttributes(profile.Attributes);
    var ddbSave2 = ddb.SaveConnections(profile.connections);  // Lambda will
   return before this call completes
                                                              // No compiler
   warnings
    return await Task.WaitAll(new Task[]{ s3Save, ddbSave }); // Did not
   "await" for ddbSave2
  }
  ```

  To use the `Task.WaitAny` method, you again pass a list of operations as an array to the method. The call returns as soon as the first operation completes, even if the others are still running.

  ```
  public async Task<SearchResult> SearchAsync(Query q)
  {
    var siteSearch1 = site1.SearchAsync(q);
    var siteSearch2 = site2.SearchAsyc(q);
    var siteSearch3 = site3.SearchAsync(q);
    var tasks[] = new Task[]{siteSearch1, siteSearch2, siteSearch3};
  ```

```
  var index = await Task.WaitAny(tasks); // Returns as soon as any of the
 tasks complete, other task may run in background
  return tasks[index].Result;
}
```

We do not recommend using `Task.WaitAny` for the above reasons.

The Context Object (C#)

You can gain useful information on how your Lambda function is interacting with the AWS Lambda runtime by adding the `ILambdaContext` parameter to your method. In return, AWS Lambda provides runtime details such as the CloudWatch log stream associated with the function or the id of the client that called your functions, which you access via the properties provided by the context object.

To do this, create a method with the following signature:

```
public void Handler(string Input, ILambdaContext context)
```

The context object properties are:

- `MemoryLimitInMB`: Memory limit, in MB, you configured for the Lambda function.
- `FunctionName`: Name of the Lambda function that is running.
- `FunctionVersion`: The Lambda function version that is executing. If an alias is used to invoke the function, then `FunctionVersion` will be the version the alias points to.
- `InvokedFunctionArn`: The ARN used to invoke this function. It can be function ARN or alias ARN. An unqualified ARN executes the `$LATEST` version and aliases execute the function version it is pointing to.
- `AwsRequestId`: AWS request ID associated with the request. This is the ID returned to the client that invoked this Lambda function. 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.
- `LogStreamName`: 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.
- `LogGroupName`: 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.
- `ClientContext`: 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.
- `Identity`: Information about the Amazon Cognito identity provider when invoked through the AWS Mobile SDK. It can be null.
- `RemainingTime`: Remaining execution time till the function will be terminated. 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. This is a `TimeSpan` field.
- `Logger`: The Lambda logger associated with the ILambdaContext object. For more information, see Logging (C#) (p. 52).

The following C# code snippet shows a simple handler function that displays the value of the input parameter and then prints some of the context information.

```
public async Task Handler(ILambdaContext context)
{
    Console.Writeline("Function name: " + context.FunctionName);
    Console.Writeline("RemainingTime: " + context.RemainingTime);
    await Task.Delay(TimeSpan.FromSeconds(0.42));
    Console.Writeline("RemainingTime after sleep: " + context.RemainingTime);
}
```

Logging (C#)

Your Lambda function can contain logging statements and, in turn, AWS Lambda writes these logs to CloudWatch Logs.

In the C# programming model, there are three ways to log data in your function:

- Use the static `Write` or `WriteLine` methods provided by the C# `Console` class. Anything written to standard out or standard error - using Console.Write or a similar method - will be logged in CloudWatch Logs.

```
public class ProductService
{
   public async Task<Product> DescribeProduct(DescribeProductRequest
 request)
    {
       Console.WriteLine("DescribeProduct invoked with Id " + request.Id);
       return await catalogService.DescribeProduct(request.Id);
    }
}
```

- Use the `Log` method on the `Amazon.Lambda.Core.LambdaLogger` class. This is a static class that can be used anywhere in your application. To use this, you must include the `Amazon.Lambda.Core` library.

```
using Amazon.Lambda.Core;

public class ProductService
{
   public async Task<Product> DescribeProduct(DescribeProductRequest
 request)
   {
       LambdaLogger.Log("DescribeProduct invoked with Id " + request.Id);
       return await catalogService.DescribeProduct(request.Id);
   }
}
```

  Each call to `LambdaLogger.Log` results in a CloudWatch Logs event, provided the event size is within the allowed limits. For information about CloudWatch Logs logs limits, see CloudWatch Logs Limits in the *Amazon CloudWatch User Guide*.

- Use the logger in `ILambdaContext`. The `ILambdaContext` object (if specified) in your method contains a `Logger` property that represents a LambdaLogger. The following is an example of using this method:

```
public class ProductService
{
   public async Task<Product> DescribeProduct(DescribeProductRequest
 request, ILambdaContext context)
   {
       context.Logger.Log("DescribeProduct invoked with Id " + request.Id);
       return await catalogService.DescribeProduct(request.Id);
```

```
    }
}
```

How to Find Logs

You can find the logs that your Lambda function writes, as follows:

- Find logs in CloudWatch Logs. The `ILambdaContext` object provides the `LogStreamName` and the `LogGroupName` properties. Using these properties, 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 `RequestResponse` (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 `Console.Write` and `Console.WriteLine` 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. 358). 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`.

Exceptions (C#)

When an exception occurs in your Lambda function, Lambda will report the exception information back to you. Exceptions can occur in two different places:

- Initialization (Lambda loading your code, validating the handler string, and creating an instance of your class if it is non-static).
- The Lambda function invocation.

The serialized exception information is returned as the payload as a modeled JSON object and outputted to CloudWatch logs.

In the initialization phase, exceptions can be thrown for invalid handler strings, a rule-breaking type or method (see Lambda Function Handler Restrictions (p. 49)), or any other validation method (such as forgetting the serializer attribute and having a POCO as your input or output type). These exceptions are of type `LambdaException`. For example:

```
{
  "errorType": "LambdaException",
  "errorMessage": "Invalid lambda function handler: 'http://
this.is.not.a.valid.handler/'.
  The valid format is 'ASSEMBLY::TYPE::METHOD'."
}
```

If your constructor throws an exception, the error type is also of type `LambdaException`, but the exception thrown during construction is provided in the `cause` property, which is itself a modeled exception object:

```
{
  "errorType": "LambdaException",
  "errorMessage": "An exception was thrown when the constructor for type
 'LambdaExceptionTestFunction.ThrowExceptionInConstructor'
   was invoked. Check inner exception for more details.",
  "cause":   {
    "errorType": "TargetInvocationException",
    "errorMessage": "Exception has been thrown by the target of an
 invocation.",
    "stackTrace": [
      "at System.RuntimeTypeHandle.CreateInstance(RuntimeType type, Boolean
 publicOnly, Boolean noCheck, Boolean&canBeCached,
      RuntimeMethodHandleInternal&ctor, Boolean& bNeedSecurityCheck)",
      "at System.RuntimeType.CreateInstanceSlow(Boolean publicOnly, Boolean
 skipCheckThis, Boolean fillCache, StackCrawlMark& stackMark)",
      "at System.Activator.CreateInstance(Type type, Boolean nonPublic)",
      "at System.Activator.CreateInstance(Type type)"
    ],
    "cause":     {
      "errorType": "ArithmeticException",
      "errorMessage": "Sorry, 2 + 2 = 5",
      "stackTrace": [
        "at LambdaExceptionTestFunction.ThrowExceptionInConstructor..ctor()"
      ]
    }
  }
}
```

As the example shows, the inner exceptions are always preserved (as the `cause` property), and can be deeply nested.

Exceptions can also occur during invocation. In this case, the exception type is preserved and the exception is returned directly as the payload and in the CloudWatch logs. For example:

```
{
  "errorType": "AggregateException",
  "errorMessage": "One or more errors occurred. (An unknown web exception
 occurred!)",
  "stackTrace": [
    "at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean
 includeTaskCanceledExceptions)",
    "at System.Threading.Tasks.Task`1.GetResultCore(Boolean
 waitCompletionNotification)",
    "at lambda_method(Closure , Stream , Stream , ContextInfo )"
  ],
  "cause":   {
    "errorType": "UnknownWebException",
    "errorMessage": "An unknown web exception occurred!",
    "stackTrace": [
      "at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()",
      "--- End of stack trace from previous location where exception was
 thrown ---",
      "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task
 task)",
      "at
 System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
 task)",
      "at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()",
      "at LambdaDemo107.LambdaEntryPoint.<CheckWebsiteStatus>d__0.MoveNext()"
    ],
    "cause":     {
      "errorType": "WebException",
      "errorMessage": "An error occurred while sending the request. SSL peer
 certificate or SSH remote key was not OK",
      "stackTrace": [
        "at System.Net.HttpWebRequest.EndGetResponse(IAsyncResult
 asyncResult)",
        "at
 System.Threading.Tasks.TaskFactory`1.FromAsyncCoreLogic(IAsyncResult
 iar, Func`2 endFunction, Action`1 endAction, Task`1 promise, Boolean
 requiresSynchronization)",
        "--- End of stack trace from previous location where exception was
 thrown ---",
        "at
 System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
        "at
 System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
 task)",
        "at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()",
        "at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()"
      ],
      "cause":       {
        "errorType": "HttpRequestException",
        "errorMessage": "An error occurred while sending the request.",
        "stackTrace": [
          "at
 System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
          "at
 System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
 task)",
          "at System.Net.Http.HttpClient.<FinishSendAsync>d__58.MoveNext()",
          "--- End of stack trace from previous location where exception was
 thrown ---",
          "at
 System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
          "at
 System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
 task)",
          "at System.Net.HttpWebRequest.<SendRequest>d__63.MoveNext()",
          "--- End of stack trace from previous location where exception was
 thrown ---",
          "at
 System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
          "at
 System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
 task)",
          "at System.Net.HttpWebRequest.EndGetResponse(IAsyncResult
 asyncResult)"
        ],
        "cause":         {
          "errorType": "CurlException",
          "errorMessage": "SSL peer certificate or SSH remote key was not
 OK",
          "stackTrace": [
            "at System.Net.Http.CurlHandler.ThrowIfCURLEError(CURLcode
 error)",
            "at
 System.Net.Http.CurlHandler.MultiAgent.FinishRequest(StrongToWeakReference`1
 easyWrapper, CURLcode messageResult)"
          ]
        }
      }
    }
  }
```

```
}
```

The method in which error information is conveyed depends on the invocation type:

- `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 of the console.
- `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 more information, see Retries on Errors (p. 154).

Creating a Deployment Package

To create a Lambda function you first create a Lambda function deployment package, a .zip or .jar file consisting of your code and any dependencies. When creating the zip, include only the code and its dependencies, not the containing folder.

- Creating a Deployment Package (Node.js) (p. 56)
- Creating a Deployment Package (Java) (p. 62)
- Creating a Deployment Package (C#) (p. 57)
- Creating a Deployment Package (Python) (p. 69)

Creating a 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.

You can create a deployment package yourself or write your code directly in the Lambda console, in which case the console creates the deployment package for you and uploads it, creating your Lambda function. Note the following to determine if you can use the console to create your 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.

Note
After you create a deployment package, you may either upload it directly or 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 using the console or the AWS CLI.

The following is an example procedure to create a deployment package (outside the console). Suppose you want to create a deployment package that includes a `filename.js` code file and your code uses the `async` library.

1. Open a text editor, and write your code. Save the file (for example, `filename.js`).

   You will use the file name to specify the handler at the time of creating the Lambda function.
2. In the same directory, use **npm** to install the libraries that your code depends on. For example, if your code uses the `async` library, use the following **npm** command.

   ```
   npm install async
   ```
3. Your directory will then have the following structure:

   ```
   filename.js
   node_modules/async
   node_modules/async/lib
   node_modules/async/lib/async.js
   node_modules/async/package.json
   ```
4. Zip the content of the folder, that is your deployment package (for example, `sample.zip`).

Then, specify the .zip file name as your deployment package at the time you create your Lambda function.

If you want to include your own binaries, including native ones, just package them in the Zip file you upload and then reference them (including the relative path within the Zip file you created) when you call them from Node.js or from other processes that you've previously started. Ensure that you include the following at the start of your function code: `process.env['PATH'] = process.env['PATH'] + ':' + process.env['LAMBDA_TASK_ROOT']`

For more information on including native binaries in your Lambda function package, see Running Executables in AWS Lambda.

Creating a Deployment Package (C#)

You can create .NET-core based AWS Lambda applications and package them for deployment in the following ways:

- Use the .NET Core CLI, which you can download here to create your Lambda application.
- Use the Lambda plugin to the AWS ToolKit for Microsoft Visual Studio, which can you download here.

Topics

.NET Core CLI

The .NET Core CLI offers a cross-platform way for you to create .NET-based Lambda applications.

Before You Begin

This section assumes you have done the following:

- Installed the .NET Core CLI. If you haven't, do so here.

Create a .NET Project

To create an application using the .NET Core CLI, open a command prompt and navigate to the folder where you installed the .NET Core runtime and follow these steps:

1. Make a directory where your project will be created using the following command: `mkdir` *`example`*
2. Navigate to that directory using the following command: `cd` *`example`*
3. Enter the following command: `dotnet new`

   This will create two files in your *`example`* directory:

   - Program.cs, which is where you write your Lambda function code.
   - project.json, which is the file is where you declare Nuget dependencies (or dependencies on local projects). NuGet is the package manager for the .NET platform. For more information, see Nuget.org.

   Note
   Lambda methods don't use the `Main()` entry point provided by default in .NET, so open the project.json file and remove the "buildOptions" property. After this, your project.json should look something like this (exact versions may differ depending on when you installed the NetCore CLI):

```
{
  "version": "1.0.0-*",
  "dependencies": {},
  "frameworks": {
    "netcoreapp1.0": {
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.1.0"
        }
      },
      "imports": "dnxcore50"
    }
  }
}
```

4. Open the `Program.cs` file using an editor of you choice, such as Microsoft Visual Studio.

   - Replace the default code that is provided with your Lambda function handler code:

     At this point, your .cs file structure should resemble this:

```
using System;
using System.IO;

namespace CSharpLambdaFunction
{
    public class LambdaHandler
    {
       public Stream myHandler(Stream inputStream)
       {
           //function logic
       }
    }
}
```

Your Lambda function handler signature should be of the format *`Assembly::Namespace.ClassName::MethodName`*. For more information, see Handler Signatures (p. 49).

Using a Serializer

For any Lambda functions that use input or output types other than a `Stream` object, you will need to add a serialization library to your application. You can do this in the following ways:

- Use Json.NET. Lambda will provide an implementation for JSON serializer using JSON.NET as a NuGet package.
- Create your own serialization library by implementing the `ILambdaSerializer` interface, which is available as part of the `Amazon.Lambda.Core` library. The interface defines two methods:
  - `T Deserialize<T>(Stream requestStream);`

    You implement this method to deserialize the request payload from the `Invoke` API into the object that is passed to the Lambda function handler.
  - `T Serialize<T>(T response, Stream responseStream);`.

    You implement this method to serialize the result returned from the Lambda function handler into the response payload that is returned by the `Invoke` API.

You use whichever serializer you wish by adding it as a dependency to your `project.json` file.

```

{
  "version": "1.0.0-*",
  "buildOptions": {
  },

  "dependencies": {
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.1"
    },

    "Newtonsoft.Json": "9.0.1",

    "Amazon.Lambda.Core": "1.0.0*",
    "Amazon.Lambda.Serialization.Json": "1.0.0",

    "Amazon.Lambda.Tools" : {
      "type" :"build",
      "version":"0.9.0-preview1"
    }
  },

  "tools": {
    "Amazon.Lambda.Tools" : "0.9.0-preview1"
  },

  "frameworks": {
    "netcoreapp1.0": {
      "imports": "dnxcore50"
    }
  }
}
```

You then add it to your AssemblyInfo.cs file. For example, if you are using the default Json.NET serializer, this is what you would add:

```
[assembly:LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]
```

Note
You can define a custom serialization attribute at the method level, which will override the default serializer specified at the assembly level. For more information, see Handling Standard Data Types (p. 48).

Create the Deployment Package

To create the deployment package, open a command prompt and navigate to the folder that contains your `project.json` file and run the following commands:

- `dotnet restore` which will restore any references to dependencies of the project that may have changed during the development process.
- `dotnet publish` which compiles the application and packages the source code and any dependencies into a folder. The output of the command window will instruct you where the folder was created. For example:

  ```
  publish: Published to C:\Users\yourname\project-folder\bin\debug
  \netcoreapp1.1\publish
  ```

  The contents of this folder represent your application and at a minimum would look something like this:

  `application-name`.deps.json

  `application-name`.dll

  `application-name`.pdb

  `application-name`.runtimeconfig.json

Zip the contents of the folder (not the folder itself). This is your deployment package.

AWS Toolkit for Visual Studio

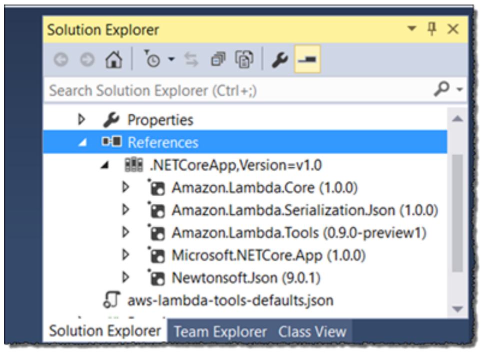
You can build .NET-based Lambda applications using the Lambda plugin to the AWS Toolkit for Visual Studio. The plugin is available as part of a Nuget package.

Step 1: Create and Build a Project

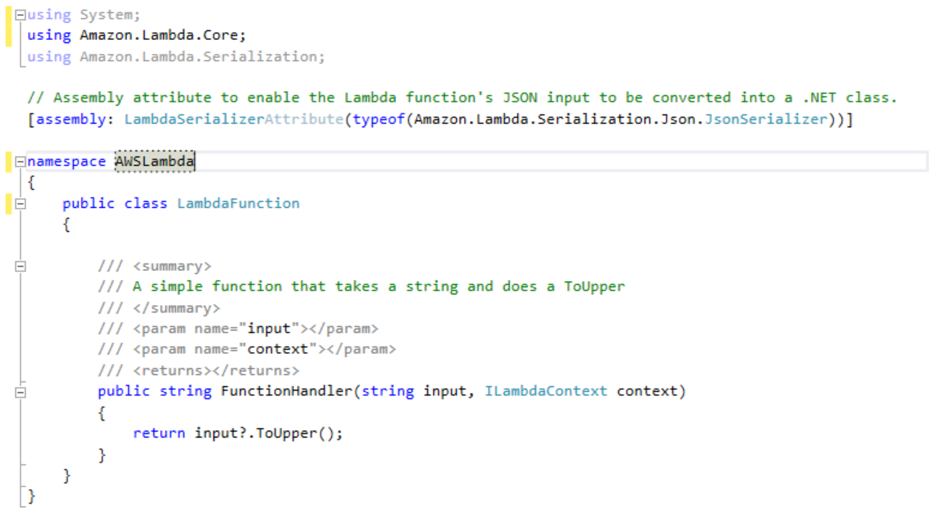
1. Launch Microsoft Visual Studio and choose **New project**.

   a. From the **File** menu, choose **New**, and then choose **Project**.

   b. In the **New Project** window, choose **AWS Lambda Project (.NET Core)** and then choose **OK**.
   c. In the **Select Blueprint** window, you will be presented with the option of selecting from a list of sample applications that will provide you with sample code to get started with creating a .NET-based Lambda application.
   d. To create a Lambda application from scratch, choose **Blank Function** and then choose **Finish**.
   e. Note that the libraries necessary for you to build a .NET-based Lambda application are provided in the **References** node of your project.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Properties
References
.NETCoreApp,Version=v1.0
Amazon.Lambda.Core (1.0.0)
Amazon.Lambda.Serialization.Json (1.0.0)
Amazon.Lambda.Tools (0.9.0-preview1)
Microsoft.NETCore.App (1.0.0)
Newtonsoft.Json (9.0.1)
aws-lambda-tools-defaults.json
Solution Explorer
Team Explorer
Class View

2. Open the **Function.cs** file. You will be provided with a template to implement your Lambda function handler code.



```
using System;
using Amazon.Lambda.Core;
using Amazon.Lambda.Serialization;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly: LambdaSerializerAttribute(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace AWSLambda
{
    public class LambdaFunction
    {

        /// <summary>
        /// A simple function that takes a string and does a ToUpper
        /// </summary>
        /// <param name="input"></param>
        /// <param name="context"></param>
        /// <returns></returns>
        public string FunctionHandler(string input, ILambdaContext context)
        {
            return input?.ToUpper();
        }
    }
}
```

3. Once you have written the code that represents your Lambda function, you can upload it by right-clicking the **Project** node in your application and then choosing **Publish to AWS Lambda**.
4. In the **Upload Lambda Function** window, do the following:

   - Specify the **Region:**
   - Specify the **Function Name:**
   - Specify the **Assembly Name:**
   - Specify the **Type Name:**
   - Specify the **Method Name:**

   Then choose **Next**
5. In the **Advanced Function Details** window, do the following:

- Specify the **Role Name:**, which is the IAM role required for your Lambda function's execution. If you have not yet created an execution role, do the following:

  1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
  2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
     - In **Role Name**, use a name that is unique within your AWS account.
     - In **Select Role Type**, choose **AWS Service Roles**, and then choose a service role that grants that service permissions to assume the role.
     - In **Attach Policy**, choose a permissions policy that is suitable to execute your Lambda function.
- (Optional) In **Environment::** specify any environment variables you wish to use. For more information, see Environment Variables (p. 89).
- (Optional)Specify the **Memory (MB):** or **Timeout (Secs):** configurations.
- (Optional)Specify any **VPC:** configurations if your Lambda function needs to access resources running inside a private VPC. For more information, see Configuring a Lambda Function to Access Resources in an Amazon VPC (p. 96).
- Choose **Next** and then choose **Upload** to deploy your application.

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:

```
project-dir/pom.xml
project-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:

```
project-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 *project-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`. 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="http://
www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>doc-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>1.1.0</version>
    </dependency>
  </dependencies>

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

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 tutorial topics in this guide, the specific tutorials 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: 1.1.0

> **Caution**
> If you are following other tutorial topics in this guide, the specific tutorials 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 tutorial topics in this guide, the specific tutorials might recommend different package name or class name.

   c. Add your Java code. If you are following other tutorial 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. 62).

The following examples use Gradle build and deployment tool to create the .zip.

Important
Gradle version 2.0 or later 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 (*project-dir*) with content having the following structure:

```
project-dir/build.gradle
project-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:

```
project-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 *project-dir*`/build/distributions` subdirectory.

1. Create a project directory (*project-dir*).
2. In the *project-dir*, create `build.gradle` file and add the following content:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

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

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

build.dependsOn buildZip
```

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 tutorial topics in this guide, the specific tutorials 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.

   ```
   project-dir> gradle build
   ```

6. Verify the resulting *`project-dir`*`.zip` file in the *`project-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 Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169).

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 (*`project-dir`*) will have the following structure:

```
project-dir/jars/           (all jars go here)
project-dir/build.gradle
```

```
project-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:

```
project-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 *`project-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.

```
project-dir> gradle build
```

Authoring Lambda Functions Using Eclipse IDE and AWS SDK Plugin (Java)

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, see the AWS Toolkit for Eclipse Getting Started Guide. For an example of using the toolkit for authoring Lambda functions, see Using AWS Lambda with the AWS Toolkit for Eclipse.

Creating a Deployment Package (Python)

To create a Lambda function you first create a Lambda function deployment package, a .zip file consisting of your code and any dependencies.

You can create a deployment package yourself or write your code directly in the Lambda console, in which case the console creates the deployment package for you and uploads it, creating your Lambda function. Note the following to determine if you can use the console to create your 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 Python.

- **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.

Note
After you create a deployment package, you may either upload it directly or 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 using the console or the AWS CLI.

The following is an example procedure to create a deployment package (outside the console).

Note
This should work for most standard installations of Python and pip when using pure Python modules in your Lambda function. If you are including modules that have native dependencies or have Python installed with Homebrew on OS X, you should see the next section which provides instructions to create a deployment package when using Virtualenv. For more information, see Create Deployment Package Using a Python Environment Created with Virtualenv (p. 71) and the Virtualenv website.

You will use `pip` to install dependencies/libraries. For information to install `pip`, go to Installation.

1. You create a directory, for example `project-dir`.
2. Save all of your Python source files (the .py files) at the root level of this directory.
3. Install any libraries using **pip**. Again, you install these libraries at the root level of the directory.

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

   For example, the following command installs the `requests` HTTP library in the `project-dir` directory.

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

   If using Mac OS X and you have Python installed using Homebrew (see Homebrew), the preceding command will not work. A simple workaround is to add a `setup.cfg` file in your *`/path/to`*`/project-dir` with the following content.

   ```
   [install]
   prefix=
   ```

4. Zip the content of the `project-dir` directory, which is your deployment package.

   Important
   Zip the directory *content*, not the directory. The contents of the Zip file are available as the current working directory of the Lambda function. For example: */project-dir/codefile.py/lib/yourlibraries*

Note
AWS Lambda includes the AWS SDK for Python (Boto 3), so you don't need to include it in your deployment package. However, if you want to use a version of Boto3 other than the one included by default, you can include it in your deployment package.

Create Deployment Package Using a Python Environment Created with Virtualenv

This section explains how to create a deployment package if you are using a Python environment that you created with the Virtualenv tool. Consider the following example:

- Created the following isolated Python environment using the Virtualenv tool and activated the environment:

  ```
  virtualenv path/to/my/virtual-env
  ```

  You can activate the environment on Windows, OS X, and Linux as follows:

  - On Windows, you activate using the `activate.bat`:

    ```
    path\to\my\virtual-env\Scripts\activate.bat
    ```

  - On OS X and Linux, you source the `activate` script:

    ```
    source  path/to/my/virtual-env/bin/activate
    ```

- Also, suppose you have installed the **requests** package in the activated environment (assume that you will you use these in your code). You can install these packages as follows :

  ```
  pip install requests
  ```

Now, to create a deployment package you do the following:

1. First, create .zip file with your Python code you want to upload to AWS Lambda.
2. Add the libraries from preceding activated virtual environment to the .zip file. That is, you add the content of the following directory to the .zip file (note again that you add the content of the directory and not the directory itself).

   For Windows the directory is:

   ```
   %VIRTUAL_ENV%\Lib\site-packages
   ```

   For OS X, Linux, the directory is:

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

   Note
   If you don't find the packages in the `site-packages` directory in your virtual environment, you might find it in the `dist-packages` directory.

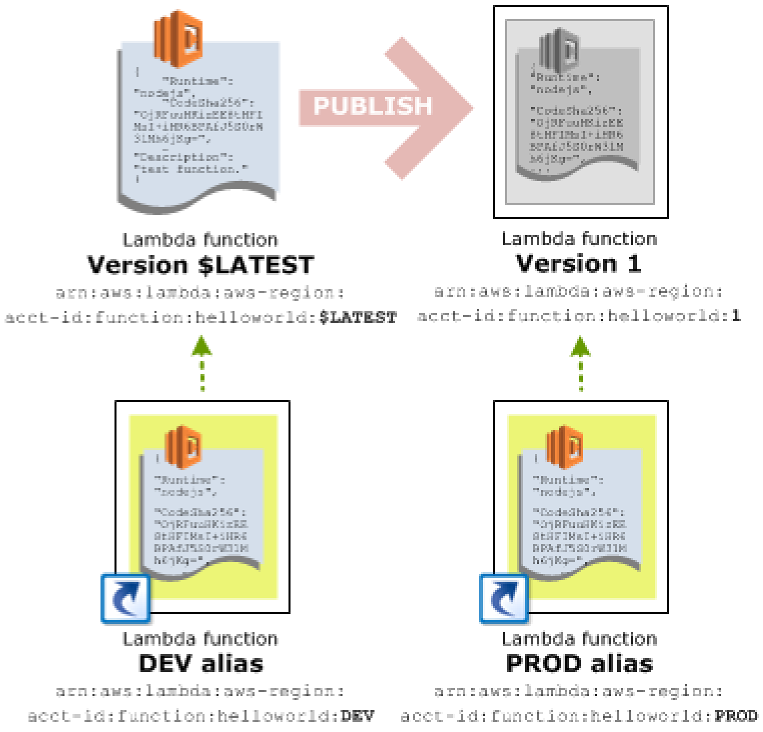
For an example of creating a Python deployment package, see Python (p. 186).

AWS Lambda Function Versioning and Aliases

Versioning allows you to better manage your in-production Lambda function code by enabling you to publish one or more versions of your Lambda function. As a result, you can work with different variations of your Lambda function in your development workflow, such as development, beta, and production. Each Lambda function version has a unique Amazon Resource Name (ARN). After you publish a version, it is immutable (that is, it can't be changed).

AWS Lambda supports creating aliases for each of your Lambda function versions. Conceptually, an AWS Lambda alias is a pointer to a specific Lambda function version, but it is also a resource similar to a Lambda function, and each alias has a unique ARN. Each alias maintains an ARN for a function version to which it points (note that an alias can only point to a function version, not to another alias). Unlike versions, which are immutable, aliases are mutable (that is, they can be changed) and can be updated to point to different versions

The following example shows two versions of a `helloworld` Lambda function (version `$LATEST`, and version 1). Each of these function versions has an alias (DEV and PROD) pointing to it.


PUBLISH
Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
Version 1
arn:aws:lambda:aws-region:
acct-id:function:helloworld:1
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV
Lambda function
PROD alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:PROD

Aliases enable you to abstract the process of promoting new Lambda function versions into production from the mapping of the Lambda function version and its event source. For more information, see How It Works (p. 151).

For example, suppose Amazon S3 is the event source that invokes your Lambda function when new objects are created in a bucket. When Amazon S3 is your event source, you store the event source mapping information in the bucket notification configuration. In the configuration you can identify the Lambda function ARN that Amazon S3 can invoke, but, in this case, each time you publish a new version of your Lambda function you need to update the notification configuration so that Amazon S3 invokes the correct version. Instead of specifying the function ARN, you can specify an alias ARN in the notification configuration (for example, PROD alias ARN). As you promote new versions of your Lambda function into production, you only need to update the PROD alias to point to the latest stable version, and you don't need to update the notification configuration in Amazon S3.

The same applies when you need to roll back to a previous version of your Lambda function. In this scenario, you just update the PROD alias to point to a different function version, and there is no need to update event source mappings.

We recommend you use versioning and aliases to deploy your Lambda functions when building applications with multiple dependencies and developers involved.

For detailed information, see the following topics:

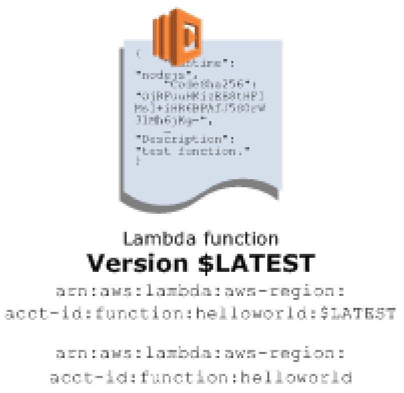
Topics

Introduction to AWS Lambda Versioning

This section explains how to create a Lambda function and publish a version from it. It also explains how to update function code and configuration information when you have one or more published versions. In addition, this section includes information on how to delete function versions, either specific versions or the entire Lambda function (with all of its versions and associated aliases).

Creating a Lambda Function (the $LATEST version)

When you create a Lambda function, there is only one version. It is the `$LATEST` version.



```
{
"Runtime":
"nodejs",
"CodeSha256":
[illegible]
"Description":
"test function."
}
```

Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld

You can refer to this function using its Amazon Resource Name (ARN). There are two ARNs associated with this initial version:

- **Qualified ARN** – The function ARN with the version suffix.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST
  ```

- **Unqualified ARN** – The function ARN without the version suffix.

  You can use this ARN in all relevant operations however you cannot use it to create an alias. For more information, see Introduction to AWS Lambda Aliases (p. 77).

  The unqualified ARN has its own resource policies.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld
  ```

Note
Unless you choose to publish versions, the $LATEST version is the only Lambda function version you have. You can use either the qualified or unqualified ARN in your event source mapping to invoke this `$LATEST` version.

The following is an example response of a `CreateFunction` API call:

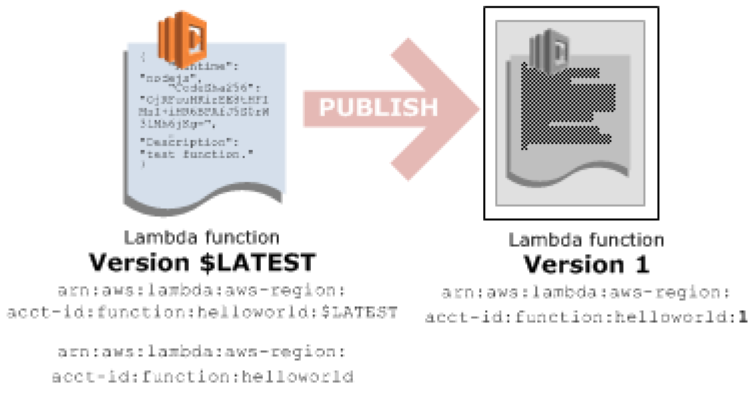
```
{
    "CodeSize": 287,
    "Description": "test function."
    "FunctionArn": "arn:aws:lambda:aws-region:acct-id:function:helloworld",
    "FunctionName": "helloworld",
    "Handler": "helloworld.handler",
    "LastModified": "2015-07-16T00:34:31.322+0000",
    "MemorySize": 128,
    "Role": "arn:aws:iam::acct-id:role/lambda_basic_execution",
    "Runtime": "nodejs4.3",
    "Timeout": 3,
    "CodeSHA256": "OjRFuuHKizEE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",
    "Version": "$LATEST"
}
```

For more information, see CreateFunction (p. 332).

In this response, AWS Lambda returns the unqualified ARN of the newly created function as well as its version, `$LATEST`. The response also shows that the `Version` is `$LATEST`. The `CodeSha256` is the checksum of the deployment package that you uploaded.

Publishing a Lambda Function Version

When you publish a version, AWS Lambda makes a snapshot copy of the Lambda function code (and configuration) in the `$LATEST` version. A published version is immutable. That is, you can't change the code or configuration information. The new version has a unique ARN that includes a version number suffix as shown:


PUBLISH
Lambda function
Version $LATEST
arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST
arn:aws:lambda:aws-region:acct-id:function:helloworld
Lambda function
Version 1
arn:aws:lambda:aws-region:acct-id:function:helloworld:1

You can publish a version using any of the following methods:

- **Publish a version explicitly** – Use the `PublishVersion` API to explicitly publish a version. For more information, see PublishVersion (p. 372). This action creates a new version using the code and configuration in the `$LATEST` version.
- **Publish a version at the time you create or update a Lambda function** – Use the `CreateFunction` or `UpdateFunctionCode` requests to also publish a version by adding the optional `publish` parameter in the request:

- Specify the `publish` parameter in your `CreateFunction` request to create a new Lambda function (the `$LATEST` version), and then immediately publish it by creating a snapshot and assigning it to be version 1. For more information about `CreateFunction`, see CreateFunction (p. 332).
- Specify the `publish` parameter in your `UpdateFunctionCode` request to update the code in the `$LATEST` version, and then publish a version from the `$LATEST`. For more information about `UpdateFunctionCode`, see UpdateFunctionCode (p. 383).

If you specify the `publish` parameter at the time you create a Lambda function, the function configuration information that AWS Lambda returns in response shows the version number of the newly published version, as shown following (in the example, the version is 1):

```
{
    "CodeSize": 287,
    "Description": "test function."
    "FunctionArn": "arn:aws:lambda:aws-region:acct-id:function:helloworld",
    "FunctionName": "helloworld",
    "Handler": "helloworld.handler",
    "LastModified": "2015-07-16T00:34:31.322+0000",
    "MemorySize": 128,
    "Role": "arn:aws:iam::acct-id:role/lambda_basic_execution",
    "Runtime": "nodejs4.3",
    "Timeout": 3,
    "CodeSHA256": "OjRFuuHKizEE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",
    "Version": "1"
}
```

Note
Lambda will only publish a new version if the code has not yet been published or if the code has changed when compared against the $LATEST version. If there is no change, the $LATEST published version will be returned.

We recommend that you publish a version at the same time that you create your Lambda function or update your Lambda function code, especially when multiple developers contribute to the same Lambda function development. You can use the `publish` parameter in your request to do this. When you have multiple developers working on a project, it is possible for developer A to create a Lambda function (`$LATEST` version) and before developer A publishes a version, developer B updates the code (deployment package) associated with the `$LATEST` version. In this case, you lose the original code that developer A uploaded. When both developers add the `publish` parameter it prevents the race condition described.

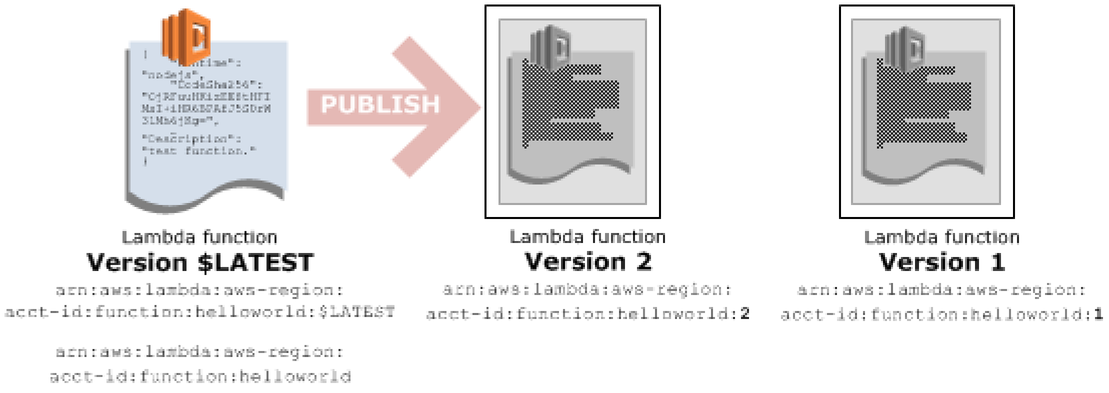
Note
The published versions are immutable. That is, you cannot change code or configuration information associated with a version.

Each version of a Lambda function is a unique resource with a Amazon Resource Name (ARN). The following example shows the ARN of version number 1 of the `helloworld` Lambda function:

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:1
```

Note
This is a qualified ARN, where the version number is a suffix. Published versions can have only qualified ARN.

You can publish multiple versions. Each time you publish a version, AWS Lambda copies `$LATEST` version (code and configuration information) to create a new version. When you publish additional versions, AWS Lambda assigns a monotonically increasing sequence number for versioning, even if the function was deleted and re-created. Version numbers are never reused, even for a function that has been deleted and re-created, so that the consumer of that version can depend on the executable of that version to never change (except if it's deleted). If you want to re-use a qualifier, use aliases with your versions. Aliases can be deleted and re-created with the same name.


PUBLISH
Lambda function
Version $LATEST
arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST
arn:aws:lambda:aws-region:acct-id:function:helloworld
Lambda function
Version 2
arn:aws:lambda:aws-region:acct-id:function:helloworld:2
Lambda function
Version 1
arn:aws:lambda:aws-region:acct-id:function:helloworld:1

Updating Lambda Function Code and Configuration

AWS Lambda maintains your latest function code in the `$LATEST` version. When you update your function code, AWS Lambda replaces the code in the `$LATEST` version of the Lambda function. For more information, see UpdateFunctionCode (p. 383).

Published versions are immutable. You cannot update code or configuration information associated with a published version.

You have the following options of publishing a new version as you update your Lambda function code:

- **Publish a version in the same update code request** – Use the `UpdateFunctionCode` API (recommended).
- **First update the code, and then explicitly publish a version** – Use the `PublishVersion` API.

You can update code and configuration information (such as description, memory size, and execution timeout) of the `$LATEST` version of the Lambda function. However, published versions are immutable. That is, you cannot change code or configuration information.

Deleting a Lambda Function and a Specific Version

With versioning, you have the following choices:

- **Delete a specific version** – You can delete a Lambda function version by specifying the version you want to delete in your `DeleteFunction` request. If there are aliases dependent on this version, the request will fail. AWS Lambda deletes the version only if there are no aliases dependent on this version. For more information about aliases, see Introduction to AWS Lambda Aliases (p. 77).
- **Delete the entire Lambda function (all of its versions and aliases)** – To delete the Lambda function and all of its versions, do not specify any version in your `DeleteFunction` request. This deletes the entire function including all of its versions and aliases.

Important
You can delete a specific function version, but you cannot delete the `$LATEST`.

Related Topics

Introduction to AWS Lambda Aliases (p. 77)

Managing Versioning Using the AWS Management Console, the AWS CLI, or Lambda APIs (p. 87)

Introduction to AWS Lambda Aliases

You can create aliases for your Lambda function. An AWS Lambda alias is like a pointer to a specific Lambda function version. For more information about versioning, see Introduction to AWS Lambda Versioning (p. 73). By using aliases, you can access the Lambda function it is pointing to (for example, to invoke the function) without the caller having to know the specific version the alias is pointing to.

AWS Lambda aliases enable the following use cases:

- **Easier support for promotion of new versions of Lambda functions and roll back when needed** – After initially creating a Lambda function (the `$LATEST` version) you can first publish a version 1 of it. By creating an alias named PROD that points to version 1, you can now use the PROD alias to invoke version 1 of the Lambda function.

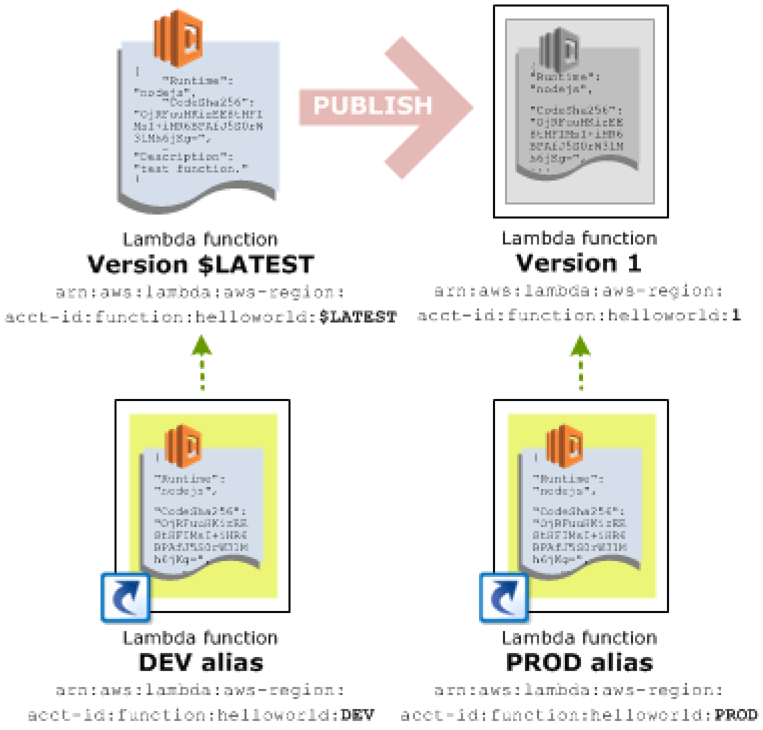
  Now, you can update the code (the `$LATEST` version) with all of your improvements, and then publish another stable and improved version (version 2). You can promote version 2 to production by remapping the PROD alias so that it points to version 2. If you find something wrong, you can easily roll back the production version to version 1 by remapping the PROD alias so that it points to version 1.

  > **Note**
  > In this context, the terms *promotion* and *roll back* refer to the remapping of aliases to different function versions.

- **Simplify management of event source mappings** – Instead of using Lambda function ARNs in event source mappings, by using an alias ARN you ensure that you don't need to update your event source mappings when you promote a new version or roll back to a previous version.

An AWS Lambda alias is a resource similar to a Lambda function. However, you can't create an alias independently. You create an alias for an existing Lambda function. If a Lambda function is a resource, you can think of an AWS Lambda alias as a subresource that is associated with a Lambda function.

Both the Lambda function and alias are AWS Lambda resources, and like all other AWS resources they both have unique Amazon Resource Names (ARNs). The following example shows a Lambda function (the `$LATEST` version), with one published version. Each version has an alias pointing to it.


PUBLISH
Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
Version 1
arn:aws:lambda:aws-region:
acct-id:function:helloworld:1
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV
Lambda function
PROD alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:PROD

You can access the function using either the function ARN or the alias ARN.

- Because the function version is $LATEST, you can access it using the qualified or unqualified function ARN.
  - Qualified function ARN (with the $LATEST version suffix):

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST
```

- When using any of the alias ARNs, you are using a qualified ARN. Each alias ARN has an alias name suffix.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:PROD
arn:aws:lambda:aws-region:acct-id:function:helloworld:BETA
arn:aws:lambda:aws-region:acct-id:function:helloworld:DEV
```

AWS Lambda provides the following APIs for you to create and manages aliases:

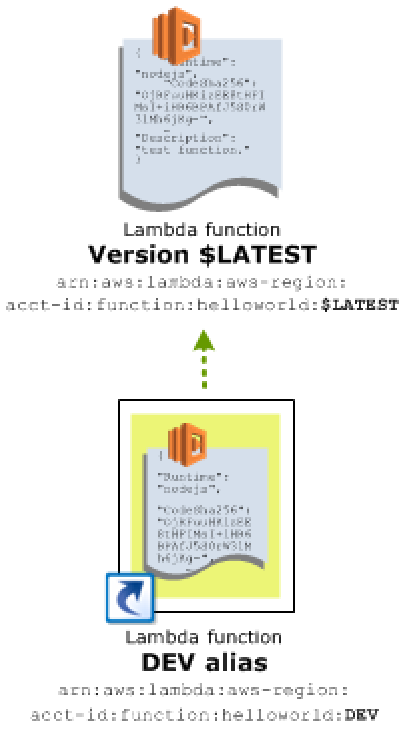
- CreateAlias (p. 325)
- UpdateAlias (p. 378)
- GetAlias (p. 345)
- ListAliases (p. 364)
- DeleteAlias (p. 338)

Example: Using Aliases to Manage Lambda Function Versions

The following is an example scenario of how to use versioning and aliases to promote new versions of Lambda functions into production.

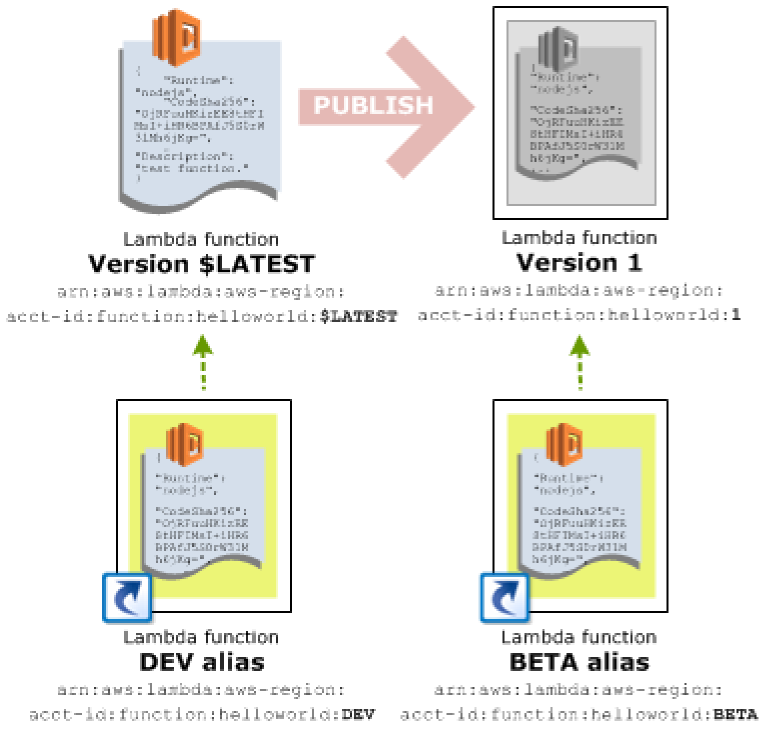
Initially, you create a Lambda function.

It is the $LATEST version. You also create an alias (DEV, for development) that points to the newly created function. Developers can use this alias to test the function with the event sources in a development environment.


Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV

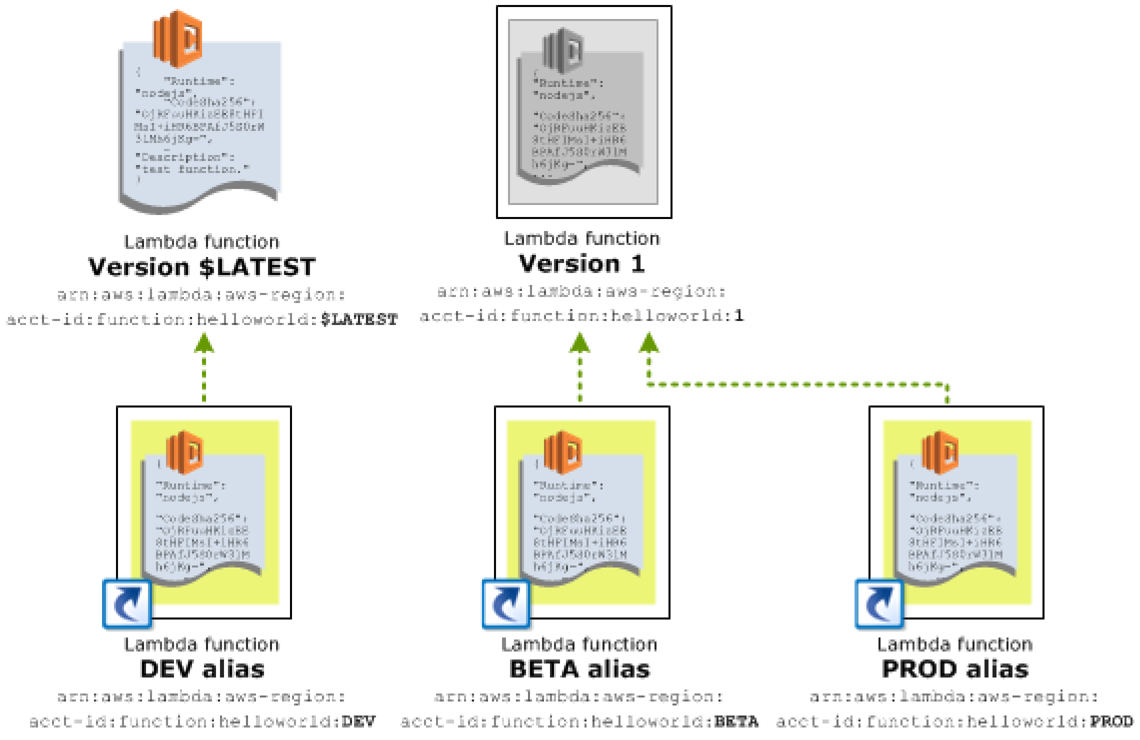
Test the function version using event sources in a beta environment, in a stable way while continuing to develop newer versions.

You publish a version from the $LATEST and have another alias (BETA) point to it. This allows you to associate your beta event sources to this specific alias. In the event source mappings, use the BETA alias to associate your Lambda function with the event source.


PUBLISH
Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
Version 1
arn:aws:lambda:aws-region:
acct-id:function:helloworld:1
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV
Lambda function
BETA alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:BETA

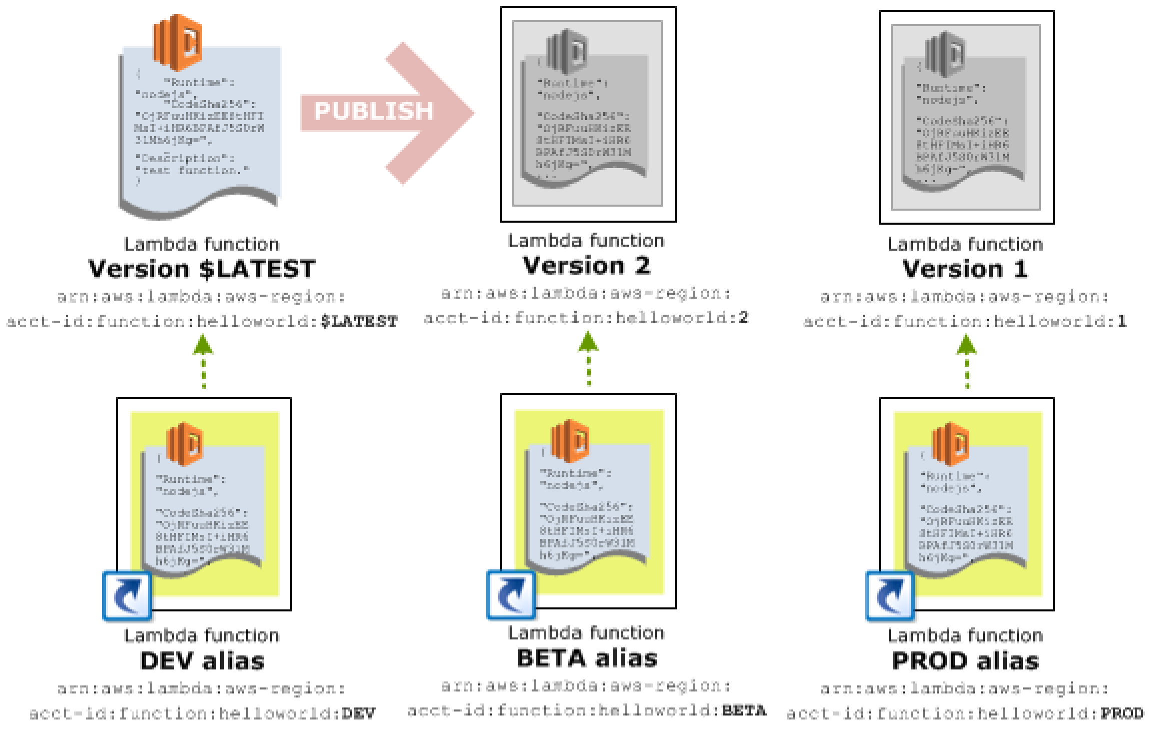
Promote the Lambda function version in production to work with event sources in production environment.

After testing the BETA version you can define the production version by creating an alias that maps to version 1. This means you want to point your production event sources to this specific version. You do this by creating a PROD alias and using the PROD alias ARN in all of your production event source mappings.


Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
Version 1
arn:aws:lambda:aws-region:
acct-id:function:helloworld:1
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV
Lambda function
BETA alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:BETA
Lambda function
PROD alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:PROD

Continue development, publish more versions, and test.

As you develop your code you can update the $LATEST version by uploading updated code and then publish to beta testing by having the BETA alias point to it. This simple remapping of the beta alias enables you put version 2 of your Lambda function into beta without changing any of your event sources. This is how aliases enable you to control which versions of your function are used with specific event sources in your development environment.


PUBLISH
Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
Version 2
arn:aws:lambda:aws-region:
acct-id:function:helloworld:2
Lambda function
Version 1
arn:aws:lambda:aws-region:
acct-id:function:helloworld:1
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV
Lambda function
BETA alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:BETA
Lambda function
PROD alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:PROD

If you want to try creating this setup using AWS CLI, see Tutorial: Using AWS Lambda Aliases (p. 81).

Related Topics

Introduction to AWS Lambda Versioning (p. 73)

Tutorial: Using AWS Lambda Aliases (p. 81)

Managing Versioning Using the AWS Management Console, the AWS CLI, or Lambda APIs (p. 87)

Tutorial: Using AWS Lambda Aliases

This AWS CLI-based tutorial creates Lambda function versions and aliases that point to it as described in the Example: Using Aliases to Manage Lambda Function Versions (p. 78).

This example uses the us-west-2 (US West, Oregon) region to create the Lambda function and aliases.

1. First, you need to create a deployment package that you can upload to create your Lambda function.

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

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

exports.handler = function(event, context, callback) {
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    callback(null, "message");

};
```

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

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

2. Create an IAM role (execution role) that you can specify at the time you create your Lambda function.

   a. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.

   b. Follow the steps in IAM Roles in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:

      • In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**.

      • In **Attach Policy**, choose the policy named **AWSLambdaBasicExecutionRole**.

   c. Write down the Amazon Resource Name (ARN) of the IAM role. You need this value when you create your Lambda function in the next step.

