AWS Lambda

Developer Guide

AWS Lambda: Developer Guide

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#, Go 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 Kinesis, or create your own back end that operates at AWS scale, performance, and security.

You can also build serverless applications composed of functions that are triggered by events and automatically deploy them using AWS CodePipeline and AWS CodeBuild. For more information, see Deploying Lambda-based Applications (p. 293).

For more information about the AWS Lambda execution environment, see Lambda Execution Environment and Available Libraries (p. 407). For information about how AWS Lambda determines compute resources required to execute your code, see Configuring Lambda Functions (p. 133).

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, Go and 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.
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. 15).
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. 3).
4. **Read the "Deploying 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 Deploying Lambda-based Applications (p. 293).

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. 177).

The following topics provide additional information about AWS Lambda:

- AWS Lambda Function Versioning and Aliases (p. 293)
- Using Amazon CloudWatch (p. 330)
- Best Practices for Working with AWS Lambda Functions (p. 402)
- AWS Lambda Limits (p. 410)

Getting Started

In this section, we introduce you to the fundamental concepts of a typical Lambda-based application and the options available to create and test your applications. In addition, you will be provided with instructions on installing the necessary tools to complete the tutorials included in this guide and create your first Lambda function.

Building Blocks of a Lambda-based Application

- **Lambda function:** The foundation, it is comprised of your custom code and any dependent libraries. For more information, see Lambda Functions (p. 15).
- **Event source:** An AWS service, such as Amazon SNS, or a custom service, that triggers your function and executes its logic. For more information, see Event Source Mapping (p. 152).
- **Downstream resources:** An AWS service, such as DynamoDB tables or Amazon S3 buckets, that your Lambda function calls once it is triggered.
- **Log streams:** While Lambda automatically monitors your function invocations and reports metrics to CloudWatch, you can annotate your function code with custom logging statements that allow you to analyze the execution flow and performance of your Lambda function to ensure it's working properly.
- **AWS SAM:** A model to define serverless applications. AWS SAM is natively supported by AWS CloudFormation and defines simplified syntax for expressing serverless resources. For more information, see Using the AWS Serverless Application Model (AWS SAM) (p. 313)

Tools to Create and Test Lambda-based Applications

There are three key tools that you use to interact with the AWS Lambda service, described below. We will cover tools for building AWS Lambda-based applications in further sections.

- **Lambda Console:** Provides a way for you to graphically design your Lambda-based application, author or update your Lambda function code, and configure event, downstream resources and IAM permissions that your function requires. It also includes advanced configuration options, outlined in Advanced Topics (p. 393).
- **AWS CLI:** A command-line interface you can use to leverage Lambda's API operations, such as creating functions and mapping event sources. For a full list of Lambda's API operations, see Actions (p. 413).
- **SAM CLI:** A command-line interface you can use to develop, test, and analyze your serverless applications locally before uploading them to the Lambda runtime. For more information, see Test Your Serverless Applications Locally Using SAM CLI (Public Beta) (p. 99).

Before you begin

In order to use the tutorials offered at the end of this section, make sure you have done the following:

- Set Up an AWS Account (p. 4)
- Set Up the AWS Command Line Interface (AWS CLI) (p. 6)
- Followed the steps to use SAM CLI, including Docker, outlined here: Install SAM CLI (p. 7).

Next Step

Set Up an AWS Account (p. 4)

Set Up an AWS Account

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:

Set Up an AWS Account and Create an Administrator User

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**.

   Note
   This might be unavailable in your browser if you previously signed into the AWS Management Console. In that case, choose **Sign in to a different account**, and then choose **Create a new 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. Use your AWS account email address and password to sign in as the *AWS account root user* to the IAM console at https://console.aws.amazon.com/iam/.

   Note
   We strongly recommend that you adhere to the best practice of using the **`Administrator`** IAM user below and securely lock away the root user credentials. Sign in as the root user only to perform a few account and service management tasks.
2. In the navigation pane of the console, choose **Users**, and then choose **Add user**.
3. For **User name**, type **`Administrator`**.
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 create a new password the next time the user signs in.
5. Choose **Next: Permissions**.
6. On the **Set permissions** page, choose **Add user to group**.
7. Choose **Create group**.
8. In the **Create group** dialog box, for **Group name** type **`Administrators`**.
9. For **Filter policies**, select the check box for **AWS managed - 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 **Create user**.

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.

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 AWS Management Console and open the 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:

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

Next Step

Set Up the AWS Command Line Interface (AWS CLI) (p. 6)

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 Set Up an AWS Account and Create an Administrator User (p. 4), 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. For more information on creating this profile, see Named Profiles.

```
[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
```

Next Step

Install SAM CLI (p. 7)

Install SAM CLI

SAM CLI is a tool that also allows faster, iterative development of your Lambda function code, which is explained at Test Your Serverless Applications Locally Using SAM CLI (Public Beta) (p. 99). To use SAM CLI, you first need to install Docker.

Installing Docker

Docker is an open-source software container platform that allows you to build, manage and test applications, whether you're running on Linux, Mac or Windows. For more information and download instructions, see Docker.

Once you have Docker installed, SAM CLI automatically provides a customized Docker image called `docker-lambda`. This image is designed specifically by an AWS partner to simulate the live AWS Lambda execution environment. This environment includes installed software, libraries, security permissions, environment variables, and other features outlined at Lambda Execution Environment and Available Libraries (p. 407).

Using `docker-lambda`, you can invoke your Lambda function locally. In this environment, your serverless applications execute and perform much as in the AWS Lambda runtime, without your having to redeploy the runtime. Their execution and performance in this environment reflect such considerations as timeouts and memory use.