3. Create a Lambda function (`helloworld`).

```
aws lambda create-function \
--region us-west-2 \
--function-name helloworld \
--zip-file fileb://file-path/helloworld.zip \
--role arn:aws:iam::account-id:role/lambda_basic_execution \
--handler helloworld.handler \
--runtime nodejs4.3 \
--profile adminuser
```

The response returns the configuration information showing `$LATEST` as the function version as shown in the following example response:

```
{
    "CodeSha256": "OjRFuuHKizEE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",
    "FunctionName": "helloworld",
    "CodeSize": 287,
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:account-
id:function:helloworld",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::account-id:role/lambda_basic_execution",
    "Timeout": 3,
    "LastModified": "2015-09-30T18:39:53.873+0000",
    "Handler": "helloworld.handler",
    "Runtime": "nodejs4.3",
    "Description": ""
}
```

4. Create an alias (`DEV`) that points to the `$LATEST` version of the `helloworld` Lambda function:

```
aws lambda create-alias \
--region us-west-2 \
--function-name helloworld \
--description "sample alias" \
--function-version "\$LATEST" \
--name DEV \
--profile adminuser
```

The response returns the alias information, including the function version it is pointing to and the alias ARN. The ARN is the same as the function ARN with an alias name suffix. The following is an example response:

```
{
    "AliasArn": "arn:aws:lambda:us-west-2:account-
id:function:helloworld:DEV",
    "FunctionVersion": "$LATEST",
    "Name": "DEV",
    "Description": "sample alias"
}
```

5. Publish a version of the `helloworld` Lambda function.

```
aws lambda publish-version \
--region us-west-2 \
--function-name helloworld \
--profile adminuser
```

The response returns configuration information of the function version, including the version number, and the function ARN with the version suffix. The following is an example response:

```
{
    "CodeSha256": "OjRFuuHKizEE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",
    "FunctionName": "helloworld",
    "CodeSize": 287,
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:account-
id:function:helloworld:1",
    "Version": "1",
    "Role": "arn:aws:iam::account-id:role/lambda_basic_execution",
    "Timeout": 3,
    "LastModified": "2015-10-03T00:48:00.435+0000",
    "Handler": "helloworld.handler",
    "Runtime": "nodejs4.3",
    "Description": ""
}
```

6. Create an alias (BETA) for the for the `helloworld` Lambda function version 1.

```
aws lambda create-alias \
--region us-west-2 \
--function-name helloworld \
--description "sample alias" \
--function-version 1 \
--name BETA \
--profile adminuser
```

Now you have two aliases for the `helloworld` function. The `DEV` alias points to the `$LATEST` function version, and the `BETA` alias points to version 1 of the Lambda function.

7. Now suppose you want to put the version 1 of the `helloworld` function in production. Create another alias (`PROD`) that points to version 1.

   ```
   aws lambda create-alias \
   --region us-west-2 \
   --function-name helloworld \
   --description "sample alias" \
   --function-version 1 \
   --name PROD \
   --profile adminuser
   ```

   At this time you have both the `BETA` and `PROD` aliases pointing to version 1 of the Lambda function.

8. You can now publish a newer version (for example, version 2), but first you need to update your code and upload a modified deployment package. If the `$LATEST` version is not changed, you cannot publish more than one version of it. Assuming you updated the deployment package, uploaded it, and published version 2, you can now change the `BETA` alias to point to version 2 of the Lambda function.

   ```
   aws lambda update-alias \
   --region us-west-2 \
   --function-name helloworld \
   --function-version 2 \
   --name BETA \
   --profile adminuser
   ```

   Now you have three aliases pointing to a different version of the Lambda function (`DEV` alias points to the `$LATEST` version, `BETA` alias points to version 2, and the `PROD` alias points to version 1 of the Lambda function.

For information about using the AWS Lambda console to manage versioning, see Managing Versioning Using the AWS Management Console, the AWS CLI, or Lambda APIs (p. 87).

Granting Permissions in a Push Model

In a push model (see Event Source Mapping (p. 120)), event sources such as Amazon S3 invoke your Lambda function. These event sources maintain a mapping that identifies a function version or alias they will invoke when events occur. Note the following:

- We recommend that you specify an existing Lambda function alias in the mapping configuration (see Introduction to AWS Lambda Aliases (p. 77)). For example, if the event source is Amazon S3, you specify the alias ARN in the bucket notification configuration so that Amazon S3 can invoke the alias when it detects specific events.
- In the push model, you grant event sources permissions using a resource policy that you attach to your Lambda function. In versioning, the permissions you add are specific to the qualifier that you specify in the `AddPermission` request (see Versioning, Aliases, and Resource Policies (p. 85)).

For example, the following AWS CLI command grants Amazon S3 permissions to invoke the PROD alias of the helloworld Lambda function (note that the `--qualifier` parameter specifies the alias name).

```
aws lambda add-permission \
--region us-west-2 \
--function-name helloworld \
--qualifier PROD \
--statement-id 1 \
--principal s3.amazonaws.com \
--action lambda:InvokeFunction \
--source-arn arn:aws:s3:::examplebucket \
--source-account 111111111111 \
--profile adminuser
```

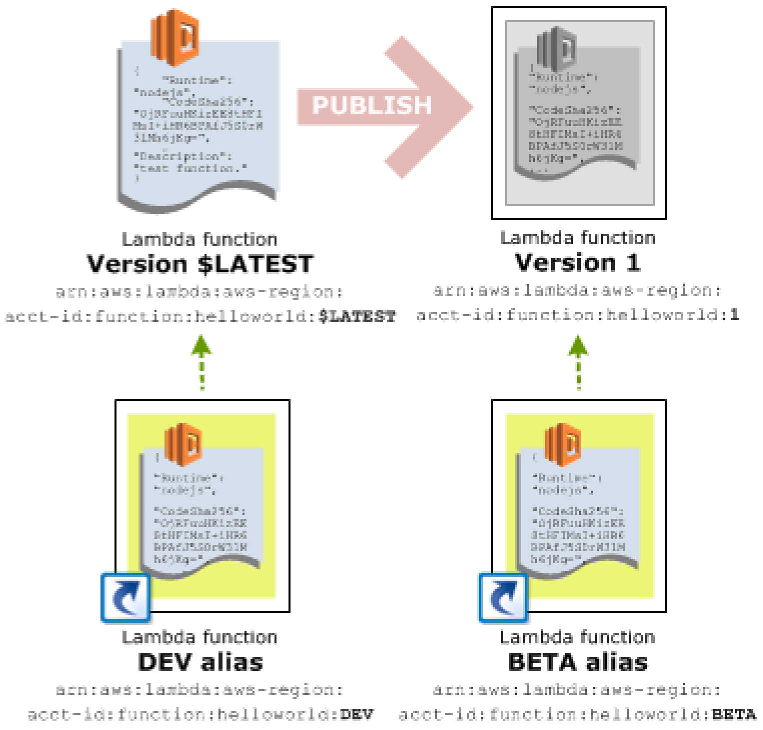
In this case, Amazon S3 is now able to invoke the PROD alias and AWS Lambda can then execute the helloworld Lambda function version that the PROD alias points to. For this to work, you must use the PROD alias ARN in the S3 bucket's notification configuration.

For information about how to handle Amazon S3 events, see Tutorial: Using AWS Lambda with Amazon S3 (p. 177).

Note
If you use the AWS Lambda console to add an event source for your Lambda function, the console adds the necessary permissions for you.

Versioning, Aliases, and Resource Policies

With versioning and aliases you can access a Lambda function using various ARNs. For example, consider the following scenario:


PUBLISH
Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
Version 1
arn:aws:lambda:aws-region:
acct-id:function:helloworld:1
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV
Lambda function
BETA alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:BETA

You can invoke for example the `helloworld` function version 1 using any of the following two ARNs:

- Using the qualified function ARN:

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:1
```

  Note
  An unqualified function ARN (function ARN without a version or alias suffix), maps to the $LATEST version.

- Using the BETA alias ARN:

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:BETA
```

In a *push* model, event sources (such as Amazon S3 and custom applications) can invoke any of the Lambda function versions as long you grant the necessary permissions to these event sources by using an access policy associated with the Lambda function. For more information about the push model, see Event Source Mapping (p. 120).

Assuming that you grant permission, the next question is, "can an event source invoke a function version using any of the associated ARNs?" The answer is, it depends on how you identified function in your add permissions request (see AddPermission (p. 322)). The key to understanding this is that the permission you grant apply only to the ARN used in the add permission request:

- If you use a qualified function name (such as `helloworld:1`), the permission is valid for invoking the `helloworld` function version 1 *only* using its qualified ARN (using any other ARNs will result in a permission error).

- If you use an alias name (such as `helloworld:BETA`), the permission is valid only for invoking the `helloworld` function using the BETA alias ARN (using any other ARNs will result in a permission error, including the function version ARN to which the alias points).
- If you use an unqualified function name (such as `helloworld`), the permission is valid only for invoking the `helloworld` function using the unqualified function ARN (using any other ARNs will result in a permission error).

  Note
  Note that even though the access policy is only on the unqualified ARN, the code and configuration of the invoked Lambda function is still from function version `$LATEST`. The unqualified function ARN maps to the `$LATEST` version but the permissions you add are ARN-specific.
- If you use a qualified function name using the `$LATEST` version (`helloworld:$LATEST`), the permission is valid for invoking the `helloworld` function version `$LATEST` *only* using its qualified ARN (using unqualified ARN will result in a permission error).

Managing Versioning Using the AWS Management Console, the AWS CLI, or Lambda APIs

You can manage Lambda function versioning programmatically using AWS SDKs (or make the AWS Lambda API calls directly, if you need to), using AWS Command Line Interface (AWS CLI), or the AWS Lambda console.

AWS Lambda provides the following APIs to manage versioning and aliases:

PublishVersion (p. 372)

ListVersionsByFunction (p. 370)

CreateAlias (p. 325)

UpdateAlias (p. 378)

DeleteAlias (p. 338)

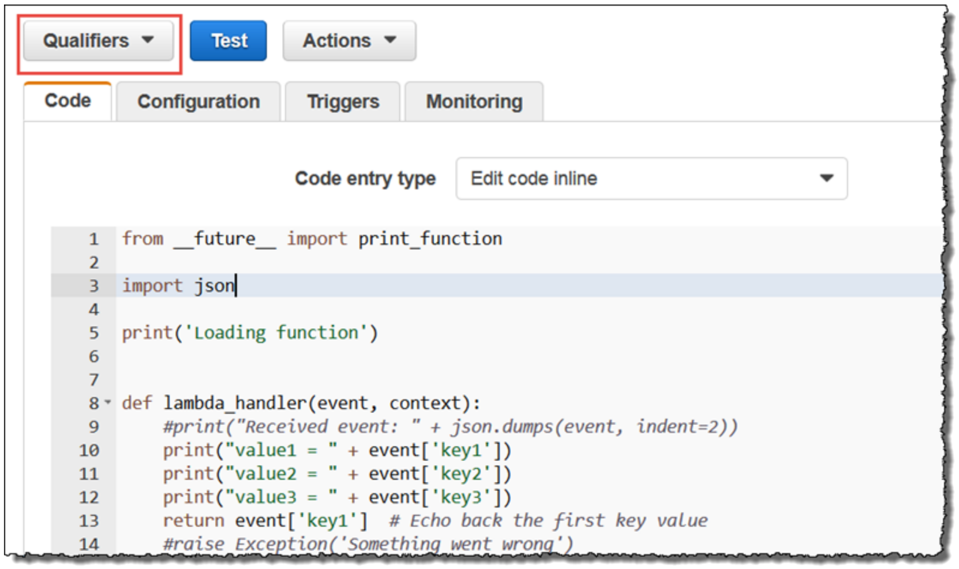
GetAlias (p. 345)

ListAliases (p. 364)

In addition to these APIs, existing relevant APIs also support versioning related operations.

For an example of how you can use the AWS CLI, see Tutorial: Using AWS Lambda Aliases (p. 81).

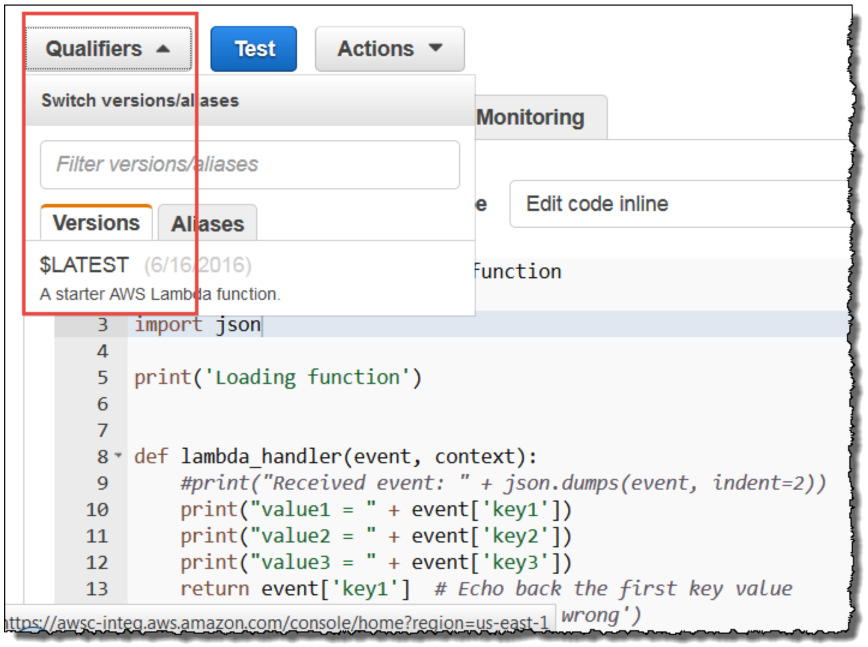
This section explains how you can use the AWS Lambda console to manage versioning. In the AWS Lambda console, choose a function and then choose **Qualifiers**.



Code entry type: Edit code inline

```
from __future__ import print_function

import json

print('Loading function')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
    #raise Exception('Something went wrong')
```

The expanded **Qualifiers** menu displays a **Versions** and **Aliases** tab, as shown in the following screen shot. In the **Versions** pane, you can see a list of versions for the selected function. If you have not previously published a version for the selected function, the **Versions** pane lists only the $LATEST version, as shown:

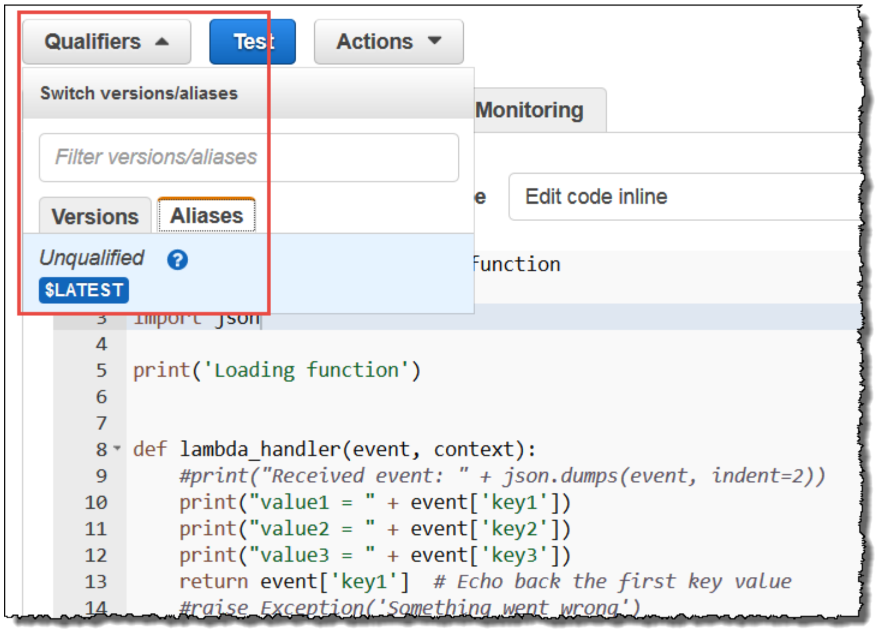


$LATEST (6/16/2016)
A starter AWS Lambda function.

```
import json

print('Loading function')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
```

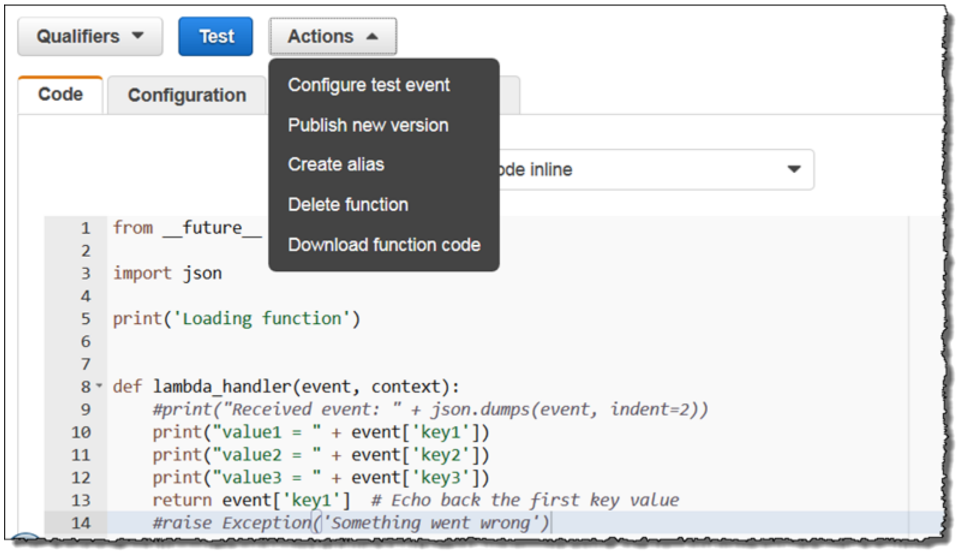
https://awsc-integ.aws.amazon.com/console/home?region=us-east-1

Choose the **Aliases** tab to see a list of aliases for the function. Initially, you won't have any aliases, as shown following:



```
import json

print('Loading function')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
    #raise Exception('Something went wrong')
```

Now, you can publish a version or create aliases for the selected Lambda function using the **Actions** menu.



```
from __future__
import json

print('Loading function')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
    #raise Exception('Something went wrong')
```

To learn about versioning and aliases, see AWS Lambda Function Versioning and Aliases (p. 72).

Environment Variables

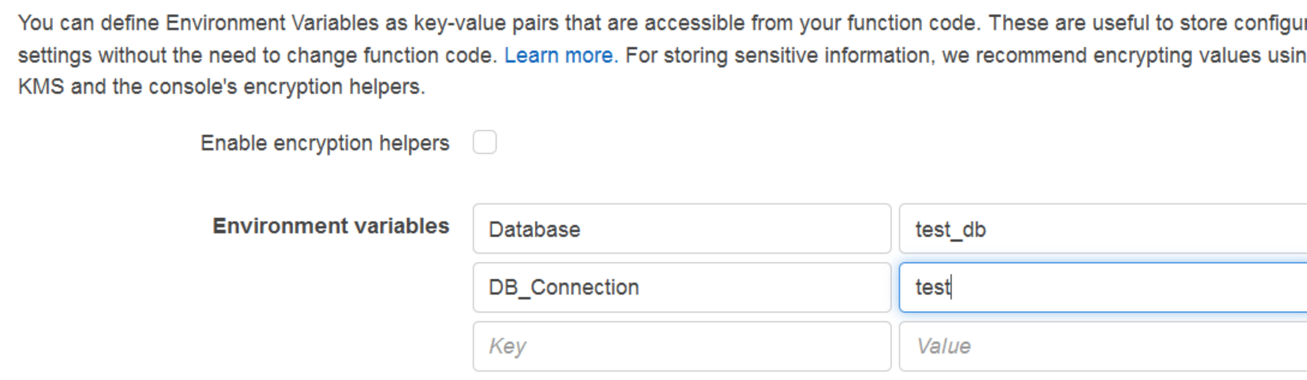
Environment variables for Lambda functions enable you to dynamically pass settings to your function code and libraries, without making changes to your code. Environment variables are key-value pairs that you create and modify as part of your function configuration, using either the AWS Lambda Console, the AWS Lambda CLI or the AWS Lambda SDK. AWS Lambda then makes these key value pairs available to your Lambda function code using standard APIs supported by the language, like `process.env` for Node.js functions.

You can use environment variables to help libraries know what directory to install files in, where to store outputs, store connection and logging settings, and more. By separating these settings from the application logic, you don't need to update your function code when you need to change the function behavior based on different settings.

Setting Up

Suppose you want a Lambda function to behave differently as it moves through lifecycle stages from development to deployment. For example, the dev, test, and production stages can contain databases that the function needs to connect to that require different connection information and use different table names. You can create environment variables to reference the database names, connection information or table names and set the value for the function based on the stage in which it’s executing (for example, development, test, production) while your function code remains unchanged.

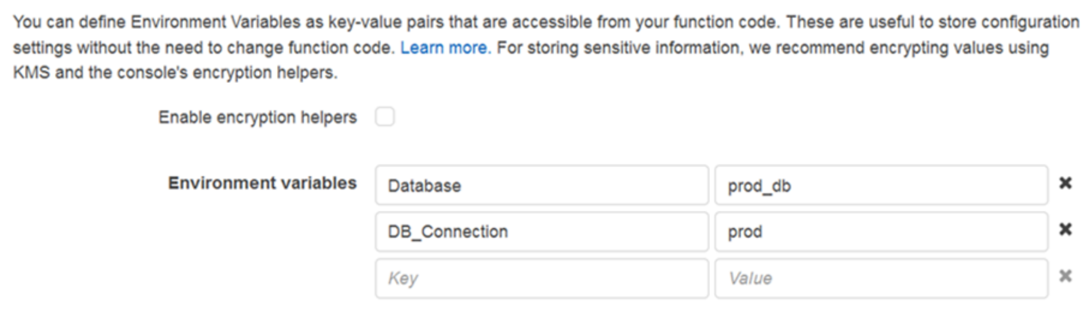
The following screenshots show how modify your function's configuration using the AWS console. The first screenshot configures the settings for the function corresponding to a test stage. The second one configures settings for a production stage.



You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configu
settings without the need to change function code. Learn more. For storing sensitive information, we recommend encrypting values usin
KMS and the console's encryption helpers.

Enable encryption helpers ☐

Environment variables	Database	test_db
	DB_Connection	test
	Key	Value



You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. Learn more. For storing sensitive information, we recommend encrypting values using KMS and the console's encryption helpers.

Enable encryption helpers ☐

Environment variables	Database	prod_db	✖
	DB_Connection	prod	✖
	Key	Value	✖

Note the **Enable encryption helpers** checkbox. You will learn more about using this in the Create a Lambda Function Using Environment Variables To Store Sensitive Information (p. 95) tutorial.

You can also use the AWS CLI to create Lambda functions that contain environment variables. For more details, see the CreateFunction (p. 332) and UpdateFunctionConfiguration (p. 387) APIs. Environment variables are also supported when creating and updating functions using AWS CloudFormation. Environment variables can also be used to configure settings specific to the language runtime or a library included in your function. For example, you can modify `PATH` to specify a directory where executables are stored. You can also set runtime-specific environment variables, such as `PYTHONPATH` for Python or `NODE_PATH` for Node.js.

The following example creates a new Lambda function that sets the `LD_LIBRARY_PATH` environment variable, which is used to specify a directory where shared libraries are dynamically loaded at runtime. In this example, the Lambda function code uses the shared library in the `/usr/bin/test/lib64` directory.

```
aws lambda create-function \
    --region us-east-1
    --function-name myTestFunction
    --zip-file fileb://path/package.zip
    --role role-arn
    --environment Variables={LD_LIBRARY_PATH=/usr/bin/test/lib64}
    --handler index.handler
    --runtime nodejs4.3
    --profile default
```

Rules for Naming Environment Variables

There is no limit to the number of environment variables you can create as long as the total size of the set does not exceed 4 KB.

Other requirements include:

- Must start with letters *[a-zA-Z]*.
- Can only contain alphanumeric characters and underscores *([a-zA-Z0-9_]*.

In addition, there are a specific set of keys that AWS Lambda reserves. If you try to set values for any of these reserved keys, you will receive an error message indicating that the action is not allowed. For more information on these keys, see Environment Variables Available to Lambda Functions (p. 158).

Environment Variables and Function Versioning

Function versioning provides a way to manage your Lambda function code by enabling you to publish one or more versions of your Lambda function as it proceeds from development to test to production. For each version of a Lambda function that you publish, the environment variables (as well as other function-specific configurations such as `MemorySize` and `Timeout` limit) are saved as a snapshot of that version and those settings are immutable (cannot be changed).

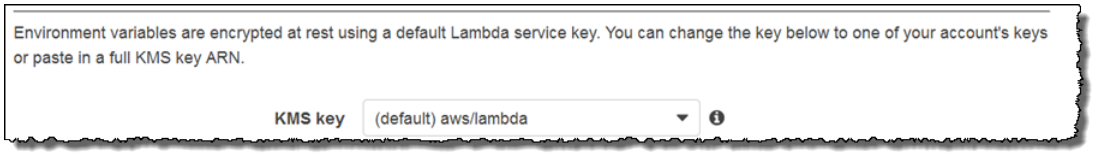
As application and configuration requirements evolve, you can create new versions of your Lambda function and update the environment variables to meet those requirements prior to the newest version being published. The current version of your function is $LATEST.

In addition, you can create aliases, which are pointers to a particular version of your function. The advantage of aliases is that if you need to roll back to a previous function version, you point the alias to that version, which contains the environment variables required for that version. For more information, see AWS Lambda Function Versioning and Aliases (p. 72).

Environment Variable Encryption

When you create or update Lambda functions that use environment variables, AWS Lambda encrypts them using the AWS Key Management Service. When your Lambda function is invoked, those values are decrypted and made available to the Lambda code.

The first time you create or update Lambda functions that use environment variables in a region, a default service key is created for you automatically within AWS KMS. This key is used to encrypt environment variables. You can alternatively select a key that you created separately using AWS Key Management Service, via the IAM console or AWS KMS Keys APIs. Creating your own key gives you more flexibility, including the ability to create, rotate, disable, and define access controls, and to audit the encryption keys used to protect your data. For more information, see the AWS Key Management Service Developer Guide.


Environment variables are encrypted at rest using a default Lambda service key. You can change the key below to one of your account's keys or paste in a full KMS key ARN.

KMS key (default) aws/lambda

If you use your own key, you will be billed per AWS Key Management Service Pricing guidelines. You will not be billed if you use the default service key provided by AWS Lambda.

If you're using the default KMS service key for Lambda, then no additional IAM permissions are required in your function execution role – your role will just work automatically without changes. If you're supplying your own (custom) KMS key, then you'll need to add `kms:Decrypt` to your execution role. In addition, the user that will be creating and updating the Lambda function must have permissions to use the KMS key. For more information on KMS keys, see the Using Key Policies in AWS KMS.

Storing Sensitive Information

As mentioned in the previous section, when you deploy your Lambda function, all the environment variables you've specified are encrypted by default. They are then decrypted automatically by AWS Lambda when the function is invoked. However, if you need to store sensitive information in an environment variable, we strongly suggest you encrypt that information before deploying your Lambda function.

Fortunately, the Lambda console makes that easier for you by providing encryption helpers that leverage AWS Key Management Service to store that sensitive information as `Ciphertext`. The Lambda console also provides decryption helper code to decrypt that information for use in your in Lambda function code. For more information, see Create a Lambda Function Using Environment Variables To Store Sensitive Information (p. 95).

Error scenarios

If your function configuration exceeds 4KB, or you use environment variable keys reserved by AWS Lambda, then your update or create operation will fail with a configuration error. During execution time, it's possible that the encryption/decryption of environment variables can fail. If AWS Lambda is unable to decrypt the environment variables due to an AWS KMS service exception, AWS KMS will return an exception message explaining what the error conditions are and what, if any, remedies you can apply to address the issue. These will be logged to your function log stream in Amazon CloudWatch logs. For example, if the KMS key you are using to access the environment variables is disabled, you will see the following error:

```
Lambda was unable to configure access to your environment variables because
 the KMS key used is disabled.
            Please check your KMS key settings.
```

Next Step

Create a Lambda Function Using Environment Variables (p. 92)

Create a Lambda Function Using Environment Variables

This section will illustrate how you can modify a Lambda function's behavior through configuration changes that require no changes to the Lambda function code.

In this tutorial, you will do the following:

- Create a deployment package with sample code that returns the value of an environment variable that specifies the name of an Amazon S3 bucket.

- Invoke a Lambda function and verify that the Amazon S3 bucket name that is returned matches the value set by the environment variable.
- Update the Lambda function by changing the Amazon S3 bucket name specified by the environment variable.
- Invoke the Lambda function again and verify that the Amazon S3 bucket name that is returned matches the updated value.

Step 1: Prepare

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account.
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Step 2: Set Up the Lambda Environment

In this section, you do the following:

- Create the Lambda function deployment package using the sample code provided.
- Create a Lambda execution role.
- Create the Lambda function by uploading the deployment package, and then test it by invoking it manually.

Step 2.1: Create the Deployment Package

The code sample below reads the environment variable of a Lambda function that returns the name of an Amazon S3 bucket.

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

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

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

        var bucketName = process.env.S3_BUCKET;
     callback(null, bucketName);
    }
```

2. Save the file as *index.js*.
3. Zip the *index.js.* file as *Test_Environment_Variables.zip*.

Step 2.2: Create an Execution Role

Create an IAM role (execution role) that you can specify at the time you create your Lambda function.

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in IAM Roles in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:

    - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**.

- In **Attach Policy**, choose the policy named **AWSLambdaBasicExecutionRole**.

3. Write down the Amazon Resource Name (ARN) of the IAM role. You need this value when you create your Lambda function in the next step.

Step 2.3 Create the Lambda function and Test It

In this section, you create a Lambda function containing an environment variable that specifies an Amazon S3 bucket named `Test`. When invoked, the function simply returns the name of the Amazon S3 bucket. Then you update the configuration by changing the Amazon S3 bucket name to `Prod` and when invoked again, the function returns the updated name of the Amazon S3 bucket.

To create the Lambda function, open a command prompt and run the following Lambda AWS CLI `create-function` command. You need to provide the .zip file path and the execution role ARN. For the runtime parameter, choose `nodejs4.3`.

```
aws lambda  create-function \
--region us-east-1 \
--function-name ReturnBucketName \
--zip-file fileb://file-path/Test_Environment_Variables.zip \
--role role-arn \
--environment Variables={S3_BUCKET=Test} \
--handler index.handler \
--runtime nodejs4.3 \
--version  version \
--profile default
```

Note
Optionally, 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 need to replace the `--zip-file` parameter with the `--code` parameter. For example:

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

Next, 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 ReturnBucketName \
--region us-east-1 \
--profile default \
outputfile.txt
```

The Lambda function will return the name of the Amazon S3 bucket as "Test".

Next, run the following Lambda CLI `update-function-configuration` command to update the Amazon S3 environment variable by pointing it to the `Prod` bucket.

```
aws lambda update-function-configuration
--invocation-type Event \
--function-name ReturnBucketName \
--region us-east-1 \
--environment Variables={S3_BUCKET=Prod} \
--profile default \
```

Run the `aws lambda invoke` command again using the same parameters. This time, the Lambda function will return the Amazon S3 bucket name as `Prod`.

Create a Lambda Function Using Environment Variables To Store Sensitive Information

Along with specifying configuration settings for your Lambda function, you can also use environment variables to store sensitive information, such as a database password, using AWS Key Management Service and the Lambda console's encryption helpers. For more information, see Environment Variable Encryption (p. 91). The following example shows you how to do this and also how to use KMS to decrypt that information.

This tutorial will demonstrate how you can use the Lambda console to encrypt an environment variable containing sensitive information and provides sample code for decrypting that information to use in your Lambda function.

Step 1: Create the Lambda Function

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Choose **Create a Lambda function**.
3. In **Select blueprint**, choose the **Blank Function** blueprint.
4. On the **Configure triggers** page, you can optionally choose a service that automatically triggers your Lambda function by choosing the gray box with ellipses (...) to display a list of available services. For this example, do not configure a trigger and choose **Next**.
5. In **Configure function**, do the following:

   - In **Name***, specify your Lambda function name.
   - In **Runtime**, specify **Node.js4.3**.

     Note that in **Lambda function code** section you can take advantage of the **Edit code inline** option to do the following:
     - Replace the Lambda function handler code with your custom code.
     - Implement the decryption helper code that Lambda provides, which you will learn about later in this exercise.
   - Check the **Enable encryption helpers** checkbox.
   - If you already have a KMS key associated with your user account, the **Encryption key** field will be auto-populated with that key. If you haven't created a KMS key for your account, you will be provided a link to the AWS IAM console to create one. The account must have have `encrypt` and `decrypt` permissions for that key.

     Note
     You cannot use the default Lambda service key for encrypting sensitive information on the client side.
   - In **Environment variables**, enter your key-value pair. If the value you provided is sensitive, choose the **Encrypt** button. This masks the value you entered and results in a call to AWS KMS to encrypt the value and return it as `Ciphertext`. Note that the **Encrypt** button toggles to **Decrypt** after you choose it. This affords you the option to update the information. Once you have done that, choose the **Encrypt** button.

     The **Code** button provides sample decrypt code specific to the runtime of your Lambda function that you can use with your application.
   - In **Role***, choose **Choose an existing role**.

- In **Existing role***, choose **lambda_basic_execution**.

  Note
  If the policy of the execution role does not have the `decrypt` permission, you will need add it.

6. In **Review**, review the configuration and then choose **Create Function**.

Configuring a Lambda Function to Access Resources in an Amazon VPC

Typically, you create resources inside Amazon Virtual Private Cloud (Amazon VPC) so that they cannot be accessed over the public Internet. These resources could be AWS service resources, such as Amazon Redshift data warehouses, Amazon ElastiCache clusters, or Amazon RDS instances. They could also be your own services running on your own EC2 instances. By default, resources within a VPC are not accessible from within a Lambda function.

AWS Lambda runs your function code securely within a VPC by default. However, to enable your Lambda function to access resources inside your private VPC, you must provide additional VPC-specific configuration information that includes VPC subnet IDs and security group IDs. AWS Lambda uses this information to set up elastic network interfaces (ENIs) that enable your function to connect securely to other resources within your private VPC.

Important
AWS Lambda does not support connecting to resources within Dedicated Tenancy VPCs. For more information, see Dedicated VPCs.

Configuring a Lambda Function for Amazon VPC Access

You add VPC information to your Lambda function configuration using the `VpcConfig` parameter, either at the time you create a Lambda function (see CreateFunction (p. 332)), or you can add it to the existing Lambda function configuration (see UpdateFunctionConfiguration (p. 387)). Following are AWS CLI examples:

- The `create-function` CLI command specifies the `--vpc-config` parameter to provide VPC information at the time you create a Lambda function.

```
$  aws lambda create-function \
--function-name ExampleFunction \
--runtime python2.7 \
--role execution-role-arn \
--zip-file fileb://path/app.zip \
--handler app.handler \
--vpc-config SubnetIds=comma-separated-vpc-subnet-
ids,SecurityGroupIds=comma-separated-security-group-ids \
--memory-size 1024 \
--profile adminuser
```

Note
The Lambda function execution role must have permissions to create, describe and delete ENIs. AWS Lambda provides a permissions policy, `AWSLambdaVPCAccessExecutionRole`, with permissions for the necessary EC2 actions (`ec2:CreateNetworkInterface`, `ec2:DescribeNetworkInterfaces`,

and `ec2:DeleteNetworkInterface`) that you can use when creating a role. You can review the policy in the IAM console. Do not delete this role immediately after your Lambda function execution. There is a delay between the time your Lambda function executes and ENI deletion. If you do delete the role immediately after function execution, you are responsible for deleting the ENIs.

- The `update-function-configuration` CLI command specifies the `--vpc-config` parameter to add VPC information to an existing Lambda function configuration.

```
$ aws lambda update-function-configuration \
--function-name ExampleFunction \
--vpc-config SubnetIds=comma-separated-vpc-subnet-
ids,SecurityGroupIds=security-group-ids
```

  To remove VPC-related information from your Lambda function configuration, use the `UpdateFunctionConfiguration` API by providing an empty list of subnet IDs and security group IDs as shown in the following example CLI command.

```
$ aws lambda update-function-configuration \
--function-name ExampleFunction \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

Note the following additional considerations:

- We recommend that you avoid DNS resolution of public host names for your VPC. This can take several seconds to resolve, which adds several seconds of billable time on your request. For example, if your Lambda function accesses an Amazon RDS instance in your VPC, launch the instance with the `no-publicly-accessible` option.

- When you add VPC configuration to a Lambda function, it can only access resources in that VPC. If a Lambda function needs to access both VPC resources and the public Internet, the VPC needs to have a Network Address Translation (NAT) instance inside the VPC.

- When a Lambda function is configured to run within a VPC, it incurs an additional ENI start-up penalty. This means address resolution may be delayed when trying to connect to network resources.

Internet Access for Lambda Functions

AWS Lambda uses the VPC information you provide to set up ENIs that allow your Lambda function to access VPC resources. Each ENI is assigned a private IP address from the IP address range within the Subnets you specify, but is not assigned any public IP addresses. *Therefore, if your Lambda function requires Internet access (for example, to access AWS services that don't have VPC endpoints, such as Amazon Kinesis), you can configure a NAT instance inside your VPC or you can use the Amazon VPC NAT gateway*. For more information, see NAT Gateways in the *Amazon VPC User Guide*. You cannot use an Internet gateway attached to your VPC, since that requires the ENI to have public IP addresses.

Important
If your Lambda function needs Internet access, do not attach it to a public subnet or to a private subnet without Internet access. Instead, attach it only to private subnets with Internet access through a NAT instance or an Amazon VPC NAT gateway.

Guidelines for Setting Up VPC-Enabled Lambda Functions

Your Lambda function automatically scales based on the number of events it processes. The following are general guidelines for setting up VPC-enabled Lambda functions to support the scaling behavior.

- If your Lambda function accesses a VPC, you must make sure that your VPC has sufficient ENI capacity to support the scale requirements of your Lambda function. You can use the following formula to approximately determine the ENI capacity.

  ```
  Projected peak concurrent executions * (Memory in GB / 1.5GB)
  ```

  Where:
  - **Projected peak concurrent execution** – Use the information in Lambda Function Concurrent Executions (p. 152) to determine this value.
  - **Memory** – The amount of memory you configured for your Lambda function.

- The subnets you specify should have sufficient available IP addresses to match the number of ENIs.

  We also recommend that you specify at least one subnet in each Availability Zone in your Lambda function configuration. By specifying subnets in each of the Availability Zones, your Lambda function can run in another Availability Zone if one goes down or runs out of IP addresses.

  Caution
  If your VPC does not have sufficient ENIs or subnet IPs, your Lambda function will not scale as requests increase, and you will see an increase in function failures. AWS Lambda currently does not log errors to CloudWatch Logs that are caused by insufficient ENIs or IP addresses. If you see an increase in errors without corresponding CloudWatch Logs, you can invoke the Lambda function synchronously to get the error responses (for example, test your Lambda function in the AWS Lambda console because the console invokes your Lambda function synchronously and displays errors).

Topics

Tutorials: Configuring a Lambda Function to Access Resources in an Amazon VPC

This section provides end-to-end example tutorials where you create and configure a Lambda function to access resources in an Amazon VPC, such as an Amazon ElastiCache cluster or an Amazon RDS database instance.

Topics

Tutorial: Configuring a Lambda Function to Access Amazon ElastiCache in an Amazon VPC

In this tutorial, you do the following:

- Create an Amazon ElastiCache cluster in your default Amazon Virtual Private Cloud (Amazon VPC) in the us-east-1 region. For more information about Amazon ElastiCache, see Amazon ElastiCache.
- Create a Lambda function to access the ElastiCache cluster. When you create the Lambda function, you provide subnet IDs in your Amazon VPC and a VPC security group to allow the Lambda function to access resources in your VPC. For illustration in this tutorial, the Lambda function generates a UUID, writes it to the cache, and retrieves it from the cache.
- Invoke the Lambda function manually and verify that it accessed the ElastiCache cluster in your VPC.

Important
This tutorial uses the default Amazon VPC in the us-east-1 region in your account. For more information about Amazon VPC, see How to Get Started with Amazon VPC in the *Amazon VPC User Guide*.

Next Step

Step 1: Create an ElastiCache Cluster (p. 99)

Step 1: Create an ElastiCache Cluster

In this step, you create an ElastiCache cluster in the default Amazon VPC in us-east-1 region in your account.

1. Run the following AWS CLI command to create a Memcached cluster in the default VPC in the us-east-1 region in your account.

```
aws elasticache create-cache-cluster \
    --cache-cluster-id ClusterForLambdaTest \
    --cache-node-type cache.m3.medium \
    --engine memcached \
    --security-group-ids your-default-vpc-security-group \
    --num-cache-nodes 1
```

   You can look up the default VPC security group in the VPC console under **Security Groups**. Your example Lambda function will add and retrieve an item from this cluster.

   You can also launch a cache cluster using the Amazon ElastiCache console. For instructions, see Getting Started with Amazon ElastiCache in the *Amazon ElastiCache User Guide*.
2. Write down the configuration endpoint for the cache cluster that you launched. You can get this from the Amazon ElastiCache console. You will specify this value in your Lambda function code in the next section.

Next Step

Step 2: Create a Lambda Function (p. 99)

Step 2: Create a Lambda Function

In this step, you do the following:

- Create a Lambda function deployment package using the sample code provided.

- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify this role so that Lambda can assume the role and then execute the function on your behalf.

  The permissions policy grants AWS Lambda permissions to set up elastic network interfaces (ENIs) to enable your Lambda function to access resources in the VPC. In this example, your Lambda function accesses an ElastiCache cluster in the VPC.
- Create the Lambda function by uploading the deployment package.

Topics

Step 2.1: Create a Deployment Package

Note
At this time, example code for the Lambda function is provided only in Python.

Python

The following example Python code reads and writes an item to your ElastiCache cluster.

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

```
from __future__ import print_function
import time
import uuid
import sys
import socket
import elasticache_auto_discovery
from pymemcache.client.hash import HashClient

#elasticache settings
elasticache_config_endpoint = "your-elasticache-cluster-endpoint:port"
nodes = elasticache_auto_discovery.discover(elasticache_config_endpoint)
nodes = map(lambda x: (x[1], int(x[2])), nodes)
memcache_client = HashClient(nodes)

def handler(event, context):
    """
    This function puts into memcache and get from it.
    Memcache is hosted using elasticache
    """

    #Create a random UUID... this will the sample element we add to the
 cache.
    uuid_inserted = uuid.uuid4().hex
    #Put the UUID to the cache.
    memcache_client.set('uuid', uuid_inserted)
    #Get item (UUID) from the cache.
    uuid_obtained = memcache_client.get('uuid')
    if uuid_obtained == uuid_inserted:
        # this print should go to the CloudWatch Logs and Lambda console.
        print ("Success: Fetched value %s from memcache" %(uuid_inserted))
    else:
        raise Exception("Value is not the same as we put :(. Expected %s
 got %s" %(uuid_inserted, uuid_obtained))
```

```
    return "Fetched value from memcache: " + uuid_obtained
```

2. Save the file as `app.py`.
3. Install the following library dependencies using **pip**:

   - `pymemcache` – The Lambda function code uses this library to create a `HashClient` object to set and get items from memcache (see pymemcache).
   - `elasticache-auto-discovery` – The Lambda function uses this library to get the nodes in your Amazon ElastiCache cluster (see elasticache-auto-discovery).
4. Zip all of these files into a file named `app.zip` to create your deployment package. For step-by-step instructions, see Creating a Deployment Package (Python) (p. 69).

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 101)

Step 2.2: Create the Execution Role (IAM Role)

In this step, you create an AWS Identity and Access Management (IAM) role using the following predefined role type and access permissions policy:

- **AWS Lambda** (AWS service role) – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaVPCAccessExecutionRole** (access permissions policy) – This is the policy that you attach to the role. The policy grants permissions for the EC2 actions that AWS Lambda needs to manage ENIs. You can view this AWS managed policy in IAM console.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:

   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-vpc-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - In **Attach Policy**, choose **AWSLambdaVPCAccessExecutionRole**. The permissions in this policy are sufficient for the Lambda function in this tutorial.
3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Next Step

Step 2.3: Create the Lambda Function (Upload the Deployment Package) (p. 101)

Step 2.3: Create the Lambda Function (Upload the Deployment Package)

In this step, you create the Lambda function (`AccessMemCache`) using the `create-function` AWS CLI command.

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

You need to update the following `create-function` command by providing the .zip file path and the execution role ARN. The `--runtime` parameter value can be `python2.7`, `nodejs` and `java8`, depending on the language you used to author your code.

Note
At this time, example code for the Lambda function is provided only in Python.

```
$ aws lambda create-function \
--function-name AccessMemCache  \
--region us-east-1 \
--zip-file fileb://path-to/app.zip \
--role execution-role-arn \
--handler app.handler \
--runtime python2.7  \
--timeout 30 \
--vpc-config SubnetIds=comma-separated-vpc-subnet-
ids,SecurityGroupIds=default-security-group-id \
--memory-size 1024 \
--profile adminuser
```

You can find the subnet IDs and the default security group ID of your VPC from the VPC console.

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

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

Note
You can also create the Lambda function using the AWS Lambda console. When creating the function, choose a VPC for the Lambda and then select the subnets and security groups from the provided fields.

Next Step

Step 3: Test the Lambda Function (Invoke Manually) (p. 102)

Step 3: Test the Lambda Function (Invoke Manually)

In this step, you invoke the Lambda function manually using the `invoke` command. When the Lambda function executes, it generates a UUID and writes it to the ElastiCache cluster that you specified in your Lambda code. The Lambda function then retrieves the item from the cache.

1. Invoke the Lambda function (`AccessMemCache`) using the AWS Lambda `invoke` command.

```
$ aws lambda invoke \
--function-name AccessMemCache  \
--region us-east-1 \
--profile adminuser \
output.txt
```

2. Verify that the Lambda function executed successfully as follows:

   - Review the output.txt file.
   - Review the results in the AWS Lambda console.
   - Verify the results in CloudWatch Logs.

What Next?

Now that you have created a Lambda function that accesses an ElastiCache cluster in your VPC, you can have the function invoked in response to events. For information about configuring event sources and examples, see Use Cases (p. 175).

Tutorial: Configuring a Lambda Function to Access Amazon RDS in an Amazon VPC

In this tutorial, you do the following:

- Launch an Amazon RDS MySQL database engine instance in your default Amazon VPC. In the MySQL instance, you create a database (ExampleDB) with a sample table (Employee) in it. For more information about Amazon RDS, see Amazon RDS.
- Create a Lambda function to access the ExampleDB database, create a table (Employee), add a few records, and retrieve the records from the table.
- Invoke the Lambda function manually and verify the query results. This is how you verify that your Lambda function was able to access the RDS MySQL instance in the VPC.

> **Important**
> This tutorial uses the default Amazon VPC in the us-east-1 region in your account. For more information about Amazon VPC, see How to Get Started with Amazon VPC in the *Amazon VPC User Guide*.

Next Step

Step 1: Create an Amazon RDS MySQL Instance and ExampleDB Database (p. 103)

Step 1: Create an Amazon RDS MySQL Instance and ExampleDB Database

In this tutorial, the example Lambda function creates a table (Employee), inserts a few records, and then retrieves the records. The table that the Lambda function creates has the following schema:

```
Employee(EmpID, Name)
```

Where `EmpID` is the primary key. Now, you need to add a few records to this table.

First, you launch an RDS MySQL instance in your default VPC with ExampleDB database. If you already have an RDS MySQL instance running in your default VPC, skip this step.

> **Important**
> This tutorial uses the RDS MySQL DB engine launched in the default VPC in the us-east-1 region.

You can launch an RDS MySQL instance using one of the following methods:

- Follow the instructions at Creating a MySQL DB Instance and Connecting to a Database on a MySQL DB Instance in the *Amazon Relational Database Service User Guide*.
- Use the following AWS CLI command:

```
$ aws rds create-db-instance \
    --db-instance-identifier MySQLForLambdaTest \
    --db-instance-class db.t2.micro \
    --engine MySQL \
    --allocated-storage 5 \
    --no-publicly-accessible \
    --db-name ExampleDB \
    --master-username username \
    --master-user-password password \
    --backup-retention-period 3 \
    --profile adminuser
```

Write down the database name, user name, and password. You also need the host address (endpoint) of the DB instance, which you can get from the RDS console (you might need to wait until the instance status is available and the Endpoint value appears in the console).

Next Step

Step 2: Create a Lambda Function (p. 104)

Step 2: Create a Lambda Function

In this step, you do the following:

- Create a Lambda function deployment package using the sample code provided.
- Create an IAM role (execution role) that you specify at the time of creating your Lambda function. This is the role AWS Lambda assumes when executing the Lambda function.

  The permissions policy associated with this role grants AWS Lambda permissions to set up elastic network interfaces (ENIs) to enable your Lambda function to access resources in the VPC.
- Create the Lambda function by uploading the deployment package.

Topics

Step 2.1: Create a Deployment Package

Note
At this time, example code for the Lambda function is provided only in Python.

Python

The following example Python code runs a SELECT query against the Employee table in the MySQL RDS instance that you created in the VPC. The code creates a table in the ExampleDB database, adds sample records, and retrieves those records.

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

```
import sys
import logging
import rds_config
import pymysql
#rds settings
rds_host  = "rds-instance-endpoint"
name = rds_config.db_username
password = rds_config.db_password
db_name = rds_config.db_name
```

```
logger = logging.getLogger()
logger.setLevel(logging.INFO)

try:
    conn = pymysql.connect(rds_host, user=name, passwd=password,
 db=db_name, connect_timeout=5)
except:
    logger.error("ERROR: Unexpected error: Could not connect to MySql
 instance.")
    sys.exit()

logger.info("SUCCESS: Connection to RDS mysql instance succeeded")
def handler(event, context):
    """
    This function fetches content from mysql RDS instance
    """

    item_count = 0

    with conn.cursor() as cur:
        cur.execute("create table Employee3 ( EmpID  int NOT NULL, Name
 varchar(255) NOT NULL, PRIMARY KEY (EmpID))")
        cur.execute('insert into Employee3 (EmpID, Name) values(1,
 "Joe")')
        cur.execute('insert into Employee3 (EmpID, Name) values(2,
 "Bob")')
        cur.execute('insert into Employee3 (EmpID, Name) values(3,
 "Mary")')
        conn.commit()
        cur.execute("select * from Employee3")
        for row in cur:
            item_count += 1
            logger.info(row)
            #print(row)


    return "Added %d items from RDS MySQL table" %(item_count)
```

Note
We recommend that `pymysql.connect()` is executed outside the handler, as shown, for better performance.

2. Save the file as `app.py`.
3. Install the following library dependencies using **pip**:

   - `pymysql` – The Lambda function code uses this library to access your MySQL instance (see PyMySQL) .
4. Create a config file that contains the following information and save it as `rds_config.py`:

```
#config file containing credentials for rds mysql instance
db_username = "username"
db_password = "password"
db_name = "databasename"
```

5. Zip all of these files into a file named `app.zip` to create your deployment package. For step-by-step instructions, see Creating a Deployment Package (Python) (p. 69).

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 106)

Step 2.2: Create the Execution Role (IAM Role)

In this step, you create an execution role (IAM role) for your Lambda function using the following predefined role type and access permissions policy:

- **AWS Lambda** (AWS service role) – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaVPCAccessExecutionRole** (access permissions policy) – This role grants AWS Lambda permissions for EC2 actions to create ENIs and your Lambda function can access VPC resources and CloudWatch Logs actions to write logs.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-vpc-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - In **Attach Policy**, choose **AWSLambdaVPCAccessExecutionRole**. The permissions in this policy are sufficient for the Lambda function in this tutorial.
3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Next Step

Step 2.3: Create the Lambda Function (Upload the Deployment Package) (p. 106)

Step 2.3: Create the Lambda Function (Upload the Deployment Package)

In this step, you create the Lambda function (`ReadMySqlTable`) using the `create-function` AWS CLI command.

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

You need to update the following `create-function` command by providing the .zip file path and the execution role ARN. The `--runtime` parameter value can be `python2.7`, `nodejs`, or `java8`, depending on the language you used to author your code.

> **Note**
> At this time, example code for the Lambda function is provided only in Python.

```
$ aws lambda create-function \
--region us-east-1 \
--function-name   CreateTableAddRecordsAndRead  \
--zip-file fileb://file-path/app.zip \
--role execution-role-arn \
--handler app.handler \
--runtime python2.7 \
--vpc-config SubnetIds=comma-separated-subnet-ids,SecurityGroupIds=default-
vpc-security-group-id \
--profile adminuser
```

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

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

Note
You can also create the Lambda function using the AWS Lambda console (use the parameter values shown in the preceding CLI command).

Next Step

Step 3: Test the Lambda Function (Invoke Manually) (p. 107)

Step 3: Test the Lambda Function (Invoke Manually)

In this step, you invoke the Lambda function manually using the `invoke` command. When the Lambda function executes, it runs the SELECT query against the Employee table in the RDS MySQL instance and prints the results (these results also go to the CloudWatch Logs).

1. Invoke the Lambda function (`ReadMySqlTable`) using the AWS Lambda `invoke` command.

```
$ aws lambda invoke \
--function-name CreateTableAddRecordsAndRead  \
--region us-east-1 \
--profile adminuser \
output.txt
```

2. Verify that the Lambda function executed successfully as follows:

   - Review the output.txt file.
   - Review the results in the AWS Lambda console.
   - Verify the results in CloudWatch Logs.

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 User 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. 109)
- AWS Lambda Metrics (p. 111)

You can insert logging statements into your code to help you validate that 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 (/aws/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 User Guide*. For information about how to access CloudWatch log entries, see Accessing Amazon CloudWatch Logs for AWS Lambda (p. 110).

> **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 grant these permissions, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).

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 Tutorial: Using AWS Lambda with Amazon S3 (p. 177). 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 your code and verify that it 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 the Lambda console using the console's test invoke functionality, or you can use the AWS CLI `Invoke` 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 /aws/lambda/<*function name*>.

   Following are some examples of 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 that 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 the IAM role and verify that it has all of the necessary permissions to access any AWS resources that your code references. To ensure that you have correctly set up the execution role, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).
   - 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 information about memory size limits, see CreateFunction (p. 332). When you change the memory setting, it can also change how you are charged for duration. For information about pricing, see the AWS Lambda product website.

2. Check your Lambda function and verify that it 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 metrics. If you do not see an increase in your CloudWatch requests, check the access permissions policy associated with the function.

Troubleshooting Scenario 2: Increased Latency in Lambda Function Execution

In this scenario, you have just finished Tutorial: Using AWS Lambda with Amazon S3 (p. 177). 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 CloudWatch errors 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's 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. Investigate the source of the execution bottleneck that is using logs.

   You can test your code locally, as you would with any other Node.js function, or you can test it within Lambda using the test invoke capability on the Lambda console, or using 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 aws/lambda/<*function name*>. Add logging statements around various parts of your code, such as callouts to other services, to see how much time it takes to execute different parts of your code.

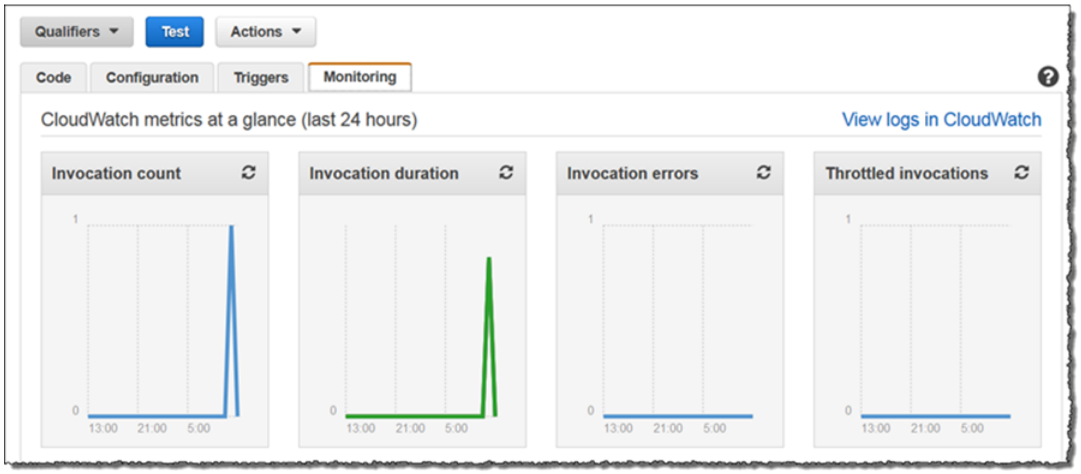
Accessing Amazon CloudWatch Metrics for AWS Lambda

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. 111). For more information about CloudWatch, see the Amazon CloudWatch User 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 AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. If you have not created a Lambda function before, see Getting Started (p. 160).
3. On the **Functions** page, choose the function name and then choose the **Monitoring** tab.



A graphical representation of the metrics for the Lambda function are shown.
4. Choose **View logs in CloudWatch** to view the logs.

To access metrics using the CloudWatch console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. From the navigation bar, choose a region.
3. In the navigation pane, choose **Metrics**.
4. In the **CloudWatch Metrics by Category** pane, choose **Lambda Metrics**.
5. (Optional) In the graph pane, choose 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.

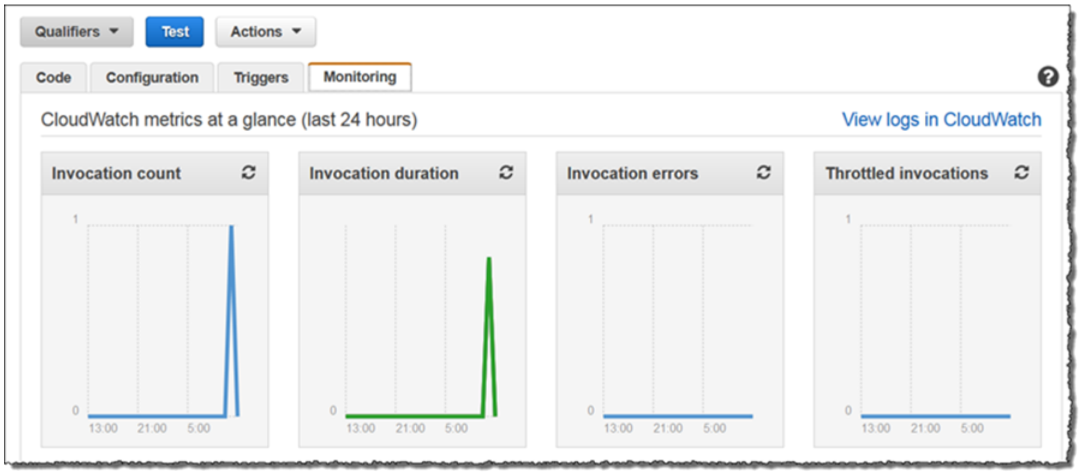
You can insert logging statements into your code to help you validate that your code is working as expected. Lambda automatically integrates with CloudWatch Logs and pushes all logs from your code to a CloudWatch Logs group associated with a Lambda function, which is named /aws/lambda/<*function name*>. 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 User 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.

Note
There is no additional charge for using Lambda logs; however, standard CloudWatch Logs charges apply. For more information, see CloudWatch Pricing.

To view logs using the Lambda console

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. If you have not created a Lambda function before, see Getting Started (p. 160).
3. On the **Functions** page, choose the function name and then choose the **Monitoring** tab.



A graphical representation of the metrics for the Lambda function are shown.
4. Choose **View logs in CloudWatch** to view the logs.

For more information on accessing CloudWatch Logs, see the following guides:

- Amazon CloudWatch User Guide
- Amazon CloudWatch Logs API Reference
- Monitoring Log Files in the *Amazon CloudWatch User Guide*

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 services) publish metrics 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 User 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
`Dead Letter Error`	The Dead Letter Error metric will be incremented when Lambda is unable to write the failed event payload to your configured Dead Letter Queues (p. 113). This could be due to the following: • Permissions errors • Throttles from downstream services • Misconfigured resources • Timeouts 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

Metric	Description
	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

Errors/Invocations Ratio
When calculating the error rate on Lambda function invocations, it's important to distinguish between an invocation request and an actual invocation. It is possible for the error rate to exceed the number of billed Lambda function invocations. Lambda reports an invocation metric only if the Lambda function code is executed. If the invocation request yields a throttling or other initialization error that prevents the Lambda function code from being invoked, Lambda will report an error, but it does not log an invocation metric.

- Lambda emits `Invocations=1` when the function is executed. If the Lambda function is not executed, nothing is emitted.
- Lambda emits a data point for `Errors` for each invoke request. `Errors=0` means that there is no function execution error. `Errors=1` means that there is a function execution error.
- Lambda emits a data point for `Throttles` for each invoke request. `Throttles=0` means there is no invocation throttle. `Throttles=1` means there is an invocation throttle.

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 metric data by Lambda function.
`Resource`	Filters the metric data by Lambda function resource.
`Version`	Filters the metric data by Lambda version.
`Alias`	Filters the metric data by Lambda alias.

Dead Letter Queues

By default, a failed Lambda function invoked asynchronously will be retried twice, after which the event will be discarded. Using Dead Letter Queues (DLQ), you can indicate to Lambda that any such unprocessed events should instead be sent to an Amazon SQS queue or Amazon SNS topic, where you can take further action.

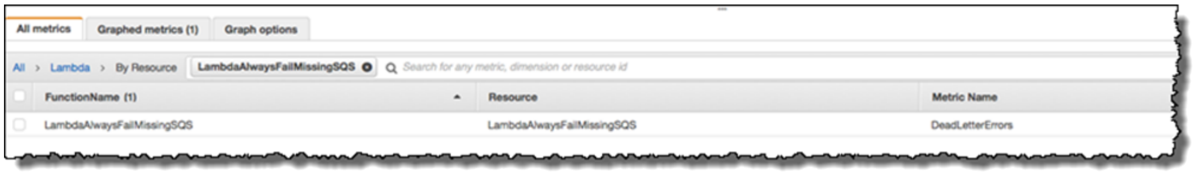
You configure a DLQ by specifying a target arn (Amazon Resource Name) on a Lambda function's `DeadLetterConfig` parameter of an Amazon SNS topic or an Amazon SQS queue where you want the event payload delivered. For more information on creating an Amazon SNS topic, see Create an SNS Topic. For more information on creating an Amazon SQS queue, see Tutorial: Creating an Amazon SQS Queue.