> **Important**
> Because this is a simulated environment, there is no guarantee that your local testing results will exactly match those in the actual AWS runtime.

For more information, see Docker Lambda on GitHub. (If you don't have a Github account, you can create one for free and then access Docker Lambda).

Installing SAM CLI

The easiest way to install SAM CLI is to use pip.

You can run SAM CLI on Linux, Mac, or Windows environments. The easiest way to install SAM CLI is to use pip.

To use pip, you must have Python installed and added to your system's Environment path.

Note
In a Windows environment, you run pip from the *python-version*\Scripts directory.

```
pip install aws-sam-cli
```

Then verify that the installation succeeded.

```
sam --version
```

You should see something similar to the following:

```
SAM CLI, version 0.3.0
```

To begin using the SAM CLI with your serverless applications, see Test Your Serverless Applications Locally Using SAM CLI (Public Beta) (p. 99)

Next Step

Create a Simple Lambda Function and Explore the Console (p. 8)

Create a Simple Lambda Function and Explore the Console

In this Getting Started exercise you first create a Lambda function using the AWS Lambda console. Next, you manually invoke the Lambda function using 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.
- 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.

Preparing for the Getting Started

First, you need to sign up for an AWS account and create an administrator user in your account. For instructions, see Set Up an AWS Account (p. 4).

Next Step

Create a Simple Lambda Function (p. 9)

Create a Simple Lambda Function

Follow the steps in this section to create a simple Lambda function.

To create a Lambda function

1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. Note that AWS Lambda offers a simple `Hello World` function upon introduction under the **How it works** label and includes a **Run** option, allowing you to invoke the function as a general introduction. This tutorial introduces additonal options you have to create, test and update your Lambda functions, as well as other features provided by the Lambda console and provides links to each, inviting you to explore each one in depth.

   Choose **Create a function** under the **Get Started** section to proceed.

   > **Note**
   > The console shows the **Get Started** 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 function** to go to the **Create function** page.

3. On the **Create function** page, you are presented with three options:

   - **Author from scratch**
   - **Blueprints**
   - **Serverless Application Repository**

   For more information on using the Serverless Application Repository, see What Is the AWS Serverless Application Repository?

   a. If you'd like to review the blueprints, choose the **Blueprints** button, which will display the available 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, choose the **`Author from scratch`** button.

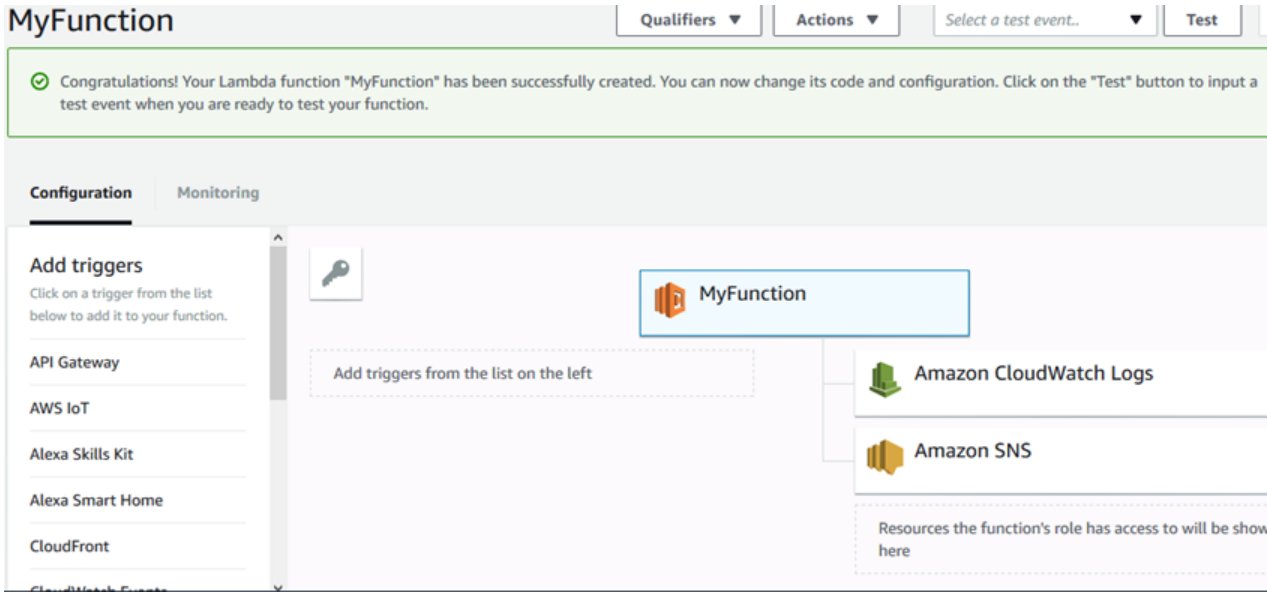
4. In **Author from scratch**, do the following:

   - In **Name***, specify your Lambda function name.
   - In **Runtime***, choose `Python 3.6`.
   - In **Role***, choose **Create new role from template(s):**
   - In **Role name***, enter a name for your role.
   - Leave the **Policy templates** field blank. For the purposes of this introduction, your Lambda function will have the necessary execution permissions.

     > **Note**
     > For an in-depth look at AWS Lambda's security polices, see Authentication and Access Control for AWS Lambda (p. 356).

   - Choose **Create Function.**

5. Under your new ***`function-name`*** page, note the following:



In the **Add triggers** panel, you can optionally choose a service that automatically triggers your Lambda function by choosing one of the service options listed.

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 example, do not configure a trigger.