```
{
    "Code": {
        "ZipFile": blob,
        "S3Bucket": “string”,
        "S3Key": “string”,
        "S3ObjectVersion": “string”
    },
    "Description": "string",
    "FunctionName": "string",
    "Handler": "string",
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
    "Publish": bool,
    "DeadLetterConfig": {
        "TargetArn": "string"
    }
}
```

Lambda will direct events that cannot be processed to the Amazon SNS topic or Amazon SQS queue that you’ve configured for the Lambda function. Functions without an associated DLQ will discard events that have exhausted their retries. For more information on retry policies, see Retries on Errors (p. 154). You will need to explicitly provide read/publish/sendMessage access to your DLQ resource as part of the execution role for your Lambda function. The payload written to the DLQ target arn will be the original event payload with no modifications to the message body. The attributes of the message, described below, will contain information to help you understand why the event wasn’t processed:

Name	Type	Value
RequestID	String	Unique request identifier
ErrorCode	Number	3-digit HTTP error code
ErrorMessage	String	Error message (truncated to 1 KB)

If for some reason, the event payload consistently fails to reach the target arn, Lambda will increment a CloudWatch metric called `DeadLetterErrors` and then delete the event payload.


All metrics
Graphed metrics (1)
Graph options
All > Lambda > By Resource
LambdaAlwaysFailMissingSQS
Search for any metric, dimension or resource id

FunctionName (1)	Resource	Metric Name
LambdaAlwaysFailMissingSQS	LambdaAlwaysFailMissingSQS	DeadLetterErrors

Building applications with AWS Lambda

When building applications on AWS Lambda, the core components are Lambda functions and event sources. An *event source* is the AWS service or custom application that publishes events, and a *Lambda function* is the custom code that processes the events. To illustrate, consider the following scenarios:

- **File processing** – Suppose you have a photo sharing application. People use your application to upload photos, and the application stores these user photos in an Amazon S3 bucket. Then, your application creates a thumbnail version of each user's photos and displays them on the user's profile page. In this scenario, you may choose to create a Lambda function that creates a thumbnail automatically. Amazon S3 is one of the supported AWS event sources that can publish *object-created events* and invoke your Lambda function. Your Lambda function code can read the photo object from the S3 bucket, create a thumbnail version, and then save it in another S3 bucket.

- **Data and analytics** – Suppose you are building an analytics application and storing raw data in a DynamoDB table. When you write, update, or delete items in a table, DynamoDB streams can publish item update events to a stream associated with the table. In this case, the event data provides the item key, event name (such as insert, update, and delete), and other relevant details. You can write a Lambda function to generate custom metrics by aggregating raw data.

- **Websites** – Suppose you are creating a website and you want to host the backend logic on Lambda. You can invoke your Lambda function over HTTP using Amazon API Gateway as the HTTP endpoint. Now, your web client can invoke the API, and then API Gateway can route the request to Lambda.

- **Mobile applications** – Suppose you have a custom mobile application that produces events. You can create a Lambda function to process events published by your custom application. For example, in this scenario you can configure a Lambda function to process the clicks within your custom mobile application.

Each of these event sources uses a specific format for the event data. For more information, see Sample Events Published by Event Sources (p. 130). When a Lambda function is invoked, it receives the event as a parameter for the Lambda function.

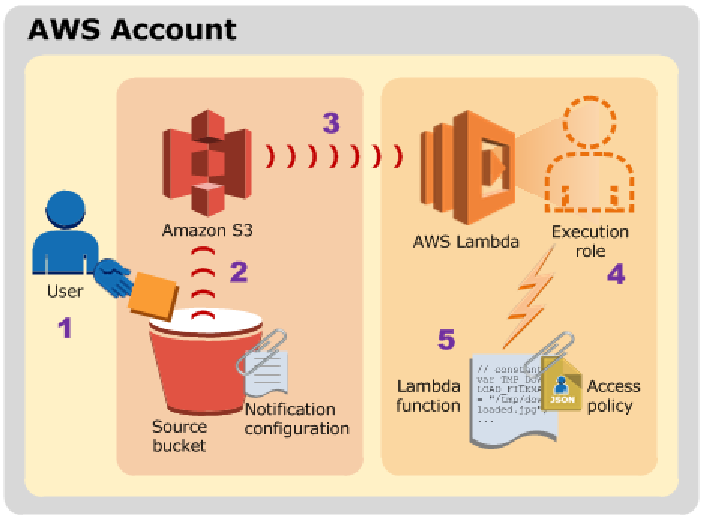
AWS Lambda supports many AWS services as event sources. For more information, see Supported Event Sources (p. 124). When you configure these event sources to trigger a Lambda function, the Lambda function is invoked automatically when events occur. You define *event source mapping*, which is how you identify what events to track and which Lambda function to invoke.

In addition to the supported AWS services, user applications can also generate events—you can build your own custom event sources. Custom event sources invoke a Lambda function using the AWS Lambda Invoke (p. 358) operation. User applications, such as client, mobile, or web applications, can publish events and invoke Lambda functions on demand using the AWS SDKs or AWS Mobile SDKs, such as the AWS Mobile SDK for Android.

The following are introductory examples of event sources and how the end-to-end experience works.

Example 1: Amazon S3 Pushes Events and Invokes a Lambda Function

Amazon S3 can publish events of different types, such as PUT, POST, COPY, and DELETE object events on a bucket. Using the bucket notification feature, you can configure an event source mapping that directs Amazon S3 to invoke a Lambda function when a specific type of event occurs, as shown in the following illustration.



The diagram illustrates the following sequence:

1. The user creates an object in a bucket.
2. Amazon S3 detects the object created event.
3. Amazon S3 invokes your Lambda function using the permissions provided by the execution role. For more information on execution roles, see Authentication and Access Control for AWS Lambda (p. 287). Amazon S3 knows which Lambda function to invoke based on the event source mapping that is stored in the bucket notification configuration.
4. AWS Lambda executes the Lambda function, specifying the event as a parameter.

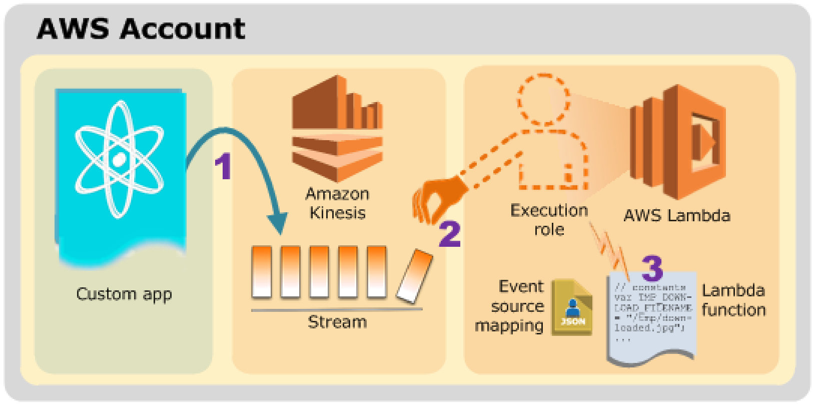
Note the following:

- The event source mapping is maintained within the event source service, Amazon S3 in this scenario. This is true for all supported AWS event sources except the stream-based sources (Amazon Kinesis and DynamoDB streams). The next example explains stream-based event sources.
- The event source (Amazon S3) invokes the Lambda function (referred to as the *push model*). Again, this is true for all supported AWS services except the stream-based event sources.
- In order for the event source (Amazon S3) to invoke your Lambda function, you must grant permissions using the permissions policy attached to the Lambda function.

Example 2: AWS Lambda Pulls Events from an Amazon Kinesis Stream and Invokes a Lambda Function

For stream-based event sources, AWS Lambda polls the stream and invokes the Lambda function when records are detected on the stream. These stream sources are special in that event source mapping information is stored in Lambda. AWS Lambda provides an API for you to create and manage these event source mappings.

The following diagram shows how a custom application writes records to an Amazon Kinesis stream.



The diagram illustrates the following sequence:

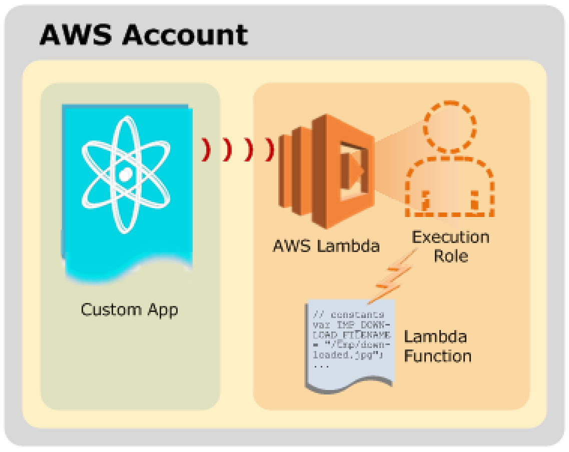
1. The custom application writes records to an Amazon Kinesis stream.
2. AWS Lambda continuously polls the stream, and invokes the Lambda function when the service detects new records on the stream. AWS Lambda knows which stream to poll and which Lambda function to invoke based on the event source mapping you create in Lambda.
3. The Lambda function is invoked with the incoming event.

Note the following:

- When working with stream-based event sources:
  - You create event source mappings in AWS Lambda.
  - AWS Lambda invokes the Lambda function (referred to as the *pull model*).
- AWS Lambda does not need permission to invoke your Lambda function, therefore you don't need to add any permissions to the permissions policy attached to your Lambda function.
- Your Lambda role needs permission to read from the stream.

Example 3: Custom Application Publishes Events and Invokes a Lambda Function

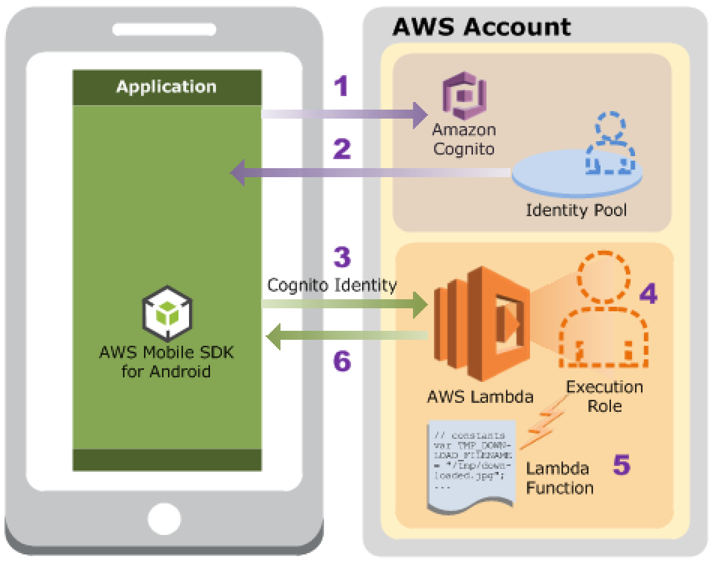
The following diagram shows how a custom application in your account invokes your Lambda function.



The diagram illustrates the following sequence:

1. The custom application invokes your Lambda function using the AWS SDK.
2. The Lambda function is invoked with the incoming event.

The following diagram shows how a custom mobile application invokes a Lambda function.



1. The mobile application sends a request to Amazon Cognito with an identity pool ID in the request (you create the identity pool as part of the setup).

2. Amazon Cognito returns temporary security credentials to the application.

   Amazon Cognito assumes the role associated with the identity pool to generate temporary credentials.

3. The mobile application invokes the Lambda function using the temporary credentials (Cognito Identity).

4. AWS Lambda assumes the execution role to execute your Lambda function on your behalf.

5. The Lambda function executes.

6. AWS Lambda returns results to the mobile application, assuming the app invoked the Lambda function using the `RequestResponse` invocation type (referred to as *synchronous invocation*).

Note the following:

- This is an example of an on-demand invocation of a Lambda function. For on-demand invocations, you don't need to preconfigure an event source mapping like you do for AWS services.

- In this example, because the custom application is using the same account credentials as the account that owns the Lambda function, it does not require additional permissions to invoke the function.

Suggested Reading

If you are new to AWS Lambda, at this time you can continue and read all of the topics in this How It Works chapter for details. You might also consider exploring the Getting Started (p. 160) exercise first to get hands-on experience creating and testing a Lambda function, and then read the topics in this chapter.

Additionally, the Building Lambda Functions (p. 5) also provides introductory information that you might find useful, before you dive deep into the technology.

Event Source Mapping

In AWS Lambda, Lambda functions and event sources are the core components in AWS Lambda. An event source is the entity that publishes events, and a Lambda function is the custom code that processes the events. Supported event sources refer to those AWS services that can be preconfigured to work with AWS Lambda. The configuration is referred to as *event source mapping*, which maps an event source to a Lambda function. It enables automatic invocation of your Lambda function when events occur.

Each event source mapping identifies the type of events to publish and the Lambda function to invoke when events occur. The specific Lambda function then receives the event information as a parameter, your Lambda function code can then process the event.

Note the following about the event sources. These event sources can be any of the following:

- **AWS services** – These are the supported AWS services that can be preconfigured to work with AWS Lambda. You can group these services as regular AWS services or stream-based services. Amazon Kinesis Streams and Amazon DynamoDB Streams are stream-based event sources, all others AWS services do not use stream-based event sources. Where you maintain the event source mapping and how the Lambda function is invoked depends on whether or not you're using a stream-based event source.

- **Custom applications** – You can have your custom applications publish events and invoke a Lambda function.

You may be wondering—where do I keep the event mapping information? Do I keep it within the event source or within AWS Lambda? The following sections explain event source mapping for each of these event source categories. These sections also explain how the Lambda function is invoked and how you manage permissions to allow invocation of your Lambda function.

Topics

Event Source Mapping for AWS Services

Except for the stream-based AWS services (Amazon Kinesis Streams and DynamoDB streams), other supported AWS services publish events and can also invoke your Lambda function (referred to as the *push model*). In the push model, note the following:

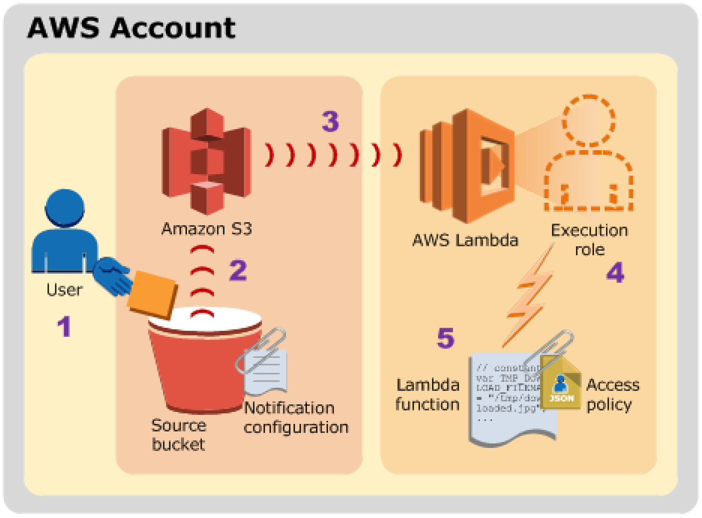
- Event source mappings are maintained within the event source. Relevant API support in the event sources enables you to create and manage event source mappings. For example, Amazon S3 provides the bucket notification configuration API. Using this API, you can configure an event source mapping that identifies the bucket events to publish and the Lambda function to invoke.

- Because the event sources invoke your Lambda function, you need to grant the event source the necessary permissions using a resource-based policy (referred to as the *Lambda function policy*). For more information, see AWS Lambda Permissions Model (p. 155).

The following example illustrates how this model works.

Example – Amazon S3 Pushes Events and Invokes a Lambda Function

Suppose that you want your AWS Lambda function invoked for each *object created* bucket event. You add the necessary event source mapping in the bucket notification configuration.



The diagram illustrates the flow:

1. The user creates an object in a bucket.
2. Amazon S3 detects the object created event.
3. Amazon S3 invokes your Lambda function according to the event source mapping described in the bucket notification configuration.
4. AWS Lambda verifies the permissions policy attached to the Lambda function to ensure that Amazon S3 has the necessary permissions. For more information on permissions policies, see Authentication and Access Control for AWS Lambda (p. 287)
5. Once AWS Lambda verifies the attached permissions policy, it executes the Lambda function. Remember that your Lambda function receives the event as a parameter.

Event Source Mapping for AWS Stream-based Services

The Amazon Kinesis Streams and DynamoDB streams are the stream-based services that you can preconfigure to use with AWS Lambda. After you do the necessary event source mapping, AWS Lambda polls the streams and invokes your Lambda function (referred to as the *pull model*). In the pull model, note the following:

- The event source mappings are maintained within the AWS Lambda. AWS Lambda provides the relevant APIs to create and manage event source mappings. For more information, see CreateEventSourceMapping (p. 328).

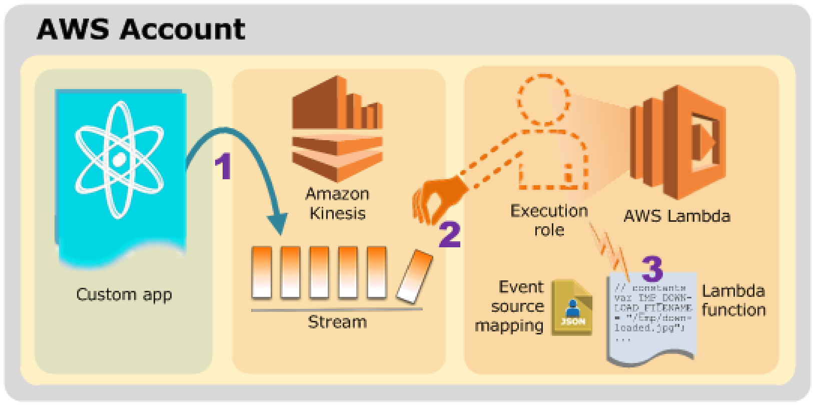
- AWS Lambda needs your permission to poll the stream and read records. You grant these permissions via the execution role, using the permissions policy associated with role that you specify when you create your Lambda function. AWS Lambda does not need any permissions to invoke your Lambda function.

The following example illustrates how this model works.

Example – AWS Lambda Pulls Events from an Amazon Kinesis Stream and Invokes a Lambda Function

The following diagram shows a custom application that writes records to an Amazon Kinesis stream and how AWS Lambda polls the stream. When AWS Lambda detects a new record on the stream, it invokes your Lambda function.

Suppose you have a custom application that writes records to an Amazon Kinesis stream. You want to invoke a Lambda function when new records are detected on the stream. You create a Lambda function and the necessary event source mapping in AWS Lambda.



The diagram illustrates the following sequence:

1. The custom application writes records to an Amazon Kinesis stream.

2. AWS Lambda continuously polls the stream, and invokes the Lambda function when the service detects new records on the stream. AWS Lambda knows which stream to poll and which Lambda function to invoke based on the event source mapping you create in AWS Lambda.

3. Assuming the attached permission policy, which allows AWS Lambda to poll the stream, is verified, AWS Lambda then executes the Lambda function. For more information on permissions policies, see Authentication and Access Control for AWS Lambda (p. 287)

The example uses an Amazon Kinesis stream but the same applies when working with a DynamoDB stream.

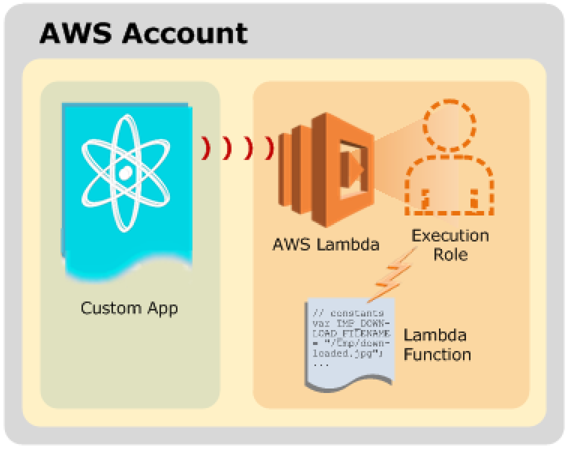
Event Source Mapping for Custom Applications

If you have custom applications that publish and process events, you can create a Lambda function to process these events. In this case, there is no preconfiguration required—you don't have to set up an event source mapping. Instead, the event source uses the AWS Lambda `Invoke` API. If the application and Lambda function are owned by different AWS accounts, the AWS account that owns the Lambda function must allow cross-account permissions in the permissions policy associated with the Lambda function.

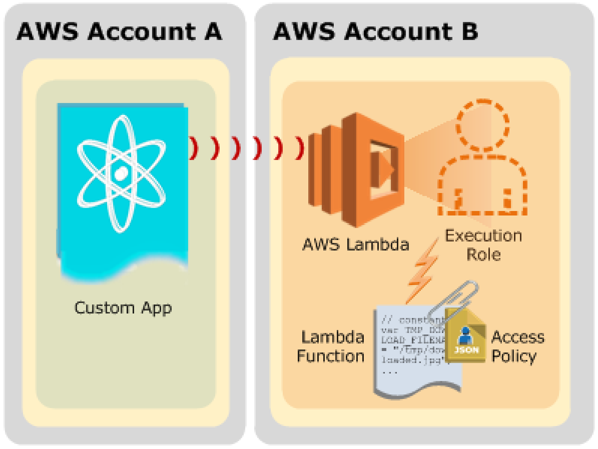
The following example illustrates how this works.

Example – Custom Application Publishes Events and Invokes a Lambda Function

The following diagram shows how a custom application in your account can invoke a Lambda function. In this example, the custom application is using the same account credentials as the account that owns the Lambda function, and, therefore, does not require additional permissions to invoke the function.



In the following example, the user application and Lambda function are owned by different AWS accounts. In this case, the AWS account that owns the Lambda function must have cross-account permissions in the permissions policy associated with the Lambda function. For more information, see AWS Lambda Permissions Model (p. 155).



Suggested Reading

If you are new to AWS Lambda, we suggest you read through all of the topics in the How It Works section to familiarize yourself with Lambda. The next topic is Supported Event Sources (p. 124).

After you read all of the topics in the How it Works section, we recommend that you review Building Lambda Functions (p. 5), try the Getting Started (p. 160) exercise, and then explore the Use Cases (p. 175). Each use case provides step-by-step instructions for you to set up the end-to-end experience.

Supported Event Sources

This topic lists the supported AWS services that you can configure as event sources for AWS Lambda functions. After you preconfigure the event source mapping, your Lambda function gets invoked automatically when these event sources detect events. For more information about invocation modes, see Event Source Mapping (p. 120).

For all of the event sources listed in this topic, note the following:

- Event sources maintain the event source mapping, except for the stream-based services (Amazon Kinesis Streams and Amazon DynamoDB Streams). For the stream-based services, AWS Lambda maintains the event source mapping. AWS Lambda provides the CreateEventSourceMapping (p. 328) operation for you to create and manage the event source mapping. For more information, see Event Source Mapping (p. 120).

- The invocation type that these event sources use when invoking a Lambda function is also preconfigured. For example, Amazon S3 always invokes a Lambda function asynchronously and Amazon Cognito invokes a Lambda function synchronously. The only time you can control the invocation type is when you are invoking the Lambda function yourself using the Invoke (p. 358) operation (for example, invoking a Lambda function on demand from your custom application).

You can also invoke a Lambda function on demand. For details, see Other Event Sources: Invoking a Lambda Function On Demand (p. 130).

For examples of events that are published by these event sources, see Sample Events Published by Event Sources (p. 130).

Topics

Amazon S3

You can write Lambda functions to process S3 bucket events, such as the object-created or object-deleted events. For example, when a user uploads a photo to a bucket, you might want Amazon S3 to invoke your Amazon S3 function so that it reads the image and creates a thumbnail for the photo.

You can use the bucket notification configuration feature in Amazon S3 to configure the event source mapping, identifying the bucket events that you want Amazon S3 to publish and which Lambda function to invoke.

For an example Amazon S3 event, see Event Message Structure, Amazon S3 Put Sample Event (p. 136), and Amazon S3 Delete Sample Event (p. 137). For an example use case, see Using AWS Lambda with Amazon S3 (p. 175).

Error handling for a given event source depends on how Lambda is invoked. Amazon S3 invokes your Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon DynamoDB

You can use Lambda functions as triggers for your Amazon DynamoDB table. Triggers are custom actions you take in response to updates made to the DynamoDB table. To create a trigger, first you enable Amazon DynamoDB Streams for your table. AWS Lambda polls the stream and your Lambda function processes any updates published to the stream.

This is a stream-based event source. For stream-based service, you create event source mapping in AWS Lambda, identifying the stream to poll and which Lambda function to invoke.

For an example DynamoDB event, see Step 2.3.2: Test the Lambda Function (Invoke Manually) (p. 210) and Amazon DynamoDB Update Sample Event (p. 134). For general format, see GetRecord in the *Amazon DynamoDB API Reference*. For an example use case, see Using AWS Lambda with Amazon DynamoDB (p. 203).

Error handling for a given event source depends on how Lambda is invoked. DynamoDB is a stream-based event source. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon Kinesis Streams

You can configure AWS Lambda to automatically poll your stream and process any new records such as website click streams, financial transactions, social media feeds, IT logs, and location-tracking events. Then, AWS Lambda polls the stream periodically (multiple times per second) for new records.

For stream-based service, you create event source mapping in AWS Lambda, identifying the stream to poll and which Lambda function to invoke.

For an example event, see Step 2.3: Create the Lambda Function and Test It Manually (p. 199) and Amazon Kinesis Streams Sample Event (p. 136). For an example use case, see Using AWS Lambda with Amazon Kinesis (p. 193).

Error handling for a given event source depends on how Lambda is invoked. Amazon Kinesis Streams is a stream-based event source. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon Simple Notification Service

You can write Lambda functions to process Amazon Simple Notification Service notifications. When a message is published to an Amazon SNS topic, the service can invoke your Lambda function by passing the message payload as parameter. Your Lambda function code can then process the event, for example publish the message to other Amazon SNS topics, or send the message to other AWS services.

This also enables you to trigger a Lambda function in response to Amazon CloudWatch alarms and other AWS services that use Amazon SNS.

You configure the event source mapping in Amazon SNS via topic subscription configuration. For more information, see Invoking Lambda functions using Amazon SNS notifications in the *Amazon Simple Notification Service Developer Guide*.

For an example event, see Appendix: Message and JSON Formats and Amazon SNS Sample Event (p. 133). For an example use case, see Using AWS Lambda with Amazon SNS from Different Accounts (p. 230).

When a user calls the SNS Publish API on a topic that your Lambda function is subscribed to, Amazon SNS will call Lambda to invoke your function asynchronously. Lambda will then return a delivery status.

If there was an error calling Lambda, Amazon SNS will retry invoking the Lambda function up to three times. After three tries, if Amazon SNS still could not successfully invoke the Lambda function, then Amazon SNS will send a delivery status failure message to CloudWatch.

Error handling for a given event source depends on how Lambda is invoked. Amazon SNS invokes your Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon Simple Email Service

Amazon Simple Email Service (Amazon SES) is a cost-effective email service. With Amazon SES, in addition to sending emails, you can also use the service to receive messages. For more information about Amazon SES, see Amazon Simple Email Service. When you use Amazon SES to receive messages, you can configure Amazon SES to call your Lambda function when messages arrive. The service can then invoke your Lambda function by passing in the incoming email event as parameter. For example scenarios, see Considering Your Use Case for Amazon SES Email Receiving.

You configure event source mapping using the rule configuration in Amazon SES. The following topics provide additional information in the *Amazon Simple Email Service Developer Guide*:

- For sample events, see Lambda Action and Amazon SES Email Receiving Sample Event (p. 131).
- For Lambda function examples, see Lambda Function Examples.

Error handling for a given event source depends on how Lambda is invoked. Amazon SES invokes your Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon Cognito

The Amazon Cognito Events feature enables you to run Lambda function in response to events in Amazon Cognito. For example, you can invoke a Lambda function for the Sync Trigger events, that is published each time a dataset is synchronized. To learn more and walk through an example, see Introducing Amazon Cognito Events: Sync Triggers in the Mobile Development blog.

You configure event source mapping using Amazon Cognito event subscription configuration. For information about event source mapping and a sample event, see Amazon Cognito Events in the *Amazon Cognito Developer Guide*. For another example event, see Amazon Cognito Sync Trigger Sample Event (p. 136)

Error handling for a given event source depends on how Lambda is invoked. Amazon Cognito is configured to invoke a Lambda function synchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

AWS CloudFormation

As part of deploying AWS CloudFormation stacks, you can specify a Lambda function as a custom resource to execute any custom commands. Associating a Lambda function with a custom resource enables you to invoke your Lambda function whenever you create, update, or delete AWS CloudFormation stacks.

You configure event source mapping in AWS CloudFormation using stack definition. For more information, see AWS Lambda-backed Custom Resources in the *AWS CloudFormation User Guide*.

For an example event, see AWS CloudFormation Create Request Sample Event (p. 131).

Error handling for a given event source depends on how Lambda is invoked. AWS CloudFormation invokes your Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon CloudWatch Logs

You can use AWS Lambda functions to perform custom analysis on Amazon CloudWatch Logs using CloudWatch Logs subscriptions. CloudWatch Logs subscriptions provide access to a real-time feed of log events from CloudWatch Logs and deliver it to your AWS Lambda function for custom processing, analysis, or loading to other systems. For more information about CloudWatch Logs, see Monitoring Log Files.

You maintain event source mapping in Amazon CloudWatch Logs using the log subscription configuration. For more information, see Real-time Processing of Log Data with Subscriptions (Example 2: AWS Lambda) in the *Amazon CloudWatch User Guide*.

For an example event, see Amazon CloudWatch Logs Sample Event (p. 133).

Error handling for a given event source depends on how Lambda is invoked. Amazon CloudWatch Logs invokes your Lambda function asynchronously (invoking a Lambda function does not block write operation into the logs). For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon CloudWatch Events

Amazon CloudWatch Events help you to respond to state changes in your AWS resources. When your resources change state, they automatically send events into an event stream. You can create rules that match selected events in the stream and route them to your AWS Lambda function to take action. For example, you can automatically invoke an AWS Lambda function to log the state of an EC2 instance or AutoScaling Group.

You maintain event source mapping in Amazon CloudWatch Events by using a rule target definition. For more information, see the PutTargets operation in the *Amazon CloudWatch Events API Reference*.

For sample events, see Supported Event Types in the *Amazon CloudWatch User Guide*.

Error handling for a given event source depends on how Lambda is invoked. Amazon CloudWatch Events invokes your Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

AWS CodeCommit

You can create a trigger for an AWS CodeCommit repository so that events in the repository will invoke a Lambda function. For example, you can invoke a Lambda function when a branch or tag is created or when a push is made to an existing branch. For more information, see Manage Triggers for an AWS CodeCommit Repository.

You maintain the event source mapping in AWS CodeCommit by using a repository trigger. For more information, see the PutRepositoryTriggers operation.

Error handling for a given event source depends on how Lambda is invoked. AWS CodeCommit invokes your Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Scheduled Events (powered by Amazon CloudWatch Events)

You can also set up AWS Lambda to invoke your code on a regular, scheduled basis using the schedule event capability in Amazon CloudWatch Events. To set a schedule you can specify a fixed rate (number of hours, days, or weeks) or specify a cron expression (see Schedule Expression Syntax for Rules in the *Amazon CloudWatch User Guide*).

You maintain event source mapping in Amazon CloudWatch Events by using a rule target definition. For more information, see the PutTargets operation in the *Amazon CloudWatch Events API Reference*.

For an example use case, see Using AWS Lambda with Scheduled Events (p. 263).

For an example event, see Scheduled Event Sample Event (p. 133).

Error handling for a given event source depends on how Lambda is invoked. Amazon CloudWatch Events is configured to invoke a Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

AWS Config

You can use AWS Lambda functions to evaluate whether your AWS resource configurations comply with your custom Config rules. As resources are created, deleted, or changed, AWS Config records these changes and sends the information to your Lambda functions. Your Lambda functions then evaluate the changes and report results to AWS Config. You can then use AWS Config to assess overall resource compliance: you can learn which resources are noncompliant and which configuration attributes are the cause of noncompliance.

You maintain event source mapping in AWS Config by using a rule target definition. For more information, see the PutConfigRule operation in the *AWS Config API reference*.

For more information, see Evaluating Resources With AWS Config Rules. For an example of setting a custom rule, see Developing a Custom Rule for AWS Config. For example Lambda functions, see Example AWS Lambda Functions for AWS Config Rules (Node.js).

Error handling for a given event source depends on how Lambda is invoked. AWS Config is configured to invoke a Lambda function asynchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon Echo

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, see Getting Started with Alexa Skills Kit.

Error handling for a given event source depends on how Lambda is invoked. Amazon Echo is configured to invoke a Lambda function synchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon Lex

Amazon Lex is an AWS service for building conversational interfaces into applications using voice and text. Amazon Lex provides pre-build integration with AWS Lambda, allowing you to create Lambda functions for use as code hook with your Amazon Lex bot. In your intent configuration, you can identify your Lambda function to perform initialization/validation, fulfillment, or both.

For more information, see Using Lambda Functions. For an example use case, see Exercise 1: Create Amazon Lex Bot Using a Blueprint.

Error handling for a given event source depends on how Lambda is invoked. Amazon Lex is configured to invoke a Lambda function synchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

Amazon API Gateway

You can invoke a Lambda function over HTTPS. You can do this by defining a custom REST API and endpoint using Amazon API Gateway. You map individual API operations, 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 Make Synchronous Calls to Lambda Functions. For an example use case, see Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS) (p. 236).

Error handling for a given event source depends on how Lambda is invoked. Amazon API Gateway is configured to invoke a Lambda function synchronously. For more information on how errors are retried, see Retries on Errors (p. 154).

In addition, you can also use Lambda functions with other AWS services that publish data to one of the supported AWS event sources listed in this topic. For example, you can:

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

Other Event Sources: Invoking a Lambda Function On Demand

In addition to invoking Lambda functions using event sources, you can also invoke your Lambda function on demand. You don't need to preconfigure any event source mapping in this case. However, make sure that the custom application has the necessary permissions to invoke your Lambda function.

For example, user applications can also generate events (build your own custom event sources). User applications such as client, mobile, or web applications can publish events and invoke Lambda functions using the AWS SDKs or AWS Mobile SDKs such as the AWS Mobile SDK for Android.

For more information, see Tools for Amazon Web Services. For an example tutorial, see Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS) (p. 236).

Sample Events Published by Event Sources

The following is a list of example events published by the supported AWS services. For more information about the supported AWS event sources, see Supported Event Sources (p. 124).

Sample Events

- AWS CloudFormation Create Request Sample Event (p. 131)
- Amazon SES Email Receiving Sample Event (p. 131)
- Scheduled Event Sample Event (p. 133)
- Amazon CloudWatch Logs Sample Event (p. 133)
- Amazon SNS Sample Event (p. 133)
- Amazon DynamoDB Update Sample Event (p. 134)
- Amazon Cognito Sync Trigger Sample Event (p. 136)
- Amazon Kinesis Streams Sample Event (p. 136)
- Amazon S3 Put Sample Event (p. 136)
- Amazon S3 Delete Sample Event (p. 137)
- Mobile Backend Sample Event (p. 138)

- Amazon Lex Sample Event (p. 138)

AWS CloudFormation Create Request Sample Event

```
{
  "StackId": stackidarn,
  "ResponseURL": "http://pre-signed-S3-url-for-response",
  "ResourceProperties": {
    "StackName": "stack-name",
    "List": [
      "1",
      "2",
      "3"
    ]
  },
  "RequestType": "Create",
  "ResourceType": "Custom::TestResource",
  "RequestId": "unique id for this create request",
  "LogicalResourceId": "MyTestResource"
}
```

Amazon SES Email Receiving Sample Event

```
     "Records": [
     {
       "eventVersion": "1.0",
       "ses": {
       "mail": {
       "commonHeaders": {
       "from": [
       "Jane Doe <janedoe@example.com>"
       ],
       "to": [
       "johndoe@example.com"
       ],
      "returnPath": "janedoe@example.com",
      "messageId": "<0123456789example.com>",
      "date": "Wed, 7 Oct 2015 12:34:56 -0700",
      "subject": "Test Subject"
      },
      "source": "janedoe@example.com",
      "timestamp": "1970-01-01T00:00:00.000Z",
      "destination": [
      "johndoe@example.com"
       ],
       "headers": [
       {
        "name": "Return-Path",
        "value": "<janedoe@example.com>"
       },
       {
         "name": "Received",
         "value": "from mailer.example.com (mailer.example.com
 [203.0.113.1]) by inbound-smtp.us-west-2.amazonaws.com with SMTP id
 o3vrnil0e2ic28trm7dfhrc2v0cnbeccl4nbp0g1x for johndoe@example.com; Wed,
 07 Oct 2015 12:34:56 +0000 (UTC)"
       },
       {
        "name": "DKIM-Signature",
        "value": "v=1; a=rsa-sha256; c=relaxed/relaxed; d=example.com;
 s=example; h=mime-version:from:date:message-id:subject:to:content-type;
 bh=jX3F0bCAI7sIbkHyy3mLYO28ieDQz2R0P8HwQkklFj4x=; b=sQwJ+LMe9RjkesGu
+vqU56asvMhrLRRYrWCbVt6WJulueecwfEwRf9JVWgkBTKiL6m2hr70xDbPWDhtLdLO
+jB3hzjVnXwK3pYIOHw3vxG6NtJ6o61XSUwjEsp9tdyxQjZf2HNYee873832l3K1EeSXKzxYk9Pwqcpi3dMC74ct
+k6khpurTQQ4sp4PZPRlgHtnj3Zzv7nmpTo7dtPG5z5S9J+L+Ba7dixT0jn3HuhaJ9b
+VThboo4YfsX9PMNhWWxGjVksSFOcGluPO7QutCPyoY4gbxtwkN9W69HA=="
       },
       {
        "name": "MIME-Version",
        "value": "1.0"
       },
       {
        "name": "From",
        "value": "Jane Doe <janedoe@example.com>"
       },
       {
         "name": "Date",
         "value": "Wed, 7 Oct 2015 12:34:56 -0700"
       },
       {
         "name": "Message-ID",
         "value": "<0123456789example.com>"
      },
      {
         "name": "Subject",
         "value": "Test Subject"
      },
      {
        "name": "To",
        value": "johndoe@example.com"
      },
      {
        "name": "Content-Type",
        "value": "text/plain; charset=UTF-8"
      }
      ],
        "headersTruncated": false,
        "messageId": "o3vrnil0e2ic28trm7dfhrc2v0clambda4nbp0g1x"
      },
        "receipt": {
        "recipients": [
        "johndoe@example.com"
        ],
       "timestamp": "1970-01-01T00:00:00.000Z",
       "spamVerdict": {
       "status": "PASS"
      },
      "dkimVerdict": {
      "status": "PASS"
      },
      "processingTimeMillis": 574,
      "action": {
      "type": "Lambda",
      "invocationType": "Event",
      "functionArn": functionarn
      },
      "spfVerdict": {
      "status": "PASS"
      },
      "virusVerdict": {
      "status": "PASS"
      }
      }
      },
      "eventSource": "aws:ses"
      }
      ]
      }]}
```

Scheduled Event Sample Event

```
{
  "account": "123456789012",
  "region": "us-east-1",
  "detail": {},
  "detail-type": "Scheduled Event",
  "source": "aws.events",
  "time": "1970-01-01T00:00:00Z",
  "id": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",
  "resources": [
    "arn:aws:events:us-east-1:123456789012:rule/my-schedule"
  ]
}
```

Amazon CloudWatch Logs Sample Event

```
{
 "awslogs": {
 "data": "H4sIAAAAAAAAAHWPwQqCQBCGX0Xm7EFtK
+smZBEUgXoLCdMhFtKV3akI8d0bLYmibvPPN3wz00CJxmQnTO41whwWQRIctmEcB6sQbFC3CjW3XW8kxpOpP
+OC22d1Wml1qZkQGtoMsScxaczKN3plG8zlaHIta5KqWsozoTYw3/
djzwhpLwivWFGHGpAFe7DL68JlBUk+l7KSN7tCOEJ4M3/qOI49vMHj
+zCKdlFqLaU2ZHV2a4Ct/an0/ivdX8oYc1UVX860fQDQiMdxRQEAAA=="
 }
 }
```

Amazon SNS Sample Event

```
{
  "Records": [
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": eventsubscriptionarn,
      "EventSource": "aws:sns",
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "1970-01-01T00:00:00.000Z",
        "Signature": "EXAMPLE",
        "SigningCertUrl": "EXAMPLE",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
        "Type": "Notification",
        "UnsubscribeUrl": "EXAMPLE",
        "TopicArn": topicarn,
        "Subject": "TestInvoke"
      }
    }
  ]
}
```

Amazon DynamoDB Update Sample Event

```
    {
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": eventsourcearn,
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": sourcearn,
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "3",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 38,
        "SequenceNumber": "333",
        "OldImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "REMOVE",
      "eventSourceARN": sourcearn,
      "eventSource": "aws:dynamodb"
    }
  ]
}
```

Amazon Cognito Sync Trigger Sample Event

```
   {
  "datasetName": "datasetName",
  "eventType": "SyncTrigger",
  "region": "us-east-1",
  "identityId": "identityId",
  "datasetRecords": {
    "SampleKey2": {
      "newValue": "newValue2",
      "oldValue": "oldValue2",
      "op": "replace"
    },
    "SampleKey1": {
      "newValue": "newValue1",
      "oldValue": "oldValue1",
      "op": "replace"
    }
  },
  "identityPoolId": "identityPoolId",
  "version": 2
}
```

Amazon Kinesis Streams Sample Event

```
  "Records": [
    {
      "eventID":
 "shardId-000000000000:49545115243490985018280067714973144582180062593244200961",
      "eventVersion": "1.0",
      "kinesis": {
        "partitionKey": "partitionKey-3",
        "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0IDEyMy4=",
        "kinesisSchemaVersion": "1.0",
        "sequenceNumber":
 "49545115243490985018280067714973144582180062593244200961"
      },
      "invokeIdentityArn": identityarn,
      "eventName": "aws:kinesis:record",
      "eventSourceARN": eventsourcearn,
      "eventSource": "aws:kinesis",
      "awsRegion": "us-east-1"
    }
  ]
}
```

Amazon S3 Put Sample Event

```
  "Records": [
    {
      "eventVersion": "2.0",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "s3": {
        "configurationId": "testConfigRule",
        "object": {
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901",
          "key": "HappyFace.jpg",
          "size": 1024
        },
        "bucket": {
          "arn": bucketarn,
          "name": "sourcebucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          }
        },
        "s3SchemaVersion": "1.0"
      },
      "responseElements": {
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH",
        "x-amz-request-id": "EXAMPLE123456789"
      },
      "awsRegion": "us-east-1",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "eventSource": "aws:s3"
    }
  ]
}
```

Amazon S3 Delete Sample Event

```
  {
  "Records": [
    {
      "eventVersion": "2.0",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "s3": {
        "configurationId": "testConfigRule",
        "object": {
          "sequencer": "0A1B2C3D4E5F678901",
          "key": "HappyFace.jpg"
        },
        "bucket": {
          "arn": bucketarn,
          "name": "sourcebucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          }
        },
        "s3SchemaVersion": "1.0"
      },
      "responseElements": {
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH",
        "x-amz-request-id": "EXAMPLE123456789"
      },
      "awsRegion": "us-east-1",
      "eventName": "ObjectRemoved:Delete",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "eventSource": "aws:s3"
    }
  ]
}
```

Mobile Backend Sample Event

```
 {
  "operation": "echo",
  "message": "Hello world!"
}
```

Amazon Lex Sample Event

```
{
  "messageVersion": "1.0",
  "invocationSource": "FulfillmentCodeHook or DialogCodeHook",
  "userId": "user-id specified in the POST request to Amazon Lex.",
  "sessionAttributes": {
     "key1": "value1",
     "key2": "value2",
  },
  "bot": {
    "name": "bot-name",
    "alias": "bot-alias",
    "version": "bot-version"
  },
  "outputDialogMode": "Text or Voice, based on ContentType request header
 in runtime API request",
  "currentIntent": {
    "name": "intent-name",
    "slots": {
      "slot-name": "value",
      "slot-name": "value",
      "slot-name": "value"
    },
    "confirmationStatus": "None, Confirmed, or Denied
      (intent confirmation, if configured)"
    }
}
```

Deploying Lambda-based Applications

Lambda-based applications (also referred to as *serverless applications*) are composed of functions triggered by events. A typical serverless application consists of one or more functions triggered by events such as object uploads to Amazon S3, Amazon SNS notifications, and API actions. Those functions can stand alone or leverage other resources such as DynamoDB tables or Amazon S3 buckets. The most basic serverless application is simply a function.

AWS Lambda provides API operations that you can use to create and update Lambda functions by providing a deployment package as a ZIP file. However, this mechanism might not be convenient for automating deployment steps for functions, or coordinating deployments and updates to other elements of a serverless application (like event sources and downstream resources). For example, in order to deploy an Amazon SNS trigger, you need to update the function, the Amazon SNS topic, the mapping between the function and the topic, and any other downstream resources required by your function such as a DynamoDB table.

Deploying Serverless Applications Using AWS CloudFormation

You can use AWS CloudFormation to specify, deploy, and configure serverless applications. AWS CloudFormation is a service that helps you model and set up your AWS resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all of the AWS resources that you want (like Lambda functions and DynamoDB tables), and AWS CloudFormation takes care of provisioning and configuring those resources for you. You don't need to individually create and configure AWS resources and figure out what's dependent on what—AWS CloudFormation handles all of that. For more information, see AWS CloudFormation Concepts in the *AWS CloudFormation User Guide*.

Using the AWS Serverless Application Model (AWS SAM)

The AWS Serverless Application Model (AWS SAM) is a model to define serverless applications. AWS SAM is natively supported by AWS CloudFormation and defines simplified syntax for expressing serverless resources. The specification currently covers APIs, Lambda functions and Amazon DynamoDB tables. The specification is available under Apache 2.0 for AWS partners and customers to adopt and extend within their own toolsets. For details on the specification, see the AWS Serverless Application Model.

Serverless Resources Within AWS CloudFormation

AWS SAM supports special resource types that simplify how to express functions, APIs, mappings, and DynamoDB tables for serverless applications, as well as some features for these services like environment variables. The AWS CloudFormation description of these resources conforms to the AWS Serverless Application Model. In order to deploy your application, simply specify the resources you need as part of your application, along with their associated permissions policies in an AWS CloudFormation template file (written in either JSON or YAML), package your deployment artifacts, and deploy the template.

An AWS CloudFormation template with serverless resources conforming to the AWS SAM model is referred to as a SAM file or template.

The examples below illustrate how to leverage AWS SAM to declare common components of a serverless application:

Lambda function

The following shows the notation you use to describe a Lambda function:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:

    FunctionName:
        Type: AWS::Serverless::Function
        Properties:
           Handler: index.handler
           Runtime: nodejs4.3
           CodeUri: s3://bucketName/codepackage.zip
```

The `handler` value of the `Handler` property points to the module containing the code your Lambda function will execute when invoked. The `index` value of the `Handler` property indicates the name of the file containing the code. You can declare as many functions as your serverless application requires.

You can also declare environment variables, which are configuration settings you can set for your application. The following shows an example of a serverless app with two Lambda functions and an environment variable that points to a DynamoDB table. You can update environment variables without needing to modify, repackage, or redeploy your Lambda function code. For more information, see Environment Variables (p. 89).

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  PutFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      Policies: AWSLambdaDynamoDBExecutionRole
      CodeUri: s3://bucketName/codepackage.zip
      Environment:
        Variables:
          TABLE_NAME: !Ref Table
  DeleteFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      Policies: AWSLambdaDynamoDBExecutionRole
      CodeUri: s3://bucketName/codepackage.zip
      Environment:
        Variables:
          TABLE_NAME: !Ref Table
      Events:
        Stream:
          Type: DynamoDB
          Properties:
            Stream: !GetAtt DynamoDBTable.StreamArn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON

   DynamoDBTable:
     Type: AWS::DynamoDB::Table
     Properties:
       AttributeDefinitions:
         - AttributeName: id
           AttributeType: S
       KeySchema:
         - AttributeName: id
           KeyType: HASH
       ProvisionedThroughput:
         ReadCapacityUnits: 5
         WriteCapacityUnits: 5
       StreamSpecification:
         StreamViewType: streamview type
```

Note the notation at the top:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
```

This is required in order to include objects defined by the AWS Serverless Application Model within an AWS CloudFormation template.

SimpleTable

`SimpleTable` is a resource that creates a DynamoDB table with a single-attribute primary key. You can use this simplified version if the data your serverless application is interacting with only needs to be accessed by a single-valued key. You could update the previous example to use a `SimpleTable`, as shown following:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  TableName:
     Type: AWS::Serverless::SimpleTable
     Properties:
       PrimaryKey:
         Name: id
         Type: String
       ProvisionedThroughput:
         ReadCapacityUnits: 5
         WriteCapacityUnits: 5
```

Events

Events are AWS resources that trigger the Lambda function, such as an Amazon API Gateway endpoint or an Amazon SNS notification. The `Events` property is an array, which allows you to set multiple events per function. The following shows the notation you use to describe a Lambda function with a DynamoDB table as an event source:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  FunctionName:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      Events:
        Stream:
          Type: DynamoDB
          Properties:
            Stream: !GetAtt DynamoDBTable.StreamArn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON
  TableName:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
```

As mentioned above, you can set multiple event sources that will trigger the Lambda function. The example below shows a Lambda function that can be triggered by either an HTTP `PUT` or `POST` event.

API

There are two ways to define an `API` using AWS SAM. The following uses Swagger to configure the underlying Amazon API Gateway resources:

```

AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  Api:
    Type: AWS::Serverless::Api
    Properties:
      StageName: prod
      DefinitionUri: swagger.yml
```

In the next example, the `AWS::Serverless::Api` resource type is implicity added from the union of `API` events defined on `AWS::Serverless::Function` resources.

```

AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  GetFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.get
      Runtime: nodejs4.3
      CodeUri: s3://bucket/api_backend.zip
      Policies: AmazonDynamoDBReadOnlyAccess
      Environment:
        Variables:
          TABLE_NAME: !Ref Table
      Events:
        GetResource:
          Type: Api
          Properties:
            Path: /resource/{resourceId}
            Method: get

  PutFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.put
      Runtime: nodejs4.3
      CodeUri: s3://bucket/api_backend.zip
      Policies: AmazonDynamoDBFullAccess
      Environment:
        Variables:
          TABLE_NAME: !Ref Table
      Events:
        PutResource:
          Type: Api
          Properties:
            Path: /resource/{resourceId}
            Method: put

  DeleteFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.delete
      Runtime: nodejs4.3
      CodeUri: s3://bucket/api_backend.zip
      Policies: AmazonDynamoDBFullAccess
      Environment:
        Variables:
          TABLE_NAME: !Ref Table
      Events:
        DeleteResource:
          Type: Api
          Properties:
            Path: /resource/{resourceId}
            Method: delete

  Table:
    Type: AWS::Serverless::SimpleTable
```

In the example above, AWS CloudFormation will automatically generate an Amazon API Gateway `API` with the path `"/resource/{resourceId}"` and with the methods `GET`, `PUT` and `DELETE`.

Permissions

You can supply an IAM role ARN to be used as this function's execution role, as shown below:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  FunctionName:
    Type: AWS::Serverless::Function
    Properties:
      Role:role arn
```

Alternatively, you could supply one or more managed policies to the Lambda function resource. AWS CloudFormation will then create a new role with the managed policies plus the default Lambda basic execution policy.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  FunctionName:
    Type: AWS::Serverless::Function
    Properties:
      Policies: AmazonDynamoDBFullAccess
```

If none of these are supplied, a default execution role is created with Lambda basic execution permissions.

Note
In addition to using the serverless resources, you can also use conventional CloudFormation syntax for expressing resources in the same template. Any resources not included in the current SAM model can still be created in the AWS CloudFormation template using AWS CloudFormation syntax. In addition, you can use AWS CloudFormation syntax to express serverless resources as an alternative to using the SAM model. For information about specifying a Lambda function using conventional CloudFormation syntax as part of your SAM template, see AWS::Lambda::Function in the AWS CloudFormation User Guide.

For a list of complete serverless application examples, see Examples of How to Use AWS Lambda (p. 175).

Next Step

Create Your Own Serverless Application (p. 144)

Create Your Own Serverless Application

In the following tutorial, you create a simple serverless application that consists of a single function that returns the name of an Amazon S3 bucket you specify as an environment variable. Follow these steps:

1. Copy and paste the following into a text file and save it as `index.js`

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


exports.handler = function(event, context, callback) {
  var bucketName = process.env.S3_BUCKET;
  callback(null, bucketName);
}
```

2. Paste the following into a text file and save it as `example.yaml`:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  TestFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      Environment:
        Variables:
          S3_BUCKET: bucket-name
```

3. Create a folder called *examplefolder* and place the `example.yaml` file and the `index.js` file inside the folder.

   Your *example* folder now contains the following two files that you can then use to package the serverless application:

   - `example.yaml`
   - `index.js`

Packaging and Deployment

After you create your Lambda function handler and your *example.yaml* file, you can use the AWS CLI to package and deploy your serverless application.