- In **Function code** note that code is provided. It returns a simple "Hello from Lambda" greeting.
- **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. 50).
- Note the embedded IDE (Integrated Development Environment). To learn more, see Creating Functions Using the AWS Lambda Console Editor (p. 108).

6. Other configuration options on this page include:

- **Environment variables** – for Lambda functions enable you to dynamically pass settings to your function code and libraries, without making changes to your code. For more information, see Environment Variables (p. 393).
- **Tags** – are key-value pairs that you attach to AWS resources to better organize them. For more information, see Tagging Lambda Functions (p. 350).
- **Execution role** – which allows you to administer security on your function, using defined roles and policies or creating new ones. For more information, see Authentication and Access Control for AWS Lambda (p. 356).
- **Basic settings** – allows you to dictate the memory allocation and timeout limit for your Lambda function. For more information, see AWS Lambda Limits (p. 410).
- **Network** – allows you to select a VPC your function will access. For more information, see Configuring a Lambda Function to Access Resources in an Amazon VPC (p. 135).

- **Debugging and error handling** – allows you to select a Dead Letter Queues (p. 401) resource to analyze failed function invocation retries. It also allows you to enable active tracing. For more information, see Using AWS X-Ray (p. 338).
- **Concurrency** – allows you to allocate a specific limit of concurrent executions allowed for this function. For more information, see Function Level Concurrent Execution Limit (p. 390).
- **Auditing and compliance** – logs function invocations for operational and risk auditing, governance and compliance. For more information, see Using AWS Lambda with AWS CloudTrail (p. 229).

Invoke the Lambda Function Manually and Verify Results, Logs, and Metrics

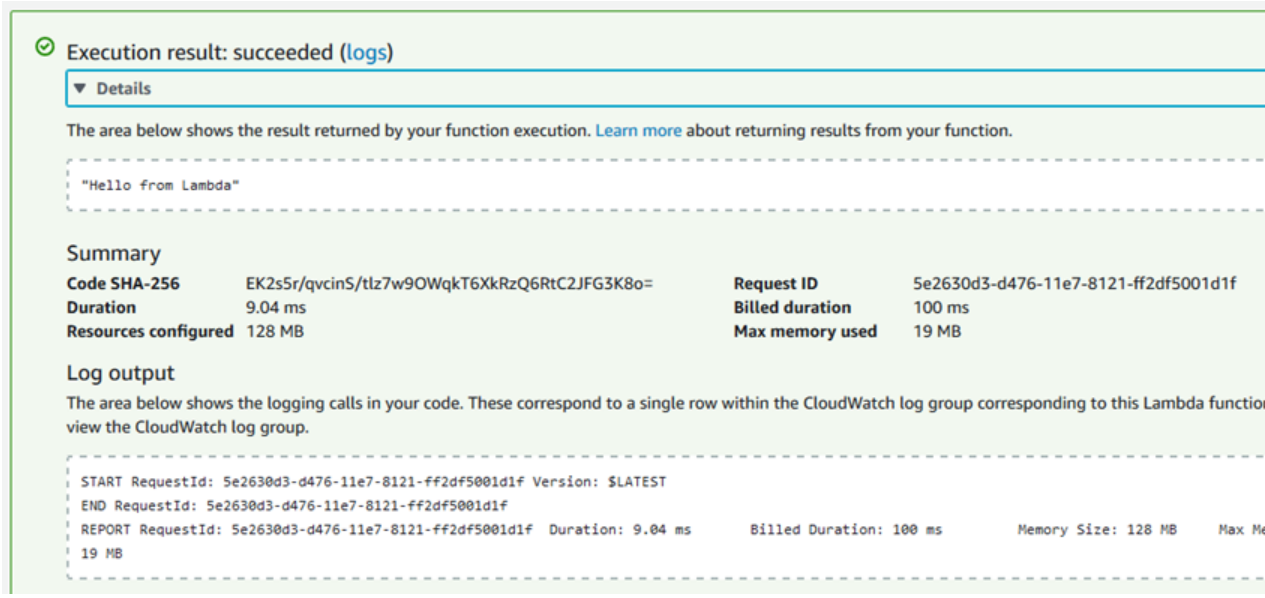
Follow the steps to invoke your Lambda function using the sample event data provided in the console.

1. On the ***yourfunction*** page, choose **Test**.
2. In the **Configure test event** page, choose **Create new test event** and in **Event template**, leave the default **Hello World** option. Enter an **Event name** and note the following sample event template:

   ```
   {
     "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.

   Note
   If you choose to delete the test event, go to the **Configure test event** page and then choose **Delete**.
3. Choose **Create** and then choose **Test**. Each user can create up to 10 test events per function. Those test events are not available to other users.
4. AWS Lambda executes your function on your behalf. The `handler` in your Lambda function receives and then processes the sample event.
5. Upon successful execution, view results in the console.



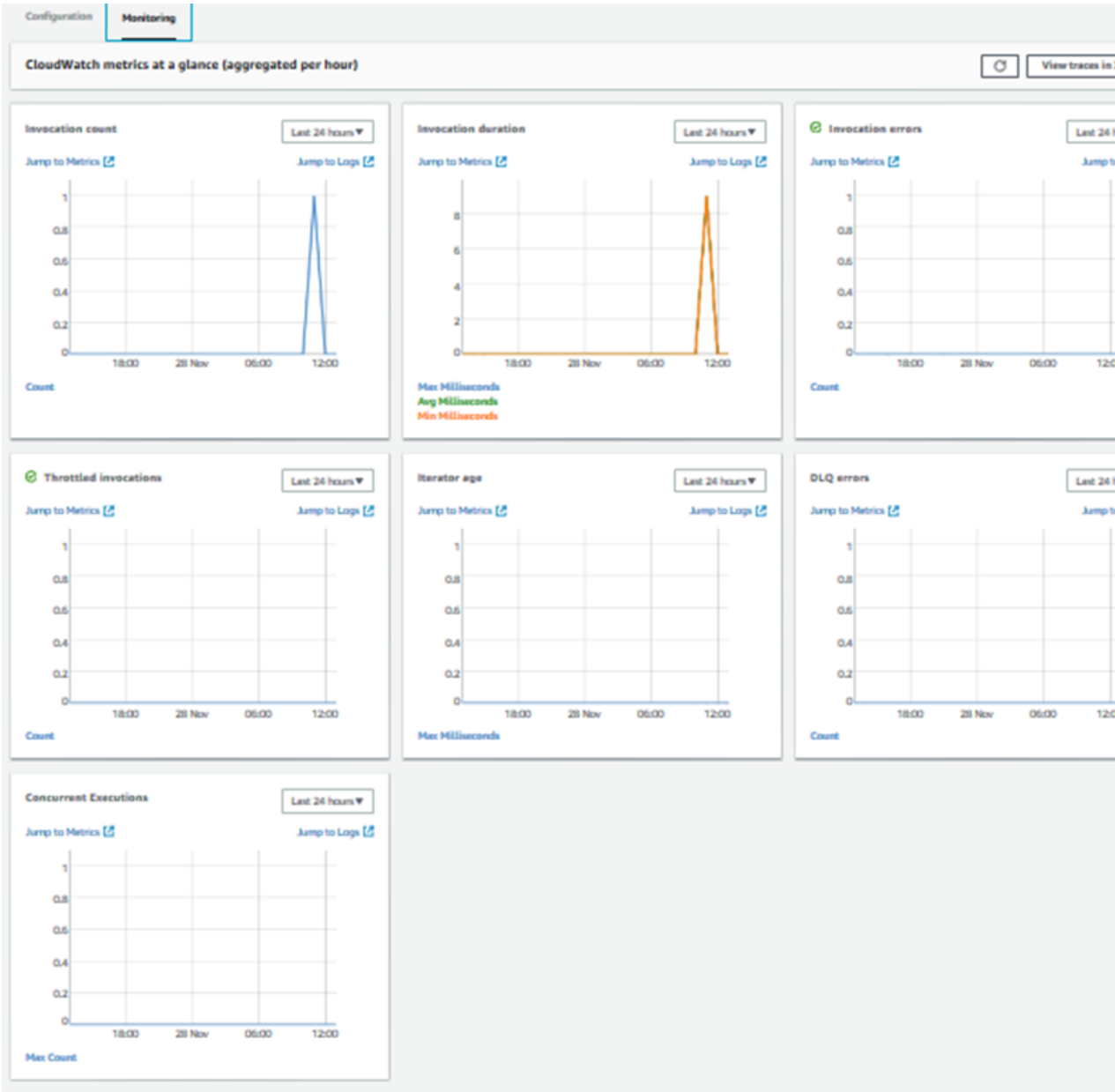
Note the following:

- 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. 151).
- 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.

6. Run the Lambda function a few times to gather some metrics that you can view in the next step.
7. Choose the **Monitoring** tab to view the CloudWatch metrics for your Lambda function. This page shows the CloudWatch metrics.



Note the following:

- The X-axis shows the past 24 hours from the current time.
- 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 AWS Lambda Limits (p. 410).
- Concurrent execution metrics show the number of concurrent invocations of your Lambda function. For more information, see Managing Concurrency (p. 389).

- 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.

For more information on these metrics and what they mean, see AWS Lambda CloudWatch Metrics (p. 336).

Lambda Functions

If you are new to AWS Lambda, you might ask: 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.

In subsequent sections, we cover how the functions you create get invoked, and how to deploy and monitor them. We also recommend reading the **Function Code** and **Function Configuration** sections at Best Practices for Working with AWS Lambda Functions (p. 402).

To begin, we introduce you to the topic that explains the fundamentals of building a Lambda function, Building Lambda Functions (p. 15).

Building Lambda Functions

You upload your application code in the form of one or more *Lambda functions* to AWS Lambda, a compute service. In turn, AWS Lambda executes 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.

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. 407). 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
Node.js	• AWS Lambda console • Visual Studio, with IDE plug-in (see AWS Lambda Support in Visual Studio) • Your own authoring environment • For more information, see Deploying Code and Creating a Lambda Function (p. 16).
Java	• Eclipse, with AWS Toolkit for Eclipse (see Using AWS Lambda with the AWS Toolkit for Eclipse) • Your own authoring environment • For more information, see Deploying Code and Creating a Lambda Function (p. 16).
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 • For more information, see Deploying Code and Creating a Lambda Function (p. 16).
Python	• AWS Lambda console • Your own authoring environment • For more information, see Deploying Code and Creating a Lambda Function (p. 16).
Go	• Your own authoring environment • For more information, see Deploying Code and Creating a Lambda Function (p. 16).

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. 18) section provides information for each of the supported languages.

> **Note**
> After you familiarize yourself with AWS Lambda, see the Use Cases (p. 177), 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

- Testing a Lambda Function (p. 17)

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. 77).

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. 429) 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. 15).

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.
- Test your Lambda function locally using Test Your Serverless Applications Locally Using SAM CLI (Public Beta) (p. 99).

The console provides sample event data. The same data is also provided in the Sample Events Published by Event Sources (p. 166) 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. 332).

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. 334).

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. 3) – The Getting Started exercise provides a console-based experience. Sample code is provided for your preferred runtimes. You can also code within the console, using the Code Editor and upload it to AWS Lambda, and test it using sample event data provided in the console.

- Use Cases (p. 177) – 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. 15). Most examples in the Use 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. 18)

Creating a Deployment Package (p. 77)

AWS Lambda Function Versioning and Aliases (p. 293)

Using Amazon CloudWatch (p. 330)

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. For more information, see Best Practices for Working with AWS Lambda Functions (p. 402)

The following language specific topics provide detailed information:

Programming Model(Node.js)

AWS Lambda currently supports the following Node.js runtimes:

- Node.js runtime v8.10 (runtime = nodejs8.10)
- Node.js runtime v6.10 (runtime = nodejs6.10)
- Node.js runtime v4.3 (runtime = nodejs4.3)*
- Node.js runtime v0.10.42 (runtime = nodejs)*

Important
*Node v0.10.42 and Node v4.3 are currently marked as deprecated. For more information, see Runtime Support Policy (p. 406). You must migrate existing functions to the newer Node.js runtime versions available on AWS Lambda (nodejs.8.10 or nodejs6.10) as soon as possible.

When you create a Lambda function, you specify the runtime that you want to use. For more information, see `runtime` parameter of CreateFunction (p. 429).

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 applies to all supported runtime versions, except where indicated.

Topics

Lambda Function Handler (Node.js)

AWS Lambda invokes your Lambda function via a `handler` object. A `handler` represents the name of your Lambda function (and serves as the entry point that AWS Lambda uses to execute your function code. For example:

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