Packaging

To package your application, create an Amazon S3 bucket that the `package` command will use to upload your ZIP deployment package (if you haven't specified one in your *example.yaml* file). You can use the following command to create the Amazon S3 bucket:

```
aws s3 mb s3://bucket-name --region region
```

Next, open a command prompt and type the following:

```
aws cloudformation package \
   --template-file example.yaml \
   --output-template-file serverless-output.yaml \
   --s3-bucket s3-bucket-name
```

The package command returns an AWS SAM template, in this case `serverless-output.yaml` that contains the `CodeUri` that points to the deployment zip in the Amazon S3 bucket that you specified. This template represents your serverless application. You are now ready to deploy it.

Deployment

To deploy the application, run the following command:

```
aws cloudformation deploy \
  --template-file serverless-output.yaml \
  --stack-name new-stack-name \
  --capabilities CAPABILITY_IAM
```

Note that the value you specify for the `--template-file` parameter is the name of the SAM template that was returned by the package command. In addition, the `--capabilities` parameter is optional. The `AWS::Serverless::Function` resource will implicitly create a role to execute the Lambda function if one is not specified in the template. You use the `--capabilities` parameter to explicitly acknowledge that AWS CloudFormation is allowed to create roles on your behalf.

When you run the `aws cloudformation deploy` command, it creates an AWS CloudFormation `ChangeSet`, which is a list of changes to the AWS CloudFormation stack, and then deploys it. Some stack templates might include resources that can affect permissions in your AWS account, for example, by creating new AWS Identity and Access Management (IAM) users. For those stacks, you must explicitly acknowledge their capabilities by specifying the `--capabilities` parameter. For more information, see CreateChangeSet in the *AWS CloudFormation API Reference*.

To verify your results, open the AWS CloudFormation console to view the newly created AWS CloudFormation stack and the Lambda console to view your function.

For a list of complete serverless application examples, see Examples of How to Use AWS Lambda (p. 175).

Exporting a Serverless Application

You can export a serverless application and re-deploy it to, for example, a different AWS region or development stage, using the Lambda console. When you export a Lambda function, you will be provided with a ZIP deployment package and a SAM template that represents your serverless application. You can then use the `package` and `deploy` commands described in the previous section for re-deployment.

You can also select one of Lambda blueprints to create a ZIP package for you to package and deploy. Follow the steps below to do this:

To export a serverless application using the Lambda console

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Do any of the following:
   - **Create a function using a Lambda blueprint** – Choose a blueprint and follow the steps to create a Lambda function. For an example, see Step 2.1: Create a Hello World Lambda Function (p. 164). When you reach the **Review** page, choose **Export function**.
   - **Create a function** – Choose **Create function**, and then create your function. After your Lambda function is created, you can export it by selecting the function. Choose **Actions**, then choose **Export function**.
   - **Open an existing Lambda function** – Open the function by choosing the **Function name**, choose **Actions**, choose **Export function**.
3. In the **Export your function** window, you have the following options:
   - Choose **Download AWS SAM file**, which defines the Lambda function and other resources that comprise your serverless application.
   - Choose **Download deployment package**, which contains your Lambda function code and any dependent libraries.

Use the AWS SAM file and the ZIP deployment package and follow the steps in Packaging and Deployment (p. 145) to re-deploy the serverless application.

Automating Deployment of Lambda-based Applications

In the previous section, you learned how to create a SAM template, generate your deployment package, and use the AWS CLI to manually deploy your serverless application. In this section, you will leverage the following AWS services to fully automate the deployment process.

- **CodePipeline**: You use CodePipeline to model, visualize, and automate the steps required to release your serverless application. For more information, see What is AWS CodePipeline?
- **CodeBuild**: You use CodeBuild to build, locally test, and package your serverless application. For more information, see What is AWS CodeBuild?
- **AWS CloudFormation**: You use AWS CloudFormation to deploy your application. For more information, see What is AWS CloudFormation?

Building a Pipeline for Your Serverless Application

In the following tutorial, you will create an AWS CodePipeline that automates the deployment of your serverless application. First, you will need to set up a **source stage** to trigger your pipeline. For the purposes of this tutorial:

- We will use GitHub. For instructions on how to create a GitHub repository, see Create a Repository in GitHub.
- You will need to create an AWS CloudFormation role and add the **AWSLambdaBasicExecutionRole** policy to that role, as outlined below:

  1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
  2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
     - In **Role Name**, use a name that is unique within your AWS account (for example, **cloudformation-lambda-execution-role**).
     - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS CloudFormation**. This grants the AWS CloudFormation service permissions to assume the role.
     - In **Attach Policy**, choose **AWSLambdaBasicExecutionRole**. The permissions in this policy are sufficient for the Lambda function in this tutorial.

Step 1: Set Up Your Repository

To set up your repository, do the following:

- Add an *index.js file* containing the code below:

```
var time = require('time');
exports.handler = (event, context, callback) => {
    var currentTime = new time.Date();
    currentTime.setTimezone("America/Los_Angeles");
    callback(null, {
        statusCode: '200',
        body: 'The time in Los Angeles is: ' + currentTime.toString(),
    });
};
```

- Add a *SAM_template.yaml* file, containing the content below. This is the SAM template that defines the resources in your application. This SAM template defines a Lambda function that is triggered by API Gateway. For more information about AWS SAM see AWS Serverless Application Model.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Outputs the time
Resources:
  TimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      Events:
        MyTimeApi:
          Type: Api
          Properties:
            Path: /TimeResource
            Method: GET
```

- Add a *buildspec.yaml* file. A build spec is a collection of build commands and related settings, in YAML format, that AWS CodeBuild uses to run a build. For more information, see Build Specification Reference for AWS CodeBuild. In this example, the build action will be:
  - Use npm to install the time package.
  - Running the `Package` command to prepare your deployment package for subsequent deployment steps in your pipeline. For more information on the package command, see Uploading Local Artifacts to an S3 Bucket

  ```
  version: 0.1
  phases:
    install:
      commands:
        - npm install time
        - aws cloudformation package
           --template-file samTemplate.yaml \
           --s3-bucket bucket-name \
           --output-template-file NewSamTemplate.yaml
  artifacts:
    type: zip
    files:
      - NewSamTemplate.yaml
  ```

  Note that you need to supply the `--s3-bucket` parameter value with the name of the your Amazon S3 bucket, similar to the step you would take if you were manually going to package the deployment package with SAM, as discussed in the Packaging (p. 145) step of the previous tutorial.

Step 2: Create Your Pipeline

Follow the steps below to create your AWS CodePipeline.

1. Sign in to the AWS Management Console and open the AWS CodePipeline console.
2. Choose **Get Started Now**.
3. In **Pipeline name:** enter a name for your pipeline and then choose **Next step**.
4. In **Source provider:** choose **GitHub**.
5. Choose **Connect to GitHub:** and then choose the **Repository** and **Branch** you want to connect to. Every git push to the branch you select will trigger your pipeline. Choose **Next step**.
6. Choose **AWS CodeBuild** as your **Build provider**.
7. Choose **Create a new build project** and enter a project name.
8. Choose **Ubuntu** as the operating system.
9. Choose **Node.js** as the runtime.
10. In **Version** choose `aws/codebuild/nodejs:4.3.2`
11. Choose **Save build project**.

    Note
    A service role for AWS CodeBuild will automatically be created on your behalf.

    Choose **Next step**.
12. In **Deployment provider:** choose **AWS CloudFormation**.

    By selecting this option, AWS CloudFormation commands will be used to deploy the SAM template. For more information see Serverless Resources Within AWS CloudFormation (p. 139).
13. In **Action mode:** choose **create or replace a change set**.
14. In **Stack name:** enter **MyBetaStack**.
15. In **Change set name:** enter **MyChangeSet**.
16. In **Template file:** enter **NewSamTemplate.yaml**.
17. In **Capabilities:** choose **CAPABILITY_IAM**.
18. In **Role** select the AWS CloudFormation role you created at the beginning of this tutorial and then choose **Next step**.
19. Choose **Create role**. Choose **Next** and then choose **Allow.** Choose **Next step**.
20. Review your pipeline and then choose **Create pipeline**.

Step 3: Complete Your Beta Deployment Stage

Follow the steps below to complete your Beta stage.

1. Choose **Edit**.
2. Choose the

   ✏

   icon for your beta stage.
3. In the beta stage, choose the

   ✏

   icon that is located below your existing action.
4. In **Category:** choose **Deploy**.
5. In **Action:** enter **execute_cs**.
6. In **Deployment provider:** choose **AWS CloudFormation**.
7. In **Action mode:** choose **execute a changeset**. This is similar to the step you would take if you were manually going to deploy the package, as discussed in the Deployment (p. 145) step of the previous tutorial. `CreateChangeSet` transforms the SAM template to the full AWS CloudFormation format and `deployChangeSet` deploys the AWS CloudFormation template.

8. In **Stack name:** enter **MyBetaStack**.
9. In **Change set name:** enter **MyChangeSet**.
10. Choose **Add action**.
11. Choose **Save pipeline changes**.
12. Choose **Save and continue**.

Your pipeline is ready. Any git push to the branch you connected to this pipeline is going to trigger a deployment. To test your pipeline and deploy your application for the first time, do one of the following:

- Perform a git push to the branch connected to your pipeline.
- Go the AWS CodePipeline console, choose the name of the pipeline you created and then choose **Release change**.

AWS Lambda: How It Works

How Does AWS Lambda Run My Code? The Container Model

When AWS Lambda executes your Lambda function on your behalf, it takes care of provisioning and managing resources needed to run your Lambda function. When you create a Lambda function, you specify configuration information, such as the amount of memory and maximum execution time that you want to allow for your Lambda function. When a Lambda function is invoked, AWS Lambda launches a container (that is, an execution environment) based on the configuration settings you provided.

> **Note**
> The content of this section is for information only. AWS Lambda manages container creations and deletion, there is no AWS Lambda API for you to manage containers.

It takes time to set up a container and do the necessary bootstrapping, which adds some latency each time the Lambda function is invoked. You typically see this latency when a Lambda function is invoked for the first time or after it has been updated because AWS Lambda tries to reuse the container for subsequent invocations of the Lambda function.

After a Lambda function is executed, AWS Lambda maintains the container for some time in anticipation of another Lambda function invocation. In effect, the service freezes the container after a Lambda function completes, and thaws the container for reuse, if AWS Lambda chooses to reuse the container when the Lambda function is invoked again. This container reuse approach has the following implications:

- Any declarations in your Lambda function code (outside the `handler` code, see Programming Model (p. 8)) remains initialized, providing additional optimization when the function is invoked again. For example, if your Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations. You can add logic in your code to check if a connection already exists before creating one.

- Each container provides some disk space in the `/tmp` directory. The directory content remains when the container is frozen, providing transient cache that can be used for multiple invocations. You can add extra code to check if the cache has the data that you stored. For disk space size, see AWS Lambda Limits (p. 285).

- Background processes or callbacks initiated by your Lambda function that did not complete when the function ended resume if AWS Lambda chooses to reuse the container. You should make sure any background processes or callbacks (in case of Node.js) in your code are complete before the code exits.

Note
When you write your Lambda function code, do not assume that AWS Lambda always reuses the container because AWS Lambda may choose not to reuse the container. Depending on various other factors, AWS Lambda may simply create a new container instead of reusing an existing container.

Lambda Function Concurrent Executions

Concurrent executions refers to the number of executions of your function code that are happening at any given time. You can estimate the concurrent execution count, but the concurrent execution count will differ depending on whether or not your Lambda function is processing events from a stream-based event source.

- **Stream-based event sources** – If you create a Lambda function that processes events from stream-based services (Amazon Kinesis Streams or DynamoDB streams), the number of shards per stream is the unit of concurrency. If your stream has 100 active shards, there will be 100 Lambda functions running concurrently. Then, each Lambda function processes events on a shard in the order that they arrive.
- **Event sources that aren't stream-based** – If you create a Lambda function to process events from event sources that aren't stream-based (for example, Amazon S3 or API Gateway), each published event is a unit of work. Therefore, the number of events (or requests) these event sources publish influences the concurrency.

  You can use the following formula to estimate your concurrent Lambda function invocations:

  ```
  events (or requests) per second * function duration
  ```

  For example, consider a Lambda function that processes Amazon S3 events. Suppose that the Lambda function takes on average three seconds and Amazon S3 publishes 10 events per second. Then, you will have 30 concurrent executions of your Lambda function.

Request Rate

Request rate refers to the rate at which your Lambda function is invoked. For all services except the stream-based services, the request rate is the rate at which the event sources generate the events. For stream-based services, AWS Lambda calculates the request rate as follow:

```
request rate = number of concurrent executions / function duration
```

For example, if there are five active shards on a stream (that is, you have five Lambda functions running in parallel) and your Lambda function takes about two seconds, the request rate is 2.5 requests/second.

Safety Limit

By default, AWS Lambda limits the total concurrent executions across all functions within a given region to 100. The default limit is a safety limit that protects you from costs due to potential runaway or recursive functions during initial development and testing. To increase this limit above the default, follow the steps in To request a limit increase for concurrent executions (p. 153).

Any invocation that causes your function's concurrent execution to exceed the safety limit is throttled, and does not execute your function. Each throttled invocation increases the CloudWatch `Throttles` metric for the function.

The throttled invocation is handled differently based on how the function is invoked:

- **Event sources that aren't stream-based** – Some of these event sources invoke a Lambda function synchronously and others invoke it asynchronously.

  - **Synchronous invocation** – If the function is invoked synchronously and is throttled, the invoking application receives a 429 error and the invoking application is responsible for retries. These event sources may have additional retries built into the integration. For example, CloudWatch Logs retries the failed batch up to five times with delays between retries. For a list of supported event sources and the invocation types that they use, see Supported Event Sources (p. 124).

    If you invoke Lambda through API Gateway, you need to make sure you map Lambda response errors to API Gateway error codes. If you invoke the function directly, such as through the AWS SDKs using the `RequestResponse` invocation mode or through API Gateway, your client receives the 429 error and you can choose to retry the invocation.

  - **Asynchronous invocation** – If your Lambda function is invoked asynchronously and is throttled, AWS Lambda automatically retries the throttled event for up to six hours, with delays between retries. Asynchronous events are queued before they are used to invoke the Lambda function.

- **Stream-based event sources** – For stream-based event sources (Amazon Kinesis Streams and DynamoDB streams), AWS Lambda polls your stream and invokes your Lambda function. Therefore, when your Lambda function is throttled, AWS Lambda attempts to process the throttled batch of records until the time the data expires, which can be up to seven days for Amazon Kinesis Streams. The throttled request is treated as blocking per shard and Lambda will not read any new records from the shard until the throttled batch of records either expires or succeeds. If there is more than one shard in the stream, Lambda will continue invokes on the non-throttled shards until one gets through.

To request a limit increase for concurrent executions

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 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.

Suggested Reading

If you are new to AWS Lambda, we suggest you read through all of the topics in the How It Works section to familiarize yourself with Lambda. The next topic is Retries on Errors (p. 154).

After you read all of the topics in the How it Works section, we recommend that you review Building Lambda Functions (p. 5), try the Getting Started (p. 160) exercise, and then explore the Use Cases (p. 175). Each use case provides step-by-step instructions for you to set up the end-to-end experience.

Retries on Errors

A Lambda function can fail for any of the following reasons:

- The function times out while trying to reach an endpoint.
- The function fails to successfully parse input data.
- The function experiences resource constraints, such as out-of-memory errors or other timeouts.

If any of these failures occur, your function will throw an exception. How the exception is handled depends upon how the Lambda function was invoked:

- **Event sources that aren't stream-based** – Some of these event sources are set up to invoke a Lambda function synchronously and others invoke it asynchronously. Accordingly, exceptions are handled as follows:
  - **Synchronous invocation** – The invoking application receives a 429 error, and is responsible for retries. For a list of supported event sources and the invocation types they use, see Supported Event Sources. These event sources may have additional retries built into the integration.

    If you invoked the Lambda function directly through AWS SDKs, or through API Gateway, your client receives the error and can choose to retry. If you are invoking Lambda through API Gateway, you need to make sure you map Lambda response errors to API Gateway error codes.
  - **Asynchronous invocation** – Asynchronous events are queued before being used to invoke the Lambda function. If AWS Lambda is unable to fully process the event, it will automatically retry the invocation twice, with delays between retries. If you have specified a Dead Letter Queue for your function, then the failed event is sent to the specified Amazon SQS queue or Amazon SNS topic. If you don't specify a Dead Letter Queue (DLQ), which is not required and is the default setting, then the event will be discarded. For more information, see Dead Letter Queues (p. 113).
- **Stream-based event sources** – For stream-based event sources (Amazon Kinesis Streams and DynamoDB streams), AWS Lambda polls your stream and invokes your Lambda function. Therefore, if a Lambda function fails, AWS Lambda attempts to process the erring batch of records until the time the data expires, which can be up to seven days for Amazon Kinesis Streams. The exception is treated as blocking, and AWS Lambda will not read any new records from the stream until the failed batch of records either expires or processed successfully. This ensures that AWS Lambda processes the stream events in order.

For more information about invocation modes, see Event Source Mapping (p. 120).

Suggested Reading

If you are new to AWS Lambda, we suggest you read through all of the topics in the How It Works section to familiarize yourself with Lambda. The next topic is AWS Lambda Permissions Model (p. 155).

After you read all of the topics in the How it Works section, we recommend that you review Building Lambda Functions (p. 5), try the Getting Started (p. 160) exercise, and then explore the Use Cases (p. 175). Each use case provides step-by-step instructions for you to set up the end-to-end experience.

AWS Lambda Permissions Model

For the end-to-end AWS Lambda-based applications to work, you have to manage various permissions. For example:

- For event sources, except for the stream-based services (Amazon Kinesis Streams and DynamoDB streams), you must grant the event source permissions to invoke your AWS Lambda function.
- For stream-based event sources (Amazon Kinesis Streams and DynamoDB streams), AWS Lambda polls the streams on your behalf and reads new records on the stream, so you need to grant AWS Lambda permissions for the relevant stream actions.
- When your Lambda function executes, it can access AWS resources in your account (for example, read an object from your S3 bucket). AWS Lambda executes your Lambda function on your behalf by assuming the role you provided at the time of creating the Lambda function. Therefore, you need to grant the role the necessary permissions that your Lambda function needs, such as permissions for Amazon S3 actions to read an object.

The following sections describe permissions management.

Topics

Manage Permissions: Using an IAM Role (Execution Role)

Each Lambda function has an IAM role (execution role) associated with it. You specify the IAM role when you create your Lambda function. Permissions you grant to this role determine what AWS Lambda can do when it assumes the role. There are two types of permissions that you grant to the IAM role:

- If your Lambda function code accesses other AWS resources, such as to read an object from an S3 bucket or write logs to CloudWatch Logs, you need to grant permissions for relevant Amazon S3 and CloudWatch actions to the role.

- If the event source is stream-based (Amazon Kinesis Streams and DynamoDB streams), AWS Lambda polls these streams on your behalf. AWS Lambda needs permissions to poll the stream and read new records on the stream so you need to grant the relevant permissions to this role.

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

Important
The user that creates the IAM role is, in effect, passing permissions to AWS Lambda to assume this role, which requires the user to have permissions for the `iam:PassRole` action. If an administrator user is creating this role, you don't need to do anything extra to set up permissions for the `iam:PassRole` action because the administrator user has full permissions, including the `iam:PassRole` action.

To simplify the process for creating an execution role, AWS Lambda provides the following AWS managed (predefined) permissions policies that you can use. These policies include common permissions for specific scenarios:

- **AWSLambdaBasicExecutionRole** – Grants permissions only for the Amazon CloudWatch Logs actions to write logs. You can use this policy if your Lambda function does not access any other AWS resources except writing logs.

- **AWSLambdaKinesisExecutionRole** – Grants permissions for Amazon Kinesis Streams actions, and CloudWatch Logs actions. If you are writing a Lambda function to process Amazon Kinesis stream events you can attach this permissions policy.

- **AWSLambdaDynamoDBExecutionRole** – Grants permissions for DynamoDB streams actions and CloudWatch Logs actions. If you are writing a Lambda function to process DynamoDB stream events you can attach this permissions policy.

- **AWSLambdaVPCAccessExecutionRole** – Grants permissions for Amazon Elastic Compute Cloud (Amazon EC2) actions to manage elastic network interfaces (ENIs). If you are writing a Lambda function to access resources in a VPC in the Amazon Virtual Private Cloud (Amazon VPC) service, you can attach this permissions policy. The policy also grants permissions for CloudWatch Logs actions to write logs.

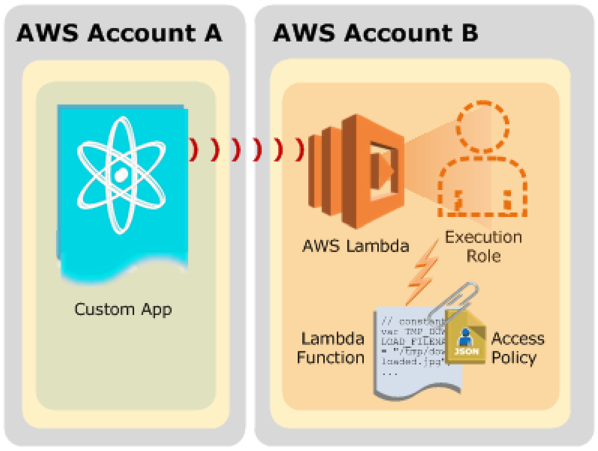
  You can find these AWS managed permissions policies in the IAM console. Search for these policies and you can see the permissions each of these policies grant.

Manage Permissions: Using a Lambda Function Policy

All supported event sources, except the stream-based services (Amazon Kinesis and DynamoDB streams), invoke your Lambda function (the *push model*), provided the you grant the necessary permissions. For example, if you want Amazon S3 to invoke your Lambda function when objects are created in a bucket, Amazon S3 needs permissions to invoke your Lambda function.

You can grant these permissions via the function policies. AWS Lambda provides APIs for you to manage permission in a function policy. For example, see AddPermission (p. 322).

You can also grant cross-account permissions using the function policy. For example, if a user-defined application and the Lambda function it invokes belong to the same AWS account, you don't need to grant explicit permissions. Otherwise, the AWS account that owns the Lambda function must allow cross-account permissions in the permissions policy associated with the Lambda function.



Note
Instead of using a Lambda function policy, you can create another IAM role that grants the event sources (for example, Amazon S3 or DynamoDB) permissions to invoke your Lambda function. However, you might find that resource policies are easier to set up and they make it easier for you to track which event sources have permissions to invoke your Lambda function.

For more information about Lambda function policies, see Using Resource-Based Policies for AWS Lambda (Lambda Function Policies) (p. 306). For more information about Lambda permissions, see Authentication and Access Control for AWS Lambda (p. 287).

Suggested Reading

If you are new to AWS Lambda, we suggest you read through all of the topics in the How It Works section to familiarize yourself with Lambda. The next topic is Lambda Execution Environment and Available Libraries (p. 157).

After you read all of the topics in the How it Works section, we recommend that you review Building Lambda Functions (p. 5), try the Getting Started (p. 160) exercise, and then explore the Use Cases (p. 175). Each use case provides step-by-step instructions for you to set up the end-to-end experience.

Lambda Execution Environment and Available Libraries

The underlying AWS Lambda execution environment is based on the following:

- Public Amazon Linux AMI version (AMI name: amzn-ami-hvm-2016.03.3.x86_64-gp2):

  For information about using an AMI, see Amazon Machine Images (AMI) in the *Amazon EC2 User Guide for Linux Instances*.
- Linux kernel version – 4.4.23-31.54.amzn1.x86_64

If you are using any native binaries in your code, make sure they are compiled in this environment. Note that only 64-bit binaries are supported on AWS Lambda.

AWS Lambda supports the following runtime versions:

- Node.js – v0.10.36, v4.3.2 (recommended)
- Java – Java 8
- Python – Python 2.7
- .NET Core – .NET Core 1.0.1 (C#)

The following libraries are available in the AWS Lambda execution environment, regardless of the supported runtime you use, so you don't need to include them:

AWS SDK – AWS SDK for JavaScript version 2.6.9

The following libraries are available in the AWS Lambda execution environment so you don't need to include them:

- AWS SDK for Python (Boto 3) version 1.4.1, Botocore version 1.4.61
- Amazon Linux build of `java-1.8.0-openjdk` for Java.

Environment Variables Available to Lambda Functions

The following is a list of environment variables that are part of the AWS Lambda execution environment and made available to Lambda functions. The table below indicates which ones are reserved by AWS Lambda and cannot be changed as well as which ones you can set when creating your Lambda function. For more information on using environment variables with your Lambda function, see Environment Variables (p. 89).

Lambda Environment Variables

Key	Reserved	Value
LAMBDA_TASK_ROOT	Yes	Contains the path to your Lambda function code.
LAMBDA_RUNTIME_DIR	Yes	Restricted to Lambda runtime-related artifacts. For example the aws-sdk for Node.js and boto3 for Python can be found under this path.
AWS_REGION	Yes	The AWS region where the Lambda function is executed.
AWS_DEFAULT_REGION	Yes	The AWS region where the Lambda function is executed.
AWS_LAMBDA_LOG_GROUP_NAME	Yes	The name of Amazon CloudWatch Logs group where log streams containing your Lambda function logs are created.
AWS_LAMBDA_LOG_STREAM_NAME	Yes	The Amazon CloudWatch Logs streams containing your Lambda function logs.
AWS_LAMBDA_FUNCTION_NAME	Yes	The name of the Lambda function.
AWS_LAMBDA_FUNCTION_MEMORY_SIZE	Yes	The size of the Lambda function in MB.

Key	Reserved	Value
AWS_LAMBDA_FUNCTION_VERSION	Yes	The version of the Lambda function.
AWS_ACCESS_KEY AWS__ACCESS_KEY_ID AWS_SECRET__KEY AWS_SECRET_ACCESS_KEY AWS_SESSION_TOKEN AWS_SECURITY_TOKEN	Yes	The security credentials required to execute the Lambda function, depending on which runtime is used. Different runtimes use a subset of these keys. They are generated via an IAM execution role specified for the function.
PATH	No	Contains /usr/local/bin, /usr/bin or /bin for running executables.
LANG	No	Set to en_US.UTF-8. This is the Locale of the runtime.
LD_LIBRARY_PATH	No	Contains /lib64, /usr/lib64, LAMBDA_TASK_ROOT, LAMBDA_TASK_ROOT/lib. Used to store helper libraries and function code.
NODE_PATH	No	Set for the Node.js runtime. It contains LAMBDA_RUNTIME_DIR, LAMBDA_RUNTIME_DIR/node_modules, LAMBDA_TASK_ROOT.
PYTHON_PATH	No	Set for the Python runtime. It contains LAMBDA_RUNTIME_DIR.

Suggested Reading

If you are new to AWS Lambda, we suggest you read through all of the topics in the How It Works section to familiarize yourself with Lambda, starting with How It Works (p. 151).

After you read all of the topics in the How it Works section, we recommend that you review Building Lambda Functions (p. 5), try the Getting Started (p. 160) exercise, and then explore the Use Cases (p. 175). Each use case provides step-by-step instructions for you to set up the end-to-end experience.

Getting Started

In this Getting Started section, you do the following:

- Set up an AWS account and AWS Command Line Interface (AWS CLI). Most tutorials use the AWS CLI commands.
- Create and test a simple Hello World Lambda function.

To get started, complete the steps in the following topics:

Topics

Step 1: Set Up an AWS Account and the AWS CLI

If you have not already done so, you need to sign up for an AWS account and create an administrator user in the account. You also need to set up the AWS Command Line Interface (AWS CLI). Many of the tutorials use the AWS CLI.

To complete the setup, follow the instructions in the following topics:

Topics

Step 1.1: Set Up an AWS Account and Create an Administrator User

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

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 AWS 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 already have an AWS account, 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 https://aws.amazon.com/, and then choose **Create an AWS Account**.
2. Follow the online 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 permissions to access the resources owned by that service. The console requires your password. You can create access keys for your AWS account to access the AWS CLI or API. However, we don't recommend that you access AWS using the credentials for your AWS account. Instead, we recommend that you use AWS Identity and Access Management (IAM). Create an IAM user, add the user to an IAM group with administrative permissions, and then grant administrative permissions to the IAM user that you created. You can then access AWS using a special URL and that IAM user's credentials.

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

The Getting Started exercises and tutorials in this guide assume you have a user (`adminuser`) with administrator privileges. When you follow the procedure, create a user with name `adminuser`.

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

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Users**, and then choose **Add user**.
3. For **User name**, type a user name, such as **`Administrator`**. The name can consist of letters, digits, and the following characters: plus (+), equal (=), comma (,), period (.), at (@), underscore (_), and hyphen (-). The name is not case sensitive and can be a maximum of 64 characters in length.
4. Select the check box next to **AWS Management Console access**, select **Custom password**, and then type the new user's password in the text box. You can optionally select **Require password reset** to force the user to select a new password the next time the user signs in.

5. Choose **Next: Permissions**.
6. On the **Set permissions for user** page, choose **Add user to group**.
7. Choose **Create group**.
8. In the **Create group** dialog box, type the name for the new group. The name can consist of letters, digits, and the following characters: plus (+), equal (=), comma (,), period (.), at (@), underscore (_), and hyphen (-). The name is not case sensitive and can be a maximum of 128 characters in length.
9. For **Filter**, choose **Job function**.
10. In the policy list, select the check box for **AdministratorAccess**. Then choose **Create group**.
11. Back in the list of groups, select the check box for your new group. Choose **Refresh** if necessary to see the group in the list.
12. Choose **Next: Review** to see the list of group memberships to be added to the new user. When you are ready to proceed, choose Add permissions.

You can use this same process to create more groups and users, and to give your users access to your AWS account resources. To learn about using policies to restrict users' permissions to specific AWS resources, go to Access Management and Example Policies for Administering AWS Resources.

To sign in as the new IAM user

1. Sign out of the AWS Management Console.
2. Use the following URL format to log in to the console:

   ```
   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 how to find your account number, see 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 Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. On the navigation pane, choose **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 choose **Yes, Create**.
5. To remove the alias, choose **Customize**, and then choose **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)
- Getting Started
- IAM User Guide

Next Step

Step 1.2: Set Up the AWS Command Line Interface (AWS CLI) (p. 163)

Step 1.2: Set Up the AWS Command Line Interface (AWS CLI)

All the exercises in this guide assume that you are using administrator user credentials (`adminuser`) in your account to perform the operations. For instructions on creating an administrator user in your AWS account, see Step 1.1: Set Up an AWS Account and Create an Administrator User (p. 160), and then follow the steps to download and configure the AWS Command Line Interface (AWS 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 AWS CLI config file. You use this profile when executing the AWS 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, see Regions and Endpoints in the *Amazon Web Services 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
```

Now that you have set up an account and AWS CLI, you can create your first Lambda function. For instructions, see Step 2: Create a HelloWorld Lambda Function and Explore the Console (p. 164).

Step 2: Create a HelloWorld Lambda Function and Explore the Console

In this Getting Started exercise you first create a Hello World Lambda function using the AWS Lambda console. Next, you manually invoke the Lambda function using a sample event data. AWS Lambda executes the Lambda function and returns results. You then verify execution results, including the logs that your Lambda function created and various CloudWatch metrics.

As you follow the steps, you will also familiarize yourself with the AWS Lambda console including:

- Explore the blueprints. Each blueprint provides sample code and sample configurations that enable you to create Lambda functions with just a few clicks. The Getting Started exercise uses the **hello-world-python** blueprint.
- View and update configuration information of your Lambda function.
- Invoke a Lambda function manually and explore results in the **Execution results** section.
- Monitor CloudWatch metrics in the console.

Although not required, we recommend you review How It Works (p. 151) first.

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 Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Next Step

Step 2.1: Create a Hello World Lambda Function (p. 164)

Step 2.1: Create a Hello World Lambda Function

Follow the steps in this section to create a Hello World Lambda function. In this step, you will do the following:

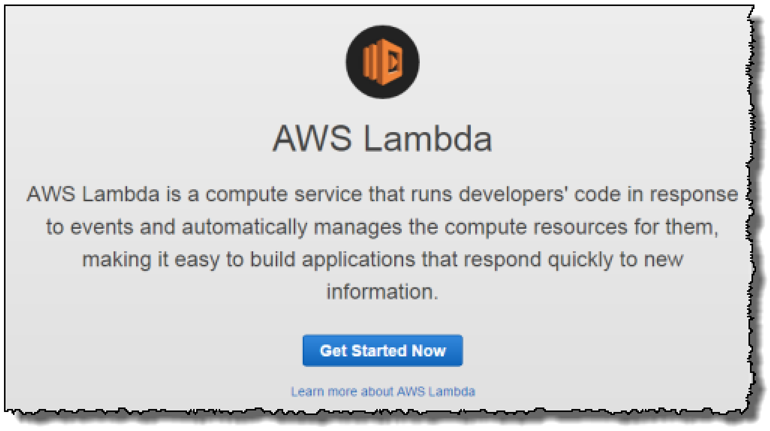
- **Select a blueprint** – For this exercise, you use the **hello-world-python** blueprint. It provides sample code authored in Python. The language used for the Lambda function does not matter for this exercise. Later you can create your own Lambda functions in any of the supported languages.

  Blueprints provide example code to do some minimal processing. Most blueprints process events from specific event sources, such as Amazon S3, DynamoDB, or custom application. For example, if you select an **s3-get-object** blueprint, it provides sample code that processes an object-created event published by Amazon S3 that Lambda receives as parameter.
- **Configure function** – Because you select a blueprint for this exercise, the console will have some of the configuration information prepopulated. For example, it preconfigures Python 2.7 as the runtime, provides example code, identifies the handler in the code sample, and other configuration information such as memory and timeout. For more information about configuring functions, see Lambda Functions (p. 3). For more information about the function configuration parameters, see CreateFunction (p. 332).

  You will also create an IAM role (referred as the *execution role*) with the necessary permissions that AWS Lambda can assume to invoke your Lambda function on your behalf.

To create a Hello World Lambda function

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



AWS Lambda

AWS Lambda is a compute service that runs developers' code in response to events and automatically manages the compute resources for them, making it easy to build applications that respond quickly to new information.

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 **Select blueprint** page, first explore the available blueprints. Then, select a specific blueprint for this Getting Started exercise.

   a. Review the blueprints. You can also use the **Filter** to search for specific blueprints. For example:

      - Enter **`S3`** in **Filter** to get only the list of blueprints available to process Amazon S3 events.
      - Enter **`dynamodb`** in **Filter** to get a list of available blueprints to process Amazon DynamoDB events.

   b. For this Getting Started exercise, enter **`hello-world-python`** in **Filter**, and then choose the **hello-world-python** blueprint.

4. On the **Configure triggers** page, you can optionally choose a service that automatically triggers your Lambda function by choosing the gray box with ellipses (...) to display a list of available services.

   a. Depending on which service you select, you are prompted to provide relevant information for that service. For example, if you select DynamoDB, you need to provide the following:

      - The name of the DynamoDB table
      - Batch size
      - Starting position

   b. For this Getting Started exercise, do not configure a trigger and choose **Next**.

5. On the **Configure function** page, do the following:

   a. Review the preconfigured Lambda function configuration information, including:

      - **Runtime** is Python 2.7.
      - Code authored in Python is provided. It reads incoming event data and logs some of the information to CloudWatch.
      - **Handler** shows **`lambda_function.lambda_handler`** value. It is the *`filename.handler-function`*. The console saves the sample code in the

`lambda_function.py` file and in the code `lambda_handler` is the function name that receives the event as a parameter when the Lambda function is invoked. For more information, see Lambda Function Handler (Python) (p. 39).

b. Enter the function name **`hello-world-python`** in **Name**.

c. In the **Lambda Function Code** section, do the following:

- Review the sample code. Note that:

  - The console saves this code as `lambda_handler.py`. The console then zips the file, and uploads it to AWS Lambda creating your Lambda function.
  - The sample code processes incoming events of the following form:

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

  After creating the Lambda function, you invoke it using sample events of this form in the next section.

d. In the **Lambda function handler and role** section, do the following:

   i. Note the **Handler*** value. It is of the form *`python-file-name`*.*`handler-function`*.

   ii. In **Role***, choose **Create new role from template(s)**.

   iii. In **Role name**, type a name for the role.

   iv. In **Role templates**, Lambda provides a list of optional templates that, should you select one, automatically creates the role with the requisite permissions attached to that policy. For a list of the **Policy templates**, see Policy Templates (p. 311). For the purpose of this tutorial, you can leave this field blank because your Lambda function already has the basic execution permission it needs.

   Note
   Optionally, you could select **Choose an existing role** if you already have a role created with specific permissions beyond basic execution. You can also select **Create a custom role**. When you choose this option, a window appears where you can edit the permissions policy inline.

e. In the **Advanced settings** section, leave the default Lambda function configuration values.

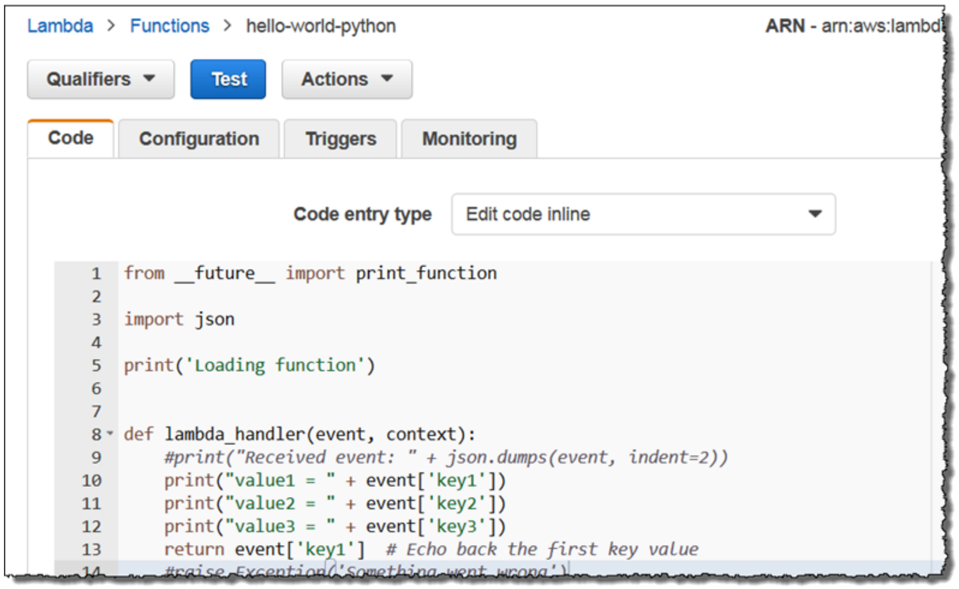
   The memory and timeout values are sufficient for the Lambda function you are creating. These configurations influence the performance of your code. For more information, see Lambda Functions (p. 3).

f. Choose **Next**.

g.

   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 package to AWS Lambda creating your Lambda function. The console shows the **hello-world-python** Lambda function, you can now perform various action including test the function:



```
from __future__ import print_function

import json

print('Loading function')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
    #raise Exception('Something went wrong')
```

Note the tabs in the console:

- **Code** – Shows the Lambda function code.
- **Configuration** – shows current function configuration and you can change the configuration as needed. After you change any configuration settings, you choose **Save** to save the updated configuration.
- **Triggers** – Shows any triggers you configured for this function (does not apply to this Getting Started exercise).
- **Monitoring** – Provides various CloudWatch metrics for your Lambda function. In the next section, you invoke your **hello-world-python** Lambda function and review these metrics.

Next Step

Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167)

Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics

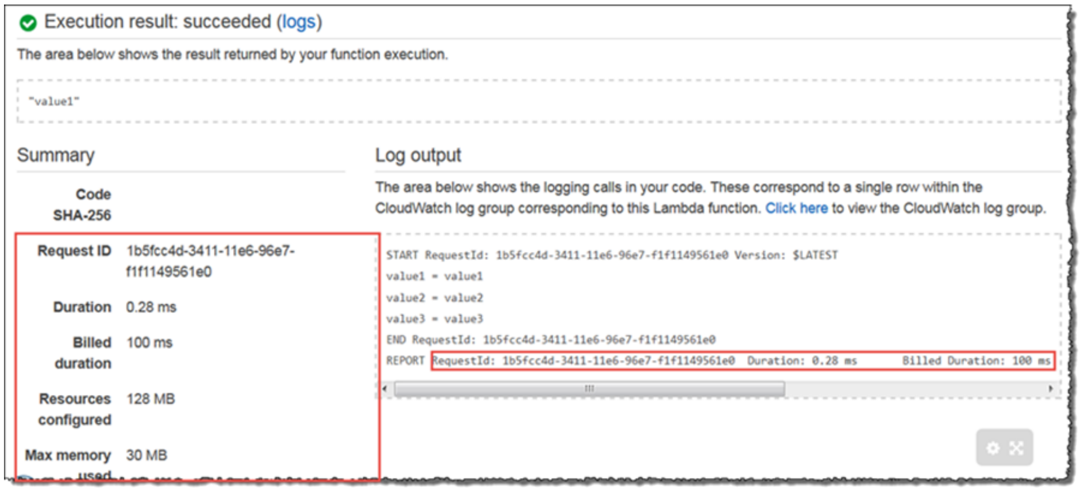
Follow the steps to invoke your Lambda function using the sample event data provided in the console.

1. On the **Lambda > Functions > HelloWorld** page, choose **Test**.
2. In the **Input test event** page, choose **Hello World** from the **Sample event template** list. The following sample event template appears in the window.

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

You can change key and values in the sample JSON, but don't change the event structure. If you do change any keys and values, you must update the sample code accordingly. Choose **Save and test**.

3. AWS Lambda executes your function on your behalf. The `handler` in your Lambda function receives and then processes the sample event.
4. Upon successful execution, view results in the console.



Execution result: succeeded (logs)

The area below shows the result returned by your function execution.

```
"value1"
```

Summary

Code SHA-256	
Request ID	1b5fcc4d-3411-11e6-96e7-f1f1149561e0
Duration	0.28 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	30 MB

Log output

The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: 1b5fcc4d-3411-11e6-96e7-f1f1149561e0 Version: $LATEST
value1 = value1
value2 = value2
value3 = value3
END RequestId: 1b5fcc4d-3411-11e6-96e7-f1f1149561e0
REPORT RequestId: 1b5fcc4d-3411-11e6-96e7-f1f1149561e0  Duration: 0.28 ms      Billed Duration: 100 ms
```

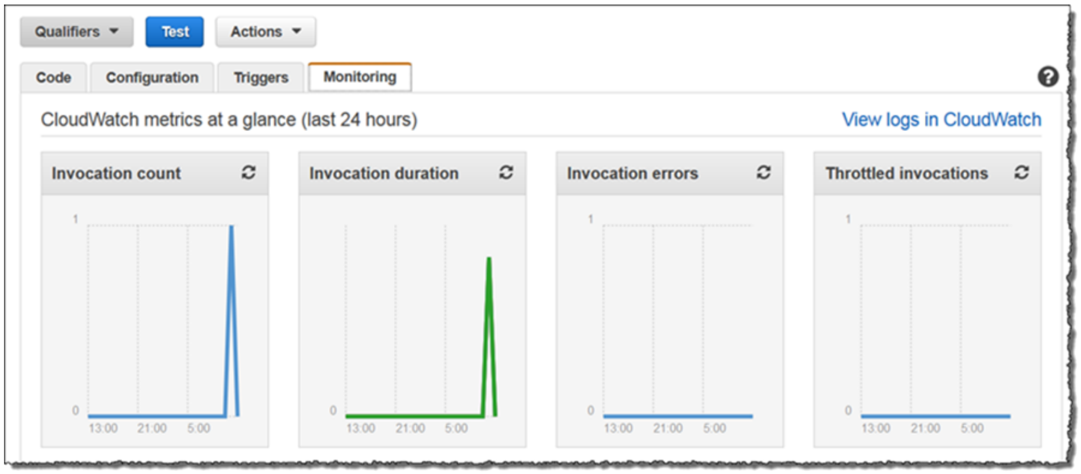
Note the following:

- The **Execution result** section shows the execution status as **succeeded** and also shows the function execution results, returned by the `return` statement.

  Note
  The console always uses the `RequestResponse` invocation type (synchronous invocation) when invoking a Lambda function which causes AWS Lambda to return a response immediately. For more information, see Invocation Types (p. 4).
- The **Summary** section shows the key information reported in the **Log output** section (the *REPORT* line in the execution log).
- The **Log output** section shows the log AWS Lambda generates for each execution. These are the logs written to CloudWatch by the Lambda function. The AWS Lambda console shows these logs for your convenience.

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

5. Run the Lambda function a few times to gather some metrics that you can view in the next step.
6. Choose the **Monitoring** tab to view the CloudWatch metrics for your Lambda function. This page shows four CloudWatch metrics.



Note the following:

- The X-axis shows the past 24 hours from the current time (for example, 2:00 pm as shown in the screen shot).
- Invocation count shows the number of invocations during this interval.
- Invocation duration shows how long it took for your Lambda function to run. It shows minimum, maximum, and average time of execution.
- Invocation errors show the number of times your Lambda function failed. You can compare the number of times your function executed and how many times it failed (if any).
- Throttled invocation metrics show whether AWS Lambda throttled your Lambda function invocation. For more information, see List of AWS Lambda Limits (p. 285).
- The AWS Lambda console shows these CloudWatch metrics for your convenience. You can see these metrics in the Amazon CloudWatch console by clicking any of these metrics.

Next Step

Step 2.3: (Optional) Create a Lambda Function Authored in Java (p. 169)

Step 2.3: (Optional) Create a Lambda Function Authored in Java

The blueprints provide sample code authored either in Python or Node.js. You can easily modify the example using the inline editor in the console. However, if you want to author code for your Lambda function in Java, there are no blueprints provided. Also, there is no inline editor for you to write Java code in the AWS Lambda console.

That means, you must write your Java code and also create your deployment package outside the console. After you create the deployment package, you can use the console to upload the package to AWS Lambda to create your Lambda function. You can also use the console to test the function by manually invoking it.

In this section you create a Lambda function using the following Java code example.

```
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. 21). For now, note the following about this code:

- When you package and upload this code to create your Lambda function, you specify the `example.Hello::myHandler` method reference as the handler.
- The handler in this example uses the `int` type for input and the `String` type for output.

  AWS Lambda supports input/output of JSON-serializable types and InputStream/OutputStream types. When you invoke this function you will pass a sample int (for example, 123).
- In this exercise you use the console to manually invoke this Lambda function. The console always uses the `RequestResponse` invocation type (synchronous) and therefore you will see the response 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. 32).

First, you need to package this code and any dependencies into a deployment package. Then, you can use the Getting Started exercise to upload the package to create your Lambda function and test using the console.

Next Step

Step 2.4: (Optional) Create a Lambda Function Authored in C# (p. 170)

Step 2.4: (Optional) Create a Lambda Function Authored in C#

The AWS Lambda console blueprints provide sample code authored either in Python or Node.js. You can easily modify the example using the inline editor in the console. However, if you want to author code for your Lambda function in C#, there are no blueprints provided. Also, there is no inline editor for you to write C# code in the AWS Lambda console.

While the Lambda console does not offer editing for compiled languages such as Java and C#, you can use your choice of IDEs, such as Visual Studio, to create and package your C# code and libraries. Once packaged as a ZIP file, you can use the AWS Lambda console to upload and test C# Lambda functions and to view logs and metrics for them.

In this section you create a Lambda function using the following C# code example.

```


using Amazon.Lambda.Core;
namespace LambdaFunctionExample{
public class Hello {
    public string MyHandler(int count, ILambdaContext context) {
        var logger = context.Logger;
        logger.log("received : " + count);
        return count.ToString();
    }
  }
}
```

Your Lambda function handler signature should be of the format *`Assembly::Namespace.ClassName::MethodName`*. The programming model explains how to write your C# 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 C# (p. 46). For now, note the following about this code:

- The handler in this example uses the `int` type for input and the `string` type for output.

  When you invoke this function you will pass a sample int (for example, 123).
- In this exercise you use the console to manually test this Lambda function. The console always uses the `RequestResponse` invocation type (synchronous) and therefore you will see the response in the console.
- The handler includes the optional `ILambdaContext` parameter. In the code we use the `LambdaLogger` provided by the `Amazon.Lambda.Core.LambdaLogger` object to write log entries to CloudWatch logs. For information about using the `ILambdaContext` object, see The Context Object (C#) (p. 51).

First, you need to package this code and any dependencies into a deployment package. Then, you can use the Getting Started exercise to upload the package to create your Lambda function and test using the console. For more information, see Creating a Deployment Package (C#) (p. 57).

Next Step

Step 3: Create a Simple Microservice using Lambda and API Gateway (p. 171)

Step 3: Create a Simple Microservice using Lambda and API Gateway

In this exercise you will use the Lambda console to create a Lambda function (`MyLambdaMicroservice`), and an Amazon API Gateway endpoint to trigger that function. You will be able to call the endpoint with any method (`GET`, `POST`, `PATCH`, etc.) to trigger your Lambda function. When the endpoint is called, the entire request will be passed through to your Lambda function. Your function action will depend on the method you call your endpoint with:

- DELETE: delete an item from a DynamoDB table
- GET: scan table and return all items
- POST: Create an item
- PUT: Update an item

Next Step

Step 3.1: Create an API Using Amazon API Gateway (p. 172)

Step 3.1: Create an API Using Amazon API Gateway

Follow the steps in this section to create a new Lambda function an API Gateway endpoint to trigger it:

1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. Choose **Create Lambda function**.
3. On the **Select blueprint** page, choose the **microservice-http-endpoint** blueprint. You can use the **Filter** to find it.
4. The **Configure triggers** page will be populated with an API Gateway trigger. The default API name that will be created is `LambdaMicroservice` (You can change this name via the **API Name** field if you wish).

   Note
   When you complete the wizard and create your function, Lambda automatically creates a proxy resource named `MyLambdaMicroservice` (your function name) under the API name you selected. For more information about proxy resources, see Configure Proxy Integration for a Proxy Resource. A proxy resource has an `AWS_PROXY` integration type and a catch-all method `ANY`. The `AWS_PROXY` integration type applies a default mapping template to pass through the entire request to the Lambda function and transforms the output from the Lambda function to HTTP responses. The `ANY` method defines the same integration setup for all the supported methods, including `GET`, `POST`, `PATCH`, `DELETE` and others.

   After reviewing your trigger, choose **Next**.
5. On the **Configure function** page, do the following:
   a. Review the preconfigured Lambda function configuration information, including:
      - **Runtime** is `Node.js 4.3`
      - Code authored in JavaScript is provided. The code performs DynamoDB operations based on the method called and payload provided.
      - **Handler** shows `index.handler`. The format is: `filename.handler-function`

   b. Enter the function name `MyLambdaMicroservice` in **Name**.

   c. In **Role**, enter a role name for the new role that will be created.

      Note
      The **microservice-http-endpoint** blueprint pre-populates the Simple Microservice permission policy template in the **Policy templates** field, to be added to your new role upon creation. This automatically adds the requisite permissions attached to that policy to your new role. For more information, see Policy Templates (p. 311).
6. Choose **Create function**.

Next Step

Step 3.2: Test Sending an HTTPS Request (p. 172)

Step 3.2: Test Sending an HTTPS Request

In this step, you will use the console to test the Lambda function. In addition, you can run a `curl` command to test the end-to-end experience. That is, send an HTTPS request to your API method and have Amazon API Gateway invoke your Lambda function.

1. With your `MyLambdaMicroService` function still open in the console, choose the **Actions** tab and then choose **Configure test event**.

2. Replace the existing text with the following:

```
{
 "httpMethod": "GET",
 "queryStringParameters": {
 "TableName": "MyTable"
    }
}
```

3. After entering the text above choose **Save and test**.

To test your microservice end-to-end, you will need to create a DynamoDB table, copy your newly created API Gateway endpoint, and invoke it using a `curl` command. To do this:

- Open a new window or tab and go to the DynamoDB console and create a new table called `MyTable`.
- Go to the Lambda console and choose your Lambda function. Choose the **Triggers** tab and then copy the endpoint that appears under your API Gateway trigger (for example, `https://someid.execute-api-region-amazon.aws.com/prod/your_function_name`.
- Run the following command to send an HTTPS `POST` method to your API endpoint:

```
curl endpoint?TableName=TableName
```

Next Step

Step 3.3: (Optional) Try Other Blueprints (p. 173)

Step 3.3: (Optional) Try Other Blueprints

You can optionally try the following exercises:

- You used the **hello-world-python** blueprint in this Getting Started exercise. This blueprint provides sample code authored in Python. There is also the **hello-world** blueprint that provides similar Lambda function code that is authored in Node.js.
- Both the **hello-world-python** and the **hello-world** blueprints process custom events. For this Getting Started exercise, you used hand-crafted sample event data. Your can write Lambda functions to process events published by event sources such as Amazon S3 and DynamoDB. This requires event source configuration in the console.

  For example, you can write a Lambda function to process Amazon S3 events. Then, you configure Amazon S3 as the event source to publish object-created events to AWS Lambda. When you upload an object to your bucket, Amazon S3 detects the event and invokes your Lambda function. Your Lambda function receives the event data as a parameter. You can verify your Lambda function executed by reviewing the CloudWatch logs either in the Lambda console or the CloudWatch console.

  The Lambda console provide blueprint to set up an example Lambda function to process Amazon S3 events. When creating a Lambda function in the console on the **Select blueprint** page, enter **s3** in the **Filter** box to search for a list of available blueprints.

  For more information about working with different event sources, see Use Cases (p. 175).

Next Step

What's Next? (p. 174)

What's Next?

This Getting Started exercise provided you with an overview of how to use the AWS Lambda console.

AWS Lambda functions can also be automatically invoked in response to events in other AWS services such as Amazon S3 and DynamoDB. Lambda functions can also be invoked on-demand over HTTPS. You can also build your own custom event sources and invoke Lambda functions on demand. For more information, see How It Works (p. 151).

Depending on your integration scenario, whether your application needs event-driven Lambda function invocation or on-demand invocation, see the following sections:

- Using AWS Lambda with Amazon S3 (p. 175)
- Using AWS Lambda with Amazon Kinesis (p. 193)
- Using AWS Lambda with Amazon DynamoDB (p. 203)
- Using AWS Lambda with AWS CloudTrail (p. 215)
- Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS) (p. 236)
- Using AWS Lambda as Mobile Application Backend (Custom Event Source: Android) (p. 250)

The console provides several blueprints for you to set up example Lambda functions quickly that can process events from these event sources. You may want to explore other blueprints in the console to get started with Lambda functions triggered by these event sources.

Examples of How to Use AWS Lambda

The use cases for AWS Lambda can be grouped into the following categories:

- **Using AWS Lambda with AWS services as event sources** – *Event sources* publish events that cause the Lambda function to be invoked. These can be AWS services such as Amazon S3. For more information and tutorials, see the following topics:

  Using AWS Lambda with Amazon S3 (p. 175)

  Using AWS Lambda with Amazon Kinesis (p. 193)

  Using AWS Lambda with Amazon DynamoDB (p. 203)

  Using AWS Lambda with AWS CloudTrail (p. 215)

  Using AWS Lambda with Amazon SNS from Different Accounts (p. 230)
- **On-demand Lambda function invocation over HTTPS (Amazon API Gateway)** – In addition to invoking Lambda functions using event sources, you can also invoke your Lambda function over HTTPS. You can do this by defining a custom REST API and endpoint using API Gateway. For more information and a tutorial, see Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS) (p. 236).
- **On-demand Lambda function invocation (build your own event sources using custom apps)** – User applications such as client, mobile, or web applications can publish events and invoke Lambda functions using the AWS SDKs or AWS Mobile SDKs, such as the AWS Mobile SDK for Android. For more information and a tutorial, see Getting Started (p. 160) and Using AWS Lambda as Mobile Application Backend (Custom Event Source: Android) (p. 250)
- **Scheduled events** – You can also set up AWS Lambda to invoke your code on a regular, scheduled basis using the AWS Lambda console. You can specify a fixed rate (number of hours, days, or weeks) or you can specify a cron expression. For more information and a tutorial, see Using AWS Lambda with Scheduled Events (p. 263).

Using AWS Lambda with Amazon S3

Amazon S3 can publish events (for example, when an object is created in a bucket) to AWS Lambda and invoke your Lambda function by passing the event data as a parameter. This integration enables you to write Lambda functions that process Amazon S3 events. In Amazon S3, you add bucket notification configuration that identifies the type of event that you want Amazon S3 to publish and the Lambda function that you want to invoke.

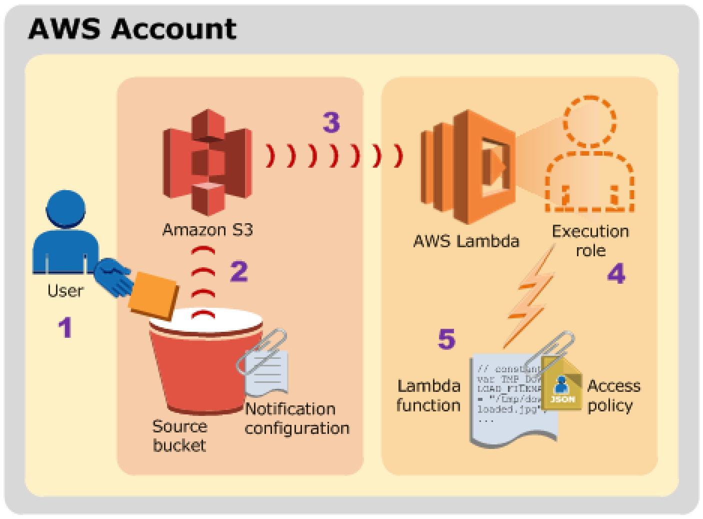
Note the following about how the Amazon S3 and AWS Lambda integration works:

- **Non-stream based (async) model** – This is a model (see Event Source Mapping (p. 120)), where Amazon S3 monitors a bucket and invokes the Lambda function by passing the event data as a parameter. In a push model, you maintain event source mapping within Amazon S3 using the bucket notification configuration. In the configuration, you specify the event types that you want Amazon S3 to monitor and which AWS Lambda function you want Amazon S3 to invoke. For more information, see Configuring Amazon S3 Event Notifications in the *Amazon Simple Storage Service Developer Guide*.
- **Asynchronous invocation** – AWS Lambda invokes a Lambda function using the `Event` invocation type (asynchronous invocation). For more information about invocation types, see Invocation Types (p. 4).
- **Event structure** – The event your Lambda function receives is for a single object and it provides information, such as the bucket name and object key name.

Note that there are two types of permissions policies that you work with when you set up the end-to-end experience:

- **Permissions for your Lambda function** – Regardless of what invokes a Lambda function, AWS Lambda executes the function by assuming the IAM role (execution role) that you specify at the time you create the Lambda function. Using the permissions policy associated with this role, you grant your Lambda function the permissions that it needs. For example, if your Lambda function needs to read an object, you grant permissions for the relevant Amazon S3 actions in the permissions policy. For more information, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).
- **Permissions for Amazon S3 to invoke your Lambda function** – Amazon S3 cannot invoke your Lambda function without your permission. You grant this permission via the permissions policy associated with the Lambda function.

The following diagram summarizes the flow:



1. User uploads an object to an S3 bucket (object-created event).
2. Amazon S3 detects the object-created event.

3. Amazon S3 invokes a Lambda function that is specified in the bucket notification configuration.
4. AWS Lambda executes the Lambda function by assuming the execution role that you specified at the time you created the Lambda function.
5. The Lambda function executes.

For a tutorial that walks you through an example setup, see Tutorial: Using AWS Lambda with Amazon S3 (p. 177).

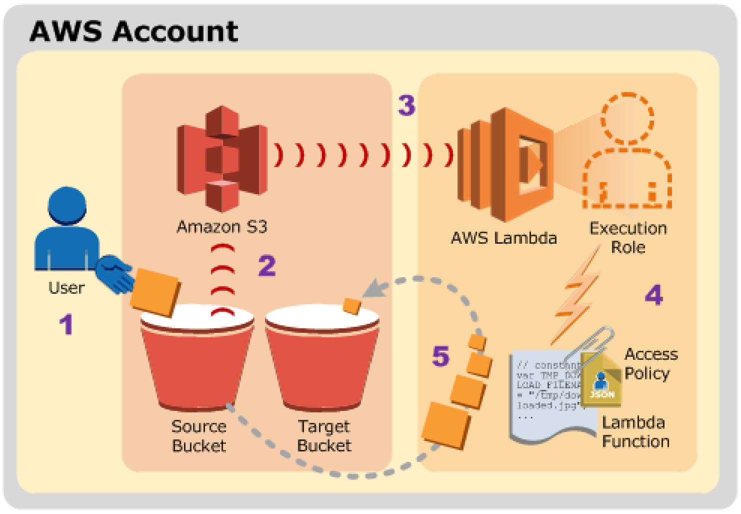
Tutorial: Using AWS Lambda with Amazon S3

Suppose you want to create a thumbnail for each image (.jpg and .png objects) that is uploaded to a bucket. You can create a Lambda function (`CreateThumbnail`) that Amazon S3 can invoke when objects are created. Then, the Lambda function can read the image object from the *`source`* bucket and create a thumbnail image target bucket (in this tutorial, it's called the *`source`*`resized` bucket).

> **Important**
> You must use two buckets. If you use the same bucket as the source and the target, each thumbnail uploaded to the source bucket triggers another object-created event, which then invokes the Lambda function again, creating an unwanted recursion.

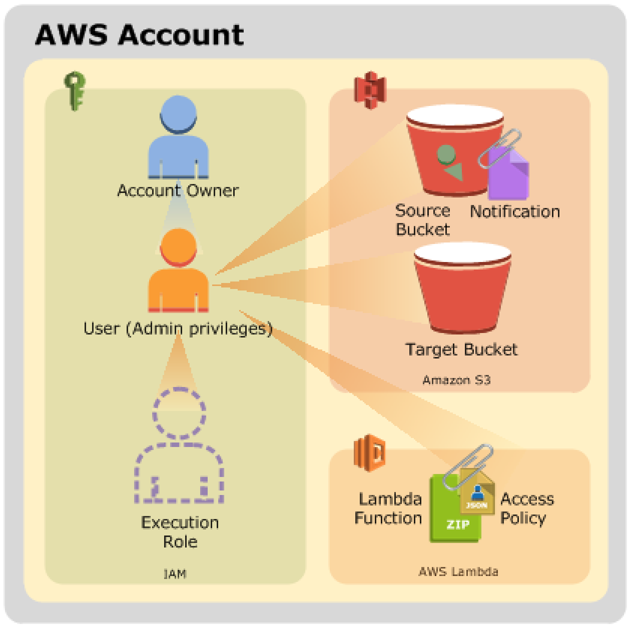
Implementation Summary

The following diagram illustrates the application flow:



1. A user uploads an object to the source bucket in Amazon S3 (object-created event).
2. Amazon S3 detects the object-created event.
3. Amazon S3 publishes the `s3:ObjectCreated:*` event to AWS Lambda by invoking the Lambda function and passing event data as a function parameter.
4. AWS Lambda executes the Lambda function by assuming the execution role that you specified at the time you created the Lambda function.
5. From the event data it receives, 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 tutorial, you will have the following Amazon S3, Lambda, and IAM resources in your account:



In Lambda:

- A Lambda function.
- An access permissions policy associated with your Lambda function – You grant Amazon S3 permissions to invoke the Lambda function using this permissions policy. You will also restrict the permissions so that Amazon S3 can invoke the Lambda function only for object-created events from a specific bucket that is 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 the 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 How It Works (p. 151).

In IAM:

- Administrator user – Called **adminuser**. Using root credentials of an AWS account is not recommended. Instead, use the **adminuser** credentials to perform the steps in this tutorial.
- An IAM role (execution role) – You grant permissions that your Lambda function needs through the permissions policy associated with this role.

In Amazon S3:

- Two buckets named *`source`* and *`source`*`resized`. Note that *`source`* is a placeholder name and you need to replace it with your actual bucket name. For example, if you have a bucket named `example` as your source, you will create `exampleresized` as the target bucket.
- Notification configuration on the source bucket – You 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, see Setting Up Notification of Bucket Events in *Amazon Simple Storage Service Developer Guide*..

Now you are ready to start the tutorial. Note that after the initial preparation, the tutorial is divided into two main sections:

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

Next Step

Step 1: Prepare (p. 179)

Step 1: Prepare

In this section, you do the following:

- Sign up for an AWS account and set up the AWS CLI.
- Create two buckets (*`source`* and *`source`*`resized` bucket) with a sample .jpg object (`HappyFace.jpg`) in the source bucket. For instructions, see the following procedure.

Step 1.1: Sign Up for AWS and Set Up the AWS CLI

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account (called **adminuser**).
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Step 1.2: 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 that both of the buckets are in the same region. In this tutorial, we use 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 *`source`* followed by **`resized`**, where *`source`* is the name of the bucket you want to use for the source. For example, `mybucket` and `mybucketresized`.

   For instructions, see 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 you connect to Amazon S3, you 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 first.

Next Step

Step 2: Create a Lambda Function and Invoke It Manually (Using Sample Event Data) (p. 180)

Step 2: Create a Lambda Function and Invoke It Manually (Using Sample Event Data)

In this section, you do the following:

- Create a Lambda function deployment package using the sample code provided.

  Note
  To see more examples of using other AWS services within your function, including calling other Lambda functions, see AWS SDK for JavaScript
- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify an IAM role (execution role) that Lambda can assume to execute the function on your behalf.
- Create the Lambda function by uploading the deployment package, and then test it by invoking it manually using sample Amazon S3 event data.

Topics

Step 2.1: Create a Deployment Package

From the **Filter View** list, choose the language you want to use for your Lambda function. The appropriate section appears with code and specific instructions for creating a deployment package.

Node.js

The deployment package is a .zip file containing your Lambda function code and dependencies.

1. Create a folder (`examplefolder`), and then create a subfolder (`node_modules`).
2. Install the Node.js platform. For more information, see the Node.js website.
3. Install dependencies. The code examples use 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 only need to install the other libraries. 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
   ```
4. Open a text editor, and then copy the following code.

   ```
   // dependencies
   var async = require('async');
   var AWS = require('aws-sdk');
   var gm = require('gm')
               .subClass({ imageMagick: true }); // Enable ImageMagick
    integration.
   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, callback) {
    // 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) {
        callback("Source and destination buckets are the same.");
        return;
    }

    // Infer the image type.
    var typeMatch = srcKey.match(/\.([^.]*)$/);
    if (!typeMatch) {
        callback("Could not determine the image type.");
        return;
    }
    var imageType = typeMatch[1];
    if (imageType != "jpg" && imageType != "png") {
        callback('Unsupported image type: ${imageType}');
        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
                );
            }

            callback(null, "message");
        }
    );
};
```

Note
The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)

5. 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 that 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`.

6. Save the file as `CreateThumbnail.js` in `examplefolder`. After you complete this step, you will have the following folder structure:

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

7. Zip the CreateThumbnail.js file and the node_modules folder as `CreateThumbnail.zip`.

   This is your Lambda function deployment package.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 187)

Java

The following is example Java code that reads incoming Amazon S3 events and creates a thumbnail. Note that it 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 a handler, see Leveraging Predefined Interfaces for Creating Handler (Java) (p. 28).

The `S3Event` type that 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.S3EventNotificationRecord;
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 an interface that requires us to specify a return type, so in this example the handler uses `String` as the return type.

Using the preceding code (in a file named `ProcessKinesisEvents.java`), create a deployment package. Make sure that you add the following dependencies:

- `aws-lambda-java-core.`
- `aws-lambda-java-events`

These can be found at aws-lambda-java-libs.

For more information, see Programming Model for Authoring Lambda Functions in Java (p. 21).

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. For examples of how to use the Maven build tool to create a standalone .jar, see Creating a .jar Deployment Package Using Maven without any IDE (Java) (p. 62) and Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java) (p. 65). For an example of how to use the Gradle build tool to create a .zip file, see Creating a .zip Deployment Package (Java) (p. 67).

After you verify that your deployment package is created, go to the next step to create an IAM role (execution role). You specify this role at the time you create your Lambda function.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 187)

Python

In this section, you create an example Python function and install dependencies.

1. Open a text editor, and copy the following code. The code uploads the resized image to a different bucket with the same image name, as shown following:

   `source-bucket/image.png` -> `source-bucketresized/image.png`

   ```
   from __future__ import print_function
   import boto3
   import os
   import sys
   import uuid
   from PIL import Image
   import PIL.Image

   s3_client = boto3.client('s3')

   def resize_image(image_path, resized_path):
       with Image.open(image_path) as image:
           image.thumbnail(tuple(x / 2 for x in image.size))
           image.save(resized_path)

   def handler(event, context):
       for record in event['Records']:
           bucket = record['s3']['bucket']['name']
           key = record['s3']['object']['key']
           download_path = '/tmp/{}{}'.format(uuid.uuid4(), key)
           upload_path = '/tmp/resized-{}'.format(key)

           s3_client.download_file(bucket, key, download_path)
           resize_image(download_path, upload_path)
           s3_client.upload_file(upload_path, '{}resized'.format(bucket),
    key)
   ```

2. Save the file as `CreateThumbnail.py`.
3. If your source code is on a local host, copy it over.

   ```
   scp -i key.pem /path/to/my_code.py ec2-user@public-ip-address:~/
   CreateThumbnail.py
   ```

4. Connect to a 64-bit Amazon Linux instance via SSH.

```
ssh -i key.pem ec2-user@public-ip-address
```

5. Ensure basic build requirements are installed.

   ```
   sudo yum install python27-devel python27-pip gcc
   ```

6. Install native dependencies required by Pillow.

   ```
   sudo yum install libjpeg-devel zlib-devel
   ```

7. Create and activate a virtual environment.

   ```
   virtualenv ~/shrink_venv
   ```

   ```
   source ~/shrink_venv/bin/activate
   ```

8. Install libraries in the virtual environment.

   ```
   pip install Pillow
   ```

   ```
   pip install boto3
   ```

   Note
   AWS Lambda includes the AWS SDK for Python (Boto 3), so you don't need to include it in your deployment package, but you can optionally include it for local testing.

9. Create a .zip file

   ```
   zip -9 ~/CreateThumbnail.zip
   ```

10. Add the contents of `lib` and `lib64` site-packages to your .zip file.

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

    ```
    zip -r9 ~/CreateThumbnail.zip *
    ```

    ```
    cd $VIRTUAL_ENV/lib64/python2.7/site-packages
    ```

    ```
    zip -r9 ~/CreateThumbnail.zip *
    ```

11. Add your python code to the .zip file

    cd ~

    ```
    zip -g CreateThumbnail.zip CreateThumbnail.py
    ```

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 187)

Step 2.2: Create the Execution Role (IAM Role)

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

- AWS service role of the type **AWS Lambda** – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaExecute** access permissions policy that you attach to the role.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:

   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-s3-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - In **Attach Policy**, choose **AWSLambdaExecute**.
3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Next Step

Step 2.3: Create the Lambda Function and Test It Manually (p. 188)

Step 2.3: Create the Lambda Function and Test It Manually

In this section, you do the following:

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

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

In this step, you upload the deployment package using the AWS CLI.

1. At the command prompt, run the following Lambda AWS CLI `create-function` command using the `adminuser` as the `--profile`. You need to update the command by providing the .zip file path and the execution role ARN. For the runtime parameter, choose between `nodejs4.3`, `python2.7` or `java8`, depending on the code sample you when you created your deployment package.

```
$ 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 runtime \
--profile adminuser \
--timeout 10 \
--memory-size 1024
```

   Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

```
--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 AWS CLI command.

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

Note
You can create the Lambda function using the AWS Lambda console, in which case note the value of the `create-function` AWS CLI command parameters. You provide the same values in the console UI.

Step 2.3.2: Test the Lambda Function (Invoke Manually)

In this step, you invoke the Lambda function manually using sample Amazon S3 event data. You can test the function using the AWS Management Console or the AWS CLI.

To test the Lambda function (console)

1. Follow the steps in the Getting Started to create and invoke the Lambda function at Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167). For the sample event for testing, choose **S3 Put** in **Sample event template**.
2. Verify that the thumbnail was created in the target bucket and monitor the activity of your Lambda function in the AWS Lambda console as follows:
   - 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.

To test the Lambda function (AWS CLI)

1. Save the following Amazon S3 sample event data in a file and save it as `input.txt`. You need to update the JSON by providing your *`sourcebucket`* name 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/
JRWeUWerMUE5JgHvANOjpD"
         },
         "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 permissions to the access policy associated with your Lambda function to grant Amazon S3 permissions to invoke your function.

3. Verify that the thumbnail was created in the target bucket and monitor the activity of your Lambda function in the AWS Lambda console as follows:

   - 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: Add an Event Source (Configure Amazon S3 to Publish Events) (p. 191)

Step 3: Add an Event Source (Configure Amazon S3 to Publish Events)

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

- Add permissions 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 the following:
  - Event type for which you want Amazon S3 to publish events. For this tutorial, you specify the `s3:ObjectCreated:*` event type so that Amazon S3 publishes events when objects are created.
  - Lambda function to invoke.

Step 3.1: Add Permissions to the Lambda Function's Access Permissions Policy

1. Run the following Lambda CLI `add-permission` command to grant Amazon S3 service principal (`s3.amazonaws.com`) permissions to perform 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 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 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 function's access policy by running the AWS CLI `get-policy` command.

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

Step 3.2: Configure 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 tutorial, select the `ObjectCreated (All)` Amazon S3 event type.
- Lambda function – This is your Lambda function that you want Amazon S3 to invoke.

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

Step 3.3: Test the Setup

You're 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 using the `CreateThumbnail` function.
3. The **adminuser** can also verify the 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.

Step 4: Deploy With AWS SAM and AWS CloudFormation

In the previous section, you used AWS Lambda APIs to create and update a Lambda function by providing a deployment package as a ZIP file. However, this mechanism may not be convenient for automating deployment steps for functions, or coordinating deployments and updates to other elements of a serverless application, like event sources and downstream resources.

You can use AWS CloudFormation to easily specify, deploy, and configure serverless applications. AWS CloudFormation is a service that helps you model and set up your Amazon Web Services resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Lambda functions and DynamoDB tables), and AWS CloudFormation takes care of provisioning and configuring those resources for you.

In addition, you can use the AWS Serverless Application Model to express resources that comprise the serverless application. These resource types, such as Lambda functions and APIs, are fully supported by AWS CloudFormation and make it easier for you to define and deploy your serverless application.

For more information, see Deploying Lambda-based Applications (p. 139).

Specification for Amazon S3 Thumbnail Application

The following contains the SAM template for this application. Copy the text below to a .yaml file and save it next to the ZIP package you created previously.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  CreateThumbnail:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      Timeout: 60
      Policies: AWSLambdaExecute
      Events:
          Type: S3
          Properties:
            Bucket: !Ref SrcBucket
            Events: s3:ObjectCreated:*

  SrcBucket:
    Type: AWS::S3::Bucket
```

Deploying the Serverless Application

For information on how to package and deploy your serverless application using the package and deploy commands, see Packaging and Deployment (p. 145).

Using AWS Lambda with Amazon Kinesis

You can create an Amazon Kinesis stream to continuously capture and store terabytes of data per hour from hundreds of thousands of sources such as website click streams, financial transactions, social media feeds, IT logs, and location-tracking events. For more information, see Amazon Kinesis.

You can subscribe Lambda functions to automatically read batches of records off your Amazon Kinesis stream and process them if records are detected on the stream. AWS Lambda then polls the stream periodically (multiple times per second) for new records.

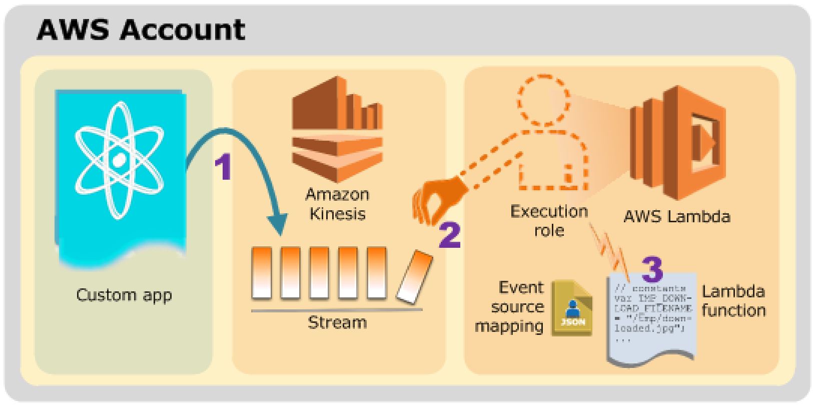
Note the following about how the Amazon Kinesis and AWS Lambda integration works:

- **Stream-based model** – This is a model (see Event Source Mapping (p. 120)), where AWS Lambda polls the stream and, when it detects new records, invokes your Lambda function by passing the new records as a parameter.

  In a stream-based model, you maintain event source mapping in AWS Lambda. The event source mapping describes which stream maps to which Lambda function. AWS Lambda provides an API (CreateEventSourceMapping (p. 328)) that you can use to create the mapping. You can also use the AWS Lambda console to create event source mappings.
- **Synchronous invocation** – AWS Lambda invokes a Lambda function using the `RequestResponse` invocation type (synchronous invocation) by polling the Kinesis Stream. For more information about invocation types, see Invocation Types (p. 4).
- **Event structure** – The event your Lambda function receives is a collection of records AWS Lambda reads from your stream. When you configure event source mapping, the batch size you specify is the maximum number of records that you want your Lambda function to receive per invocation.

Regardless of what invokes a Lambda function, AWS Lambda always executes a Lambda function on your behalf. If your Lambda function needs to access any AWS resources, you need to grant the relevant permissions to access those resources. You also need to grant AWS Lambda permissions to poll your Amazon Kinesis stream. You grant all of these permissions to an IAM role (execution role) that AWS Lambda can assume to poll the stream and execute the Lambda function on your behalf. You create this role first and then enable it at the time you create the Lambda function. For more information, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).

The following diagram illustrates the application flow:



1. Custom app writes records to the stream.
2. AWS Lambda polls the stream and, when it detects new records in the stream, invokes your Lambda function.

3. AWS Lambda executes the Lambda function by assuming the execution role you specified at the time you created the Lambda function.

For a tutorial that walks you through an example setup, see Tutorial: Using AWS Lambda with Amazon Kinesis (p. 194).

Tutorial: Using AWS Lambda with Amazon Kinesis

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

The tutorial is divided into two main sections:

- First, you perform the necessary setup to create a Lambda function and then you test it by invoking it manually using sample event data (you don't need an Amazon Kinesis stream).
- Second, you create an Amazon Kinesis stream (event source). You add an event source mapping in AWS Lambda to associate the stream with your Lambda function. AWS Lambda starts polling the stream, you add test records to the stream using the Amazon Kinesis API, and then you verify that AWS Lambda executed your Lambda function.

> **Important**
> Both the Lambda function and the Amazon Kinesis stream must be in the same AWS region. This tutorial assumes that you create these resources in the `us-west-2` region.

In this tutorial, you use the AWS Command Line Interface to perform AWS Lambda operations such as creating a Lambda function, creating a stream, and adding records to the stream. You use the AWS Lambda console to manually invoke the function before you create a Amazon Kinesis stream. You verify return values and logs in the console UI.

Next Step

Step 1: Prepare (p. 194)

Step 1: Prepare

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account.
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Next Step

Step 2: Create a Lambda Function and Invoke It Manually (Using Sample Event Data) (p. 194)

Step 2: Create a Lambda Function and Invoke It Manually (Using Sample Event Data)

In this section, you do the following:

- Create a Lambda function deployment package using the sample code provided. The sample Lambda function code that you'll use to process Amazon Kinesis events is provided in various languages. Select one of the languages and follow the corresponding instructions to create a deployment package.

Note
To see more examples of using other AWS services within your function, including calling other Lambda functions, see AWS SDK for JavaScript

- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify an IAM role (execution role) that Lambda can assume to execute the function on your behalf.
- Create the Lambda function by uploading the deployment package, and then test it by invoking it manually using sample Amazon Kinesis event data.

Topics

Step 2.1: Create a Deployment Package

From the **Filter View** list, choose the language you want to use for your Lambda function. The appropriate section appears with code and specific instructions for creating a deployment package.

Node.js

The following is example Node.js code that receives Amazon Kinesis event records as input and processes them. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

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

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

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

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

Note
The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)

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

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 198)

Java

The following is example Java code that receives Amazon Kinesis event record data as a input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `recordHandler` is the handler. The handler uses the predefined `KinesisEvent` class that is 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.KinesisEventRecord;

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 without exceptions, 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.

Using the preceding code (in a file named `ProcessKinesisEvents.java`), create a deployment package. Make sure that you add the following dependencies:

- `aws-lambda-java-core`
- `aws-lambda-java-events`

For more information, see Programming Model for Authoring Lambda Functions in Java (p. 21).

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. For examples of how to use the Maven build tool to create a standalone .jar, see Creating a .jar Deployment Package Using Maven without any IDE (Java) (p. 62) and Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java) (p. 65). For an example of how to use the Gradle build tool to create a .zip file, see Creating a .zip Deployment Package (Java) (p. 67).

After you verify that your deployment package is created, go to the next step to create an IAM role (execution role). You specify this role at the time you create your Lambda function.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 198)

C#

The following is example C# code that receives Amazon Kinesis event record data as a input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `HandleKinesisRecord` is the handler. The handler uses the predefined `KinesisEvent` class that is defined in the `Amazon.Lambda.KinesisEvents` library.

```

using System;
using System.IO;
using System.Text;

using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;

namespace KinesisStreams
{
    public class KinesisSample
    {
     [LambdaSerializer(typeof(JsonSerializer))]
        public void HandleKinesisRecord(KinesisEvent kinesisEvent)
        {
            Console.WriteLine($"Beginning to process
 {kinesisEvent.Records.Count} records...");

            foreach (var record in kinesisEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventId}");
                Console.WriteLine($"Event Name: {record.EventName}");

                string recordData = GetRecordContents(record.Kinesis);
                Console.WriteLine($"Record Data:");
                Console.WriteLine(recordData);
            }

            Console.WriteLine("Stream processing complete.");
        }

        private string GetRecordContents(KinesisEvent.Record streamRecord)
        {
            using (var reader = new StreamReader(streamRecord.Data,
 Encoding.ASCII))
            {
                return reader.ReadToEnd();
            }
        }
    }
}
```

To create a deployment package, follow the steps outlined in .NET Core CLI (p. 57). In doing so, note the following after you've created your .NET project:

- Rename the default *Program.cs file* with a file name of your choice, such as *ProcessingKinesisEvents.cs*.
- Replace the default contents of the renamed *Program.cs* file with the code example above.
- In the *project.json* file, make sure the following references are included in the `dependencies` node.
  - `"Amazon.Lambda.Core": "1.0.0-*"`
  - `"Amazon.Lambda.KinesisEvents":"1.0.0-*"`
  - `"Amazon.Lambda.Serialization.Json":"1.0.0-*"`

After you verify that your deployment package is created, go to the next step to create an IAM role (execution role). You specify this role at the time you create your Lambda function.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 198)

Python

The following is example Python code that receives Amazon Kinesis event record data as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

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

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

```
from __future__ import print_function
#import json
import base64
def lambda_handler(event, context):
    for record in event['Records']:
       #Kinesis data is base64 encoded so decode here
       payload=base64.b64decode(record["kinesis"]["data"])
       print("Decoded payload: " + payload)
```

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

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 198)

Step 2.2: Create the Execution Role (IAM Role)

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

- AWS service role of the type **AWS Lambda** – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaKinesisExecutionRole** – This is the access permissions policy that you attach to the role.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-kinesis-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - In **Attach Policy**, choose **AWSLambdaKinesisExecutionRole**. The permissions in this policy are sufficient for the Lambda function in this tutorial.
3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Next Step

Step 2.3: Create the Lambda Function and Test It Manually (p. 199)

Step 2.3: Create the Lambda Function and Test It Manually

In this section, you do the following:

- Create a Lambda function by uploading the deployment package.
- Test the Lambda function by invoking it manually. Instead of creating an event source, you use sample Amazon Kinesis event data.

In the next section, you create an Amazon Kinesis stream and test the end-to-end experience.

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

In this step, you upload the deployment package using the AWS CLI.

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

You need to update the command by providing the .zip file path and the execution role ARN. The `--runtime` parameter value can be `python2.7`, `nodejs4.3`, or `java8`, depending on the language you used to author your code.

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

The `--handler` parameter value for Java should be `example.ProcessKinesisRecords::recordHandler`. For Node.js, it should be `ProcessKinesisRecords.handler` and for Python it should be `ProcessKinesisRecords.lambda_handler`.

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

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

Note
You can create the Lambda function using the AWS Lambda console, in which case note the value of the `create-function` AWS CLI command parameters. You provide the same values in the console UI.

Step 2.3.2: Test the Lambda Function (Invoke Manually)

Invoke the function manually using sample Amazon Kinesis event data. We recommend that you invoke the function using the console because the console UI provides a user-friendly interface for reviewing the execution results, including the execution summary, logs written by your code, and the results returned by the function (because the console always performs synchronous execution—invokes the Lambda function using the `RequestResponse` invocation type).

To test the Lambda function (console)

1. Follow the steps in the Getting Started to create and invoke the Lambda function at Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167). For the sample event for testing, choose **Kinesis** in **Sample event template**.
2. Verify the results in the console.

To test the Lambda function (AWS CLI)

1. Copy the following JSON into a file and save it as `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::account-id:role/
testLEBRole",
            "eventVersion": "1.0",
            "eventName": "aws:kinesis:record",
            "eventSourceARN": "arn:aws:kinesis:us-
west-2:35667example:stream/examplestream",
            "awsRegion": "us-west-2"
        }
    ]
}
```

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
In this tutorial example, the message is saved in the `outputfile.txt` file. If you request synchronous execution (`RequestResponse` as the invocation type), the function returns the string message in the response body.
For Node.js, it could be one of the following (whatever one you specify in the code):
`context.succeed("message")`
`context.fail("message")`
`context.done(null, "message)`
For Python or Java, it is the message in the return statement:

```
        return "message"
```

Next Step

Step 3: Add an Event Source (Create an Amazon Kinesis Stream and Associate It with Your Lambda Function) (p. 201)

Step 3: Add an Event Source (Create an Amazon Kinesis Stream and Associate It with Your Lambda Function)

In this section, you create an Amazon Kinesis stream, and then you add an event source in AWS Lambda to associate the Amazon Kinesis stream with your Lambda function.

After you create an event source, AWS Lambda starts polling the stream. You 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` AWS 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
```

You can get a list of event source mappings by running the following command.

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

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

> **Note**
> If you disable the event source mapping, AWS Lambda stops polling the Amazon Kinesis stream. If you re-enable event source mapping, it will resume polling from the sequence number where it stopped, so each record is processed either before you disabled the mapping or after you enabled it. If the sequence number falls behind `TRIM_HORIZON`, when you re-enable it polling will start from `TRIM_HORIZON`. However, if you create a new event source mapping, polling will always start from `TRIM_HORIZON`, `LATEST` or `AT_TIMESTAMP`, depending on the starting position you specify. This applies even if you delete an event source mapping and create a new one with the same configuration as the deleted one.

Step 3.3: Test the Setup

You're 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 invokes your Lambda function by passing in the event data from the stream.

   AWS Lambda assumes the execution role to poll the stream. You have granted the role permissions for the necessary Amazon Kinesis actions so that 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.

Step 4: Deploy With AWS SAM and AWS CloudFormation

In the previous section, you used AWS Lambda APIs to create and update a Lambda function by providing a deployment package as a ZIP file. However, this mechanism may not be convenient for automating deployment steps for functions, or coordinating deployments and updates to other elements of a serverless application, like event sources and downstream resources.

You can use AWS CloudFormation to easily specify, deploy, and configure serverless applications. AWS CloudFormation is a service that helps you model and set up your Amazon Web Services resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Lambda functions and DynamoDB tables), and AWS CloudFormation takes care of provisioning and configuring those resources for you.

In addition, you can use the AWS Serverless Application Model to express resources that comprise the serverless application. These resource types, such as Lambda functions and APIs, are fully supported by AWS CloudFormation and make it easier for you to define and deploy your serverless application.

For more information, see Deploying Lambda-based Applications (p. 139).

Specification for Amazon Kinesis Application

The following contains the SAM template for this application. Copy the text below to a .yaml file and save it next to the ZIP package you created previously.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  ProcessKinesisRecords:
    Type: AWS::Serverless::Function
    Properties:
      Handler: ProcessKinesisRecords.handler
      Runtime: nodejs4.3
      Policies: AWSLambdaKinesisExecutionRole
      Events:
        Stream:
          Type: Kinesis
          Properties:
            Stream: !GetAtt ExampleStream.Arn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON

  ExampleStream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 1
```

Deploying the Serverless Application

For information on how to package and deploy your serverless application using the package and deploy commands, see Packaging and Deployment (p. 145).

Using AWS Lambda with Amazon DynamoDB

You can use Lambda functions as triggers for your Amazon DynamoDB table. Triggers are custom actions you take in response to updates made to the DynamoDB table. To create a trigger, first you enable Amazon DynamoDB Streams for your table. Then, you write a Lambda function to process the updates published to the stream.

Note the following about how the Amazon DynamoDB and AWS Lambda integration works:

- **Stream-based model** – This is a model (see Event Source Mapping (p. 120)), where AWS Lambda polls the stream and, when it detects new records, invokes your Lambda function by passing the update event as parameter.

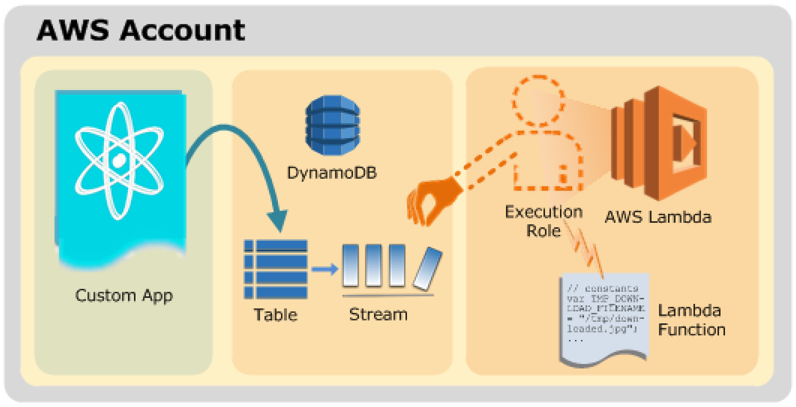
  In a stream-based model, you maintain event source mapping in AWS Lambda. The event source mapping describes which stream maps to which Lambda function. AWS Lambda provides an API

(CreateEventSourceMapping (p. 328)) for you to create the mapping. You can also the AWS Lambda console to create event source mappings.

- **Synchronous invocation** – AWS Lambda invokes a Lambda function using the `RequestResponse` invocation type (synchronous invocation). For more information about invocation types, see Invocation Types (p. 4).
- **Event structure** – The event your Lambda function receives is the table update information AWS Lambda reads from your stream. When you configure event source mapping, the batch size you specify is the maximum number of records that you want your Lambda function to receive per invocation.

Regardless of what invokes a Lambda function, AWS Lambda always executes a Lambda function on your behalf. If your Lambda function needs to access any AWS resources, you need to grant the relevant permissions to access those resources. You also need to grant AWS Lambda permissions to poll your DynamoDB stream. You grant all of these permissions to an IAM role (execution role) that AWS Lambda can assume to poll the stream and execute the Lambda function on your behalf. You create this role first and then enable it at the time you create the Lambda function. For more information, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).

The following diagram illustrates the application flow:



1. Custom app updates the DynamoDB table.
2. Amazon DynamoDB publishes item updates to the stream.
3. AWS Lambda polls the stream and invokes your Lambda function when it detects new records in the stream.
4. AWS Lambda executes the Lambda function by assuming the execution role you specified at the time you created the Lambda function.

For a tutorial that walks you through an example setup, see Tutorial: Using AWS Lambda with Amazon DynamoDB (p. 204).

Tutorial: Using AWS Lambda with Amazon DynamoDB

In this tutorial, you create a Lambda function to consume events from a DynamoDB stream.

The tutorial is divided into two main sections:

- First, you perform the necessary setup to create a Lambda function and then you test it by invoking it manually using sample event data.

- Second, you create an DynamoDB stream-enabled table and add an event source mapping in AWS Lambda to associate the stream with your Lambda function. AWS Lambda starts polling the stream. Then, you test the end-to-end setup. As you create, update, and delete items from the table, Amazon DynamoDB writes records to the stream. AWS Lambda detects the new records as it polls the stream and executes your Lambda function on your behalf.

  Important
  Both the Lambda function and the DynamoDB stream must be in the same AWS region. This tutorial assumes that you create these resources in the `us-east-1` region.

In this tutorial, you use the AWS Command Line Interface to perform AWS Lambda operations such as creating a Lambda function, creating a stream, and adding records to the stream. You use the AWS Lambda console to manually invoke the function before you create a DynamoDB stream. You verify return values and logs in the console UI.

Next Step

Step 1: Prepare (p. 205)

Step 1: Prepare

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account.
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Next Step

Step 2: Create a Lambda Function and Invoke It Manually (Using Sample Event Data) (p. 205)

Step 2: Create a Lambda Function and Invoke It Manually (Using Sample Event Data)

In this section, you do the following:

- Create a Lambda function deployment package using the sample code provided. The sample Lambda function code that you'll use to process DynamoDB events is provided in various languages. Select one of the languages and follow the corresponding instructions to create a deployment package.

  Note
  To see more examples of using other AWS services within your function, including calling other Lambda functions, see AWS SDK for JavaScript
- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify an IAM role (execution role) that Lambda can assume to execute the function on your behalf. For example, AWS Lambda needs permissions for DynamoDB actions so it can poll the stream and read records from the stream. In the *pull* model you must also grant AWS Lambda permissions to invoke your Lambda function. The example Lambda function writes some of the event data to CloudWatch, so it needs permissions for necessary CloudWatch actions.
- Create the Lambda function by uploading the deployment package, and then test it by invoking it manually using sample DynamoDB event data. 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 to use, and the handler. For more information about these parameters, see CreateFunction (p. 332). After creating the Lambda function, you invoke it using sample Amazon DynamoDB event data.

Topics

Step 2.1: Create a Lambda Function Deployment Package

From the **Filter View** list, choose the language you want to use for your Lambda function. The appropriate section appears with code and specific instructions for creating a deployment package.

Node.js

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

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

exports.lambda_handler = function(event, context, callback) {
    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);
    });
    callback(null, "message");
};
```

Note
The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)

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

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 209)

Java

In the following 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.DynamodbStreamRecord;
```

```
public class DDBEventProcessor implements
        RequestHandler<DynamodbEvent, String> {

    public String handleRequest(DynamodbEvent ddbEvent, Context context) {

        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
           System.out.println(record.getEventID());
           System.out.println(record.getEventName());
           System.out.println(record.getDynamodb().toString());

        }
        return "Successfully processed " + ddbEvent.getRecords().size() + "
 records.";
    }
}
```

If the handler returns normally without exceptions, 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.

Using the preceding code (in a file named `DDBEventProcessor.java`), create a deployment package. Make sure that you add the following dependencies:

- `aws-lambda-java-core`
- `aws-lambda-java-events`

For more information, see Programming Model for Authoring Lambda Functions in Java (p. 21).

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. For examples of how to use the Maven build tool to create a standalone .jar, see Creating a .jar Deployment Package Using Maven without any IDE (Java) (p. 62) and Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java) (p. 65). For an example of how to use the Gradle build tool to create a .zip file, see Creating a .zip Deployment Package (Java) (p. 67).

After you verify that your deployment package is created, go to the next step to create an IAM role (execution role). You specify this role at the time you create your Lambda function.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 209)

C#

In the following code, `ProcessDynamoEvent` is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined `DynamoDbEvent` class, which is defined in the `Amazon.Lambda.DynamoDBEvents` library.

```


using System;
using System.IO;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

using Amazon.Lambda.Serialization.Json;
```

```
namespace DynamoDBStreams
{
    public class DdbSample
    {
        private static readonly JsonSerializer _jsonSerializer = new
 JsonSerializer();

        public void ProcessDynamoEvent(DynamoDBEvent dynamoEvent)
        {
            Console.WriteLine($"Beginning to process
 {dynamoEvent.Records.Count} records...");

            foreach (var record in dynamoEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventID}");
                Console.WriteLine($"Event Name: {record.EventName}");

                string streamRecordJson = SerializeObject(record.Dynamodb);
                Console.WriteLine($"DynamoDB Record:");
                Console.WriteLine(streamRecordJson);
            }

            Console.WriteLine("Stream processing complete.");
        }

        private string SerializeObject(object streamRecord)
        {
            using (var ms = new MemoryStream())
            {
                _jsonSerializer.Serialize(streamRecord, ms);
                return Encoding.UTF8.GetString(ms.ToArray());
            }
        }
    }
}
```

To create a deployment package, follow the steps outlined in .NET Core CLI (p. 57). In doing so, note the following after you've created your .NET project:

- Rename the default *Program.cs file* with a file name of your choice, such as *ProcessingDynamoDBStreams.cs*.
- Replace the default contents of the renamed *Program.cs* file with the code example above.
- In the *project.json* file, add the following references to the `dependencies` node.
  - `"Amazon.Lambda.Core":"1.0.0-*"`
  - `"Amazon.Lambda.Serialiation.Json":"1.0.0-*"`
  - `"Amazon.Lambda.DynamoDBEvents":"1.0.0-*"`

After you verify that your deployment package is created, go to the next step to create an IAM role (execution role). You specify this role at the time you create your Lambda function.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 209)

Python

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

```
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print(record['eventID'])
        print(record['eventName'])
    print('Successfully processed %s records.' %
 str(len(event['Records'])))
```

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

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 209)

Step 2.2: Create the Execution Role (IAM Role)

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

- AWS service role of the type **AWS Lambda** – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaDynamoDBExecutionRole** – This is the access permissions policy that you attach to the role.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-dynamodb-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - In **Attach Policy**, choose **AWSLambdaDynamoDBExecutionRole**. The permissions in this policy are sufficient for the Lambda function in this tutorial.
3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Next Step

Step 2.3: Create the Lambda Function and Test It Manually (p. 209)

Step 2.3: Create the Lambda Function and Test It Manually

In this section, you do the following:

- Create a Lambda function by uploading the deployment package.
- Test the Lambda function by invoking it manually. Instead of creating an event source, you use sample DynamoDB event data.

In the next section, you create an DynamoDB stream and test the end-to-end experience.

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

In this step, you upload the deployment package using the AWS CLI.

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

You need to update the command by providing the .zip file path and the execution role ARN. The `--runtime` parameter value can be `python2.7`, `nodejs4.3`, or `java8`, depending on the language you used to author your code.

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

> **Note**
> If you choose `Java 8` as the runtime, the handler value must be `packageName::methodName`.

For more information, see CreateFunction (p. 332). AWS Lambda creates the function and returns function configuration information.

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

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

Step 2.3.2: Test the Lambda Function (Invoke Manually)

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

1. Copy the following JSON into a file and save it as `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 that the `invoke` command specifies the `RequestResponse` as the invocation type, which requests synchronous execution. For more information, see Invoke (p. 358). The function returns the string message (message in the `context.succeed()` in the code) in the response body.

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

   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 (Create a DynamoDB Stream and Associate It with Your Lambda Function) (p. 212)

Step 3: Add an Event Source (Create a DynamoDB Stream and Associate It with Your Lambda Function)

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 associates the DynamoDB stream with your Lambda function. After you create this event source mapping, AWS Lambda starts polling the stream.
- Test the end-to-end experience. As you perform table updates, DynamoDB writes event records to the stream. As AWS Lambda polls the stream, it detects new records in the stream and executes 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 and open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
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 where you created the Lambda function. This tutorial assumes 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 need this in the next step when you associate the stream with your Lambda function.

Step 3.2: Add an Event Source 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 by running the following command.

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

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

Step 3.3: Test the Setup

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

1. In the DynamoDB console, add, update, delete items to the table. DynamoDB writes records of these actions to the stream.
2. AWS Lambda polls the stream and when it detects updates to the stream, it invokes your Lambda function by passing in the event data it finds 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.

Step 4: Deploy With AWS SAM and AWS CloudFormation

In the previous section, you used AWS Lambda APIs to create and update a Lambda function by providing a deployment package as a ZIP file. However, this mechanism may not be convenient for automating deployment steps for functions, or coordinating deployments and updates to other elements of a serverless application, like event sources and downstream resources.

You can use AWS CloudFormation to easily specify, deploy, and configure serverless applications. AWS CloudFormation is a service that helps you model and set up your Amazon Web Services resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Lambda functions and DynamoDB tables), and AWS CloudFormation takes care of provisioning and configuring those resources for you.

In addition, you can use the AWS Serverless Application Model to express resources that comprise the serverless application. These resource types, such as Lambda functions and APIs, are fully supported by AWS CloudFormation and make it easier for you to define and deploy your serverless application.

For more information, see Deploying Lambda-based Applications (p. 139).

Specification for DynamoDB Application

The following contains the SAM template for this application. Copy the text below to a .yaml file and save it next to the ZIP package you created previously.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  ProcessDynamoDBStream:
    Type: AWS::Serverless::Function
    Properties:
      Handler: ProcessDynamoDBStream.handler
      Runtime: nodejs4.3
      Policies: AWSLambdaDynamoDBExecutionRole
      Events:
        Stream:
          Type: DynamoDB
          Properties:
            Stream: !GetAtt DynamoDBTable.StreamArn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON

  DynamoDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
      StreamSpecification:
        StreamViewType: NEW_IMAGE
```

Deploying the Serverless Application

For information on how to package and deploy your serverless application using the package and deploy commands, see Packaging and Deployment (p. 145).

Using AWS Lambda with AWS CloudTrail

You can enable CloudTrail in your AWS account to get logs of API calls and related events history in your account. CloudTrail records all of the API access events as objects in your Amazon S3 bucket that you specify at the time you enable CloudTrail.

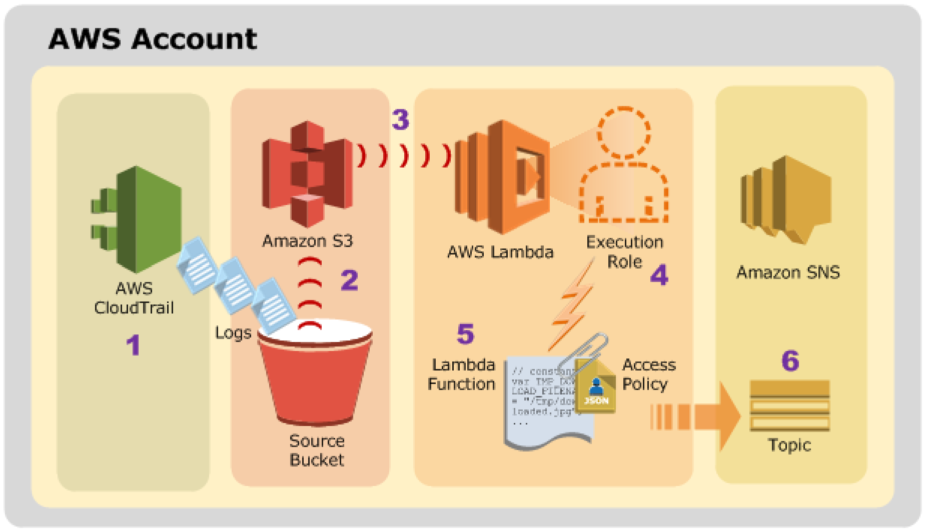
You can take advantage of Amazon S3's bucket notification feature and direct Amazon S3 to publish object-created events to AWS Lambda. Whenever CloudTrail writes logs to your S3 bucket, Amazon S3 can then invoke your Lambda function by passing the Amazon S3 object-created event as a parameter. The S3 event provides information, including the bucket name and key name of the log object that CloudTrail created. Your Lambda function code can read the log object and process the access records logged by CloudTrail. For example, you might write Lambda function code to notify you if specific API call was made in your account.

In this scenario, you enable CloudTrail so it can write access logs to your S3 bucket. As for AWS Lambda, Amazon S3 is the event source so Amazon S3 publishes events to AWS Lambda and invokes your Lambda function.

Note
Amazon S3 can only support one event destination.

For detailed information about how to configure Amazon S3 as the event source, see Using AWS Lambda with Amazon S3 (p. 175).

The following diagram summarizes the flow:



1. AWS CloudTrail saves logs to an S3 bucket (object-created event).
2. Amazon S3 detects the object-created event.

3. Amazon S3 publishes the `s3:ObjectCreated:*` event to AWS Lambda by invoking the Lambda function, as specified in the bucket notification configuration. Because the Lambda function's access permissions policy includes permissions for Amazon S3 to invoke the function, Amazon S3 can invoke the function.
4. AWS Lambda executes the Lambda function by assuming the execution role that you specified at the time you created the Lambda function.
5. The Lambda function reads the Amazon S3 event it receives as a parameter, determines where the CloudTrail object is, reads the CloudTrail object, and then it processes the log records in the CloudTrail object.
6. If the log includes a record with specific `eventType` and `eventSource` values, it publishes the event to your Amazon SNS topic. In Tutorial: Using AWS Lambda with AWS CloudTrail (p. 216), you subscribe to the SNS topic using the email protocol, so you get email notifications.

For a tutorial that walks you through an example scenario, see Tutorial: Using AWS Lambda with AWS CloudTrail (p. 216).

Tutorial: Using AWS Lambda with AWS CloudTrail

Suppose you have turned on AWS CloudTrail for your AWS account to maintain records (logs) of AWS API calls made on your account and you want to be notified anytime an API call is made to create an SNS topic. As API calls are made in your account, CloudTrail writes logs to an Amazon S3 bucket that you configured. In this scenario, 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 passes an S3 event identifying, among other things, the bucket name and key name of the object that CloudTrail created. Your Lambda function can read the log object, and it knows the API calls that were reported in the log.

Each object CloudTrail creates in your S3 bucket 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",
             ...
         },
         {
            ...
         },
         ...
}
```

For illustration, the Lambda function notifies you by email if an API call to create an Amazon SNS topic is reported in the log. That is, when your Lambda function parses the log, it looks for records with the following:

- `eventSource = "sns.amazonaws.com"`
- `eventName = "CreateTopic"`

If found, it publishes the event to your Amazon SNS topic (you configure this topic to notify you by email).

Implementation Summary

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

> **Note**
> This tutorial assumes that you create these resources in the `us-west-2` region.

In Lambda:

- A Lambda function.
- An access policy associated with your Lambda function – You grant Amazon S3 permissions to invoke the Lambda function using this permissions policy. You will also restrict the permissions so that Amazon S3 can invoke the Lambda function only for object-created events from a specific bucket that is 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 How It Works (p. 151).

In IAM:

- An IAM role (execution role) – You grant permissions that your Lambda function needs through the permissions policy associated with this role.

In Amazon S3:

- A bucket – In this tutorial, the bucket name is *`examplebucket`*. When you turn the trail on in the CloudTrail console, you specify this bucket for CloudTrail to save the logs.
- Notification configuration on the *`examplebucket`* – In the configuration, you direct Amazon S3 to publish object-created events to Lambda, by invoking your Lambda function. For more information about the Amazon S3 notification feature, see Setting Up Notification of Bucket Events in *Amazon Simple Storage Service Developer Guide*.
- Sample CloudTrail log object (`ExampleCloudTrailLog.json`) in *`examplebucket`* bucket – In the first half of this exercise, you create and test your Lambda function by manually invoking it using a sample S3 event. This sample event identifies *`examplebucket`* as the bucket name and this sample object key name. Your Lambda function then reads the object and sends you email notifications using an SNS topic.

In Amazon SNS

- An SNS topic – You subscribe to this topic by specifying email as the protocol.

Now you are ready to start the tutorial.

Next Step

Step 1: Prepare (p. 218)

Step 1: Prepare

In this section you do the following:

- Sign up for an AWS account and set up the AWS CLI.
- Turn on CloudTrail in your account.
- Create an SNS topic and subscribe to it.

Follow the steps in the following sections to walk through the setup process.

> **Note**
> In this tutorial, we assume that you are setting the resources in the `us-west-2` region.

Step 1.1: Sign Up for AWS and Set Up the AWS CLI

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account (called **adminuser**).
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Step 1.2: 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, see Creating and Updating Your Trail in the *AWS CloudTrail User Guide*.

> **Note**
> Although you turn CloudTrail on now, you do not perform any additional configuration for your Lambda function to process the real CloudTrail logs in the first half of this exercise. Instead, you will use sample CloudTrail log objects (that you will upload) and sample S3 events to manually invoke and test your Lambda function. In the second half of this tutorial, you perform additional configuration steps that enable your Lambda function to process the CloudTrail logs.

Step 1.3: 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, see 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, see 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 a Lambda Function and Invoke It Manually (Using Sample Event Data) (p. 219)

Step 2: Create a Lambda Function and Invoke It Manually (Using Sample Event Data)

In this section, you do the following:

- Create a Lambda function deployment package using the sample code provided. The sample Lambda function code that you'll use to process Amazon S3 events is provided in various languages. Select one of the languages and follow the corresponding instructions to create a deployment package.

  Note
  Your Lambda function uses an S3 event that provides the bucket name and key name of the object CloudTrail created. Your Lambda function then reads that object to process CloudTrail records.
- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify an IAM role (execution role) that Lambda can assume to execute the function on your behalf.
- Create the Lambda function by uploading the deployment package, and then test it by invoking it manually using sample CloudTrail event data.

Topics

Step 2.1: Create a Deployment Package

The deployment package is a .zip file containing your Lambda function code. For this tutorial, you will need to install the `async` library. To do this, open a command window and navigate to the directory where you intend to store the code file file you will copy and save below. Use *npm* to install the async library as shown below :

```
npm install async
```

Node.js

1. Open a text editor, and then 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       = 'The ARN of your SNS topic';

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

```
exports.handler = function(event, context, callback) {
    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 + '
 notification(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.');
        }
        callback(null,"message");
    });
};
```

Note
The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)

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

Note
We're using Node.js in this tutorial example, but you can author your Lambda functions in Java or Python too.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 221)

Step 2.2: Create the Execution Role (IAM Role)

Now you create an IAM role (execution role) that you specify when creating your Lambda function. This role has a permissions policy that grant the necessary permissions that your Lambda function needs, such as permissions to write CloudWatch logs, permissions to read CloudTrail log objects from an S3 bucket, and permissions to publish events to your SNS topic when your Lambda function finds specific API calls in the CloudTrail records.

For more information about the execution role, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Create a managed policy and attach it to the IAM role. In this step, you modify an existing AWS Managed Policy, save it using a different name, and then attach the permissions policy to an IAM role that you create.
   a. In the navigation pane of the IAM console, choose **Policies**, and then choose **Create Policy**.
   b. Next to **Copy an AWS Managed Policy**, choose **Select**.
   c. Next to **AWSLambdaExecute**, choose **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 that you created.

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

3. Note the permissions policy name because you will use it in the next step.
4. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role and then attach the permissions policy you just created to the role. As you follow the steps to create a role, note the following:

   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-cloudtrail-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**.
   - In **Attach Policy**, choose the policy you created in the previous step.

Next Step

Step 2.3: Create the Lambda Function and Test It Manually (p. 222)

Step 2.3: Create the Lambda Function and Test It Manually

In this section, you do the following:

- Create a Lambda function by uploading the deployment package.
- Test the Lambda function by invoking it manually.

  In this step, you use a sample S3 event that identifies your bucket name and the sample object (that is, an example CloudTrail log). In the next section you configure your S3 bucket notification to publish object-created events and test the end-to-end experience.

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

In this step, you upload the deployment package using the AWS CLI and provide configuration information when you create the Lambda function. At the command prompt, run the following Lambda CLI `create-function` command using the `adminuser profile`.

> **Note**
> You need to update the command by providing the .zip file path (*`//file-path/CloudTrailEventProcessing.zip \`*) and the execution role ARN (*`execution-role-arn`*). If you used the sample code provided earlier in this tutorial, set the `--runtime` parameter value to `node.js`.
> You can author your Lambda functions in Java or Python too. If you use another language, change the `--runtime` parameter value to `java8` or `python2.7` as needed.

```
$ 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 nodejs4.3 \
--profile adminuser \
--timeout 10 \
--memory-size 1024
```

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter as shown:

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

Note
You can create the Lambda function using the AWS Lambda console, in which case note the value of the `create-function` AWS CLI command parameters. You provide the same values in the console.

Step 2.3.2: Test the Lambda Function (Invoke Manually)

In this section, you invoke the Lambda function manually using sample Amazon S3 event data. When the Lambda function executes, it reads the S3 object (a sample CloudTrail log) from the bucket identified in the S3 event data, and then it publishes an event to your SNS topic if the sample CloudTrail log reports use a specific API. For this tutorial, 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. Save the following sample CloudTrail log to a file (`ExampleCloudTrailLog.json`).

   Note
   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 an event of this type, it publishes the event to the Amazon SNS topic that you created and then you 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"
       }
    ]
}
```

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

   ```
   $ gzip ExampleCloudTrailLog.json
   ```

   This creates `ExampleCloudTrailLog.json.gz` file.

3. Upload the `ExampleCloudTrailLog.json.gz` file to the *`examplebucket`* that you specified in the CloudTrail configuration.

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

4. Save the following JSON (an example S3 event) in a file, `input.txt`. Note the bucket name and the object key name values.

   You provide this sample event when you invoke your Lambda function. For more information about the S3 event structure, see 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/
JRWeUWerMUE5JgHvANOjpD"
            },
            "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"
                }
            }
        }
    ]
}
```

5. In the AWS Management Console, invoke the function manually using sample Amazon S3 event data. For instructions, see the Getting Started exercise Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167). In the console, use the following sample Amazon S3 event data.

   Note
   We recommend that you invoke the function using the console because the console UI provides a user-friendly interface for reviewing the execution results, including the execution summary, logs written by your code, and the results returned by the function

(because the console always performs synchronous execution—invokes the Lambda function using the `RequestResponse` invocation type).