```
    ... function code
    callback(null, "some success message");
   // or
   // callback("some error type");
}
```

- *myHandler* – This is the name of the function AWS Lambda invokes. Suppose you save this code as `helloworld.js`. Then, `myHandler` is the function that contains your Lambda function code and `helloworld` is the name of the file that represents your deployment package. For more information, see Creating a Deployment Package (Node.js) (p. 78).

  AWS Lambda supports two invocation types:

  - **RequestResponse**, or *synchronous execution*: AWS Lambda returns the result of the function call to the client invoking the Lambda function. If the handler code of your Lambda function does not specify a return value, AWS Lambda will automatically return `null` for that value. For a simple sample, see Example (p. 21).
  - **Event**, or *asynchronous execution*: AWS Lambda will discard any results of the function call.

    Note
    If you discover that your Lambda function does not process the event using asynchronous invocation, you can investigate the failure using Dead Letter Queues (p. 401).

    Event sources can range from a supported AWS service or custom applications that invoke your Lambda function. For examples, see Sample Events Published by Event Sources (p. 166). For a simple sample, see Example (p. 21).
- `context` – AWS Lambda uses this parameter to provide details of your Lambda function's execution. For more information, see The Context Object (Node.js) (p. 22).
- `callback` (optional)– See Using the Callback Parameter (p. 20).

Using the Callback Parameter

The Node.js runtimes v6.10 and v8.10 support 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.

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 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. 467)), 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.

If you are using runtime version 8.10, you can include the `async` keyword:

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

            // return information to the caller.
}
```

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, *myHandler*

In the function, the `console.log()` statements log some of the incoming event data to CloudWatch Logs. When the `callback` parameter is called, the Lambda function exits only after the event loop passed is empty.

If you want to use the `async` feature provided by the v8.10 runtime, consider the following code sample:

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

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.
- The sample uses **nodejs6.10** as the **runtime** but you can also select **nodejs8.10**. The code samples provided will work for any version.

For instructions to create a Lambda function using the console, see Create a Simple Lambda Function (p. 9).

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.

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. 19).

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.log_group_name);
    console.log('logStreamName =', context.log_stream_name);
    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 **nodejs6.10**. For instructions on how to do this, see Create a Simple Lambda Function (p. 9).
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 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 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. 20).

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 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 examples:

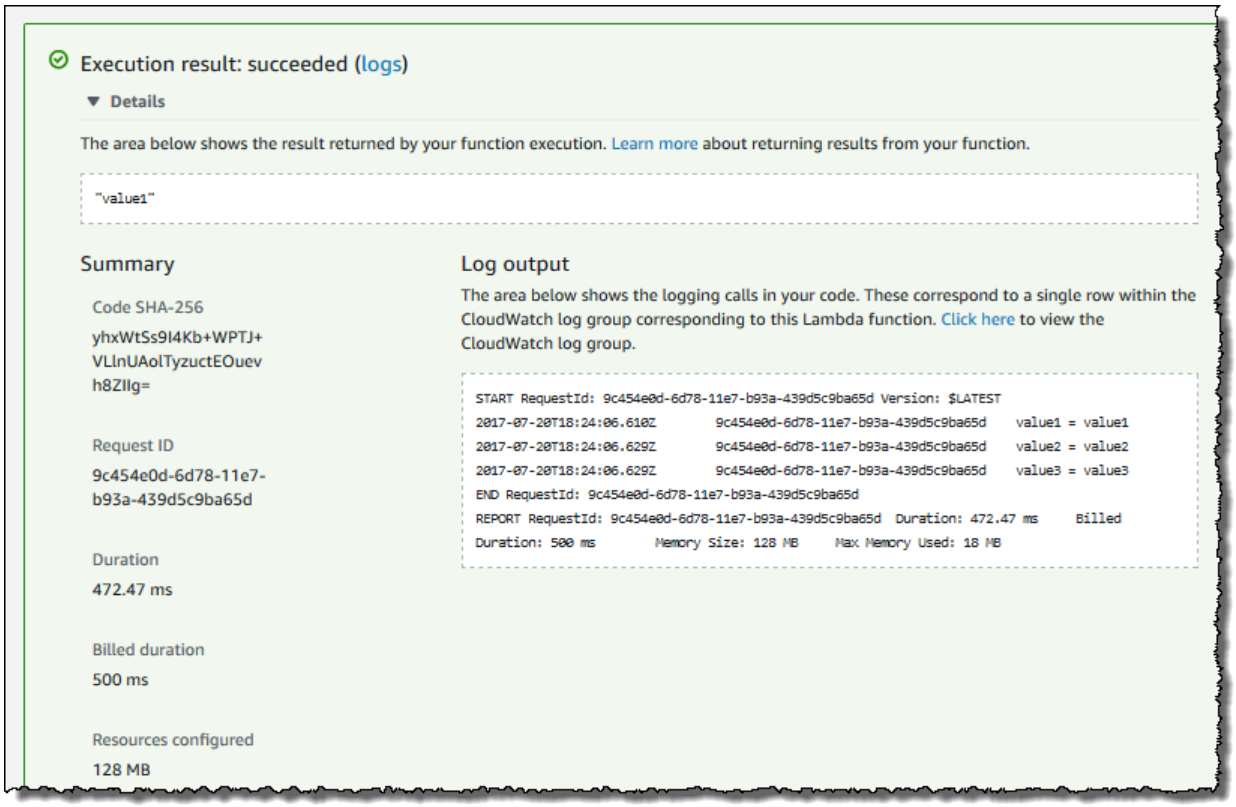
- The first sample can be written using either runtime version 6.10 or 4.3.

```
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 second sample uses the Node.js `async` feature, available only in runtime versions 8.10 or later.

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

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

In either case, the following screenshot shows an example **Log output** section in the Lambda console. You can examine the same information in CloudWatch Logs. For more information, see Accessing Amazon CloudWatch Logs for AWS Lambda (p. 334).



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 Create a Simple Lambda Function (p. 9).
2. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console. 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. 3).

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.

Function Errors (Node.js)

If your Lambda function notifies AWS Lambda that it failed to execute properly, 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 the 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)"
  ]
}
```

You would get the same result if you write the function using the async feature of Node.js runtime version 8.10. For example:

```
exports.handler = async function(event, context) {
    function AccountAlreadyExistsError(message) {
        this.name = "AccountAlreadyExistsError";
        this.message = message;
    }
    AccountAlreadyExistsError.prototype = new Error();

    const error = new AccountAlreadyExistsError("Account is in use!");
    throw error
};
```

Again, when this Lambda function is invoked, it will notify AWS Lambda that function execution completed with an error and passes the error information to AWS Lambda. AWS Lambda returns the error information back to the client:

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

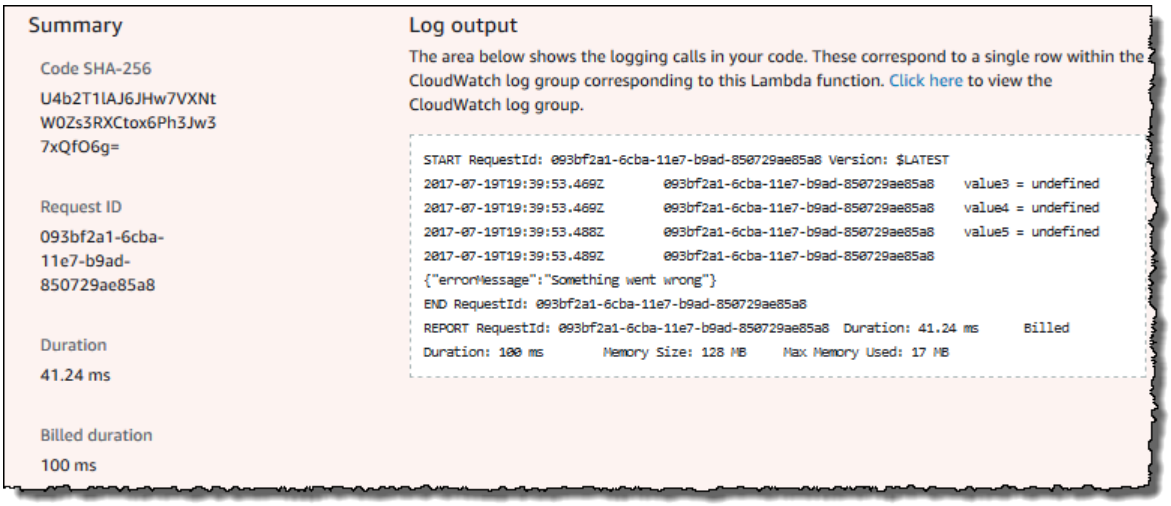
Note that the error information is returned as the `stackTrace` JSON array of stack trace elements.

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:

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



- 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 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 Understanding Retry Behavior (p. 155).

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 Create a Simple Lambda Function (p. 9).
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.

Function Error Handling

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see Handling Error Conditions Using a State Machine.

Consider a `CreateAccount` state is a task that writes a customer's details to a database using a Lambda function.

- If the task succeeds, an account is created and a welcome email is sent.
- If a user tries to create an account for a username that already exists, the Lambda function raises an error, causing the state machine to suggest a different username and to retry the account-creation process.