```
{
    "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/
JRWeUWerMUE5JgHvANOjpD"
            },
            "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"
                }
            }
        }
    ]
}
```

6. Execute the following AWS CLI command to invoke the function manually using the `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 posts 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. 107).

7. (Optional) Manually invoke the Lambda function using AWS CLI as follows:

   a. Save the JSON from Step 2 earlier in this procedure to a file called `input.txt`.
   b. Execute the following invoke command:

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

Note
In this tutorial example, the message is saved in the `outputfile.txt` file. If you request synchronous execution (`RequestResponse` as the invocation type), the function returns the string message in the response body.
For Node.js, it could be one of the following (whatever one you specify in the code):
`context.succeed("message")`
`context.fail("message")`
`context.done(null, "message)`
For Python or Java, it is the message in the return statement:
`return "message"`

Next Step

Step 3: Add Event Source (Configure Amazon S3 to Publish Events) (p. 227)

Step 3: Add Event Source (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 permissions 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 following:
  - Event type for which you want Amazon S3 to publish events. For this tutorial, you specify the `s3:ObjectCreated:*` event type.
  - Lambda function to invoke.

Step 3.1: Add Permissions to the Lambda Function's Access Permissions Policy

1. Run the following Lambda CLI `add-permission` command to grant Amazon S3 service principal (`s3.amazonaws.com`) permissions to perform 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 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 function's access policy by running the AWS CLI `get-policy` command.

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

Step 3.2: Configure Notification on the 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 tutorial, these can be any event types that create objects.
- Lambda function ARN – This is your Lambda function that you want Amazon S3 to 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 has the following ARN:

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

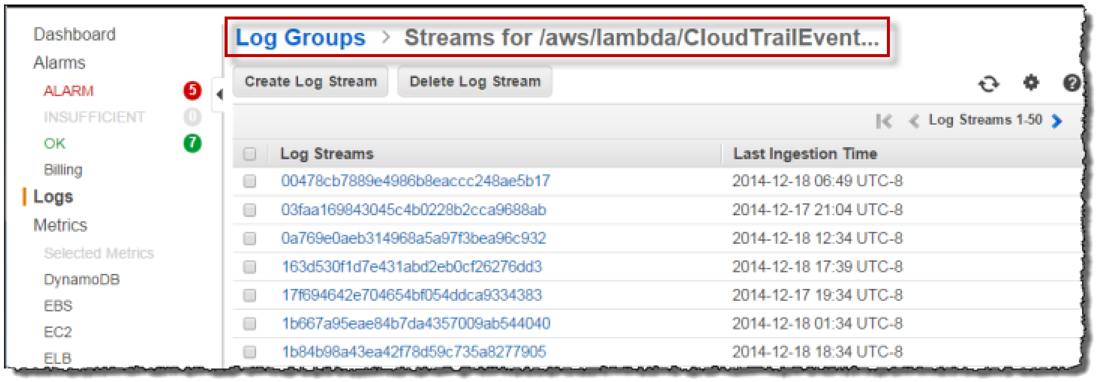
For instructions on adding notification configuration to a bucket, see Enabling Event Notifications in the *Amazon Simple Storage Service Console User Guide*.

Step 3.3: Test the Setup

You're all done! Now you can 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 receive an email notification about this event.
3. AWS CloudTrail creates a log object in your bucket.
4. If you open the log object (.gz file), the log shows the `CreateTopic` SNS event.
5. For each object AWS CloudTrail creates, Amazon S3 invokes your Lambda function by passing in the log object as event data.
6. Lambda executes your function. The function parses the log, finds a `CreateTopic` SNS event, and then you receive 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. 107).



Log Streams	Last Ingestion Time
00478cb7889e4986b8eaccc248ae5b17	2014-12-18 06:49 UTC-8
03faa169843045c4b0228b2cca9688ab	2014-12-17 21:04 UTC-8
0a769e0aeb314968a5a97f3bea96c932	2014-12-18 12:34 UTC-8
163d530f1d7e431abd2eb0cf26276dd3	2014-12-18 17:39 UTC-8
17f694642e704654bf054ddca9334383	2014-12-17 19:34 UTC-8
1b667a95eae84b7da4357009ab544040	2014-12-18 01:34 UTC-8
1b84b98a43ea42f78d59c735a8277905	2014-12-18 18:34 UTC-8

Step 4: Deploy With AWS SAM and AWS CloudFormation

In the previous section, you used AWS Lambda APIs to create and update a Lambda function by providing a deployment package as a ZIP file. However, this mechanism may not be convenient for automating deployment steps for functions, or coordinating deployments and updates to other elements of a serverless application, like event sources and downstream resources.

You can use AWS CloudFormation to easily specify, deploy, and configure serverless applications. AWS CloudFormation is a service that helps you model and set up your Amazon Web Services resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Lambda functions and DynamoDB tables), and AWS CloudFormation takes care of provisioning and configuring those resources for you.

In addition, you can use the AWS Serverless Application Model to express resources that comprise the serverless application. These resource types, such as Lambda functions and APIs, are fully supported by AWS CloudFormation and make it easier for you to define and deploy your serverless application.

For more information, see Deploying Lambda-based Applications (p. 139).

Specification for Amazon API Gateway Application

The following contains the SAM template for this application. Copy the text below to a .yaml file and save it next to the ZIP package you created in the previous section.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Parameters:
  NotificationEmail:
    Type: String
Resources:
  CloudTrailEventProcessing:
    Type: AWS::Serverless::Function
    Properties:
      Handler: CloudTrailEventProcessing.handler
      Runtime: nodejs4.3
      Timeout: 10
      MemorySize: 1024
      Policies:
      - Effect: Allow
```

```
        Action: s3:GetObject
        Resource: !Sub 'arn:aws:s3:::${Bucket}/*'
      - Effect: Allow
        Action: sns:Publish
        Resource: !Ref Topic
      Events:
        PhotoUpload:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
      Environment:
        Variables:
          SNS_TOPIC_ARN: !Ref Topic

  Bucket:
    Type: AWS::S3::Bucket

  Trail:
    Type: AWS::CloudTrail::Trail
    Properties:
      IsLogging: true
      S3BucketName: !Ref Bucket

  Topic:
    Type: AWS::SNS::Topic
    Properties:
      Subscription:
      - Protocol: email
        Endpoint: !Ref NotificationEmail
```

Deploying the Serverless Application

For information on how to package and deploy your serverless application using the package and deploy commands, see Packaging and Deployment (p. 145).

Using AWS Lambda with Amazon SNS from Different Accounts

In order to perform cross account Amazon SNS deliveries to Lambda, you need to authorize your Lambda function to be invoked from Amazon SNS. In turn, Amazon SNS needs to allow the Lambda account to subscribe to the Amazon SNS topic. For example, if the Amazon SNS topic is in account A and the Lambda function is in account B, both accounts must grant permissions to the other to access their respective resources. Since not all the options for setting up cross-account permissions are available from the AWS console, you use the AWS CLI to set up the entire process.

For a tutorial that walks you through an example setup, see Tutorial: Using AWS Lambda with Amazon SNS (p. 230).

Tutorial: Using AWS Lambda with Amazon SNS

In this tutorial, you create a Lambda function in one AWS account to subscribe to an Amazon SNS topic in a separate AWS account.

The tutorial is divided into three main sections:

- First, you perform the necessary setup to create a Lambda function.
- Second, you create an Amazon SNS topic in a separate AWS account.
- Third, you grant permissions from each account in order for the Lambda function to subscribe to the Amazon SNS topic. Then, you test the end-to-end setup.

Important
This tutorial assumes that you create these resources in the `us-east-1` region.

In this tutorial, you use the AWS Command Line Interface to perform AWS Lambda operations such as creating a Lambda function, creating an Amazon SNS topic and granting permissions to allow these two resources to access each other.

Next Step

Step 1: Prepare (p. 231)

Step 1: Prepare

- Sign up for an AWS account and create an administrator user in the account (called **adminuser**).
- Install and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Next Step

Step 2: Create a Lambda Function (p. 231)

Step 2: Create a Lambda Function

In this section, you do the following:

- Create a Lambda function deployment package using the sample code provided. The sample Lambda function code that you'll use to subscribe to an Amazon SNS topic is provided in various languages. Select one of the languages and follow the corresponding instructions to create a deployment package.
- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify an IAM role (execution role) that Lambda can assume to execute the function on your behalf.

Topics

Step 2.1: Create a Lambda Function Deployment Package

From the **Filter View** list, choose the language you want to use for your Lambda function. The appropriate section appears with code and specific instructions for creating a deployment package.

Node.js

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

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

```
exports.handler = function(event, context, callback) {
// console.log('Received event:', JSON.stringify(event, null, 4));

    var message = event.Records[0].Sns.Message;
    console.log('Message received from SNS:', message);
    callback(null, "Success");
};
```

> **Note**
> The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)

2. Save the file as `index.js`.
3. Zip the `index.js` file as `LambdaWithSNS.zip`.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 233)

Java

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

```
package example;

import java.text.SimpleDateFormat;
import java.util.Calendar;

import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;

public class LogEvent implements RequestHandler<SNSEvent, Object> {
    public Object handleRequest(SNSEvent request, Context context){
    String timeStamp = new SimpleDateFormat("yyyy-MM-
dd_HH:mm:ss").format(Calendar.getInstance().getTime());
    context.getLogger().log("Invocation started: " + timeStamp);


 context.getLogger().log(request.getRecords().get(0).getSNS().getMessage());

    timeStamp = new SimpleDateFormat("yyyy-MM-
dd_HH:mm:ss").format(Calendar.getInstance().getTime());
    context.getLogger().log("Invocation completed: " + timeStamp);
           return null;
     }
}
```

Using the preceding code (in a file named `LambdaWithSNS.java`), create a deployment package. Make sure that you add the following dependencies:

- `aws-lambda-java-core`
- `aws-lambda-java-events`

For more information, see Programming Model for Authoring Lambda Functions in Java (p. 21).

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. For examples of how to use the Maven build tool to create a standalone .jar, see Creating a .jar Deployment Package Using Maven without any IDE (Java) (p. 62) and Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java) (p. 65). For an example of how to use the Gradle build tool to create a .zip file, see Creating a .zip Deployment Package (Java) (p. 67).

After you verify that your deployment package is created, go to the next step to create an IAM role (execution role). You specify this role at the time you create your Lambda function.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 233)

Python

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

```
from __future__ import print_function
import json
print('Loading function')

def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    message = event['Records'][0]['Sns']['Message']
    print("From SNS: " + message)
    return message
```

2. Save the file as `lambda_handler.py`.
3. Zip the `lambda_handler.py` file as `LambdaWithSNS.zip`.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 233)

Step 2.2: Create the Execution Role (IAM Role)

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

- AWS service role of the type **AWS Lambda** – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaBasicExecutionRole** – This is the access permissions policy that you attach to the role.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:

   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-sns-execution-role**).

- In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
- In **Attach Policy**, choose **AWSLambdaBasicExecutionRole**. The permissions in this policy are sufficient for the Lambda function in this tutorial.

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

Step 3: Set Up Cross-Account Permissions

In this section, you use CLI commands to set permissions across the Lambda function account and the Amazon SNS topic account and then test the subscription.

1. From account A, create the Amazon SNS topic:

```
aws sns create-topic \
    --name lambda-x-account
```

   Note the topic arn that is returned by the command. You will need it when you add permissions to the Lambda function to subscribe to the topic.

2. From account B, create the Lambda function. For the runtime parameter, select either `nodejs4.3`, `python2.7` or `java8`, depending on the code sample you selected when you created your deployment package.

```
aws lambda create-function \
    --function-name SNS-X-Account \
    --runtime runtime language \
    --role role arn \
    --handler index.handler \
    --description "SNS X Account Test Function" \
    --timeout 60 \
    --memory-size 128 \
    --zip-file fileb://path/LambdaWithSNS.zip
```

   Note the function arn that is returned by the command. You will need it when you add permissions to allow Amazon SNS to invoke your function.

3. From account A add permission to account B to subscribe to the topic:

```
aws sns add-permission \
    --region us-east-1 \
    --topic-arn Amazon SNS topic arn \
    --label lambda-access \
    --aws-account-id B \
    --action-name Subscribe ListSubscriptionsByTopic Receive
```

4. From account B add the Lambda permission to allow invocation from Amazon SNS:

```
aws lambda add-permission \
    --function-name SNS-X-Account \
    --statement-id sns-x-account \
    --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com \
    --source-arn Amazon SNS topic arn
```

In response, Lambda returns the following JSON code. The Statement value is a JSON string version of the statement added to the Lambda function policy:

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
\"arn:aws:lambda:us-east-1:B:function:SNS-X-Account\"}},\"Action
\":[\"lambda:InvokeFunction\"],\"Resource\":\"arn:aws:lambda:us-
east-1:A:function:SNS-X-Account\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"sns.amazonaws.com\"},\"Sid\":\"sns-x-account1\"}"
}
```

> **Note**
> Do not use the --source-account parameter to add a source account to the Lambda policy when adding the policy. Source account is not supported for Amazon SNS event sources and will result in access being denied. This has no security impact as the source account is included in the source ARN.

5. From account B subscribe the Lambda function to the topic:

```
aws sns subscribe \
    --topic-arn Amazon SNS topic arn \
    --protocol lambda \
    --notification-endpoint arn:aws:lambda:us-east-1:B:function:SNS-X-
Account
```

You should see JSON output similar to the following:

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:A:lambda-x-
account:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

6. From account A you can now test the subscription. Type "Hello World" into a text file and save it as message.txt. Then run the following command:

```
aws sns publish \
    --topic-arn arn:aws:sns:us-east-1:A:lambda-x-account \
    --message file://message.txt \
    --subject Test
```

This will return a message id with a unique identifier, indicating the message has been accepted by the Amazon SNS service. Amazon SNS will then attempt to deliver it to the topic's subscribers.

> **Note**
> Alternatively, you could supply a JSON string directly to the `message` parameter, but using a text file allows for line breaks in the message.

For more information on Amazon SNS, see What is Amazon Simple Notification Service?

Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS)

You can invoke AWS Lambda functions over HTTPS. You can do this by defining a custom REST API and endpoint using Amazon API Gateway, and then mapping individual methods, such as `GET` and `PUT`, to specific Lambda functions. Alternatively, you could add a special method named ANY to map all supported methods (`GET`, `POST`, `PATCH`, `DELETE`) to your Lambda function. When you send an HTTPS request to the API endpoint, the Amazon API Gateway service invokes the corresponding Lambda function. For more information about the `ANY` method, see Step 3: Create a Simple Microservice using Lambda and API Gateway (p. 171).

Amazon API Gateway also adds a layer between your application users and your app logic that enables the following:

- Ability to throttle individual users or requests.
- Protect against Distributed Denial of Service attacks.
- Provide a caching layer to cache response from your Lambda function.

Note the following about how the Amazon API Gateway and AWS Lambda integration works:

- **Push-event model** – This is a model (see Event Source Mapping (p. 120)), where Amazon API Gateway invokes the Lambda function by passing data in the request body as parameter to the Lambda function.
- **Synchronous invocation** – The Amazon API Gateway can invoke the Lambda function and get a response back in real time by specifying `RequestResponse` as the invocation type. For information about invocation types, see Invocation Types (p. 4).
- **Event structure** – The event your Lambda function receives is the body from the HTTPS request that Amazon API Gateway receives and your Lambda function is the custom code written to process the specific event type.

Note that there are two types of permissions policies that you work with when you set up the end-to-end experience:

- **Permissions for your Lambda function** – Regardless of what invokes a Lambda function, AWS Lambda executes the function by assuming the IAM role (execution role) that you specify at the time you create the Lambda function. Using the permissions policy associated with this role, you grant your Lambda function the permissions that it needs. For example, if your Lambda function needs to read an object, you grant permissions for the relevant Amazon S3 actions in the permissions policy. For more information, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).
- **Permission for Amazon API Gateway to invoke your Lambda function** – Amazon API Gateway cannot invoke your Lambda function without your permission. You grant this permission via the permission policy associated with the Lambda function.

For a tutorial that walks you through an example setup, see Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS) (p. 236).

Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS)

In this example you create a simple API (`DynamoDBOperations`) using Amazon API Gateway. An Amazon API Gateway is a collection of resources and methods. For this tutorial, you create one resource (`DynamoDBManager`) and define one method (`POST`) on it. The method is backed by a Lambda function (`LambdaFunctionForAPIGateway`). That is, when you invoke the method through an HTTPS endpoint, Amazon API Gateway invokes the Lambda function.

The `POST` method on the `DynamoDBManager` resource supports the following DynamoDB operations:

- Create, update, and delete an item.
- Read an item.
- Scan an item.
- Other operations (echo, ping), not related to DynamoDB, that you can use for testing.

The request payload you send in the `POST` request identifies the DynamoDB operation and provides necessary data. For example:

- The following is a sample request payload for a DynamoDB put item operation:

```
{
    "operation": "create",
    "tableName": "LambdaTable",
    "payload": {
        "Item": {
            "Id": "1",
            "name": "Bob"
        }
    }
}
```

- The following is a sample request payload for a DynamoDB read item operation:

```
{
    "operation": "read",
    "tableName": "LambdaTable",
    "payload": {
        "Key": {
            "Id": "1"
        }
    }
}
```

- The following is a sample request payload for a the `echo` operation. You will then send HTTPS PUT request to the endpoint, using the following data in the request body.

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

You can also create and manage API endpoints from the AWS Lambda console. For example, search for the **microservice** in the blueprints. This tutorial does not use the console, instead it uses AWS CLI to provide you with more details of how the API works.

Note
API Gateway offers advanced capabilities, such as:

- **Pass through the entire request** – A Lambda function can receive the entire HTTP request (instead of just the request body) and set the HTTP response (instead of just the response body) using the `AWS_PROXY` integration type.
- **Catch-all methods** – Map all methods of an API resource to a single function with a single mapping, using the `ANY` catch-all method.
- **Catch-all resources** – Map all sub-paths of a resource to a Lambda function without any additional configuration using the new path parameter (`{proxy+}`).

To learn more about these API Gateway features, see Configure Proxy Integration for a Proxy Resource.

Next Step

Step 1: Prepare (p. 238)

Step 1: Prepare

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account.
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Important
This example uses the `us-east-1` region to create an API using Amazon API Gateway and a Lambda function.

Next Step

Step 2: Create a Lambda Function and Test It Manually (p. 238)

Step 2: Create a Lambda Function and Test It Manually

In this section, you do the following:

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

Topics

Step 2.1: Create a Deployment Package

From the **Filter View** list, choose the language you want to use for your Lambda function. The appropriate section appears with code and specific instructions for creating a deployment package.

Node.js

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

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

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

var AWS = require('aws-sdk');
var dynamo = new AWS.DynamoDB.DocumentClient();

/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of the operations in the switch statement below
 *   - tableName: required for operations that interact with DynamoDB
 *   - payload: a parameter to pass to the operation being performed
 */
exports.handler = function(event, context, callback) {
    //console.log('Received event:', JSON.stringify(event, null, 2));

    var operation = event.operation;

    if (event.tableName) {
        event.payload.TableName = event.tableName;
    }

    switch (operation) {
        case 'create':
            dynamo.put(event.payload, callback);
            break;
        case 'read':
            dynamo.get(event.payload, callback);
            break;
        case 'update':
            dynamo.update(event.payload, callback);
            break;
        case 'delete':
            dynamo.delete(event.payload, callback);
            break;
        case 'list':
            dynamo.scan(event.payload, callback);
            break;
        case 'echo':
            callback(null, "Success");
            break;
        case 'ping':
            callback(null, "pong");
            break;
        default:
            callback('Unknown operation: ${operation}');
    }
};
```

   Note
   The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)

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

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

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 241)

Python

Follow the instructions to create AWS Lambda function deployment package.

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

```
from __future__ import print_function

import boto3
import json

print('Loading function')


def handler(event, context):
    '''Provide an event that contains the following keys:

      - operation: one of the operations in the operations dict below
      - tableName: required for operations that interact with DynamoDB
      - payload: a parameter to pass to the operation being performed
    '''
    #print("Received event: " + json.dumps(event, indent=2))

    operation = event['operation']

    if 'tableName' in event:
        dynamo = boto3.resource('dynamodb').Table(event['tableName'])

    operations = {
        'create': lambda x: dynamo.put_item(**x),
        'read': lambda x: dynamo.get_item(**x),
        'update': lambda x: dynamo.update_item(**x),
        'delete': lambda x: dynamo.delete_item(**x),
        'list': lambda x: dynamo.scan(**x),
        'echo': lambda x: x,
        'ping': lambda x: 'pong'
    }

    if operation in operations:
        return operations[operation](event.get('payload'))
    else:
        raise ValueError('Unrecognized operation "{}"'.format(operation))
```

2. Save the file as `LambdaFunctionOverHttps.py`.
3. Zip the `LambdaFunctionOverHttps.py` file as `LambdaFunctionOverHttps.zip`.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 241)

Step 2.2: Create the Execution Role (IAM Role)

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

- AWS service role of the type **AWS Lambda** – This role grants AWS Lambda permissions to assume the role.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:

   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-gateway-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - You create an IAM role without attaching a permissions policy in the console. After you create the role, you update the role, and then attach the following permissions policy to the role.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

3. Write down the role ARN (Amazon Resource Name). You need it in the next step when you create your Lambda function.

Next Step

Step 2.3: Create the Lambda Function and Test It Manually (p. 242)

Step 2.3: Create the Lambda Function and Test It Manually

In this section, you do the following:

- Create a Lambda function by uploading the deployment package.
- Test the Lambda function by invoking it manually and passing sample event data.

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

In this step, you upload the deployment package using the AWS CLI.

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

You need to update the command by providing the .zip file path and the execution role ARN. The `--runtime` parameter value can be `python2.7`, `nodejs4.3`, or `java8`, depending on the language you used to author your code.

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

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

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

Note
You can create the Lambda function using the AWS Lambda console, in which case note the value of the `create-function` AWS CLI command parameters. You provide the same values in the console UI.

Step 2.3.2: Test the Lambda Function (Invoke Manually)

Invoke the function manually using the sample event data. We recommend that you invoke the function using the console because the console UI provides a user-friendly interface for reviewing the execution results, including the execution summary, logs written by your code, and the results returned by the function (because the console always performs synchronous execution—invokes the Lambda function using the `RequestResponse` invocation type).

To test the Lambda function (AWS Management Console)

1. Follow the steps in the Getting Started exercise to create and invoke the Lambda function at Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167). For the sample event for testing, choose **Hello World** in **Sample event template**, and then replace the data using the following:

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

2. To test one of the `dynamo` operations, such as `read`, change the input data to the following:

```
{
    "operation": "read",
    "tableName": "the name of your stream table"
    "payload": {
        "Key": {
          "the primary key of the table": "the value of the key"
        }
    }
}
```

3. Verify the results in the console.

To test the Lambda function (AWS CLI)

1. Copy the following JSON into a file and save it as `input.txt`.

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

2. Execute the following `invoke` command:

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

Note
In this tutorial example, the message is saved in the `outputfile.txt` file if you request synchronous execution (`RequestResponse` as the invocation type). The function returns the string message in the response body. If you use the `Event` invocation type, no message is returned to the output file. In either case, the *outputfile.txt* parameter is required.
For Node.js, it could be one of the following (whatever one you specify in the code):
`context.succeed("message")`
`context.fail("message")`
`context.done(null, "message)`
For Python or Java, it is the message in the return statement:

```
        return "message"
```

Next Step

Step 3: Create an API Using Amazon API Gateway and Test It (p. 244)

Step 3: Create an API Using Amazon API Gateway and Test It

In this step, you associate your Lambda function with a method in the API that you created using Amazon API Gateway and test the end-to-end experience. That is, when an HTTPS request is sent to an API method, Amazon API Gateway invokes your Lambda function.

First, you create an API (`DynamoDBOperations`) using Amazon API Gateway with one resource (`DynamoDBManager`) and one method (`POST`). You associate the `POST` method with your Lambda function. Then, you test the end-to-end experience.

Step 3.1: Create the API

Run the following `create-rest-api` command to create the `DynamoDBOperations` API for this tutorial.

```
$ aws apigateway create-rest-api \
--name DynamoDBOperations
```

The following is an example response:

```
{
    "name": "DynamoDBOperations",
    "id": "api-id",
    "createdDate": 1447724091
}
```

Note the API ID.

You also need the ID of the API root resource. To get the ID, run the `get-resources` command.

```
$ aws apigateway get-resources \
--rest-api-id api-id
```

The following is example response (at this time you only have the root resource, but you add more resources in the next step):

```
{
    "items": [
        {
            "path": "/",
            "id": "root-id"
        }
    ]
}
```

Step 3.2: Create a Resource (DynamoDBManager) in the API

Run the following `create-resource` command to create a resource (`DynamoDBManager`) in the API that you created in the preceding section.

```
$ aws apigateway create-resource \
--rest-api-id api-id \
--parent-id root-id \
--path-part DynamoDBManager
```

The following is an example response:

```
{
    "path": "/DynamoDBManager",
    "pathPart": "DynamoDBManager",
    "id": "resource-id",
    "parentId": "root-id"
}
```

Note the ID in the response. This is the ID of the resource (`DynamoDBManager`) that you created.

Step 3.3: Create Method (POST) on the Resource

Run the following `put-method` command to create a method (`POST`) on the resource (`DynamoDBManager`) in your API (`DynamoDBOperations`).

```
$ aws apigateway put-method \
--rest-api-id api-id \
--resource-id resource-id \
--http-method POST \
--authorization-type NONE
```

We specify `NONE` for the `--authorization-type` parameter, which means that unauthenticated requests for this method are supported. This is fine for testing but in production you should use either the key-based or role-base authentication.

The following is an example response:

```
{
    "apiKeyRequired": false,
    "httpMethod": "POST",
    "authorizationType": "NONE"
}
```

Step 3.4: Set the Lambda Function as the Destination for the POST Method

Run the following command to set the Lambda function as the integration point for the `POST` method (this is the method Amazon API Gateway invokes when you make an HTTPS request for the `POST` method endpoint).

```
$ aws apigateway put-integration \
--rest-api-id api-id \
--resource-id resource-id \
--http-method POST \
--type AWS \
--integration-http-method POST \
--uri arn:aws:apigateway:aws-region:lambda:path/2015-03-31/functions/
arn:aws:lambda:aws-region:aws-acct-id:function:your-lambda-function-name/
invocations
```

Note

- `--rest-api-id` is the ID of the API (`DynamoDBOperations`) that you created in Amazon API Gateway.
- `--resource-id` is the resource ID of the resource (`DynamoDBManager`) you created in the API
- `--http-method` is the API Gateway method and `--integration-http-method` is the method that API Gateway uses to communicate with AWS Lambda.
- `--uri` is unique identifier for the endpoint to which Amazon API Gateway can send request.

The following is an example response:

```
{
    "httpMethod": "POST",
    "type": "AWS",
    "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:aws-acct-id:function:LambdaFunctionForAPIGateway/
invocations",
    "cacheNamespace": "resource-id"
}
```

Set `content-type` of the `POST` method response and integration response to JSON as follows:

- Run the following command to set the `POST` method response to JSON. This is the response type that your API method returns.

  ```
  $ aws apigateway put-method-response \
  --rest-api-id api-id \
  --resource-id resource-id \
  --http-method POST \
  --status-code 200 \
  --response-models "{\"application/json\": \"Empty\"}"
  ```

- Run the following command to set the `POST` method integration response to JSON. This is the response type that Lambda function returns.

  ```
  $ aws apigateway put-integration-response \
  --rest-api-id api-id \
  --resource-id resource-id \
  --http-method POST \
  --status-code 200 \
  --response-templates "{\"application/json\": \"\"}"
  ```

Step 3.5: Deploy the API

In this step, you deploy the API that you created to a stage called `prod`.

```
$ aws apigateway create-deployment \
--rest-api-id api-id \
--stage-name prod
```

The following is an example response:

```
{
    "id": "deployment-id",
    "createdDate": 1447726017
}
```

Step 3.6: Grant Permissions that Allows Amazon API Gateway to Invoke the Lambda Function

Now that you have an API created using Amazon API Gateway and you've deployed it, you can test. First, you need to add permissions so that Amazon API Gateway can invoke your Lambda function when you send HTTPS request to the `POST` method.

To do this, you need to add a permissions to the permissions policy associated with your Lambda function. Run the following `add-permission` AWS Lambda command to grant the Amazon API Gateway service principal (`apigateway.amazonaws.com`) permissions to invoke your Lambda function (`LambdaFunctionForAPIGateway`).

```
$ aws lambda add-permission \
--function-name LambdaFunctionOverHttps \
--statement-id apigateway-test-2 \
--action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:us-east-1:aws-acct-id:api-id/*/POST/
DynamoDBManager"
```

You must grant this permission to enable testing (if you go to the Amazon API Gateway and choose **Test** to test the API method, you need this permission). Note the `--source-arn` specifies a wildcard character (*) as the stage value (indicates testing only). This allows you to test without deploying the API.

Now, run the same command again, but this time you grant to your deployed API permissions to invoke the Lambda function.

```
$ aws lambda add-permission \
--function-name LambdaFunctionOverHttps \
--statement-id apigateway-prod-2 \
--action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:us-east-1:aws-acct-id:api-id/prod/POST/
DynamoDBManager"
```

You grant this permission so that your deployed API has permissions to invoke the Lambda function. Note that the `--source-arn` specifies a `prod` which is the stage name we used when deploying the API.

Step 3.7: Test Sending an HTTPS Request

In this step, you are ready to send an HTTPS request to the `POST` method endpoint. You can use either Curl or a method (`test-invoke-method`) provided by Amazon API Gateway.

If you want to test operations that your Lambda function supports on a DynamoDB table, first you need to create a table in Amazon DynamoDB `LambdaTable (Id)`, where Id is the hash key of string type.

If you are testing the `echo` and `ping` operations that your Lambda function supports, you don't need to create the DynamoDB table.

You can use Amazon API Gateway CLI commands to send an HTTPS `POST` request to the resource (`DynamoDBManager`) endpoint. Because you deployed your Amazon API Gateway, you can use Curl to invoke the methods for the same operation.

The Lambda function supports using the `create` operation to create an item in your DynamoDB table. To request this operation, use the following JSON:

```

{
    "operation": "create",
    "tableName": "LambdaTable",
    "payload": {
        "Item": {
            "Id": "foo",
            "number": 5
        }
    }
}
```

Run the `test-invoke-method` Amazon API Gateway command to send an HTTPS `POST` method request to the resource (`DynamoDBManager`) endpoint with the preceding JSON in the request body.

```
$ aws apigateway test-invoke-method \
--rest-api-id api-id \
--resource-id resource-id \
--http-method POST \
--path-with-query-string "" \
--body "{\"operation\":\"create\",\"tableName\":\"LambdaTable\",\"payload\":
{\"Item\":{\"Id\":\"1\",\"name\":\"Bob\"}}}"
```

Or, you can use the following Curl command:

```
curl -X POST -d "{\"operation\":\"create\",\"tableName\":\"LambdaTable
\",\"payload\":{\"Item\":{\"Id\":\"1\",\"name\":\"Bob\"}}}" https://api-
id.execute-api.aws-region.amazonaws.com/prod/DynamoDBManager
```

To send request for the `echo` operation that your Lambda function supports, you can use the following request payload:

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

Run the `test-invoke-method` Amazon API Gateway CLI command to send an HTTPS `POST` method request to the resource (`DynamoDBManager`) endpoint using the preceding JSON in the request body.

```
$ aws apigateway test-invoke-method \
--rest-api-id api-id \
--resource-id resource-id \
--http-method POST \
--path-with-query-string "" \
--body "{\"operation\":\"echo\",\"payload\":{\"somekey1\":\"somevalue1\",
\"somekey2\":\"somevalue2\"}}"
```

Or, you can use the following Curl command:

```
curl -X POST -d "{\"operation\":\"echo\",\"payload\":{\"somekey1\":
\"somevalue1\",\"somekey2\":\"somevalue2\"}}" https://api-id.execute-
api.region.amazonaws.com/prod/DynamoDBManager
```

Step 4: Deploy With AWS SAM and AWS CloudFormation

In the previous section, you used AWS Lambda APIs to create and update a Lambda function by providing a deployment package as a ZIP file. However, this mechanism may not be convenient for automating deployment steps for functions, or coordinating deployments and updates to other elements of a serverless application, like event sources and downstream resources.

You can use AWS CloudFormation to easily specify, deploy, and configure serverless applications. AWS CloudFormation is a service that helps you model and set up your Amazon Web Services resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Lambda functions and DynamoDB tables), and AWS CloudFormation takes care of provisioning and configuring those resources for you.

In addition, you can use the AWS Serverless Application Model to express resources that comprise the serverless application. These resource types, such as Lambda functions and APIs, are fully supported by AWS CloudFormation and make it easier for you to define and deploy your serverless application.

For more information, see Deploying Lambda-based Applications (p. 139).

Specification for Amazon API Gateway Application

The following contains the SAM template for this application. Copy the text below to a .yaml file and save it next to the ZIP package you created previously.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  LambdaFunctionOverHttps:
    Type: AWS::Serverless::Function
    Properties:
      Handler: LambdaFunctionOverHttps.handler
      Runtime: nodejs4.3
      Policies: AmazonDynamoDBFullAccess
      Events:
        HttpPost:
          Type: Api
          Properties:
            Path: 'DynamoDBOperations/DynamoDBManager'
            Method: post
```

Deploying the Serverless Application

For information on how to package and deploy your serverless application using the package and deploy commands, see Packaging and Deployment (p. 145).

Using AWS Lambda as Mobile Application Backend (Custom Event Source: Android)

You can use AWS Lambda to host backend logic for mobile applications. That is, some of your mobile app code can be run as Lambda functions. This allows you to put minimal logic in the mobile application itself making it easy to scale and update (for example, you only apply code updates to the Lambda function, instead of having to deploy code updates in your app clients).

After you create the Lambda function, you can invoke it from your mobile app using AWS Mobile SDKs, such as the AWS SDK for Android. For more information, see Tools for Amazon Web Services.

> **Note**
> You can also invoke your Lambda function over HTTP using Amazon API Gateway (instead of using any of the AWS SDKs). Amazon API Gateway adds an additional layer between your mobile users and your app logic that enable the following:
>
> - Ability to throttle individual users or requests.
> - Protect against Distributed Denial of Service attacks.
> - Provide a caching layer to cache response from your Lambda function.

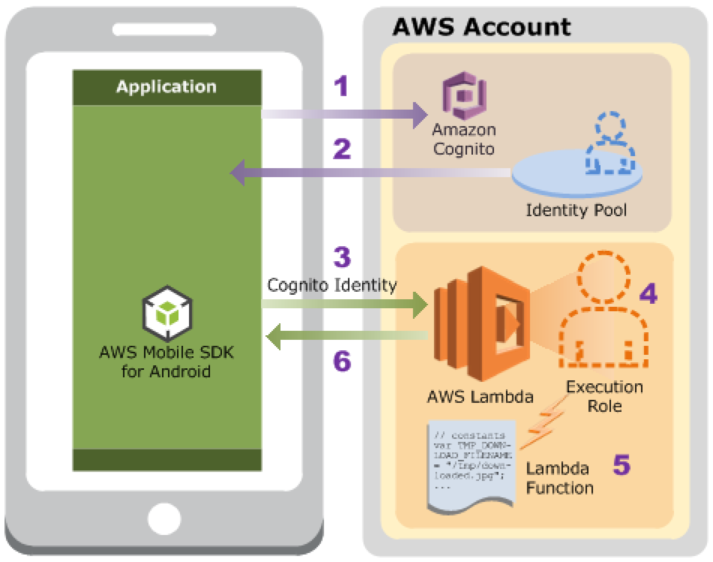
Note the following about how the mobile application and AWS Lambda integration works:

- **Push-event model** – This is a model (see Event Source Mapping (p. 120)), where the app invokes the Lambda function by passing the event data as parameter.
- **Synchronous or asynchronous invocation** – The app can invoke the Lambda function and get a response back in real time by specifying `RequestResponse` as the invocation type (or use the `Event` invocation type for asynchronous invocation). For information about invocation types, see Manage Permissions: Using a Lambda Function Policy (p. 156).
- **Event structure** – The event your Lambda function receives is defined by your application, and your Lambda function is the custom code written to process the specific event type.

Note that there are two types of permissions policies that you work with in setting the end-to-end experience:

- **Permissions for your Lambda function** – Regardless of what invokes a Lambda function, AWS Lambda executes the function by assuming the IAM role (execution role) that you specify at the time you create the Lambda function. Using the permissions policy associated with this role, you grant your Lambda function the permissions that it needs. For example, if your Lambda function needs to read an object, you grant permissions for the relevant Amazon S3 actions in the permissions policy. For more information, see Manage Permissions: Using an IAM Role (Execution Role) (p. 155).
- **Permissions for the mobile app to invoke your Lambda function** – The application must have valid security credentials and permissions to invoke a Lambda function. For mobile applications, you can use the Amazon Cognito service to manage user identities, authentication, and permissions.

The following diagram illustrates the application flow (the illustration assumes a mobile app using AWS Mobile SDK for Android to make the API calls):



1. The mobile application sends a request to Amazon Cognito with an identity pool ID in the request (you create the identity pool as part the setup).
2. Amazon Cognito returns temporary security credentials back to the application.

   Amazon Cognito assumes the role associated with the identity pool to generate temporary credentials. What the application can do using the credentials is limited to the permissions defined in the permissions policy associated with the role Amazon Cognito used in obtaining the temporary credential.

   Note
   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.
3. The mobile application invokes the Lambda function using temporary credentials (Cognito Identity).
4. AWS Lambda assumes the execution role to execute your Lambda function on your behalf.
5. The Lambda function executes.
6. AWS Lambda returns results to the mobile application, assuming the app invoked the Lambda function using the `RequestResponse` invocation type (synchronous invocation).

For a tutorial that walks you through an example setup, see Tutorial: Using AWS Lambda as Mobile Application Backend (p. 251).

Tutorial: Using AWS Lambda as Mobile Application Backend

In this tutorial, you create a simple Android mobile application. The primary purpose of this tutorial 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 app itself is simple, we will assume following:

- The sample mobile application will generate event data consisting of a name (first name and last name) in this format:

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

- You use Lambda function to process the event. That is, the app (using the AWS Mobile SDK for Android) invokes a Lambda function (`ExampleAndroidEventProcessor`) by passing the event data to it. The Lambda function in this tutorial does the following:
  - Logs incoming event data to Amazon CloudWatch Logs.
  - Upon successful execution, returns a simple string in the response body. Your mobile app displays the message using the Android `Toast` class.

  Note
  The way that the mobile application invokes a Lambda function as shown in this tutorial is an example of the AWS Lambda request-response model in which an application invokes a Lambda function and then receives a response in real time. For more information, see Programming Model (p. 8).

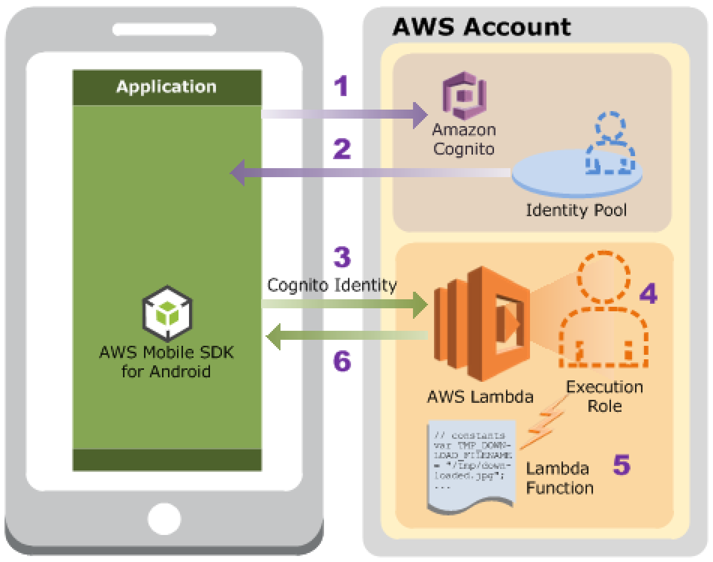
Implementation Summary

The tutorial is divided into two main sections:

- First, you perform the necessary setup to create a Lambda function and test it by invoking it manually using sample event data (you don't need mobile app to test your Lambda function).
- Second, you create an Amazon Cognito identity pool to manage authentication and permissions, and create the example Android application. Then, you run the application and it invokes the Lambda function. You can then verify the end-to-end experience. In this tutorial example:
  - You use the Amazon Cognito service to manage user identities, authentication, and permissions. The mobile application must have valid security credentials and permissions to invoke a Lambda function. 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 mobile application does not require its users 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 tutorial is limited and assumes that the mobile application users are unauthenticated. Therefore, when you configure Amazon Cognito identity pool you will do the following:
    - Enable access for unauthenticated identities.

      Amazon Cognito provides a unique identifier and temporary AWS credentials for these users to invoke the Lambda function.
    - In the access permissions policy associated with the IAM role for unauthenticated users, add permissions to invoke the Lambda function. An identity pool has two associated IAM roles, one for authenticated and one for unauthenticated application users. In this example, Amazon Cognito assumes the role for unauthenticated users to obtain temporary credentials. When the app uses these temporary credentials to invoke your Lambda function, it can do so only if has necessary permissions (that is, credentials may be valid, but you also need permissions). You do this by updating the permissions policy that Amazon Cognito uses to obtain the temporary credentials.

The following diagram illustrates the application flow:



Now you are ready to start the tutorial.

Next Step

Step 1: Prepare (p. 253)

Step 1: Prepare

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account.
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

> **Note**
> The tutorial creates a Lambda function and an Amazon Cognito identity pool in the `us-east-1` region. If you want to use a different AWS region, you must 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 the Lambda Function and Invoke It Manually (Using Sample Event Data) (p. 253)

Step 2: Create the Lambda Function and Invoke It Manually (Using Sample Event Data)

In this section, you do the following:

- Create a Lambda function deployment package using the sample code provided. The sample Lambda function code to process your mobile application events is provided in various languages. Select one of the languages and follow the corresponding instructions to create a deployment package.

  > **Note**
  > To see more examples of using other AWS services within your function, including calling other Lambda functions, see AWS SDK for JavaScript

- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify an IAM role (execution role). This is the role that AWS Lambda assumes to invoke your Lambda function on your behalf.
- Create the Lambda function by uploading the deployment package, and then test it by invoking it manually using sample event data.

Topics

Step 2.1: Create a Deployment Package

From the **Filter View** list, choose the language you want to use for your Lambda function. The appropriate section appears with code and specific instructions for creating a deployment package.

Node.js

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

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

```
exports.handler = function(event, context, callback) {
   console.log("Received event: ", event);
   callback(null, "message");
}
```

   Note
   The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)
2. Save the file as `AndroidBackendLambdaFunction.js`.
3. Zip the `AndroidBackendLambdaFunction.js` file as `AndroidBackendLambdaFunction.zip`.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 256)

Java

Use the following Java code to create your Lambda function (`AndroidBackendLambdaFunction`). The code receives Android app event data as the first parameter to the handler. Then, the code processes event data (for illustration this code writes some of the event data to CloudWatch Logs and returns a string in response).

In the code, the `handler` (`myHandler`) uses the `RequestClass` and `ResponseClass` types for the input and output. The code provides implementation for these types.

Important
You 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);
```

```
        context.getLogger().log(greetingString);
        return new ResponseClass(greetingString);
    }
}
```

Save the preceding code in a file (`HelloPojo.java`). Your can now create a deployment package. You need to include the following dependency:

- `aws-lambda-java-core`

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. For examples of how to use the Maven build tool to create a standalone .jar, see Creating a .jar Deployment Package Using Maven without any IDE (Java) (p. 62) and Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java) (p. 65). For an example of how to use the Gradle build tool to create a .zip file, see Creating a .zip Deployment Package (Java) (p. 67).

After you verify that your deployment package (`lambda-java-example-1.0-SNAPSHOT.jar`) is created, go to the next section to create an IAM role (execution role). You specify the role when you create your Lambda function.

Next Step

Step 2.2: Create the Execution Role (IAM Role) (p. 256)

Step 2.2: Create the Execution Role (IAM Role)

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

- AWS service role of the type **AWS Lambda** – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaBasicExecute** – This is the access permissions policy that you attach to the role. This Lambda function only writes logs to CloudWatch Logs. So it only needs permission for specific CloudWatch actions. This policy provides these permissions.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-android-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - In **Attach Policy**, choose **AWSLambdaBasicExecute**. The permissions in this policy are sufficient for the Lambda function in this tutorial.
3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Next Step

Step 2.3: Create the Lambda Function and Invoke It Manually (Using Sample Event Data) (p. 257)

Step 2.3: Create the Lambda Function and Invoke It Manually (Using Sample Event Data)

In this section, you do the following:

- Create a Lambda function, by uploading the deployment package.
- Test the Lambda function by invoking it manually. Instead of creating an event source, you use sample event data. In the next section, you create an Android mobile app and test the end-to-end experience.

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

In this step, you upload the deployment package using the AWS CLI.

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

You need to update the command by providing the .zip file path and the execution role ARN. The `--runtime` parameter value can be `nodejs4.3`, or `java8`, depending on the language you chose to author your code.

```
$ aws lambda create-function \
--region us-east-1 \
--function-name AndroidBackendLambdaFunction  \
--zip-file fileb://file-path-to-jar-or-zip-deployment-package \
--role execution-role-arn  \
--handler handler-name \
--runtime runtime-value \
--profile adminuser
```

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

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

Note
You can create the Lambda function using the AWS Lambda console, in which case note the value of the `create-function` AWS CLI command parameters. You provide the same values in the console UI.

Step 2.3.2: Test the Lambda Function (Invoke Manually)

Invoke the function manually using the sample event data. We recommend that you invoke the function using the console because the console UI provides a user-friendly interface for reviewing the execution results, including the execution summary, logs written by your code, and the results returned by the function (because the console always performs synchronous execution—invokes the Lambda function using the `RequestResponse` invocation type).

To test the Lambda function (AWS Management Console)

1. Follow the steps in the Getting Started exercise to create and invoke the Lambda function at Step 2.2: Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics (p. 167). After

you choose the Lambda function, choose **Configure test event** from the **Actions** menu to specify the following sample event data:

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

2. Verify the results in the console.

   - **Execution result** should be `Succeeded` with the following return value:

   ```
   {
     "greetings": "Hello first-name, last-name."
   }
   ```

   - Review the **Summary** and the **Log output** sections.

To test the Lambda function (AWS CLI)

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

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

2. Execute the following `invoke` command:

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

> **Note**
> In this tutorial example, the message is saved in the `outputfile.txt` file. If you request synchronous execution (`RequestResponse` as the invocation type), the function returns the string message in the response body.
> For Node.js, it could be one of the following (whatever one you specify in the code):
> `context.succeed("message")`
> `context.fail("message")`
> `context.done(null, "message)`
> For Java, it is the message in the return statement:
> `return "message"`

Next Step

Step 3: Create an Amazon Cognito Identity Pool (p. 258)

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 permissions to execute the `AndroidBackendLambdaFunction` Lambda function.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. For more information about Amazon Cognito services, see 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). Therefore, make sure to select the **Enable access to unauthenticated identities** option.
   - The unauthenticated application users need permission to invoke the Lambda function. To enable this, you will add the following statement to the permission policy associated with the unauthenticated identities (it allows permission for the for the `lambda:InvokeFunction` action on the specific Lambda function (you must update the resource ARN by providing your account ID).

   ```
   {
           "Effect": "Allow",
           "Action": [
               "lambda:InvokeFunction"
           ],
           "Resource": [
              "arn:aws:lambda:us-east-1:account-
   id:function:AndroidBackendLambdaFunction"
           ]
   }
   ```

   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-east-1:account-
   id:function:AndroidBackendLambdaFunction"
            ]
         }
      ]
   }
   ```

Note
You can update policy at the time of creating the identity pool. You can also update the policy after you create the identity pool, in which case make sure you write down the IAM role name for the unauthenticated users from the Amazon Cognito console. Then, go to the IAM console and search for the specific role and edit the access permissions policy.

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

3. Note down the identity pool ID. You specify this ID in your mobile application you create in the next section. The app uses this ID when it sends request to Amazon Cognito to request for temporary security credentials.

Next Step

Step 4: Create a Mobile Application for Android (p. 260)

Step 4: Create a Mobile 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, add the following two classes (`RequestClass` and `ResponseClass`). Note that the POJO is same as the POJO you created in your Lambda function in the preceding section.
   - `RequestClass`. The instances of this class act as the POJO (Plain Old Java Object) for event data which consists of first and last name. If you are using Java example for your Lambda

function you created in the preceding section, this POJO is same as the POJO you created in your Lambda function code.

```
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() {
    }
}
```

- `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() {
    }
}
```

7. In the same package, create interface called `MyInterface` for invoking the `AndroidBackendLambdaFunction` 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, see AWS Lambda in the *AWS Mobile SDK for Android Developer Guide*.

```
package com.example.....lambdaeventgenerator;
import com.amazonaws.mobileconnectors.lambdainvoker.LambdaFunction;
public interface MyInterface {

    /**
     * Invoke the Lambda function "AndroidBackendLambdaFunction".
     * The function name is the method name.
     */
    @LambdaFunction
     ResponseClass AndroidBackendLambdaFunction(RequestClass request);

}
```

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
 CognitoCachingCredentialsProvider(
        this.getApplicationContext(), "identity-pool-id",
 Regions.US_WEST_2);

// Create LambdaInvokerFactory, to be used to instantiate the Lambda
 proxy.
LambdaInvokerFactory factory = new
 LambdaInvokerFactory(this.getApplicationContext(),
        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.AndroidBackendLambdaFunction(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);
```

9. Run the code and verify it 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.

Using AWS Lambda with Scheduled Events

You can create a Lambda function and direct AWS Lambda to execute it on a regular schedule. You can specify a fixed rate (for example, execute a Lambda function every hour or 15 minutes), or you can specify a Cron expression. For more information on expressions schedules, see Schedule Expressions Using Rate or Cron (p. 267).

This functionality is available when you create a Lambda function using the AWS Lambda console or the AWS CLI. To configure it using the AWS CLI, see Run an AWS Lambda Function on a Schedule Using the AWS CLI. The console provides the **CloudWatch Events - Schedule** as an event source. At the time of creating a Lambda function, you choose this event source and specify a time interval.

If you have made any manual changes to the permissions on your function, you may need to reapply the scheduled event access to your function. You can do that by using the following CLI command.

```
aws lambda add-permission \
    --statement-id 'statement id' \
    --action 'lambda:InvokeFunction' \
    --principal events.amazonaws.com \
    --source-arn arn:aws:events:region:account-id:rule/rule_name
    --function-name function:MyFunction
    --region region
```

Note
Each AWS account can have up to 50 unique event sources of the **CloudWatch Events - Schedule** source type. Each of these can be the event source for up to five Lambda functions. That is, you can have up to 250 Lambda functions that can be executing on a schedule in your AWS account.

The console also provides a blueprint (**lambda-canary**) that uses the **CloudWatch Events - Schedule** source type. Using this blueprint, you can create a sample Lambda function and test this feature. The example code that the blueprint provides checks for the presence of a specific webpage and specific text string on the webpage. If either the webpage or the text string is not found, the Lambda function throws an error.

For a tutorial that walks you through an example setup, see Tutorial: Using AWS Lambda with Scheduled Events (p. 264).

Tutorial: Using AWS Lambda with Scheduled Events

In this tutorial, you do the following:

- Create a Lambda function using the **lambda-canary** blueprint. You configure the Lambda function to run every minute. Note that if the function returns an error, AWS Lambda logs error metrics to CloudWatch.
- Configure a CloudWatch alarm on the `Errors` metric of your Lambda function to post a message to your Amazon SNS topic when AWS Lambda emits error metrics to CloudWatch. You subscribe to the Amazon SNS topics to get email notification. In this tutorial, you do the following to set this up:
  - Create an Amazon SNS topic.
  - Subscribe to the topic so you can get email notifications when a new message is posted to the topic.
  - In Amazon CloudWatch, set an alarm on the `Errors` metric of your Lambda function to publish a message to your SNS topic when errors occur.

Next Step

Step 1: Create a Lambda Function (p. 264)

Step 1: Create a Lambda Function

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Choose **Create a Lambda function**.
3. In **Select blueprint**, choose the **lambda-canary** blueprint.
4. In **Configure triggers**:
   - Choose **CloudWatch Events - Schedule**.
   - In **Rule name**, type a name (for example, **`CheckWebsiteScheduledEvent`**).
   - In **Rule description**, type a description (for example, **`CheckWebsiteScheduledEvent trigger`**).
   - In **Schedule expression**, specify **`rate(1 minute)`**. Note that you can specify the value as a `rate` or in the `cron` expression format. All schedules use the UTC time zone, and the minimum precision for schedules is one minute.

     Note
     When setting a rate expression, the first execution is immediate and subsequent executions occur based on the rate schedule. In the preceding example, the subsequent execution rate would be every minute.

     For more information on expressions schedules, see Schedule Expressions Using Rate or Cron (p. 267).
   - In **Enable trigger**, we recommend that you leave the trigger in a disabled state until you have tested it.
   - Choose **Next**.
5. In **Configure function**, do the following:
   - Specify your Lambda function name (for example, **`CheckWebsitePeriodically`**).
   - In **Runtime**, specify **Python 2.7** or **Node.js4.3**, depending on your preferred language.

- Review the code provided by the template. Later in this tutorial, you will update the function code so that the function will return an error. You can either specify a non-existing URL or replace search text to a string that is not on the page.
- In **Role***, choose **Create new role from template(s)**.
- In **Role name**, type a name for the role.
- In **Policy templates**, Lambda provides a list of optional, additional templates that extend the basic Lambda permissions. For the purpose of this tutorial, you can leave this field blank because your Lambda function already has the basic execution permission it needs.
- In **Advanced settings**, leave the default configurations and choose **Next**.

6. In **Review**, review the configuration and then choose **Create Function**.

Next Step

Step 2: Test the Lambda Function (Using a Sample Test Event) (p. 265)

Step 2: Test the Lambda Function (Using a Sample Test Event)

1. Choose the function you created in the previous step and then choose **Test**.
2. On the **Input sample event** page, choose **Scheduled Event** in the **Sample event** list.

   Note the event time in the sample event. This value will be different when AWS Lambda invokes the function at scheduled intervals. The sample Lambda function code logs this time to CloudWatch Logs.
3. Choose **Save and test** and verify that the **Execution result** section shows success.

Next Step

Step 3: Create an Amazon SNS Topic and Subscribe to It (p. 265)

Step 3: Create an Amazon SNS Topic and Subscribe to It

1. Create an SNS topic using the Amazon SNS console. For instructions, see Create a Topic in the *Amazon Simple Notification Service Developer Guide*.
2. Subscribe to the topic. For this exercise, use email as the communication protocol. For instructions, see Subscribe to a Topic in the *Amazon Simple Notification Service Developer Guide*.

You use this Amazon SNS topic in the next step when you configure a CloudWatch alarm so that when AWS Lambda emits an error the alarm will publish a notification to this topic.

Next Step

Step 4: Configure a CloudWatch Alarm (p. 265)

Step 4: Configure a CloudWatch Alarm

To configure a CloudWatch alarm, follow the instructions at Create Alarm in the *Amazon CloudWatch User Guide*. As you follow the steps, note the following:

- In **Create Alarm** (**1. Select Metric** step), choose **Lambda Metrics**, and then choose the **Errors** (**Metric Name** is **Errors**) for the Lambda function you created. Also, on the statistics drop-down, change the settings from **Average** to **Sum** statistics.

- In **Create Alarm** (**2. Define Metric** step), set the alarm threshold to **Whenever: Errors is >= 1** and select your Amazon SNS topic from the **Send notification to:** list.

Next Step

Step 5: Test the Lambda Function Again (p. 266)

Step 5: Test the Lambda Function Again

Now test the Lambda function again. This time, update the code by specifying either a non-existing webpage URL or a text string. This causes the function to return an error that AWS Lambda sends to CloudWatch error metrics. CloudWatch posts this message to the Amazon SNS topic and you get an email notification.

Step 6: Deploy With AWS SAM and AWS CloudFormation

In the previous section, you used AWS Lambda APIs to create and update a Lambda function by providing a deployment package as a ZIP file. However, this mechanism may not be convenient for automating deployment steps for functions, or coordinating deployments and updates to other elements of a serverless application, like event sources and downstream resources.

You can use AWS CloudFormation to easily specify, deploy, and configure serverless applications. AWS CloudFormation is a service that helps you model and set up your Amazon Web Services resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Lambda functions and DynamoDB tables), and AWS CloudFormation takes care of provisioning and configuring those resources for you.

In addition, you can use the AWS Serverless Application Model to express resources that comprise the serverless application. These resource types, such as Lambda functions and APIs, are fully supported by AWS CloudFormation and make it easier for you to define and deploy your serverless application.

For more information, see Deploying Lambda-based Applications (p. 139).

Specification for Scheduled Event Application

The following contains the SAM template for this application. Copy the text below to a .yaml file and save it next to the ZIP package you created previously.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Parameters:
  NotificationEmail:
    Type: String
Resources:
  CheckWebsitePeriodically:
    Type: AWS::Serverless::Function
    Properties:
      Handler: LambdaFunctionOverHttps.handler
      Runtime: python2.7
      Policies: AmazonDynamoDBFullAccess
      Events:
        CheckWebsiteScheduledEvent:
          Type: Schedule
          Properties:
            Schedule: rate(1 minute)

  AlarmTopic:
    Type: AWS::SNS::Topic
    Properties:
      Subscription:
      - Protocol: email
        Endpoint: !Ref NotificationEmail

  Alarm:
    Type: AWS::CloudWatch::Alarm
    Properties:
      AlarmActions:
        - !Ref AlarmTopic
      ComparisonOperator: GreaterThanOrEqualToThreshold
      Dimensions:
        - Name: FunctionName
          Value: !Ref CheckWebsitePeriodically
      EvaluationPeriods: String
      MetricName: Errors
      Namespace: AWS/Lambda
      Period: '60'
      Statistic: Sum
      Threshold: '1'
```

Deploying the Serverless Application

For information on how to package and deploy your serverless application using the package and deploy commands, see Packaging and Deployment (p. 145).