The following code samples demonstrate how to do this. Note that custom errors in Node.js must extend the error prototype.

```
exports.handler = function(event, context, callback) {
    function AccountAlreadyExistsError(message) {
        this.name = "AccountAlreadyExistsError";
        this.message = message;
    }
    AccountAlreadyExistsError.prototype = new Error();

    const error = new AccountAlreadyExistsError("Account is in use!");
    callback(error);
};
```

You can configure Step Functions to catch the error using a `Catch` rule:

```
{
   "StartAt": "CreateAccount",
   "States": {
      "CreateAccount": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:CreateAccount",
         "Next": "SendWelcomeEmail",
         "Catch": [
            {
               "ErrorEquals": ["AccountAlreadyExistsError"],
               "Next": "SuggestAccountName"
            }
         ]
      },
      …
   }
}
```

At runtime, AWS Step Functions catches the error, transitioning to the `SuggestAccountName` state as specified in the `Next` transition.

Note
The name property of the `Error` object must match the `ErrorEquals` value.

Custom error handling makes it easier to create serverless applications. This feature integrates with all the languages supported by the Lambda Programming Model (p. 18), allowing you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see AWS Step Functions.

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. 40)) 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. 36).
- **aws-lambda-java-events** – This library provides predefined types that you can use when writing Lambda functions to process events published by Amazon S3, 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 Log4j2.8** – You can use the custom Log4j (see Apache Log4j 2) appender provided by AWS Lambda for logging from your lambda functions. Every call to Log4j methods, such as log.info() or log.error(), will result in a CloudWatch Logs event. The custom appender is called LambdaAppender and must be used in the log4j2.xml file. You must include the aws-lambda-java-log4j2 artifact (artifactId:aws-lambda-java-log4j2) in the deployment package (.jar file).For more information, see Logging (Java) (p. 42).
- **Custom Appender for Log4j1.2** – You can use the custom Log4j (see Apache Log4j 1.2) appender provided by AWS Lambda for logging from your lambda functions. For more information, see Logging (Java) (p. 42).

  Note
  Support for the Log4j v1.2 custom appender is marked for End-Of-Life. It will not receive ongoing updates and is not recommended for use.

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. 36).

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 or 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. 32).
- 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. 40).

For example, consider the following Java example code.

```
package example;

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

public class Hello implements RequestHandler<Integer, String>{
    public String myHandler(int myCount, Context context) {
        return String.valueOf(myCount);
    }
}
```

In this example input is of type Integer 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 Integer object (as in this example) or any custom data object.

For instructions to create a Lambda function using this Java code, see (Optional) Create a Lambda Function Authored in Java (p. 49).

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. 32).
- For information about using predefined interfaces to create a handler, see Leveraging Predefined Interfaces for Creating Handler (Java) (p. 36).

  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 Function Errors (Java) (p. 46).

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. 35).)

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. 467).

To test an end-to-end example, see (Optional) Create a Lambda Function Authored in Java (p. 49).

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. 33).

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" }
```

For more information, see (Optional) Create a Lambda Function Authored in Java (p. 49). 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::handleRequest` (*`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)
 throws IOException{
   ...
}
```

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 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 (Optional) Create a Lambda Function Authored in Java (p. 49). 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. 30)).

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 region \
--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. For more information, see Creating a Deployment Package (Java) (p. 89)
- 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 (Optional) Create a Lambda Function Authored in Java (p. 49)). 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 (Optional) Create a Lambda Function Authored in Java (p. 49)). 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).
4

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 that called 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 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. 42).

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 (Optional) Create a Lambda Function Authored in Java (p. 49). 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™ 2

AWS Lambda recommends Log4j 2 to provide 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.info()` or `log.error()`, will result in a CloudWatch Logs event. The custom appender is called `LambdaAppender` and must be used in the `log4j2.xml` file. You must include the `aws-lambda-java-log4j2` artifact (`artifactId:aws-lambda-java-log4j2`) in the deployment package (.jar file). For an example, see Example 1: Writing Logs Using Log4J v2.8 (p. 43).

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. 45).

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. 467). 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 v2.8

- The following shows how to build your artifact with Maven to correctly include the Log4j v2.8 plugins:
  - For Maven pom.xml:

```
 <dependencies>
  ...
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-log4j2</artifactId>
    <version>1.0.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.8.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.8.2</version>
  </dependency>
  ....
</dependencies>
```

  - If using the Maven shade plugin, set the plugin configuration as follows:

```
<plugins>
  ...
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.4.3</version>
    <executions>
      <execution>
        <phase>package</phase>
        <goals>
          <goal>shade</goal>
        </goals>
        <configuration>
          <transformers>
            <transformer

 implementation="com.github.edwgiz.mavenShadePlugin.log4j2CacheTransformer.PluginsCacheFileTransfor
            </transformer>
          </transformers>
        </configuration>
      </execution>
    </executions>
    <dependencies>
```

```
      <dependency>
        <groupId>com.github.edwgiz</groupId>
        <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
        <version>2.8.1</version>
      </dependency>
    </dependencies>
  </plugin>
  ...
</plugins>
```

- The following Java code example shows how to use Log4j with Lambda:

```
package example;

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

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Hello {
    // Initialize the Log4j logger.
    static final Logger logger = LogManager.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");


        logger.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 preceding uses the following log4j2.xml file to load properties