Schedule Expressions Using Rate or Cron

Rate expression

```
rate(Value Unit)
```

Where:

Value can be a positive integer.

Unit can be minute(s), hour(s), or day(s).

For example:

Example	Cron expression
Invoke Lambda function every 5 minutes	`rate(5 minutes)`
Invoke Lambda function every hour	`rate(1 hour)`
Invoke Lambda function every seven days	`rate(7 days)`

Note the following:

- Rate frequencies of less than one minute are not supported.
- For a singular value the unit must be singular (for example, `rate(1 day)`), otherwise plural (for example, `rate(5 days)`).

Cron expression

```
cron(Minutes Hours Day-of-month Month Day-of-week Year)
```

All fields are required and time zone is UTC only. The following table describes these fields.

Field	Values	Wildcards
Minutes	0-59	, - * /
Hours	0-23	, - * /
Day-of-month	1-31	, - * ? / L W
Month	1-12 or JAN-DEC	, - * /
Day-of-week	1-7 or SUN-SAT	, - * ? / L #
Year	1970-2199	, - * /

The following table describes the wildcard characters.

Character	Definition	Example
/	Specifies increments	0/15 in the minutes field directs execution to occur every 15 minutes.
L	Specifies "Last"	If used in Day-of-month field, specifies last day of the month. If used in Day-of-week field, specifies last day of the week (Saturday).
W	Specifies *Weekday*	When used with a date, such as `5/W`, specifies the closest weekday to 5th day of the month. If the 5th falls on a Saturday, execution occurs on Friday. If the 5th falls on a Sunday, execution occurs on Monday.
#	Specifies the *nd* or *nth* day of the month	Specifying `3#2` means the second Tuesday of the month (Tuesday is the third day of the 7-day week).
*	Specifies *All values*	If used in the Day-of-month field, it means all days in the month.
?	No specified value	Used in conjunction with another specified value. For example, if a specific date is specified, but you don't care what day of the week it falls on.
-	Specifies ranges	`10-12` would mean 10, 11 and 12
,	Specifies additional values	`SUN, MON, TUE` means Sunday, Monday and Tuesday
/	Specifies increments	`5/10` means 5, 15, 25, 35, etc.

The following table lists common examples of cron expressions.

Example	Cron expression
Invoke Lambda function at 10:00am (UTC) everyday	`cron(0 10 * * ? *)`
Invoke a Lambda function 12:15pm (UTC) everyday	`cron(15 12 * * ? *)`
Invoke Lambda function at 06:00pm (UTC) every Mon-Fri	`cron(0 18 ? * MON-FRI *)`
Invoke Lambda function at 8:00am (UTC) every first day of the month	`cron(0 8 1 * ? *)`
Invoke Lambda function every 10 min Mon-Fri	`cron(0/10 * ? * MON-FRI *)`
Invoke Lambda function every 5 minutes Mon-Fri between 8:00am and 5:55pm (UTC)	`cron(0/5 8-17 ? * MON-FRI *)`

Note the following:

- Cron expressions that lead to rates faster than one minute are not supported.
- One of the day-of-month or day-of-week values must be a question mark (?).

Using AWS Lambda with Custom User Applications

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 invokes your Lambda function. Instead, the tutorial provided in this section provides sample event data that you can use and then you invoke your Lambda function manually.

When a user application invokes a Lambda function, 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 How It Works (p. 151).

For a tutorial that walks you through an example setup, see Tutorial: Using AWS Lambda with Custom User Applications (p. 269).

Tutorial: Using AWS Lambda with Custom User Applications

In this tutorial, you use the AWS CLI to create and invoke a Lambda function and explore other AWS Lambda APIs.

You'll do the following:

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

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

exports.handler = function(event, context, callback) {
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    callback(null,"Success");

};
```

Note
The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)

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 that sends 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 exercise (see Getting Started (p. 160)). The difference is that the Getting Started exercise provides a console-based experience. The console does many things for you, which simplifies the experience. When using the AWS CLI, you get the experience of making the API calls, which can help you develop a better understanding of the AWS Lambda operations. In addition to creating and invoking a Lambda function, you can explore other Lambda APIs.

Next Step

Step 1: Prepare (p. 270)

Step 1: Prepare

Make sure you have completed the following steps:

- Signed up for an AWS account and created an administrator user in the account.
- Installed and set up the AWS CLI.

For instructions, see Step 1: Set Up an AWS Account and the AWS CLI (p. 160).

Next Step

Step 2: Create a Lambda Function and Invoke It Manually (p. 270)

Step 2: Create a Lambda Function and Invoke It Manually

In this section, you do the following:

- Create a deployment package. A deployment package is a .zip file that contains your code and any dependencies. For this tutorial there are no dependencies, you only have a simple example code.
- Create an IAM role (execution role). At the time you upload the deployment package, you need to specify an IAM role (execution role) that Lambda can assume to execute the function on your behalf.

  You also grant this role the permissions that your Lambda function needs. The code in this tutorial writes logs to Amazon CloudWatch Logs. So you need to grant permissions for CloudWatch actions. For more information, see AWS Lambda Watch Logs.
- Create a Lambda function (`HelloWorld`) using the `create-function` CLI command. For more information about the underlying API and related parameters, see CreateFunction (p. 332).

Topics

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 then copy the following code.

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

exports.handler = function(event, context, callback) {
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    callback(null, "Success");

};
```

   Note
   The code sample is compliant with the Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9)
2. Save the file as `helloworld.js`.
3. Zip the `helloworld.js` file as `helloworld.zip`.

Note
To see more examples of using other AWS services within your function, including calling other Lambda functions, see AWS SDK for JavaScript

Step 2.2: Create the Execution Role (IAM Role)

When the Lambda function in this tutorial executes, it needs permissions to write logs to Amazon CloudWatch. You grant these permissions by creating an IAM role (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, see IAM Roles in the *IAM User Guide*.

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

- AWS service role of the type **AWS Lambda** – This role grants AWS Lambda permissions to assume the role.
- **AWSLambdaBasicExecutionRole** access permissions policy that you attach to the role.

For more information about IAM roles, see IAM Roles in the *IAM User Guide*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. Follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
   - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-custom-app-execution-role**).
   - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
   - In **Attach Policy**, choose **AWSLambdaBasicExecutionRole**.
3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

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 nodejs4.3 \
--profile adminuser
```

Optionally, 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 need to replace the `--zip-file` parameter by the `--code` parameter, as shown following:

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

For more information, see CreateFunction (p. 332). 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": "nodejs4.3",
    "Description": ""
}
```

Next Step

Step 3: Invoke the Lambda Function (AWS CLI) (p. 273)

Step 3: Invoke the Lambda Function (AWS CLI)

In this section, you invoke your Lambda function manually using the invoke AWS 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 (for example, `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 function execution. Alternatively, you can 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. 358).

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 tutorial, you 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 permissions to another AWS account or grant permissions to another an AWS service to execute the function, you must add a permissions to the access permissions policy associated with the function. The Amazon S3 tutorial, which uses Amazon S3 as the event source (see Tutorial: Using AWS Lambda with Amazon S3 (p. 177)), grants such permissions to Amazon S3 to invoke the function.

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

- Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.

  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 choose the **logs** link to view the CloudWatch logs directly.

Next Step

Step 4: Try More CLI Commands (AWS CLI) (p. 274)

Step 4: Try More CLI Commands (AWS CLI)

Step 4.1: List the Lambda Functions in Your Account

In this section, you try AWS Lambda list function operations. Execute the following AWS CLI `list-functions` command to retrieve a list of functions that 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. 368). The following is an example response.

```
{
    "Functions": [
        {
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "CodeSize": 412,
            "FunctionArn": "arn:aws:lambda:us-east-1:account-
id:function:ProcessKinesisRecords",
            "Handler": "ProcessKinesisRecords.handler",
            "Role": "arn:aws:iam::account-id:role/LambdaExecRole",
            "Timeout": 3,
            "LastModified": "2015-02-22T21:03:01.172+0000",
            "Runtime": "nodejs4.3",
            "Description": ""
        },
        {
            "FunctionName": "ProcessKinesisRecords",
            "MemorySize": 128,
            "CodeSize": 412,
            "FunctionArn": "arn:aws:lambda:us-east-1:account-
id:function:ProcessKinesisRecords",
            "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": "nodejs4.3",
            "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` AWS 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 a URL to Download Previously Uploaded Lambda Function Deployment Packages

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. 349).

```
$ 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": "nodejs4.3",
        "Description": ""
    }

}
```

If you want the function configuration information only (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 (AWS CLI) (p. 276)

Step 5: Delete the Lambda Function and IAM Role (AWS CLI)

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

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

Delete the 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 the *IAM User Guide*.

AWS Lambda@Edge (Preview)

Lambda@Edge (preview) allows you to run Lambda functions at the AWS Edge locations in response to CloudFront events, without provisioning or managing servers, by using the AWS Lambda serverless programming model. Before Lambda@Edge, while you could use CloudFront to deliver pages faster, customized processing still required requests to be forwarded back to compute resources at centralized servers. This slows down the end user experience. You can use Lambda@Edge in response to CloudFront events and to customize content delivered through CloudFront at reduced network latency with execution at the AWS edge locations.

Functions deployable to edge locations are written in Node.js. With Lambda@Edge, a virtually unlimited set of solutions is available, including the following:

- Write Lambda functions to inspect cookies to rewrite URLs to different versions of a site for A/B testing.
- Send different objects to your users based on the user-agent header (for example, send images that have different resolutions to users based on their devices).
- Inspect headers or authorized tokens, inserting a corresponding header and allowing access control before forwarding a request to the origin.
- Add, drop, and modify headers, and rewrite the URL path to direct users to different objects in cache.

While the runtime (`nodejs4.3-edge`) is the same as the standard Node environment supported in AWS Lambda (`nodejs4.3`), there are some additional restrictions. For example, there are no built-in libraries available. You can use Lambda functions to respond to four different trigger points for a CloudFront request:

- When a request is first received from a viewer (viewer request).
- When the Lambda function receives a response from the origin (origin response).
- When the Lambda function forwards a request to the origin (origin request).
- Before the Lambda function responds to the end viewer (viewer response).

For more information about setting up CloudFront with Lambda@Edge, see Using CloudFront with Lambda@Edge.

How It Works

After you have identified a decision that can be made at the AWS Edge location (where all the information needed to make the decision is available within the function and the request/response itself), first identify which cache behaviors and at what point in the request flow (viewer request, origin request, origin response, or viewer response) the logic applies to.

You then create a Lambda function to encapsulate the logic. Functions that run on Lambda@Edge are written in `nodejs4.3`; however, instead of selecting the standard `nodejs4.3` as the runtime of your Lambda function, you must select `nodejs4.3-edge`. Next, you use CloudFront API operations or the AWS Lambda console to associate the appropriate events (viewer request etc.) for a specific CloudFront distribution to your Lambda function. This the same model as all other event sources—for example, when triggering Lambda functions for Amazon S3 events, you associate specific events like Amazon S3 PUT for a specific bucket. Once saved, the next time an applicable request is made to your distribution, the function is propagated to the CloudFront edge, scaled, and executed as needed.

Authoring Functions for Lambda@Edge

The programming model for using Node.js with Lambda@Edge is the same as using Lambda in a region. For more information, see Programming Model (Node.js) (p. 9). You can also include third-party packages. However, when responding to *request events*, the request object should be returned as part of callback so that CloudFront can modify the request. The same applies while responding to *response events*. The response object should be returned.

```
exports.origin_response_handler = function(event, context, callback) {
    var headers = event.Records[0].cf.response.headers;
    for (var header in headers) {
        /* your custom header logic */;
    }
    callback(null, event.Records[0].cf.response);
}
```

```
exports.origin_request_handler = function(event, context, callback) {
    var request = event.Records[0].cf.request;
    request.uri = /* your custom uri logic  */;
   var headers = request.headers;
    for (var header in headers) {
       /* your custom header logic */;
    }
     callback(null , event.Records[0].cf.request);
}
```

Note
Lambda@Edge functions will not have write access to write to the local file system, and cannot make remote network calls to external services.

Permissions

If you are using the AWS CLI, Lambda@Edge requires an additional permission. You will need to add the following principal to the resource policy, as shown below:

```
aws lambda add-permission
```

```
--function-name arn
--statement-id statement-id
--action lambda:GetFunction
--principal edgelambda.amazonaws.com
```

When using the Lambda console, these permissions are added automatically.

Testing and Debugging

Lambda@Edge functions can be tested on the Lambda console with test events modeled after the Request or Response events. For sample events, see Lambda@Edge Event Structure. However, the testing in the console only validates logic, and does not apply service limits specific to Lambda@Edge which come into play if your function is deployed to the AWS Edge. To avoid discrepancies in behavior while testing in a region, ensure that your function does not make remote calls (either through the AWS SDK or via direct http/https requests) or use environment variables, and configure the function to approximate its eventual edge deployment by selecting 128MB and 1 second of duration. Note that the actual duration limit during the preview is 50ms. For more information, see Lambda@Edge Limits (p. 280). Functions running on Lambda@Edge do not have CloudWatch Logs available at this time. However, errors in Lambda@Edge functions will appear in the corresponding CloudFront distribution request logs.

Simple Setup

In the Lambda console:

1. Choose the **cloudfront-ab-test** blueprint.
2. In **Triggers**, choose the CloudFront distribution, cache behavior, and an event type.
   - Choose **CloudFront** from the service list.
   - Choose **distribution**.
   - Choose **cache behavior identified by the URL pattern**.

     With Lambda@Edge, you associate functions at the cache behavior level. To specify different caching policies (how long to cache an object, what headers to include in the cache key, which origin has the content) you have the following options:
     - `*.jpg`
     - `*.php`
     - `* (default)`
   - Choose the **CloudFront** event type.
3. Choose **Edge-Node.js 4.3** as the runtime for your function. Edge-Node.js 4.3 is the only supported runtime for functions that run on Lambda@Edge today.
4. Choose **Create function**. When the specified CloudFront distribution receives a request, the function is propagated to the edge location, and your function will start triggering to the associated event.

Note

Note that you need to add a special permission to allow Lambda@Edge to propagate your functions to edge locations. When using the console, these are automatically added. If you create your functions using the API or AWS CLI, see Permissions (p. 278).

Lambda@Edge Limits

Due to the constrained execution environment, Lambda@Edge has additional restrictions over and above the default Lambda limits. Functions running in Edge locations do not count against your concurrency limit for the region. For more information, see AWS Lambda Limits (p. 285).

Item	Default Limit
Max memory setting	128 MB
Maximum duration	50 ms
Size of code/dependencies that you can zip into a deployment package (uncompressed zip/jar size)	1 MB
Maximum Global TPS	100

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. 322)
- CreateEventSourceMapping (p. 328)
- CreateFunction (p. 332)

  (The `ZipFile` parameter is omitted from the CloudTrail logs for `CreateFunction`.)
- DeleteEventSourceMapping (p. 339)
- DeleteFunction (p. 341)
- GetEventSourceMapping (p. 347)
- GetFunction (p. 349)
- GetFunctionConfiguration (p. 352)
- GetPolicy (p. 356)
- ListEventSourceMappings (p. 366)
- ListFunctions (p. 368)
- RemovePermission (p. 376)
- UpdateEventSourceMapping (p. 380)

- UpdateFunctionCode (p. 383)

  (The `ZipFile` parameter is omitted from the CloudTrail logs for `UpdateFunctionCode`.)
- UpdateFunctionConfiguration (p. 387)

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 Working with CloudTrail Log Files.

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"
    }
  ]
}
```

Note
The `eventName` may include date and version information, such as `"GetFunction20150331"`, but it is still referring to the same public API.

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.
- Instantiate AWS clients outside the scope of the handler to take advantage of connection re-use.
- Make sure you have set `+rx` permissions on your files in the uploaded ZIP to ensure Lambda can execute code on your behalf.
- 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

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	Default 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	300 seconds
Invoke (p. 358) request body payload size (RequestResponse)	6 MB
Invoke (p. 358) request body payload size (Event)	128 K
Invoke (p. 358) response body payload size (RequestResponse)	6 MB

The following table lists the Lambda account limits per region.

AWS Lambda Account Limits Per Region

Resource	Default Limit
Concurrent executions (see Lambda Function Concurrent Executions (p. 152))	100

To request a limit increase for concurrent execution

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 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.

The following table lists service limits for deploying a Lambda function.

AWS Lambda Deployment Limits

Item	Default Limit
Lambda function deployment package size (.zip/.jar file)	50 MB
Total size of all the deployment packages that can be uploaded per region	75 GB
Size of code/dependencies that you can zip into a deployment package (uncompressed zip/jar size)	250 MB
Total size of environment variables set	4 KB

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.

Authentication and Access Control for AWS Lambda

Access to AWS Lambda requires credentials that AWS can use to authenticate your requests. Those credentials must have permissions to access AWS resources, such as an AWS Lambda function or an Amazon S3 bucket. The following sections provide details on how you can use AWS Identity and Access Management (IAM) and Lambda to help secure your resources by controlling who can access them:

Authentication

You can access AWS as any of the following types of identities:

- **AWS account root user** – When you sign up for AWS, you provide an email address and password that is associated with your AWS account. These are your *root credentials* and they provide complete access to all of your AWS resources.

  Important
  For security reasons, we recommend that you use the root credentials only to create an *administrator user*, which is an *IAM user* with full permissions to your AWS account. Then, you can use this administrator user to create other IAM users and roles with limited permissions. For more information, see IAM Best Practices and Creating an Admin User and Group in the *IAM User Guide*.
- **IAM user** – An IAM user is simply an identity within your AWS account that has specific custom permissions (for example, permissions to create a function in Lambda). You can use an IAM user name and password to sign in to secure AWS webpages like the AWS Management Console, AWS Discussion Forums, or the AWS Support Center.

In addition to a user name and password, you can also generate access keys for each user. You can use these keys when you access AWS services programmatically, either through one of the several SDKs or by using the AWS Command Line Interface (CLI). The SDK and CLI tools use the access keys to cryptographically sign your request. If you don't use the AWS tools, you must sign the request yourself. Lambda supports *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the *AWS General Reference*.

- **IAM role** – An IAM role is another IAM identity you can create in your account that has specific permissions. It is similar to an *IAM user*, but it is not associated with a specific person. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources. IAM roles with temporary credentials are useful in the following situations:

  - **Federated user access** – Instead of creating an IAM user, you can use preexisting user identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated Users and Roles in the *IAM User Guide*.

  - **Cross-account access** – You can use an IAM role in your account to grant another AWS account permissions to access your account's resources. For an example, see Tutorial: Delegate Access Across AWS Accounts Using IAM Roles in the *IAM User Guide*.

  - **AWS service access** – You can use an IAM role in your account to grant an AWS service permissions to access your account's resources. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data stored in the bucket into an Amazon Redshift cluster. For more information, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.

  - **Applications running on Amazon EC2** – Instead of storing access keys within the EC2 instance for use by applications running on the instance and making AWS API requests, you can use an IAM role to manage temporary credentials for these applications. To assign an AWS role to an EC2 instance and make it available to all of its applications, you can create an instance profile that is attached to the instance. An instance profile contains the role and enables programs running on the EC2 instance to get temporary credentials. For more information, see Using Roles for Applications on Amazon EC2 in the *IAM User Guide*.

Access Control

You can have valid credentials to authenticate your requests, but unless you have permissions you cannot create or access AWS Lambda resources. For example, you must have permissions to create a Lambda function, add an event source, and publish a version of your Lambda function.

The following sections describe how to manage permissions for AWS Lambda. We recommend that you read the overview first.

- Overview of Managing Access Permissions to Your AWS Lambda Resources (p. 289)

- Using Identity-Based Policies (IAM Policies) for AWS Lambda (p. 293)
- Using Resource-Based Policies for AWS Lambda (Lambda Function Policies) (p. 306)

Overview of Managing Access Permissions to Your AWS Lambda Resources

Every AWS resource is owned by an AWS account, and permissions to create or access a resource are governed by permissions policies. An account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles), and some services (such as AWS Lambda) also support attaching permissions policies to resources.

Note
An *account administrator* (or administrator user) is a user with administrator privileges. For more information, see IAM Best Practices in the *IAM User Guide*.

When granting permissions, you decide who is getting the permissions, the resources they get permissions for, and the specific actions that you want to allow on those resources.

Topics

AWS Lambda Resources and Operations

In AWS Lambda, the primary resources are a Lambda *function* and an *event source mapping*. You create an event source mapping in the AWS Lambda pull model to associate a Lambda function with an event source. For more information, see Event Source Mapping (p. 120).

AWS Lambda also supports additional resource types, *alias* and *version*. However, you can create aliases and versions only in the context of an existing Lambda function. These are referred to as *subresources*.

These resources and subresources have unique Amazon Resource Names (ARNs) associated with them as shown in the following table.

Resource Type	ARN Format
Function	arn:aws:lambda:*region*:*account-id*:function:*function-name*
Function alias	arn:aws:lambda:*region*:*account-id*:function:*function-name*:*alias-name*
Function version	arn:aws:lambda:*region*:*account-id*:function:*function-name*:*version*
Event source mapping	arn:aws:lambda:*region*:*account-id*:event-source-mappings:*event-source-mapping-id*

AWS Lambda provides a set of operations to work with the Lambda resources. For a list of available operations, see Actions (p. 320).

Understanding Resource Ownership

A *resource owner* is the AWS account that created the resource. That is, the resource owner is the AWS account of the *principal entity* (the root account, an IAM user, or an IAM role) that authenticates the request that creates the resource. The following examples illustrate how this works:

- If you use the root account credentials of your AWS account to create a Lambda function, your AWS account is the owner of the resource (in Lambda, the resource is the Lambda function).
- If you create an IAM user in your AWS account and grant permissions to create a Lambda function to that user, the user can create a Lambda function. However, your AWS account, to which the user belongs, owns the Lambda function resource.
- If you create an IAM role in your AWS account with permissions to create a Lambda function, anyone who can assume the role can create a Lambda function. Your AWS account, to which the role belongs, owns the Lambda function resource.

Managing Access to Resources

A *permissions policy* describes who has access to what. The following section explains the available options for creating permissions policies.

> **Note**
> This section discusses using IAM in the context of AWS Lambda. It doesn't provide detailed information about the IAM service. For complete IAM documentation, see What Is IAM? in the *IAM User Guide*. For information about IAM policy syntax and descriptions, see AWS IAM Policy Reference in the *IAM User Guide*.

Policies attached to an IAM identity are referred to as *identity-based* policies (IAM polices) and policies attached to a resource are referred to as *resource-based* policies. AWS Lambda supports both identity-based (IAM policies) and resource-based policies.

Topics

Identity-Based Policies (IAM Policies)

You can attach policies to IAM identities. For example, you can do the following:

- **Attach a permissions policy to a user or a group in your account** – An account administrator can use a permissions policy that is associated with a particular user to grant permissions for that user to create a Lambda function.
- **Attach a permissions policy to a role (grant cross-account permissions)** – You can attach an identity-based permissions policy to an IAM role to grant cross-account permissions. For example, the administrator in Account A can create a role to grant cross-account permissions to another AWS account (for example, Account B) or an AWS service as follows:

  1. Account A administrator creates an IAM role and attaches a permissions policy to the role that grants permissions on resources in Account A.
  2. Account A administrator attaches a trust policy to the role identifying Account B as the principal who can assume the role.

3. Account B administrator can then delegate permissions to assume the role to any users in Account B. Doing this allows users in Account B to create or access resources in Account A. The principal in the trust policy can also be an AWS service principal if you want to grant an AWS service permissions to assume the role.

For more information about using IAM to delegate permissions, see Access Management in the *IAM User Guide*.

The following is an example policy that grants permissions for the `lambda:ListFunctions` action on all resources. In the current implementation, Lambda doesn't support identifying specific resources using the resource ARNs (also referred to as resource-level permissions) for some of the API actions, so you must specify a wildcard character (*).

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListExistingFunctions",
            "Effect": "Allow",
            "Action": [
                "lambda:ListFunctions"
            ],
            "Resource": "*"
        }
    ]
}
```

For more information about using identity-based policies with Lambda, see Using Identity-Based Policies (IAM Policies) for AWS Lambda (p. 293). For more information about users, groups, roles, and permissions, see Identities (Users, Groups, and Roles) in the *IAM User Guide*.

Resource-Based Policies (Lambda Function Policies)

Each Lambda function can have resource-based permissions policies associated with it. For Lambda, a Lambda function is the primary resource and these policies are referred to as *Lambda function policies*. You can use a Lambda function policy to grant cross-account permissions as an alternative to using identity-based policies with IAM roles. For example, you can grant Amazon S3 permissions to invoke your Lambda function by simply adding permissions to the Lambda function policy instead of creating an IAM role.

Important
Lambda function policies are primarily used when you are setting up an event source in AWS Lambda to grant a service or an event source permissions to invoke your Lambda function (see Invoke (p. 358)). An exception to this is when an event source (for example, Amazon DynamoDB or Amazon Kinesis) uses the pull model, where permissions are managed in the Lambda function execution role instead. For more information, see Event Source Mapping (p. 120).

The following is an example Lambda function policy that has one statement. The statement allows the Amazon S3 service principal permission for the `lambda:InvokeFunction` action on a Lambda function called HelloWorld. The condition ensures that the bucket where the event occurred is owned by the same account that owns the Lambda function.

```
{
   "Policy":{
      "Version":"2012-10-17",
      "Statement":[
         {
            "Effect":"Allow",
            "Principal":{
               "Service":"s3.amazonaws.com"
            },
            "Action":"lambda:InvokeFunction",
            "Resource":"arn:aws:lambda:region:account-
id:function:HelloWorld",
            "Sid":"65bafc90-6a1f-42a8-a7ab-8aa9bc877985",
            "Condition":{
               "StringEquals":{
                  "AWS:SourceAccount":"account-id"
               },
               "ArnLike":{
                  "AWS:SourceArn":"arn:aws:s3:::ExampleBucket"
               }
            }
         }
      ]
   }
}
```

For more information about using resource-based policies with Lambda, see Using Resource-Based Policies for AWS Lambda (Lambda Function Policies) (p. 306). For additional information about using IAM roles (identity-based policies) as opposed to resource-based policies, see How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.

Specifying Policy Elements: Actions, Effects, Resources, and Principals

For each AWS Lambda resource (see AWS Lambda Resources and Operations (p. 289)), the service defines a set of API operations (see Actions (p. 320)). To grant permissions for these API operations, Lambda defines a set of actions that you can specify in a policy. Note that, performing an API operation can require permissions for more than one action. When granting permissions for specific actions, you also identify the resource on which the actions are allowed or denied.

The following are the most basic policy elements:

- **Resource** – In a policy, you use an Amazon Resource Name (ARN) to identify the resource to which the policy applies. For more information, see AWS Lambda Resources and Operations (p. 289).
- **Action** – You use action keywords to identify resource operations that you want to allow or deny. For example, the `lambda:InvokeFunction` permission allows the user permissions to perform the AWS Lambda `Invoke` operation.
- **Effect** – You specify the effect when the user requests the specific action—this can be either allow or deny. If you don't explicitly grant access to (allow) a resource, access is implicitly denied. You can also explicitly deny access to a resource, which you might do to make sure that a user cannot access it, even if a different policy grants access.
- **Principal** – In identity-based policies (IAM policies), the user that the policy is attached to is the implicit principal. For resource-based policies, you specify the user, account, service, or other entity that you want to receive permissions (applies to resource-based policies only).

To learn more about IAM policy syntax and descriptions, see AWS IAM Policy Reference in the *IAM User Guide*.

For a table showing all of the AWS Lambda API actions and the resources that they apply to, see Lambda API Permissions: Actions, Resources, and Conditions Reference (p. 310).

Specifying Conditions in a Policy

When you grant permissions, you can use the IAM policy language to specify the conditions when a policy should take effect. For example, you might want a policy to be applied only after a specific date. For more information about specifying conditions in a policy language, see Condition in the *IAM User Guide*.

To express conditions, you use predefined condition keys. There are no condition keys specific to Lambda. However, there are AWS-wide condition keys that you can use as appropriate. For a complete list of AWS-wide keys, see Available Keys for Conditions in the *IAM User Guide*.

Using Identity-Based Policies (IAM Policies) for AWS Lambda

This topic provides examples of identity-based policies in which an account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles).

> **Important**
> We recommend that you first review the introductory topics that explain the basic concepts and options available for you to manage access to your AWS Lambda resources. For more information, see Overview of Managing Access Permissions to Your AWS Lambda Resources (p. 289).

The sections in this topic cover the following:

- Permissions Required to Use the AWS Lambda Console (p. 294)
- AWS Managed (Predefined) Policies for AWS Lambda (p. 294)
- Customer Managed Policy Examples (p. 295)

The following shows an example of a permissions policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CreateFunctionPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction"
            ],
            "Resource": "*"
        },
        {
            "Sid": "PermissionToPassAnyRole",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::account-id:role/*"
        }
    ]
}
```

The policy has two statements:

- The first statement grants permissions for the AWS Lambda action (`lambda:CreateFunction`) on a resource by using the Amazon Resource Name (ARN) for the Lambda function. Currently, AWS Lambda doesn't support permissions for this particular action at the resource-level. Therefore, the policy specifies a wildcard character (*) as the `Resource` value.
- The second statement grants permissions for the IAM action (`iam:PassRole`) on IAM roles. The wildcard character (*) at the end of the `Resource` value means that the statement allows permission for the `iam:PassRole` action on any IAM role. To limit this permission to a specific role, replace the wildcard character (*) in the resource ARN with the specific role name.

The policy doesn't specify the `Principal` element because in an identity-based policy you don't specify the principal who gets the permission. When you attach policy to a user, the user is the implicit principal. When you attach a permission policy to an IAM role, the principal identified in the role's trust policy gets the permissions.

For a table showing all of the AWS Lambda API actions and the resources that they apply to, see Lambda API Permissions: Actions, Resources, and Conditions Reference (p. 310).

Permissions Required to Use the AWS Lambda Console

The AWS Lambda console provides an integrated environment for you to create and manage Lambda functions. The console provides many features and workflows that often require permissions to create a Lambda function in addition to the API-specific permissions documented in the Lambda API Permissions: Actions, Resources, and Conditions Reference (p. 310). For more information about these additional console permissions, see Permissions Required to Use the AWS Lambda Console (p. 298).

AWS Managed (Predefined) Policies for AWS Lambda

AWS addresses many common use cases by providing standalone IAM policies that are created and administered by AWS. Managed policies grant necessary permissions for common use cases so you can avoid having to investigate what permissions are needed. For more information, see AWS Managed Policies in the *IAM User Guide*.

The following AWS managed policies, which you can attach to users in your account, are specific to AWS Lambda and are grouped by use case scenario:

- **AWSLambdaReadOnlyAccess** – Grants read-only access to AWS Lambda resources. Note that this policy doesn't grant permission for the `lambda:InvokeFunction` action. If you want a user to invoke a Lambda function, you can also attach the `AWSLambdaRole` AWS managed policy.
- **AWSLambdaFullAccess** – Grants full access to AWS Lambda resources.
- **AWSLambdaRole** – Grants permissions to invoke any Lambda function.

Note
You can review these permissions policies by signing in to the IAM console and searching for specific policies there.

In addition, there are other AWS-managed policies that are suitable for use with IAM role (execution role) you specify at the time of creating a Lambda function. For more information, see AWS Lambda Permissions Model (p. 155).

You can also create your own custom IAM policies to allow permissions for AWS Lambda API actions and resources. You can attach these custom policies to the IAM users or groups that require those permissions or to custom execution roles (IAM roles) that you create for your Lambda functions.

Customer Managed Policy Examples

The examples in this section provide a group of sample policies that you can attach to a user. If you are new to creating policies, we recommend that you first create an IAM user in your account and attach the policies to the user in sequence, as outlined in the steps in this section.

You can use the console to verify the effects of each policy as you attach the policy to the user. Initially, the user doesn't have permissions and the user won't be able to do anything in the console. As you attach policies to the user, you can verify that the user can perform various actions in the console.

We recommend that you use two browser windows: one to create the user and grant permissions, and the other to sign in to the AWS Management Console using the user's credentials and verify permissions as you grant them to the user.

For examples that show how to create an IAM role that you can use as an execution role for your Lambda function, see Creating IAM Roles in the *IAM User Guide*.

Example Steps

Step 1: Create an IAM User

First, you need to create an IAM user, add the user to an IAM group with administrative permissions, and then grant administrative permissions to the IAM user that you created. You can then access AWS using a special URL and that IAM user's credentials.

For instructions, see Creating Your First IAM User and Administrators Group in the *IAM User Guide*.

Step 2: Allow a User to List Lambda Functions

An IAM user in your account must have permissions for the `lambda:ListFunctions` action before the user can see anything in the console. When you grant these permissions, the console can show the list of Lambda functions in the AWS account created in the specific AWS Region the user belongs to.

```
{
   "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListExistingFunctions",
            "Effect": "Allow",
            "Action": [
                "lambda:ListFunctions"
            ],
            "Resource": "*"
        }
    ]
}
```

Step 3: Allow a User to View Details of a Lambda Function

A user can select a Lambda function and view details of the function (such as aliases, versions, and other configuration information), provided that the user has permissions for the following AWS Lambda actions:

```
{
    "Version": "2012-10-17",
    "Statement": [
          {
              "Sid": "DisplayFunctionDetailsPermissions",
              "Effect": "Allow",
              "Action": [
                  "lambda:ListVersionsByFunction",
                  "lambda:ListAliases",
                  "lambda:GetFunction",
                  "lambda:GetFunctionConfiguration",
                  "lambda:ListEventSourceMappings",
                  "lambda:GetPolicy"
              ],
              "Resource": "*"
          }
    ]
}
```

Step 4: Allow a User to Invoke a Lambda Function

If you want to allow a user permissions to manually invoke a function, you need to grant permissions for the `lambda:InvokeFunction` action, as shown following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "InvokePermission",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "*"
        }
    ]
}
```

Step 5: Allow a User to Monitor a Lambda Function and View CloudWatch Logs

When a user invokes a Lambda function, AWS Lambda executes it and returns results. The user needs additional permissions to monitor the Lambda function.

To enable the user to see the Lambda function's CloudWatch metrics on the console's **Monitoring** tab, or on the grid view on the console home page, you must grant the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CloudWatchPermission",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricStatistics"
            ],
            "Resource": "*"
        }
    ]
}
```

To enable a user to click the links to CloudWatch Logs in the AWS Lambda console and view log output in CloudWatch Logs, you must grant the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CloudWatchLogsPerms",
            "Effect": "Allow",
            "Action": [
                "cloudwatchlog:DescribeLogGroups",
                "cloudwatchlog:DescribeLogStreams",
                "cloudwatchlog:GetLogEvents"

              ],
            "Resource": "arn:aws:logs:region:account-id:log-group:/aws/
lambda/*"
        }
    ]
}
```

Step 6: Allow a User to Create a Lambda Function

If you want a user to be able to create a Lambda function, you must grant the following permissions. The permissions for IAM-related actions are required because when a user creates a Lambda function, the user needs to select an IAM execution role, which AWS Lambda assumes to execute the Lambda function.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListExistingRolesAndPolicies",
            "Effect": "Allow",
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "CreateFunctionPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction"
            ],
            "Resource": "*"
        },
        {
            "Sid": "PermissionToPassAnyRole",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::account-id:role/*"
        }
    ]
}
```

If you want a user to be able to create an IAM role when the user is creating a Lambda function, the user needs permissions to perform the `iam:PutRolePolicy` action, as shown following:

```
{
    "Sid": "CreateARole",
    "Effect": "Allow",
    "Action": [
        "iam:PutRolePolicy"
    ],
    "Resource": "arn:aws:iam::account-id:role/*"
}
```

Important
Each IAM role has a permissions policy attached to it, which grants specific permissions to the role. Regardless of whether the user creates a new role or uses an existing role, the user must have permissions for all of the actions granted in the permissions policy associated with the role. You must grant the user additional permissions accordingly.

Permissions Required to Use the AWS Lambda Console

To take advantage of the integrated experience provided by the AWS Lambda console, a user must often have more permissions than the API-specific permissions described in the references table, depending on what you want the user to be able to do. For more information about Lambda API operations, see Lambda API Permissions: Actions, Resources, and Conditions Reference (p. 310).

For example, suppose you allow an IAM user in your account permissions to create a Lambda function to process Amazon S3 object-created events. To enable the user to configure Amazon S3 as the event source, the console drop-down list will display a list of your buckets. However, the console can show the bucket list only if the signed-in user has permissions for the relevant Amazon S3 actions.

The following sections describe required additional permissions for different integration points. For information about integration points, see How It Works (p. 151).

If you are new to managing permissions, we recommend that you start with the example walkthrough where you create an IAM user, grant the user incremental permissions, and verify the permissions work using the AWS Lambda console (see Customer Managed Policy Examples (p. 295)).

Topics

Note
All of these permissions policies grant the specific AWS services permissions to invoke a Lambda function. The user who is configuring this integration must have permissions to invoke the Lambda function. Otherwise, the user can't set the configuration. You can attach the `AWSLambdaRole` AWS managed (predefined) permissions policy to the user to provide these permissions.

Amazon API Gateway

When you configure an API endpoint in the console, the console makes several API Gateway API calls. These calls require permissions for the `apigateway:*` action, as shown following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ApiGatewayPermissions",
            "Effect": "Allow",
            "Action": [
                "apigateway:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AddPermissionToFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission",
                "lambda:GetPolicy"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        },
        {
            "Sid": "ListEventSourcePerm",
            "Effect": "Allow",
            "Action": [
                "lambda:ListEventSourceMappings"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon CloudWatch Events

You can schedule when to invoke a Lambda function. After you select an existing CloudWatch Events rule (or create a new one), AWS Lambda creates a new target in CloudWatch that invokes your Lambda function. For target creation to work, you need to grant the following additional permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EventPerms",
            "Effect": "Allow",
            "Action": [
                "events:PutRule",
                "events:ListRules",
                "events:ListRuleNamesByTarget",
                "events:PutTargets",
                "events:RemoveTargets",
                "events:DescribeRule",
                "events:TestEventPattern",
                "events:ListTargetsByRule",
                "events:DeleteRule"

              ],
            "Resource": "arn:aws:events:region:account-id:*"
        },
        {
            "Sid": "AddPermissionToFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission",
                "lambda:GetPolicy"
              ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        }
    ]
}
```

Amazon CloudWatch Logs

You can have the Amazon CloudWatch Logs service publish events and invoke your Lambda function. When you configure this service as an event source, the console lists log groups in your account. For this listing to occur, you need to grant the `logs:DescribeLogGroups` permissions, as shown following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CloudWatchLogsPerms",
            "Effect": "Allow",
            "Action": [
                "logs:FilterLogEvents",
                "logs:DescribeLogGroups",
                "logs:PutSubscriptionFilter",
                "logs:DescribeSubscriptionFilters",
                "logs:DeleteSubscriptionFilter",
                "logs:TestMetricFilter"
            ],
            "Resource": "arn:aws:logs:region:account-id:*"
        },
        {
            "Sid": "AddPermissionToFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission",
                "lambda:GetPolicy"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        },
        {
            "Sid": "ListEventSourceMappingsPerms",
            "Effect": "Allow",
            "Action": [
                "lambda:ListEventSourceMappings"
            ],
            "Resource": "*"
        }
    ]
}
```

Note
The additional permissions shown are required for managing subscription filters.

Amazon Cognito

The console lists identity pools in your account. After you select a pool, you can configure the pool to have the **Cognito sync trigger** as the event source type. To do this, you need to grant the following additional permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CognitoPerms1",
            "Effect": "Allow",
            "Action": [
                "cognito-identity:ListIdentityPools"
            ],
            "Resource": [
                "arn:aws:cognito-identity:region:account-id:*"
            ]
        },
        {
            "Sid": "CognitoPerms2",
            "Effect": "Allow",
            "Action": [
                "cognito-sync:GetCognitoEvents",
                "cognito-sync:SetCognitoEvents"
            ],
            "Resource": [
                "arn:aws:cognito-sync:region:account-id:*"
            ]
        },
        {
            "Sid": "AddPermissionToFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission",
                "lambda:GetPolicy"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        },
        {
            "Sid": "ListEventSourcePerms",
            "Effect": "Allow",
            "Action": [
                "lambda:ListEventSourceMappings"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon DynamoDB

The console lists all of the tables in your account. After you select a table, the console checks to see if a DynamoDB stream exists for that table. If not, it creates the stream. If you want the user to be able to configure a DynamoDB stream as an event source for a Lambda function, you need to grant the following additional permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DDBpermissions1",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:DescribeTable",
                "dynamodb:UpdateTable"
            ],
            "Resource": "arn:aws:dynamodb:region:account-id:table/*"
        },
        {
            "Sid": "DDBpermissions2",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ListStreams",
                "dynamodb:ListTables"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaGetPolicyPerm",
            "Effect": "Allow",
            "Action": [
                "lambda:GetPolicy"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        },
        {
            "Sid": "LambdaEventSourcePerms",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateEventSourceMapping",
                "lambda:DeleteEventSourceMapping",
                "lambda:GetEventSourceMapping",
                "lambda:ListEventSourceMappings",
                "lambda:UpdateEventSourceMapping"
            ],
            "Resource": "*"
        }
    ]
}
```

Important
For a Lambda function to read from a DynamoDB stream, the execution role associated with the Lambda function must have the correct permissions. Therefore, the user must also have the same permissions before you can grant the permissions to the execution role. You can grant these permissions by attaching the `AWSLambdaDynamoDBExecutionRole` predefined policy, first to the user and then to the execution role.

Amazon Kinesis Streams

The console lists all Amazon Kinesis streams in your account. After you select a stream, the console creates event source mappings in AWS Lambda. For this to work, you need to grant the following additional permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PermissionForDescribeStream",
            "Effect": "Allow",
            "Action": [
                "kinesis:DescribeStream"
            ],
            "Resource": "arn:aws:kinesis:region:account-id:stream/*"
        },
        {
            "Sid": "PermissionForListStreams",
            "Effect": "Allow",
            "Action": [
                "kinesis:ListStreams"
            ],
            "Resource": "*"
        },
        {
            "Sid": "PermissionForGetFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:GetPolicy"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        },
        {
            "Sid": "LambdaEventSourcePerms",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateEventSourceMapping",
                "lambda:DeleteEventSourceMapping",
                "lambda:GetEventSourceMapping",
                "lambda:ListEventSourceMappings",
                "lambda:UpdateEventSourceMapping"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon S3

The console prepopulates the list of buckets in the AWS account and finds the bucket location for each bucket. When you configure Amazon S3 as an event source, the console updates the bucket notification configuration. For this to work, you need to grant the following additional permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3Permissions",
            "Effect": "Allow",
            "Action": [
                "s3:GetBucketLocation",
                "s3:GetBucketNotification",
                "s3:PutBucketNotification",
                "s3:ListAllMyBuckets"
            ],
            "Resource": "arn:aws:s3:::*"
        },
        {
            "Sid": "AddPermissionToFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        }
    ]
}
```

Amazon SNS

The console lists Amazon Simple Notification Service (Amazon SNS) topics in your account. After you select a topic, AWS Lambda subscribes your Lambda function to that Amazon SNS topic. For this work, you need to grant the following additional permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SNSPerms",
            "Effect": "Allow",
            "Action": [
                "sns:ListSubscriptions",
                "sns:ListSubscriptionsByTopic",
                "sns:ListTopics",
                "sns:Subscribe",
                "sns:Unsubscribe"
            ],
            "Resource": "arn:aws:sns:region:account-id:*"
        },
        {
            "Sid": "AddPermissionToFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission",
                "lambda:GetPolicy"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        },
        {
            "Sid": "LambdaListESMappingsPerms",
            "Effect": "Allow",
            "Action": [
                "lambda:ListEventSourceMappings"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS IoT

The console lists all of the AWS IoT rules. After you select a rule, the console populates the rest of the information associated with that rule in the user interface. If you select an existing rule, the console updates it with information so that events are sent to AWS Lambda. You can also create a new rule. To do these things, the user must have the following additional permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "IoTperms",
            "Effect": "Allow",
            "Action": [
                "iot:GetTopicRule",
                "iot:CreateTopicRule",
                "iot:ReplaceTopicRule"
            ],
            "Resource": "arn:aws:iot:region:account-id:*"
        },
        {
            "Sid": "IoTlistTopicRulePerms",
            "Effect": "Allow",
            "Action": [
                "iot:ListTopicRules"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaPerms",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission",
                "lambda:GetPolicy"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:*"
        }
    ]
}
```

Using Resource-Based Policies for AWS Lambda (Lambda Function Policies)

A Lambda function is one of the resources in AWS Lambda. You can add permissions to the policy associated with a Lambda function. Permissions policies attached to Lambda functions are referred to as *resource-based policies* (or *Lambda function policies* in Lambda). You use Lambda function policies to manage Lambda function invocation permissions (see Invoke (p. 358)).

Important
Before you create resource-based policies, we recommend that you first review the introductory topics that explain the basic concepts and options available for you to manage access to your AWS Lambda resources. For more information, see Overview of Managing Access Permissions to Your AWS Lambda Resources (p. 289).

Lambda function policies are primarily used when you are setting up an event source in AWS Lambda to grant a service or an event source permissions to invoke your Lambda function (see Invoke (p. 358)). An exception to this is when an event source (for example, Amazon DynamoDB or Amazon Kinesis) uses the pull model, where permissions are managed in the Lambda function execution role instead. For more information, see Event Source Mapping (p. 120).

Lambda function policies also make it easy to grant cross-account permissions to invoke your Lambda function. Suppose you want to grant cross-account permissions (for example, permissions to Amazon S3) to invoke your Lambda function. Instead of creating an IAM role to grant cross-account permissions, you can add the relevant permissions in a Lambda function policy.

Note
If the custom application and the Lambda function it invokes belong to the same AWS account, you don't need to grant explicit permissions using the policy attached to the Lambda function.

AWS Lambda provides the following API operations to manage a permissions policy associated with a Lambda function:

- AddPermission (p. 322)
- GetPolicy (p. 356)

- RemovePermission (p. 376)

> **Note**
> The AWS Lambda console is the easiest way to manage event sources and their permissions in a Lambda function policy. If the AWS service console for the event source supports configuring event source mapping, you can use that console too. As you configure new event sources or modify existing event sources, the console automatically modifies the permissions policy associated with the Lambda function.

The console doesn't support directly modifying permissions in a function policy. You must use either the AWS CLI or the AWS SDKs. The following are AWS CLI examples of the API operations listed earlier in this topic:

Examples

Example 1: Allow Amazon S3 to Invoke a Lambda Function

To grant Amazon S3 permission to invoke a Lambda function, you configure permissions as follows:

- Specify `s3.amazonaws.com` as the `principal` value.
- Specify `lambda:InvokeFunction` as the `action` for which you are granting permissions.

To ensure that the event is generated from a specific bucket that is owned by a specific AWS account, you also specify the following:

- Specify the bucket ARN as the `source-arn` value to restrict events from a specific bucket.
- Specify the AWS account ID that owns the bucket, to ensure that the named bucket is owned by the account.

The following example AWS CLI command adds a permission to the `helloworld` Lambda function policy granting Amazon S3 permissions to invoke the function.

```
aws lambda add-permission \
--region us-west-2 \
--function-name helloworld \
--statement-id 1 \
--principal s3.amazonaws.com \
--action lambda:InvokeFunction \
--source-arn arn:aws:s3:::examplebucket \
--source-account 111111111111 \
--profile adminuser
```

The example assumes that the `adminuser` (who has full permissions) is adding this permission. Therefore, the `--profile` parameter specifies the `adminuser` profile.

In response, AWS Lambda returns the following JSON code. The `Statement` value is a JSON string version of the statement added to the Lambda function policy.

```
{
    "Statement": "{\"Condition\":{\"StringEquals\":{\"AWS:SourceAccount\":
\"111111111111\"},
                   \"ArnLike\":{\"AWS:SourceArn\":
\"arn:aws:s3:::examplebucket\"}},
                  \"Action\":[\"lambda:InvokeFunction\"],
                  \"Resource\":\"arn:aws:lambda:us-
west-2:111111111111:function:helloworld\",
                  \"Effect\":\"Allow\",\"Principal\":{\"Service\":
\"s3.amazonaws.com\"},
                  \"Sid\":\"1\"}"
}
```

For information about the push model, see Event Source Mapping (p. 120).

Example 2: Allow Amazon API Gateway to Invoke a Lambda Function

To grant permissions to allow Amazon API Gateway to invoke a Lambda function, do the following:

- Specify `apigateway.amazonaws.com` as the `principal` value.
- Specify `lambda:InvokeFunction` as the action for which you are granting permissions.
- Specify the API Gateway endpoint ARN as the `source-arn` value.

The following example AWS CLI command adds a permission to the `helloworld` Lambda function policy granting API Gateway permissions to invoke the function.

```
aws lambda add-permission \
--region us-west-2 \
--function-name helloworld \
--statement-id 5 \
--principal apigateway.amazonaws.com \
--action lambda:InvokeFunction \
--source-arn arn:aws:execute-api:region:account-id:api-id/stage/method/
resource-path \
--profile adminuser
```

In response, AWS Lambda returns the following JSON code. The `Statement` value is a JSON string version of the statement added to the Lambda function policy.

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
\"arn:aws:apigateway:us-east-1::my-api-id:/test/petstorewalkthrough/pets\"}},
                  \"Action\":[\"lambda:InvokeFunction\"],
                  \"Resource\":\"arn:aws:lambda:us-west-2:account-
id:function:helloworld\",
                  \"Effect\":\"Allow\",
                  \"Principal\":{\"Service\":\"apigateway.amazonaws.com\"},
                  \"Sid\":\"5\"}"
}
```

Example 3: Allow a User Application Created by Another AWS Account to Invoke a Lambda Function (Cross-Account Scenario)

To grant permissions to another AWS account (that is, to create a cross-account scenario), you specify the AWS account ID as the `principal` value as shown in the following AWS CLI command:

```
aws lambda add-permission \
--region us-west-2 \
--function-name helloworld \
--statement-id 3 \
--principal 111111111111 \
--action lambda:InvokeFunction \
--profile adminuser
```

In response, AWS Lambda returns the following JSON code. The `Statement` value is a JSON string version of the statement added to the Lambda function policy.

```
{
    "Statement": "{\"Action\":[\"lambda:InvokeFunction\"],
                   \"Resource\":\"arn:aws:lambda:us-west-2:account-
id:function:helloworld\",
                   \"Effect\":\"Allow\",
                   \"Principal\":{\"AWS\":\"account-id\"},
                   \"Sid\":\"3\"}"
}
```

Example 4: Retrieve a Lambda Function Policy

To retrieve your Lambda function policy, you use the `get-policy` command:

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

Example 5: Remove Permissions from a Lambda Function Policy

To remove permissions from your Lambda function policy, you use the `remove-permission` command, specifying the function name and statement ID:

```
aws lambda remove-permission \
--function-name example \
--statement-id 1 \
--profile adminuser
```

Example 6: Working with Lambda Function Versioning, Aliases, and Permissions

For more information about permissions policies for Lambda function versions and aliases, see Versioning, Aliases, and Resource Policies (p. 85).

Lambda API Permissions: Actions, Resources, and Conditions Reference

When you are setting up Access Control (p. 288) and writing permissions policies that you can attach to an IAM identity (identity-based policies), you can use the following table as a reference. The list includes each AWS Lambda API operation, the corresponding actions for which you can grant permissions to perform the action, and the AWS resource for which you can grant the permissions. You specify the actions in the policy's `Action` field, and you specify the resource value in the policy's `Resource` field.

> **Note**
> Permissions for the AWS Lambda `Invoke` API in the following table can also be granted by using resource-based policies. For more information, see Using Resource-Based Policies for AWS Lambda (Lambda Function Policies) (p. 306).

You can use AWS-wide condition keys in your AWS Lambda policies to express conditions. For a complete list of AWS-wide keys, see Available Keys for Conditions in the *IAM User Guide*.

> **Note**
> To specify an action, use the `lambda:` prefix followed by the API operation name (for example, `lambda:CreateFunction`).

AWS Lambda API and Required Permissions for Actions

AddPermission (p. 322)
: **Action(s):** `lambda:AddPermission`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?/*`

CreateEventSourceMapping (p. 328)
: **Action(s):** `lambda:CreateEventSourceMapping`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

CreateFunction (p. 332)
: **Action(s):** `lambda:CreateFunction`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

DeleteEventSourceMapping (p. 339)
: **Action(s):** `lambda:DeleteEventSourceMapping`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

DeleteFunction (p. 341)
: **Action(s):** `lambda:DeleteFunction`,

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

GetEventSourceMapping (p. 347)
: **Action(s):** `lambda:GetEventSourceMapping`

Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

GetFunction (p. 349)
: **Action(s):** `lambda:GetFunction`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

GetFunctionConfiguration (p. 352)
: **Action(s):** `lambda:DescribeMountTargetSecurityGroups,`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

GetPolicy (p. 356)
: **Action(s):** `lambda:DescribeMountTargets`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

Invoke (p. 358)
: **Action(s):** `lambda:DescribeTags`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

InvokeAsync (p. 362)
: **Action(s):** `lambda:ModifyMountTargetSecurityGroups`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

ListEventSourceMappings (p. 366)
: **Action(s):** `lambda:ListEventSourceMappings`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

ListFunctions (p. 368)
: **Action(s):** `lambda:ListFunctions`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

RemovePermission (p. 376)
: **Action(s):** `lambda:RemovePermission`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

UpdateEventSourceMapping (p. 380)
: **Action(s):** `lambda:UpdateEventSourceMapping`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

UpdateFunctionCode (p. 383)
: **Action(s):** `lambda:UpdateFunctionCode`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

UpdateFunctionConfiguration (p. 387)
: **Action(s):** `lambda:UpdateFunctionConfiguration`

  Resource: `arn:aws:lambda:`*`region`*`:`*`account-id`*`:?`

Policy Templates

When you create an AWS Lambda function in the console using one of the blueprints, Lambda allows you to create a role for your function from a list of Lambda policy templates.

By selecting one of these templates, your Lambda function automatically creates the role with the requisite permissions attached to that policy.

The following lists the permissions that are applied to each policy template in the **Policy templates** list. The policy templates are named after the blueprints to which they correspond. Lambda will automatically populate the placeholder items (such as *region* and *accountID*) with the appropriate information. For more information on creating a Lambda function using policy templates, see Step 2.1: Create a Hello World Lambda Function (p. 164).

The following templates are automatically applied depending upon the type of Lambda function you are creating:

Basic: 'Basic Lambda Permissions'

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":"logs:CreateLogGroup",
         "Resource":"arn:aws:logs:region:accountId:*"
      },
      {
         "Effect":"Allow",
         "Action":[
            "logs:CreateLogStream",
            "logs:PutLogEvents"
         ],
         "Resource":[
            "arn:aws:logs:region:accountId:log-group:[[logGroups]]:*"
         ]
      }
   ]
}
```

VPCAccess: 'Lambda VPC Access Permissions'

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
      ],
      "Resource": "*"
    }
  ]
}
```

Kinesis: 'Lambda Kinesis stream poller permissions'

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":"lambda:InvokeFunction",
         "Resource":"arn:aws:lambda:region:accountId:function:functionName*"
      },
      {
         "Effect":"Allow",
         "Action":"kinesis:ListStreams",
         "Resource":"arn:aws:kinesis:region:accountId:stream/*"
      },
      {
         "Effect":"Allow",
         "Action":[
            "kinesis:DescribeStream",
            "kinesis:GetRecords",
            "kinesis:GetShardIterator"
         ],
         "Resource":"arn:aws:kinesis:region:accountId:stream/streamName"
      }
   ]
}
```

DynamoDB: 'Lambda DynamoDB stream poller permissions'

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":"lambda:InvokeFunction",
         "Resource":"arn:aws:lambda:region:accountId:function:functionName*"
      },
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:DescribeStream",
            "dynamodb:GetRecords",
            "dynamodb:GetShardIterator",
            "dynamodb:ListStreams"
         ],
         "Resource":"arn:aws:dynamodb:region:accountId:table/tableName/
stream/*"
      }
   ]
}
```

Edge: 'Basic Edge Lambda permissions'

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

RedrivePolicySNS: ‘Dead letter queue SNS permissions’

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sns:Publish"
      ],
      "Resource": "arn:aws:sns:region:accountId:topicName"
    }
  ]
}
```

RedrivePolicySQS: 'Dead letter queue SQS permissions'

```
{
  "Version": "2012-10-17",
  "Statement": [
  {
    "Effect": "Allow",
    "Action": [
       "sqs:SendMessage"
     ],
    "Resource": "arn:aws:sqs:region:accountId:queueName"
  }
 ]
}
```

The following templates are selected depending upon which blueprint you choose. You can also select them from the dropdown to add extra permissions:

CloudFormation: 'CloudFormation stack read-only permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudformation:DescribeStacks"
            ],
            "Resource": "*"
        }
    ]
}
```

AMI: 'AMI read-only permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeImages"
            ],
            "Resource": "*"
        }
    ]
}
```

KMS: 'KMS decryption permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "*"
        }
    ]
}
```

S3: 'S3 object read-only permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*"
        }
    ]
}
```

Elasticsearch: 'Elasticsearch permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "es:ESHttpPost"
            ],
            "Resource": "*"
        }
    ]
}
```

SES: 'SES bounce permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ses:SendBounce"
            ],
            "Resource": "*"
        }
    ]
}
```

TestHarness: 'Test Harness permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:region:accountId:table/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:region:accountId:function:*"
        }
    ]
}
```

Microservice: 'Simple Microservice permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:region:accountId:table/*"
        }
    ]
}
```

VPN: 'VPN Connection Monitor permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeRegions",
                "ec2:DescribeVpnConnections"
            ],
            "Resource": "*"
        }
    ]
}
```

SQS: 'SQS Poller permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sqs:DeleteMessage",
                "sqs:ReceiveMessage"
            ],
            "Resource": "arn:aws:sqs:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource":
 "arn:aws:lambda:region:accountId:function:functionName*"
        }
    ]
}
```

IoTButton: 'AWS IoT Button permissions'

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sns:ListSubscriptionsByTopic",
                "sns:CreateTopic",
                "sns:SetTopicAttributes",
                "sns:Subscribe",
                "sns:Publish"
            ],
            "Resource": "*"
        }
    ]
}
```

RekognitionNoDataAccess:'Amazon Rekognition no data permissions'

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rekognition:CompareFaces",
        "rekognition:DetectFaces",
        "rekognition:DetectLabels"
      ],
      "Resource": "*"
    }
  ]
}
```

RekognitionReadOnlyAccess: 'Amazon Rekognition read-only permissions'

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rekognition:ListCollections",
        "rekognition:ListFaces",
        "rekognition:SearchFaces",
        "rekognition:SearchFacesByImage"
      ],
      "Resource": "*"
    }
  ]
}
```

RekognitionWriteOnlyAccess: 'Amazon Rekognition write-only permissions'

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rekognition:CreateCollection",
        "rekognition:IndexFaces"
      ],
      "Resource": "*"
    }
  ]
}
```

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, see 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 How It Works (p. 151).

You can use the AWS CLI to explore the AWS Lambda API. This guide provides several tutorials that use the AWS CLI.

Topics

Actions

The following actions are supported:

- AddPermission (p. 322)
- CreateAlias (p. 325)
- CreateEventSourceMapping (p. 328)
- CreateFunction (p. 332)
- DeleteAlias (p. 338)
- DeleteEventSourceMapping (p. 339)
- DeleteFunction (p. 341)
- GetAccountSettings (p. 343)
- GetAlias (p. 345)
- GetEventSourceMapping (p. 347)
- GetFunction (p. 349)
- GetFunctionConfiguration (p. 352)
- GetPolicy (p. 356)
- Invoke (p. 358)
- InvokeAsync (p. 362)

AddPermission

Adds a permission to the resource policy associated with the specified AWS Lambda function. You use resource policies to grant permissions to event sources that use *push* model. In a *push* model, event sources (such as Amazon S3 and custom applications) invoke your Lambda function. Each permission you add to the resource policy allows an event source, permission to invoke the Lambda function.

For information about the push model, see AWS Lambda: How it Works.

If you are using versioning, the permissions you add are specific to the Lambda function version or alias you specify in the `AddPermission` request via the `Qualifier` parameter. For more information about versioning, see AWS Lambda Function Versioning and Aliases.

This operation requires permission for the `lambda:AddPermission` action.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
   "Action": "string",
   "EventSourceToken": "string",
   "Principal": "string",
   "SourceAccount": "string",
   "SourceArn": "string",
   "StatementId": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 322)

Name of the Lambda function whose resource policy you are updating by adding a new permission.

You can specify a 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 partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Qualifier (p. 322)

You can use this optional query parameter to describe a qualified ARN using a function version or an alias name. The permission will then apply to the specific qualified ARN. For example, if you specify function version 2 as the qualifier, then permission applies only when request is made using qualified function ARN:

`arn:aws:lambda:aws-region:acct-id:function:function-name:2`

If you specify an alias name, for example `PROD`, then the permission is valid only for requests made using the alias ARN:

`arn:aws:lambda:aws-region:acct-id:function:function-name:PROD`

If the qualifier is not specified, the permission is valid only when requests is made using unqualified function ARN.

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

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[a-zA-Z0-9$_-]+)`

Request Body

The request accepts the following data in JSON format.

Action (p. 322)

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) . 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

EventSourceToken (p. 322)

A unique token that must be supplied by the principal invoking the function. This is currently only used for Alexa Smart Home functions.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Pattern: `[a-zA-Z0-9._\-]+`

Required: No

Principal (p. 322)

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 (p. 322)

This parameter is used for S3, SES, CloudWatch Logs and CloudWatch Rules only. 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 (p. 322)

This is optional; however, when granting Amazon S3 permission to invoke your function, you should specify this field with the Amazon Resource Name (ARN) as its value. This ensures that only events generated from the specified source 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 (p. 322)

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 (p. 324)

The permission statement you specified in the request. The response returns the same as a string using a backslash ("\") 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: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

CreateAlias

Creates an alias that points to the specified Lambda function version. For more information, see Introduction to AWS Lambda Aliases.

Alias names are unique for a given function. This requires permission for the lambda:CreateAlias action.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/aliases HTTP/1.1
Content-type: application/json

{
   "Description": "string",
   "FunctionVersion": "string",
   "Name": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 325)

Name of the Lambda function for which you want to create an alias.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Request Body

The request accepts the following data in JSON format.

Description (p. 325)

Description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion (p. 325)

Lambda function version for which you are creating the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Required: Yes

Name (p. 325)

Name for the alias you are creating.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([a-zA-Z0-9-_]+)`

Required: Yes

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
   "AliasArn": "string",
   "Description": "string",
   "FunctionVersion": "string",
   "Name": "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.

AliasArn (p. 326)

Lambda function ARN that is qualified using the alias name as the suffix. For example, if you create an alias called `BETA` that points to a helloworld function version, the ARN is `arn:aws:lambda:aws-regions:acct-id:function:helloworld:BETA`.

Type: String

Pattern: `arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Description (p. 326)

Alias description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

FunctionVersion (p. 326)

Function version to which the alias points.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Name (p. 326)

Alias name.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([a-zA-Z0-9-_]+)`

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: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

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 association between a stream source and a Lambda function is called the event source mapping.

> **Important**
> This event source mapping is relevant only in the AWS Lambda pull model, where AWS Lambda invokes the function. For more information, see AWS Lambda: How it Works in the *AWS Lambda Developer Guide*.

You provide mapping information (for example, which stream to read from and which Lambda function to invoke) 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.

If you are using versioning, you can specify a specific function version or an alias via the function name parameter. For more information about versioning, see AWS Lambda Function Versioning and Aliases.

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",
   "StartingPositionTimestamp": number
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

BatchSize (p. 328)

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: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

Enabled (p. 328)

Indicates whether AWS Lambda should begin polling the event source. By default, `Enabled` is true.

Type: Boolean

Required: No

EventSourceArn (p. 328)

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 (p. 328)

The Lambda function to invoke when AWS Lambda detects an event on the stream.

You can specify the 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`).

If you are using versioning, you can also provide a qualified function ARN (ARN that is qualified with function version or alias name as suffix). For more information about versioning, see AWS Lambda Function Versioning and Aliases

AWS Lambda also allows you to specify only the function name with 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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: Yes

StartingPosition (p. 328)

The position in the stream where AWS Lambda should start reading. Valid only for Kinesis streams. For more information, see ShardIteratorType in the *Amazon Kinesis API Reference*.

Type: String

Valid Values: `TRIM_HORIZON | LATEST | AT_TIMESTAMP`

Required: Yes

StartingPositionTimestamp (p. 328)

The timestamp of the data record from which to start reading. Used with shard iterator type AT_TIMESTAMP. If a record with this exact timestamp does not exist, the iterator returned is for the next (later) record. If the timestamp is older than the current trim horizon, the iterator returned is for the oldest untrimmed data record (TRIM_HORIZON). Valid only for Kinesis streams.

Type: Timestamp

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 (p. 329)
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: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

EventSourceArn (p. 329)
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 (p. 329)
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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

LastModified (p. 329)
The UTC time string indicating the last time the event mapping was updated.

Type: Timestamp

LastProcessingResult (p. 329)
The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State (p. 329)
The state of the event source mapping. It can be `Creating`, `Enabled`, `Disabled`, `Enabling`, `Disabling`, `Updating`, or `Deleting`.

Type: String

StateTransitionReason (p. 329)
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 (p. 329)
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: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

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.

If you are using versioning, you can also publish a version of the Lambda function you are creating using the `Publish` parameter. For more information about versioning, see AWS Lambda Function Versioning and Aliases.

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
   },
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Variables": {
         "string" : "string"
      }
   },
   "FunctionName": "string",
   "Handler": "string",
   "KMSKeyArn": "string",
   "MemorySize": number,
   "Publish": boolean,
   "Role": "string",
   "Runtime": "string",
   "Timeout": number,
   "VpcConfig": {
      "SecurityGroupIds": [ "string" ],
      "SubnetIds": [ "string" ]
   }
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

Code (p. 332)

The code for the Lambda function.

Type: FunctionCode (p. 401) object

Required: Yes

DeadLetterConfig (p. 332)

The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

Type: DeadLetterConfig (p. 396) object

Required: No

Description (p. 332)

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

Environment (p. 332)

The parent object that contains your environment's configuration settings.

Type: Environment (p. 397) object

Required: No

FunctionName (p. 332)

The name you want to assign to the function you are uploading. The function names appear in the console and are returned in the ListFunctions (p. 368) API. Function names are used to specify functions to other AWS Lambda API operations, such as Invoke (p. 358).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: Yes

Handler (p. 332)

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: Maximum length of 128.

Pattern: `[^\s]+`

Required: Yes

KMSKeyArn (p. 332)

The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If not provided, AWS Lambda will use a default service key.

Type: String

Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

Required: No

MemorySize (p. 332)

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: Integer

Valid Range: Minimum value of 128. Maximum value of 1536.

Required: No

Publish (p. 332)

This boolean parameter can be used to request AWS Lambda to create the Lambda function and publish a version as an atomic operation.

Type: Boolean

Required: No

Role (p. 332)

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 (p. 332)

The runtime environment for the Lambda function you are uploading.

To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

> **Note**
> You can no longer create functions using the v0.10.42 runtime version as of November, 2016. Existing functions will be supported until early 2017, but we recommend you migrate them to nodejs4.3 runtime version as soon as possible.

Type: String

Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

Required: Yes

Timeout (p. 332)

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: Integer

Valid Range: Minimum value of 1.

Required: No

VpcConfig (p. 332)

If your Lambda function accesses resources in a VPC, you provide this parameter identifying the list of security group IDs and subnet IDs. These must belong to the same VPC. You must provide at least one security group and one subnet ID.

Type: VpcConfig (p. 405) object

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "Variables": {
         "string" : "string"
      }
   },
   "FunctionArn": "string",
   "FunctionName": "string",
   "Handler": "string",
   "KMSKeyArn": "string",
   "LastModified": "string",
   "MemorySize": number,
   "Role": "string",
   "Runtime": "string",
   "Timeout": number,
   "Version": "string",
   "VpcConfig": {
      "SecurityGroupIds": [ "string" ],
      "SubnetIds": [ "string" ],
      "VpcId": "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.

CodeSha256 (p. 334)
It is the SHA256 hash of your function deployment package.
Type: String

CodeSize (p. 334)
The size, in bytes, of the function .zip file you uploaded.
Type: Long

DeadLetterConfig (p. 334)
The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.
Type: DeadLetterConfig (p. 396) object

Description (p. 334)
The user-provided description.
Type: String
Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 334)
The parent object that contains your environment's configuration settings.
Type: EnvironmentResponse (p. 399) object

FunctionArn (p. 334)
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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionName (p. 334)
The name of the function.
Type: String
Length Constraints: Minimum length of 1. Maximum length of 140.
Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Handler (p. 334)
: The function Lambda calls to begin executing your function.

  Type: String

  Length Constraints: Maximum length of 128.

  Pattern: `[^\s]+`

KMSKeyArn (p. 334)
: The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If empty, it means you are using the AWS Lambda default service key.

  Type: String

  Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

LastModified (p. 334)
: The time stamp of the last time you updated the function.

  Type: String

MemorySize (p. 334)
: The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

  Type: Integer

  Valid Range: Minimum value of 128. Maximum value of 1536.

Role (p. 334)
: 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 (p. 334)
: The runtime environment for the Lambda function.

  To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

  Type: String

  Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

Timeout (p. 334)
: 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: Integer

  Valid Range: Minimum value of 1.

Version (p. 334)
: The version of the Lambda function.

  Type: String

  Length Constraints: Minimum length of 1. Maximum length of 1024.

  Pattern: `(\$LATEST|[0-9]+)`

VpcConfig (p. 334)
: VPC configuration associated with your Lambda function.

  Type: VpcConfigResponse (p. 406) object

Errors

CodeStorageExceededException
: You have exceeded your maximum total code size per account. Limits

  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: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

DeleteAlias

Deletes the specified Lambda function alias. For more information, see Introduction to AWS Lambda Aliases.

This requires permission for the lambda:DeleteAlias action.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 338)

The Lambda function name for which the alias is created. Deleting an alias does not delete the function version to which it is pointing.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Name (p. 338)

Name of the alias to delete.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([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

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

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

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 (p. 339)
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 (p. 339)
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: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

EventSourceArn (p. 339)
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 (p. 339)
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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

LastModified (p. 339)
: The UTC time string indicating the last time the event mapping was updated.

  Type: Timestamp

LastProcessingResult (p. 339)
: The result of the last AWS Lambda invocation of your Lambda function.

  Type: String

State (p. 339)
: The state of the event source mapping. It can be `Creating`, `Enabled`, `Disabled`, `Enabling`, `Disabling`, `Updating`, or `Deleting`.

  Type: String

StateTransitionReason (p. 339)
: 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 (p. 339)
: 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: 404

ServiceException
: The AWS Lambda service encountered an internal error.

  HTTP Status Code: 500

TooManyRequestsException
: HTTP Status Code: 429

DeleteFunction

Deletes the specified Lambda function code and configuration.

If you are using the versioning feature and you don't specify a function version in your `DeleteFunction` request, AWS Lambda will delete the function, including all its versions, and any aliases pointing to the function versions. To delete a specific function version, you must provide the function version via the `Qualifier` parameter. For information about function versioning, see AWS Lambda Function Versioning and Aliases.

When you delete a function the associated resource 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?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 341)

The Lambda function to delete.

You can specify the 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`). If you are using versioning, you can also provide a qualified function ARN (ARN that is qualified with function version or alias name as suffix). AWS Lambda also allows you to specify only the function name with 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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Qualifier (p. 341)

Using this optional parameter you can specify a function version (but not the `$LATEST` version) to direct AWS Lambda to delete a specific function version. If the function version has one or more aliases pointing to it, you will get an error because you cannot have aliases pointing to it. You can delete any function version but not the `$LATEST`, that is, you cannot specify `$LATEST` as the value of this parameter. The `$LATEST` version can be deleted only when you want to delete all the function versions and aliases.

You can only specify a function version, not an alias name, using this parameter. You cannot delete a function version using its alias.

If you don't specify this parameter, AWS Lambda will delete the function, including all of its versions and aliases.

Length Constraints: Minimum length of 1. Maximum length of 128.

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

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: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

GetAccountSettings

Returns a customer's account settings.

You can use this operation to retrieve Lambda limits information, such as code size and concurrency limits. For more information about limits, see AWS Lambda Limits. You can also retrieve resource usage statistics, such as code storage usage and function count.

Request Syntax

```
GET /2016-08-19/account-settings/ HTTP/1.1
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
   "AccountLimit": {
      "CodeSizeUnzipped": number,
      "CodeSizeZipped": number,
      "ConcurrentExecutions": number,
      "TotalCodeSize": number
   },
   "AccountUsage": {
      "FunctionCount": number,
      "TotalCodeSize": 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.

AccountLimit (p. 343)
Provides limits of code size and concurrency associated with the current account and region.
Type: AccountLimit (p. 393) object

AccountUsage (p. 343)
Provides code size usage and function count associated with the current account and region.
Type: AccountUsage (p. 394) object

Errors

ServiceException
The AWS Lambda service encountered an internal error.
HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

GetAlias

Returns the specified alias information such as the alias ARN, description, and function version it is pointing to. For more information, see Introduction to AWS Lambda Aliases.

This requires permission for the `lambda:GetAlias` action.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 345)

Function name for which the alias is created. An alias is a subresource that exists only in the context of an existing Lambda function so you must specify the function name.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Name (p. 345)

Name of the alias for which you want to retrieve information.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([a-zA-Z0-9-_]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
   "AliasArn": "string",
   "Description": "string",
   "FunctionVersion": "string",
   "Name": "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.

AliasArn (p. 345)

Lambda function ARN that is qualified using the alias name as the suffix. For example, if you create an alias called `BETA` that points to a helloworld function version, the ARN is `arn:aws:lambda:aws-regions:acct-id:function:helloworld:BETA`.

Type: String

Pattern: `arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Description (p. 345)
Alias description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

FunctionVersion (p. 345)
Function version to which the alias points.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Name (p. 345)
Alias name.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([a-zA-Z0-9-_]+)`

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: 404

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 429

GetEventSourceMapping

Returns configuration information for the specified event source mapping (see CreateEventSourceMapping (p. 328)).

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 (p. 347)
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 (p. 347)
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: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

EventSourceArn (p. 347)
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 (p. 347)
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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

LastModified (p. 347)
The UTC time string indicating the last time the event mapping was updated.

Type: Timestamp

LastProcessingResult (p. 347)
The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State (p. 347)
The state of the event source mapping. It can be `Creating`, `Enabled`, `Disabled`, `Enabling`, `Disabling`, `Updating`, or `Deleting`.

Type: String

StateTransitionReason (p. 347)
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 (p. 347)
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: 404

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 429

GetFunction

Returns the configuration information of the Lambda function and a presigned URL link to the .zip file you uploaded with CreateFunction (p. 332) 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.

Using the optional `Qualifier` parameter, you can specify a specific function version for which you want this information. If you don't specify this parameter, the API uses unqualified function ARN which return information about the `$LATEST` version of the Lambda function. For more information, see AWS Lambda Function Versioning and Aliases.

This operation requires permission for the `lambda:GetFunction` action.

Request Syntax

```
GET /2015-03-31/functions/FunctionName?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 349)

The Lambda function name.

You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Qualifier (p. 349)

Using this optional parameter to specify a function version or an alias name. If you specify function version, the API uses qualified function ARN for the request and returns information about the specific Lambda function version. If you specify an alias name, the API uses the alias ARN and returns information about the function version to which the alias points. If you don't provide this parameter, the API uses unqualified function ARN and returns information about the `$LATEST` version of the Lambda function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[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": {
      "CodeSha256": "string",
      "CodeSize": number,
      "DeadLetterConfig": {
         "TargetArn": "string"
      },
      "Description": "string",
      "Environment": {
         "Error": {
            "ErrorCode": "string",
            "Message": "string"
         },
         "Variables": {
            "string" : "string"
         }
      },
      "FunctionArn": "string",
      "FunctionName": "string",
      "Handler": "string",
      "KMSKeyArn": "string",
      "LastModified": "string",
      "MemorySize": number,
      "Role": "string",
      "Runtime": "string",
      "Timeout": number,
      "Version": "string",
      "VpcConfig": {
         "SecurityGroupIds": [ "string" ],
         "SubnetIds": [ "string" ],
         "VpcId": "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.

Code (p. 349)
: The object for the Lambda function location.

  Type: FunctionCodeLocation (p. 402) object

Configuration (p. 349)
: A complex type that describes function metadata.

  Type: FunctionConfiguration (p. 403) object

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

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. 332).

If you are using the versioning feature, you can retrieve this information for a specific function version by using the optional `Qualifier` parameter and specifying the function version or alias that points to it. If you don't provide it, the API returns information about the $LATEST version of the function. For more information about versioning, see AWS Lambda Function Versioning and Aliases.

This operation requires permission for the `lambda:GetFunctionConfiguration` operation.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/configuration?Qualifier=Qualifier
 HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 352)

The name of the Lambda function for which you want to retrieve the configuration information.

You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Qualifier (p. 352)

Using this optional parameter you can specify a function version or an alias name. If you specify function version, the API uses qualified function ARN and returns information about the specific function version. If you specify an alias name, the API uses the alias ARN and returns information about the function version to which the alias points.

If you don't specify this parameter, the API uses unqualified function ARN, and returns information about the `$LATEST` function version.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[a-zA-Z0-9$_-]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "Variables": {
         "string" : "string"
      }
   },
   "FunctionArn": "string",
   "FunctionName": "string",
   "Handler": "string",
   "KMSKeyArn": "string",
   "LastModified": "string",
   "MemorySize": number,
   "Role": "string",
   "Runtime": "string",
   "Timeout": number,
   "Version": "string",
   "VpcConfig": {
      "SecurityGroupIds": [ "string" ],
      "SubnetIds": [ "string" ],
      "VpcId": "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.

CodeSha256 (p. 352)
It is the SHA256 hash of your function deployment package.
Type: String

CodeSize (p. 352)
The size, in bytes, of the function .zip file you uploaded.
Type: Long

DeadLetterConfig (p. 352)
The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.
Type: DeadLetterConfig (p. 396) object

Description (p. 352)
The user-provided description.
Type: String
Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 352)
The parent object that contains your environment's configuration settings.
Type: EnvironmentResponse (p. 399) object

FunctionArn (p. 352)
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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionName (p. 352)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Handler (p. 352)

The function Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

KMSKeyArn (p. 352)

The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If empty, it means you are using the AWS Lambda default service key.

Type: String

Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

LastModified (p. 352)

The time stamp of the last time you updated the function.

Type: String

MemorySize (p. 352)

The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 1536.

Role (p. 352)

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 (p. 352)

The runtime environment for the Lambda function.

To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

Type: String

Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

Timeout (p. 352)

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: Integer

Valid Range: Minimum value of 1.

Version (p. 352)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

VpcConfig (p. 352)

VPC configuration associated with your Lambda function.

Type: VpcConfigResponse (p. 406) object

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

GetPolicy

Returns the resource policy associated with the specified Lambda function.

If you are using the versioning feature, you can get the resource policy associated with the specific Lambda function version or alias by specifying the version or alias name using the `Qualifier` parameter. For more information about versioning, see AWS Lambda Function Versioning and Aliases.

For information about adding permissions, see AddPermission (p. 322).

You need permission for the `lambda:GetPolicy action.`

Request Syntax

```
GET /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 356)

Function name whose resource policy you want to retrieve.

You can specify the 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`). If you are using versioning, you can also provide a qualified function ARN (ARN that is qualified with function version or alias name as suffix). AWS Lambda also allows you to specify only the function name with 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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Qualifier (p. 356)

You can specify this optional query parameter to specify a function version or an alias name in which case this API will return all permissions associated with the specific qualified ARN. If you don't provide this parameter, the API will return permissions that apply to the unqualified function ARN.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[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 (p. 356)
The resource policy associated with the specified function. The response returns the same as a string using a backslash ("\") 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

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 429

Invoke

Invokes a specific Lambda function. For an example, see Create the Lambda Function and Test It Manually.

If you are using the versioning feature, you can invoke the specific function version by providing function version or alias name that is pointing to the function version using the `Qualifier` parameter in the request. If you don't provide the `Qualifier` parameter, the `$LATEST` version of the Lambda function is invoked. Invocations occur at least once in response to an event and functions must be idempotent to handle this. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

This operation requires permission for the `lambda:InvokeFunction` action.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/invocations?Qualifier=Qualifier
 HTTP/1.1
X-Amz-Invocation-Type: InvocationType
X-Amz-Log-Type: LogType
X-Amz-Client-Context: ClientContext

Payload
```

URI Request Parameters

The request requires the following URI parameters.

ClientContext (p. 358)

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, see PutEvents in the *Amazon Mobile Analytics API Reference and User Guide*.

The ClientContext JSON must be base64-encoded.

FunctionName (p. 358)

The Lambda function name.

You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

InvocationType (p. 358)

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 (p. 358)

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-result` header.

Valid Values: `None | Tail`

Qualifier (p. 358)

You can use this optional parameter to specify a Lambda function version or alias name. If you specify a function version, the API uses the qualified function ARN to invoke a specific Lambda function. If you specify an alias name, the API uses the alias ARN to invoke the Lambda function version to which the alias points.

If you don't provide this parameter, then the API uses unqualified function ARN which results in invocation of the `$LATEST` version.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[a-zA-Z0-9$_-]+)`

Request Body

The request accepts the following data in JSON format.

Payload (p. 358)

JSON that you want to provide to your Lambda function as input.

Type: Binary data object

Required: No

Response Syntax

```
HTTP/1.1 StatusCode
X-Amz-Function-Error: FunctionError
X-Amz-Log-Result: LogResult

Payload
```

Response Elements

If the action is successful, the service sends back the following HTTP response.

StatusCode (p. 359)

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 (p. 359)

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 (p. 359)

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.

<varlistentry> **Payload (p. 359)**

It is the JSON representation of the object returned by the Lambda function. 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.

</varlistentry>

Errors

EC2AccessDeniedException

HTTP Status Code: 502

EC2ThrottledException

AWS Lambda was throttled by Amazon EC2 during Lambda function initialization using the execution role provided for the Lambda function.

HTTP Status Code: 502

EC2UnexpectedException

AWS Lambda received an unexpected EC2 client exception while setting up for the Lambda function.

HTTP Status Code: 502

ENILimitReachedException

AWS Lambda was not able to create an Elastic Network Interface (ENI) in the VPC, specified as part of Lambda function configuration, because the limit for network interfaces has been reached.

HTTP Status Code: 502

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

InvalidRequestContentException

The request body could not be parsed as JSON.

HTTP Status Code: 400

InvalidSecurityGroupIDException

The Security Group ID provided in the Lambda function VPC configuration is invalid.

HTTP Status Code: 502

InvalidSubnetIDException

The Subnet ID provided in the Lambda function VPC configuration is invalid.

HTTP Status Code: 502

InvalidZipFileException

AWS Lambda could not unzip the function zip file.

HTTP Status Code: 502

KMSAccessDeniedException

Lambda was unable to decrypt the environment variables because KMS access was denied. Check the Lambda function's KMS permissions.

HTTP Status Code: 502

KMSDisabledException

Lambda was unable to decrypt the environment variables because the KMS key used is disabled. Check the Lambda function's KMS key settings.

HTTP Status Code: 502

KMSInvalidStateException

Lambda was unable to decrypt the environment variables because the KMS key used is in an invalid state for Decrypt. Check the function's KMS key settings.

HTTP Status Code: 502

KMSNotFoundException

Lambda was unable to decrypt the environment variables because the KMS key was not found. Check the function's KMS key settings.

HTTP Status Code: 502

RequestTooLargeException

The request payload exceeded the `Invoke` request body JSON input limit. For more information, see Limits.

HTTP Status Code: 413

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

SubnetIPAddressLimitReachedException

AWS Lambda was not able to set up VPC access for the Lambda function because one or more configured subnets has no available IP addresses.

HTTP Status Code: 502

TooManyRequestsException

HTTP Status Code: 429

UnsupportedMediaTypeException

The content type of the `Invoke` request body is not JSON.

HTTP Status Code: 415

InvokeAsync

Important

This API is deprecated. We recommend you use `Invoke` API (see Invoke (p. 358)).

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 (p. 362)

The Lambda function name.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Request Body

The request accepts the following data in JSON format.

InvokeArgs (p. 362)

JSON that you want to provide to your Lambda function as input.

Type: Binary data object

Required: Yes

Response Syntax

```
HTTP/1.1 Status
```

Response Elements

If the action is successful, the service sends back the following HTTP response.

Status (p. 362)

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

Example

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
```

ListAliases

Returns list of aliases created for a Lambda function. For each alias, the response includes information such as the alias ARN, description, alias name, and the function version to which it points. For more information, see Introduction to AWS Lambda Aliases.

This requires permission for the lambda:ListAliases action.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/aliases?
FunctionVersion=FunctionVersion&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 364)
Lambda function name for which the alias is created.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionVersion (p. 364)
If you specify this optional parameter, the API returns only the aliases that are pointing to the specific Lambda function version, otherwise the API returns all of the aliases created for the Lambda function.

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Marker (p. 364)
Optional string. An opaque pagination token returned from a previous `ListAliases` operation. If present, indicates where to continue the listing.

MaxItems (p. 364)
Optional integer. Specifies the maximum number of aliases 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

{
   "Aliases": [
      {
         "AliasArn": "string",
         "Description": "string",
         "FunctionVersion": "string",
         "Name": "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.

Aliases (p. 364)
A list of aliases.

Type: array of AliasConfiguration (p. 395) objects

NextMarker (p. 364)
A string, present if there are more aliases.

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: 404

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 429

ListEventSourceMappings

Returns a list of event source mappings you created using the `CreateEventSourceMapping` (see CreateEventSourceMapping (p. 328)).

For each mapping, the API returns configuration information. You can optionally specify filters to retrieve specific event source mappings.

If you are using the versioning feature, you can get list of event source mappings for a specific Lambda function version or an alias as described in the `FunctionName` parameter. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

This operation requires permission for the `lambda:ListEventSourceMappings` action.

Request Syntax

```
GET /2015-03-31/event-source-mappings/?
EventSourceArn=EventSourceArn&FunctionName=FunctionName&Marker=Marker&MaxItems=MaxItems
 HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

EventSourceArn (p. 366)

The Amazon Resource Name (ARN) of the Amazon Kinesis stream. (This parameter is optional.)

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

FunctionName (p. 366)

The name of the Lambda function.

You can specify the 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`). If you are using versioning, you can also provide a qualified function ARN (ARN that is qualified with function version or alias name as suffix). AWS Lambda also allows you to specify only the function name with 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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Marker (p. 366)

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 (p. 366)

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 (p. 366)
An array of `EventSourceMappingConfiguration` objects.

Type: array of EventSourceMappingConfiguration (p. 400) objects

NextMarker (p. 366)
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: 404

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 429

ListFunctions

Returns a list of your Lambda functions. For each function, the response includes the function configuration information. You must use GetFunction (p. 349) to retrieve the code for your function.

This operation requires permission for the `lambda:ListFunctions` action.

If you are using versioning feature, the response returns list of $LATEST versions of your functions. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

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 (p. 368)
Optional string. An opaque pagination token returned from a previous `ListFunctions` operation. If present, indicates where to continue the listing.

MaxItems (p. 368)
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": [
      {
         "CodeSha256": "string",
         "CodeSize": number,
         "DeadLetterConfig": {
            "TargetArn": "string"
         },
         "Description": "string",
         "Environment": {
            "Error": {
               "ErrorCode": "string",
               "Message": "string"
            },
            "Variables": {
               "string" : "string"
            }
         },
         "FunctionArn": "string",
         "FunctionName": "string",
         "Handler": "string",
         "KMSKeyArn": "string",
         "LastModified": "string",
         "MemorySize": number,
         "Role": "string",
         "Runtime": "string",
         "Timeout": number,
         "Version": "string",
         "VpcConfig": {
            "SecurityGroupIds": [ "string" ],
            "SubnetIds": [ "string" ],
            "VpcId": "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.

Functions (p. 368)
A list of Lambda functions.
Type: array of FunctionConfiguration (p. 403) objects

NextMarker (p. 368)
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: 429

ListVersionsByFunction

List all versions of a function. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/versions?
Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 370)

Function name whose versions to list. You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Marker (p. 370)

Optional string. An opaque pagination token returned from a previous `ListVersionsByFunction` operation. If present, indicates where to continue the listing.

MaxItems (p. 370)

Optional integer. Specifies the maximum number of AWS Lambda function versions 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

{
   "NextMarker": "string",
   "Versions": [
      {
         "CodeSha256": "string",
         "CodeSize": number,
         "DeadLetterConfig": {
            "TargetArn": "string"
         },
         "Description": "string",
         "Environment": {
            "Error": {
               "ErrorCode": "string",
               "Message": "string"
            },
            "Variables": {
               "string" : "string"
            }
         },
         "FunctionArn": "string",
         "FunctionName": "string",
         "Handler": "string",
         "KMSKeyArn": "string",
         "LastModified": "string",
         "MemorySize": number,
         "Role": "string",
         "Runtime": "string",
         "Timeout": number,
         "Version": "string",
         "VpcConfig": {
            "SecurityGroupIds": [ "string" ],
            "SubnetIds": [ "string" ],
            "VpcId": "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.

NextMarker (p. 370)
A string, present if there are more function versions.

Type: String

Versions (p. 370)
A list of Lambda function versions.

Type: array of FunctionConfiguration (p. 403) objects

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: 404

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 429

PublishVersion

Publishes a version of your function from the current snapshot of $LATEST. That is, AWS Lambda takes a snapshot of the function code and configuration information from $LATEST and publishes a new version. The code and configuration cannot be modified after publication. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/versions HTTP/1.1
Content-type: application/json

{
   "CodeSha256": "string",
   "Description": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 372)

The Lambda function name. You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Request Body

The request accepts the following data in JSON format.

CodeSha256 (p. 372)

The SHA256 hash of the deployment package you want to publish. This provides validation on the code you are publishing. If you provide this parameter value must match the SHA256 of the $LATEST version for the publication to succeed.

Type: String

Required: No

Description (p. 372)

The description for the version you are publishing. If not provided, AWS Lambda copies the description from the $LATEST version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json
```

```
{
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "Variables": {
         "string" : "string"
      }
   },
   "FunctionArn": "string",
   "FunctionName": "string",
   "Handler": "string",
   "KMSKeyArn": "string",
   "LastModified": "string",
   "MemorySize": number,
   "Role": "string",
   "Runtime": "string",
   "Timeout": number,
   "Version": "string",
   "VpcConfig": {
      "SecurityGroupIds": [ "string" ],
      "SubnetIds": [ "string" ],
      "VpcId": "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.

CodeSha256 (p. 372)
: It is the SHA256 hash of your function deployment package.

  Type: String

CodeSize (p. 372)
: The size, in bytes, of the function .zip file you uploaded.

  Type: Long

DeadLetterConfig (p. 372)
: The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

  Type: DeadLetterConfig (p. 396) object

Description (p. 372)
: The user-provided description.

  Type: String

  Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 372)
: The parent object that contains your environment's configuration settings.

  Type: EnvironmentResponse (p. 399) object

FunctionArn (p. 372)

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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionName (p. 372)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Handler (p. 372)

The function Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

KMSKeyArn (p. 372)

The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If empty, it means you are using the AWS Lambda default service key.

Type: String

Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

LastModified (p. 372)

The time stamp of the last time you updated the function.

Type: String

MemorySize (p. 372)

The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 1536.

Role (p. 372)

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 (p. 372)

The runtime environment for the Lambda function.

To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

Type: String

Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

Timeout (p. 372)

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: Integer

Valid Range: Minimum value of 1.

Version (p. 372)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

VpcConfig (p. 372)

VPC configuration associated with your Lambda function.

Type: VpcConfigResponse (p. 406) object

Errors

CodeStorageExceededException

You have exceeded your maximum total code size per account. Limits

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

RemovePermission

You can remove individual permissions from an resource policy associated with a Lambda function by providing a statement ID that you provided when you added the permission.

If you are using versioning, the permissions you remove are specific to the Lambda function version or alias you specify in the `AddPermission` request via the `Qualifier` parameter. For more information about versioning, see AWS Lambda Function Versioning and Aliases.

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/policy/StatementId?
Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 376)

Lambda function whose resource policy you want to remove a permission from.

You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Qualifier (p. 376)

You can specify this optional parameter to remove permission associated with a specific function version or function alias. If you don't specify this parameter, the API removes permission associated with the unqualified function ARN.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[a-zA-Z0-9$_-]+)`

StatementId (p. 376)

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

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

UpdateAlias

Using this API you can update the function version to which the alias points and the alias description. For more information, see Introduction to AWS Lambda Aliases.

This requires permission for the lambda:UpdateAlias action.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
Content-type: application/json

{
   "Description": "string",
   "FunctionVersion": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 378)

The function name for which the alias is created.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Name (p. 378)

The alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([a-zA-Z0-9-_]+)`

Request Body

The request accepts the following data in JSON format.

Description (p. 378)

You can change the description of the alias using this parameter.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion (p. 378)

Using this parameter you can change the Lambda function version to which the alias points.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
   "AliasArn": "string",
   "Description": "string",
   "FunctionVersion": "string",
   "Name": "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.

AliasArn (p. 378)

Lambda function ARN that is qualified using the alias name as the suffix. For example, if you create an alias called `BETA` that points to a helloworld function version, the ARN is `arn:aws:lambda:aws-regions:acct-id:function:helloworld:BETA`.

Type: String

Pattern: `arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Description (p. 378)

Alias description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

FunctionVersion (p. 378)

Function version to which the alias points.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Name (p. 378)

Alias name.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([a-zA-Z0-9-_]+)`

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

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.

If you are using the versioning feature, you can update the event source mapping to map to a specific Lambda function version or alias as described in the `FunctionName` parameter. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

If you disable the event source mapping, AWS Lambda stops polling. If you enable again, it will resume polling from the time it had stopped polling, so you don't lose processing of any records. However, if you delete event source mapping and create it again, it will reset.

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 (p. 380)
The event source mapping identifier.

Request Body

The request accepts the following data in JSON format.

BatchSize (p. 380)
The maximum number of stream records that can be sent to your Lambda function for a single invocation.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

Enabled (p. 380)
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 (p. 380)
The Lambda function to which you want the stream records sent.

You can specify a 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 a partial ARN (for example, `account-id:Thumbnail`).

If you are using versioning, you can also provide a qualified function ARN (ARN that is qualified with function version or alias name as suffix). For more information about versioning, see AWS Lambda Function Versioning and Aliases

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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[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 (p. 381)

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: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

EventSourceArn (p. 381)

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 (p. 381)

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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

LastModified (p. 381)

The UTC time string indicating the last time the event mapping was updated.

Type: Timestamp

LastProcessingResult (p. 381)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State (p. 381)

The state of the event source mapping. It can be `Creating`, `Enabled`, `Disabled`, `Enabling`, `Disabling`, `Updating`, or `Deleting`.

Type: String

StateTransitionReason (p. 381)
: 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 (p. 381)
: 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: 409

ResourceNotFoundException
: The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

  HTTP Status Code: 404

ServiceException
: The AWS Lambda service encountered an internal error.

  HTTP Status Code: 500

TooManyRequestsException
: HTTP Status Code: 429

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.

If you are using the versioning feature, note this API will always update the $LATEST version of your Lambda function. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

This operation requires permission for the `lambda:UpdateFunctionCode` action.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/code HTTP/1.1
Content-type: application/json

{
   "Publish": boolean,
   "S3Bucket": "string",
   "S3Key": "string",
   "S3ObjectVersion": "string",
   "ZipFile": blob
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 383)

The existing Lambda function name whose code you want to replace.

You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Request Body

The request accepts the following data in JSON format.

Publish (p. 383)

This boolean parameter can be used to request AWS Lambda to update the Lambda function and publish a version as an atomic operation.

Type: Boolean

Required: No

S3Bucket (p. 383)

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 (p. 383)

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 (p. 383)

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 (p. 383)

The contents of your zip file containing your deployment package. If you are using the web API directly, the contents of the zip file must be base64-encoded. If you are using the AWS SDKs or the AWS CLI, the SDKs or CLI will do the encoding for you. For more information about creating a .zip file, see Execution Permissions in the *AWS Lambda Developer Guide*.

Type: Base64-encoded binary data object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "Variables": {
         "string" : "string"
      }
   },
   "FunctionArn": "string",
   "FunctionName": "string",
   "Handler": "string",
   "KMSKeyArn": "string",
   "LastModified": "string",
   "MemorySize": number,
   "Role": "string",
   "Runtime": "string",
   "Timeout": number,
   "Version": "string",
   "VpcConfig": {
      "SecurityGroupIds": [ "string" ],
      "SubnetIds": [ "string" ],
      "VpcId": "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.

CodeSha256 (p. 384)
: It is the SHA256 hash of your function deployment package.

  Type: String

CodeSize (p. 384)
: The size, in bytes, of the function .zip file you uploaded.

  Type: Long

DeadLetterConfig (p. 384)
: The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

  Type: DeadLetterConfig (p. 396) object

Description (p. 384)
: The user-provided description.

  Type: String

  Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 384)
: The parent object that contains your environment's configuration settings.

  Type: EnvironmentResponse (p. 399) object

FunctionArn (p. 384)
: 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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionName (p. 384)
: The name of the function.

  Type: String

  Length Constraints: Minimum length of 1. Maximum length of 140.

  Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Handler (p. 384)
: The function Lambda calls to begin executing your function.

  Type: String

  Length Constraints: Maximum length of 128.

  Pattern: `[^\s]+`

KMSKeyArn (p. 384)
: The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If empty, it means you are using the AWS Lambda default service key.

  Type: String

  Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

LastModified (p. 384)
: The time stamp of the last time you updated the function.

  Type: String

MemorySize (p. 384)
: The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 1536.

Role (p. 384)

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 (p. 384)

The runtime environment for the Lambda function.

To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

Type: String

Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

Timeout (p. 384)

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: Integer

Valid Range: Minimum value of 1.

Version (p. 384)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

VpcConfig (p. 384)

VPC configuration associated with your Lambda function.

Type: VpcConfigResponse (p. 406) object

Errors

CodeStorageExceededException

You have exceeded your maximum total code size per account. Limits

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

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.

If you are using the versioning feature, note this API will always update the $LATEST version of your Lambda function. For information about the versioning feature, see AWS Lambda Function Versioning and Aliases.

This operation requires permission for the `lambda:UpdateFunctionConfiguration` action.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/configuration HTTP/1.1
Content-type: application/json

{
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Variables": {
         "string" : "string"
      }
   },
   "Handler": "string",
   "KMSKeyArn": "string",
   "MemorySize": number,
   "Role": "string",
   "Runtime": "string",
   "Timeout": number,
   "VpcConfig": {
      "SecurityGroupIds": [ "string" ],
      "SubnetIds": [ "string" ]
   }
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 387)

The name of the Lambda function.

You can specify a 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 a partial ARN (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 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Request Body

The request accepts the following data in JSON format.

DeadLetterConfig (p. 387)

The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

Type: DeadLetterConfig (p. 396) object

Required: No

Description (p. 387)

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

Environment (p. 387)

The parent object that contains your environment's configuration settings.

Type: Environment (p. 397) object

Required: No

Handler (p. 387)

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: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

KMSKeyArn (p. 387)

The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If you elect to use the AWS Lambda default service key, pass in an empty string ("") for this parameter.

Type: String

Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

Required: No

MemorySize (p. 387)

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: Integer

Valid Range: Minimum value of 128. Maximum value of 1536.

Required: No

Role (p. 387)

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

Runtime (p. 387)

The runtime environment for the Lambda function.

To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

> **Note**
> You can no longer downgrade to the v0.10.42 runtime version. This version will no longer be supported as of early 2017.

Type: String

Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

Required: No

Timeout (p. 387)

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: Integer

Valid Range: Minimum value of 1.

Required: No

VpcConfig (p. 387)

If your Lambda function accesses resources in a VPC, you provide this parameter identifying the list of security group IDs and subnet IDs. These must belong to the same VPC. You must provide at least one security group and one subnet ID.

Type: VpcConfig (p. 405) object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "Variables": {
         "string" : "string"
      }
   },
   "FunctionArn": "string",
   "FunctionName": "string",
   "Handler": "string",
   "KMSKeyArn": "string",
   "LastModified": "string",
   "MemorySize": number,
   "Role": "string",
   "Runtime": "string",
   "Timeout": number,
   "Version": "string",
   "VpcConfig": {
      "SecurityGroupIds": [ "string" ],
      "SubnetIds": [ "string" ],
      "VpcId": "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.

CodeSha256 (p. 389)
: It is the SHA256 hash of your function deployment package.

  Type: String

CodeSize (p. 389)
: The size, in bytes, of the function .zip file you uploaded.

  Type: Long

DeadLetterConfig (p. 389)
: The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

  Type: DeadLetterConfig (p. 396) object

Description (p. 389)
: The user-provided description.

  Type: String

  Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 389)
: The parent object that contains your environment's configuration settings.

  Type: EnvironmentResponse (p. 399) object

FunctionArn (p. 389)
: 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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionName (p. 389)
: The name of the function.

  Type: String

  Length Constraints: Minimum length of 1. Maximum length of 140.

  Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Handler (p. 389)
: The function Lambda calls to begin executing your function.

  Type: String

  Length Constraints: Maximum length of 128.

  Pattern: `[^\s]+`

KMSKeyArn (p. 389)
: The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If empty, it means you are using the AWS Lambda default service key.

  Type: String

  Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

LastModified (p. 389)
: The time stamp of the last time you updated the function.

  Type: String

MemorySize (p. 389)
: The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 1536.

Role (p. 389)

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 (p. 389)

The runtime environment for the Lambda function.

To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

Type: String

Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

Timeout (p. 389)

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: Integer

Valid Range: Minimum value of 1.

Version (p. 389)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

VpcConfig (p. 389)

VPC configuration associated with your Lambda function.

Type: VpcConfigResponse (p. 406) object

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: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 429

Data Types

The following data types are supported:

AccountLimit

Provides limits of code size and concurrency associated with the current account and region.

Contents

CodeSizeUnzipped

Size, in bytes, of code/dependencies that you can zip into a deployment package (uncompressed zip/jar size) for uploading. The default limit is 250 MB.

Type: Long

Required: No

CodeSizeZipped

Size, in bytes, of a single zipped code/dependencies package you can upload for your Lambda function(.zip/.jar file). Try using Amazon S3 for uploading larger files. Default limit is 50 MB.

Type: Long

Required: No

ConcurrentExecutions

Number of simultaneous executions of your function per region. For more information or to request a limit increase for concurrent executions, see Lambda Function Concurrent Executions. The default limit is 100.

Type: Integer

Required: No

TotalCodeSize

Maximum size, in megabytes, of a code package you can upload per region. The default size is 75 GB.

Type: Long

Required: No

AccountUsage

Provides code size usage and function count associated with the current account and region.

Contents

FunctionCount

The number of your account's existing functions per region.

Type: Long

Required: No

TotalCodeSize

Total size, in bytes, of the account's deployment packages per region.

Type: Long

Required: No

AliasConfiguration

Provides configuration information about a Lambda function version alias.

Contents

AliasArn

Lambda function ARN that is qualified using the alias name as the suffix. For example, if you create an alias called `BETA` that points to a helloworld function version, the ARN is `arn:aws:lambda:aws-regions:acct-id:function:helloworld:BETA`.

Type: String

Pattern: `arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

Description

Alias description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion

Function version to which the alias points.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Required: No

Name

Alias name.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[0-9]+$)([a-zA-Z0-9-_]+)`

Required: No

DeadLetterConfig

The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

Contents

TargetArn

The Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic you specify as your Dead Letter Queue (DLQ).

Type: String

Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

Required: No

Environment

The parent object that contains your environment's configuration settings.

Contents

Variables

The key-value pairs that represent your environment's configuration settings. The value you specify cannot contain a ",".

Type: String to String map

Pattern: `[^,]*`

Required: No

EnvironmentError

The parent object that contains error information associated with your configuration settings.

Contents

ErrorCode

The error code returned by the environment error object.

Type: String

Required: No

Message

The message returned by the environment error object.

Type: String

Required: No

EnvironmentResponse

The parent object returned that contains your environment's configuration settings or any error information associated with your configuration settings.

Contents

Error

The parent object that contains error information associated with your configuration settings.

Type: EnvironmentError (p. 398) object

Required: No

Variables

The key-value pairs returned that represent your environment's configuration settings or error information.

Type: String to String map

Pattern: `[^,]*`

Required: No

EventSourceMappingConfiguration

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: Integer

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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

LastModified

The UTC time string indicating the last time the event mapping was updated.

Type: Timestamp

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

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

The contents of your zip file containing your deployment package. If you are using the web API directly, the contents of the zip file must be base64-encoded. If you are using the AWS SDKs or the AWS CLI, the SDKs or CLI will do the encoding for you. For more information about creating a .zip file, see Execution Permissions in the *AWS Lambda Developer Guide*.

Type: Base64-encoded binary data object

Required: No

FunctionCodeLocation

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

A complex type that describes function metadata.

Contents

CodeSha256

It is the SHA256 hash of your function deployment package.

Type: String

Required: No

CodeSize

The size, in bytes, of the function .zip file you uploaded.

Type: Long

Required: No

DeadLetterConfig

The parent object that contains the target Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

Type: DeadLetterConfig (p. 396) object

Required: No

Description

The user-provided description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

Environment

The parent object that contains your environment's configuration settings.

Type: EnvironmentResponse (p. 399) object

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-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

FunctionName

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

Handler

The function Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

KMSKeyArn

The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If empty, it means you are using the AWS Lambda default service key.

Type: String

Pattern: `(arn:aws:[a-z0-9-.]+:.*)|()`

Required: No

LastModified

The time stamp 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: Integer

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.

To use the Node.js runtime v4.3, set the value to "nodejs4.3". To use earlier runtime (v0.10.42), set the value to "nodejs".

Type: String

Valid Values: `nodejs | nodejs4.3 | java8 | python2.7 | dotnetcore1.0 | nodejs4.3-edge`

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: Integer

Valid Range: Minimum value of 1.

Required: No

Version

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Required: No

VpcConfig

VPC configuration associated with your Lambda function.

Type: VpcConfigResponse (p. 406) object

Required: No

VpcConfig

If your Lambda function accesses resources in a VPC, you provide this parameter identifying the list of security group IDs and subnet IDs. These must belong to the same VPC. You must provide at least one security group and one subnet ID.

Contents

SecurityGroupIds

A list of one or more security groups IDs in your VPC.

Type: array of Strings

Array Members: Maximum number of 5 items.

Required: No

SubnetIds

A list of one or more subnet IDs in your VPC.

Type: array of Strings

Array Members: Maximum number of 16 items.

Required: No

VpcConfigResponse

VPC configuration associated with your Lambda function.

Contents

SecurityGroupIds

A list of security group IDs associated with the Lambda function.

Type: array of Strings

Array Members: Maximum number of 5 items.

Required: No

SubnetIds

A list of subnet IDs associated with the Lambda function.

Type: array of Strings

Array Members: Maximum number of 16 items.

Required: No

VpcId

The VPC ID associated with you Lambda function.

Type: String

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**: November 30, 2016

Change	Description	Date
Introduced AWS Lambda support for the .NET runtime, Lambda@Edge (Preview), Dead Letter Queues and automated deployment of serverless applications.	AWS Lambda introduces the following features in this release. • AWS Lambda added support for C#. For more information, see Programming Model for Authoring Lambda Functions in C# (p. 46). • Lambda@Edge (Preview) allows you to run Lambda functions at the AWS Edge locations in response to CloudFront events. For more information, see AWS Lambda@Edge (Preview) (p. 277). • Added a tutorial for automating deployment of serverless applications using AWS CodePipeline, AWS CodeBuild and AWS CloudFormation. For more information, see Automating Deployment of Lambda-based Applications (p. 147). • Updated Troubleshooting and Monitoring AWS Lambda Functions with Amazon CloudWatch (p. 107) to include a section on Dead Letter Queues (p. 113), which you can configure to retrieve information on failed asynchronous invocations of Lambda functions.	In this release
AWS Lambda adds Amazon Lex as a supported event source.	Using Lambda and Amazon Lex, you can quickly build chat bots for various services like Slack and Facebook. For more information, see Amazon Lex (p. 129).	November 30, 2016
US West (N. California) Region	AWS Lambda is now available in the US West (N. California) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the *AWS General Reference*.	November 21, 2016

Change	Description	Date
Introduced the AWS Serverless Application Model for creating and deploying Lambda-based applications and using environment variables for Lambda function configuration settings.	AWS Lambda introduces the following features in this release. • AWS Serverless Application Model: You can now use the AWS SAM to define the syntax for expressing resources within a serverless application. In order to deploy your application, simply specify the resources you need as part of your application, along with their associated permissions policies in a AWS CloudFormation template file (written in either JSON or YAML), package your deployment artifacts, and deploy the template. For more information, see Deploying Lambda-based Applications (p. 139). • Environment variables: You can use environment variables to specify configuration settings for your Lambda function outside of your function code. For more information, see Environment Variables (p. 89).	November 18, 2016
Added a tutorial under Getting Started (p. 160) for creating an Amazon API Gateway endpoint using the Lambda console	The tutorial instructs how to seamlessly integrate a Lambda function with an API via new features introduced in Configure Proxy Integration for a Proxy Resource. For more information, see Step 3: Create a Simple Microservice using Lambda and API Gateway (p. 171).	August 29, 2016
Asia Pacific (Seoul) Region	AWS Lambda is now available in the Asia Pacific (Seoul) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the *AWS General Reference*.	August 29, 2016
Asia Pacific (Sydney) Region	Lambda is now available in the Asia Pacific (Sydney) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the *AWS General Reference*.	June 23, 2016
Updates to the Lambda console	The Lambda console has been updated to simplify the role-creation process. For more information, see Step 2.1: Create a Hello World Lambda Function (p. 164).	June 23, 2016
AWS Lambda now supports Node.js runtime v4.3	AWS Lambda added support for Node.js runtime v4.3. For more information, see Programming Model (Node.js) (p. 9).	April 07, 2016
EU (Frankfurt) region	Lambda is now available in the EU (Frankfurt) region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the *AWS General Reference*.	March 14, 2016
VPC support	You can now configure a Lambda function to access resources in your VPC. For more information, see Configuring a Lambda Function to Access Resources in an Amazon VPC (p. 96). For example walkthroughs, see Tutorials: Configuring a Lambda Function to Access Resources in an Amazon VPC (p. 98).	February 11, 2016

Change	Description	Date
Content reorganization	The reorganized content now provides the following: • Getting Started (p. 160) – Contains a console-based exercise in which you create a Hello World Lambda function. You explore the AWS Lambda console features, including blueprints that enable you to create Lambda functions with just a few clicks. • Use Cases (p. 175) – Provides examples of how to use AWS Lambda with other AWS services or your custom applications as event sources, invoke over HTTPS, and set up AWS Lambda to invoke your Lambda function at scheduled interval. • Programming Model (p. 8) – Explains programming model core concepts and describes language-specific details. Regardless of the language you choose, there is a common pattern to writing code for a Lambda function. • Creating a Deployment Package (p. 56) – Explains how to create deployment packages for Lambda function code that is authored in languages supported by AWS Lambda (Python, Java, and Node.js).	December 9, 2015
AWS Lambda runtime has been updated.	AWS Lambda runtime has been updated with the following SDK and Linux kernel versions in this release: • AWS SDK for JavaScript: 2.2.12 • Boto SDK: 1.2.1 • Linux kernel version: 3.14.48-33.39.amzn1.x86_6. For more information, see Lambda Execution Environment and Available Libraries (p. 157).	November 4, 2015
Versioning support, Python for developing code for Lambda functions, scheduled events, and increase in execution time	AWS Lambda introduces the following features in this release. • Python: You can now develop your Lambda function code using Python. For more information, see Programming Model (p. 8). • Versioning: You can maintain one or more versions of your Lambda function. Versioning allows you to control which Lambda function version is executed in different environments (for example, development, testing, or production). For more information, see AWS Lambda Function Versioning and Aliases (p. 72). • Scheduled events: You can also set up AWS Lambda to invoke your code on a regular, scheduled basis using the AWS Lambda console. You can specify a fixed rate (number of hours, days, or weeks) or you can specify a cron expression. For an example, see Using AWS Lambda with Scheduled Events (p. 263). • Increase in execution time: You can now set up your Lambda functions to run for up to five minutes allowing longer running functions such as large volume data ingestion and processing jobs.	October 08, 2015

Change	Description	Date
Two new walkthroughs	The following new walkthroughs are added. They both use Java Lambda function. Tutorial: Using AWS Lambda with Amazon DynamoDB (p. 204) Using AWS Lambda as Mobile Application Backend (Custom Event Source: Android) (p. 250)	August 27, 2015
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 Tutorial: Using AWS Lambda with Amazon DynamoDB (p. 204) .	July 14, 2015
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 Using AWS Lambda with Amazon API Gateway (On-Demand Over HTTPS) (p. 236). For more information about the Amazon API Gateway, see https://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 (p. 160).	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 Programming Model (p. 8).	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: How It Works (p. 151) Getting Started (p. 160) AWS Lambda	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*.