```
 <?xml version="1.0" encoding="UTF-8"?>
<Configuration packages="com.amazonaws.services.lambda.runtime.log4j2">
  <Appenders>
    <Lambda name="Lambda">
      <PatternLayout>
          <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1}:%L - %m%n</
pattern>
      </PatternLayout>
    </Lambda>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="Lambda" />
```

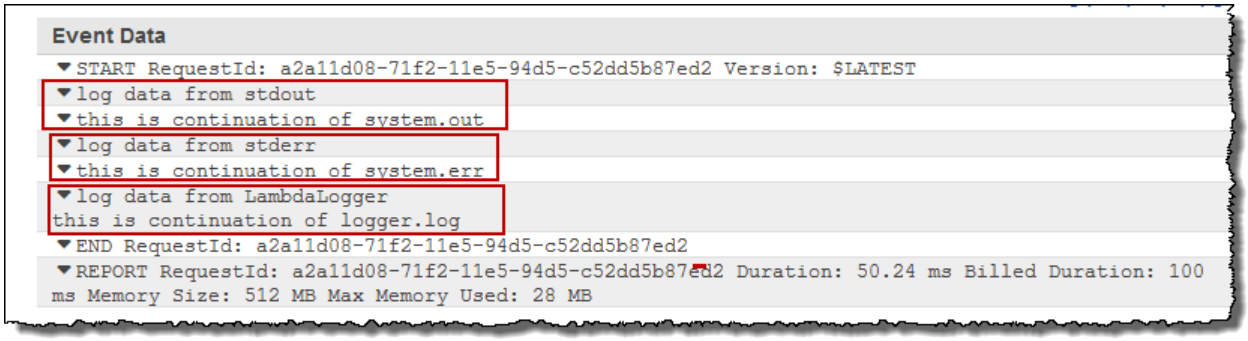
```
    </Root>
  </Loggers>
</Configuration>
```

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.



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 (Optional) Create a Lambda Function Authored in Java (p. 49). 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.

Function Errors (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:

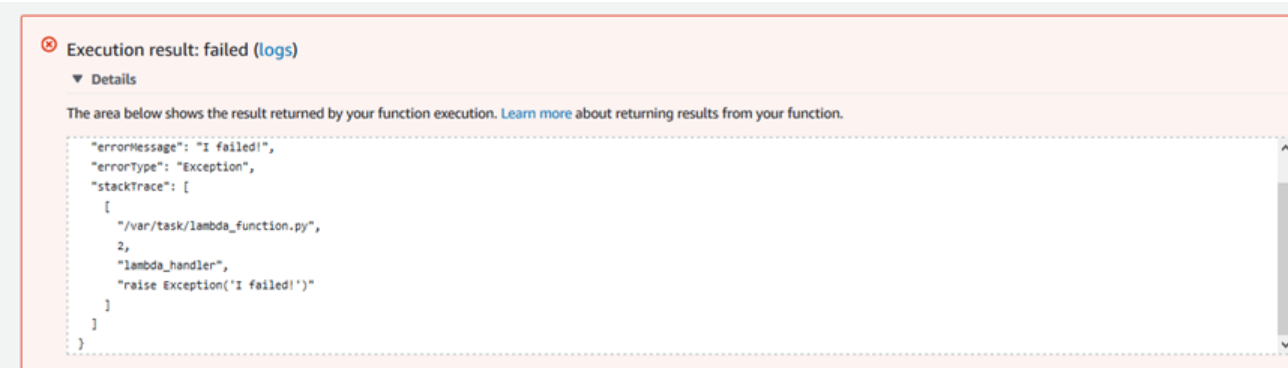
```
{
  "errorMessage": "Name John Doe is invalid. Exception occurred...",
  "errorType": "java.lang.Exception",
  "stackTrace": [
    "example.Hello.handler(Hello.java:9)",
    "sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)",
    "sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)",

 "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.



- `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 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.

Function Error Handling

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see Handling Error Conditions Using a State Machine.

Consider a `CreateAccount` state is a task that writes a customer's details to a database using a Lambda function.

- If the task succeeds, an account is created and a welcome email is sent.
- If a user tries to create an account for a username that already exists, the Lambda function raises an error, causing the state machine to suggest a different username and to retry the account-creation process.

The following code samples demonstrate how to do this. Note that custom errors in Java must extend the `Exception` class.

```
package com.example;

public static class AccountAlreadyExistsException extends Exception {
    public AccountAlreadyExistsException(String message) {
        super(message);
    }
}

package com.example;

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

public class Handler {
    public static void CreateAccount(String name, Context context) throws
 AccountAlreadyExistsException {
        throw new AccountAlreadyExistsException ("Account is in use!");
    }
}
```

You can configure Step Functions to catch the error using a `Catch` rule. Lambda automatically sets the error name to the fully-qualified class name of the exception at runtime:

```
{
   "StartAt": "CreateAccount",
   "States": {
      "CreateAccount": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:CreateAccount",
         "Next": "SendWelcomeEmail",
         "Catch": [
            {
               "ErrorEquals": ["com.example.AccountAlreadyExistsException"],
               "Next": "SuggestAccountName"
            }
         ]
      },
      …
   }
}
```

At runtime, AWS Step Functions catches the error, transitioning to the `SuggestAccountName` state as specified in the `Next` transition.

Custom error handling makes it easier to create serverless applications. This feature integrates with all the languages supported by the Lambda Programming Model (p. 18), allowing you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see AWS Step Functions.

Using Earlier Custom Appender for Log4j™ 1.2 (Not Recommended)

Note
Support for the Log4j v1.2 custom appender is marked for End-Of-Life. It will not receive ongoing updates and is not recommended for use. For more information, see Log4j 1.2

AWS Lambda supports Log4j 1.2 by providing a custom appender. You can use the custom Log4j (see Apache log4j 1.2) appender provided by Lambda for logging from your lambda functions. Every call to Log4j methods, such as `log.info()` 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: Writing Logs Using Log4J v1.2 (Not Recommended) (p. 48).

Example: Writing Logs Using Log4J v1.2 (Not Recommended)

Note
Versions 1.x of Log4j have been marked as end of life. For more information, see Log4j 1.2

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");



        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 = INFO, 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}:%m%n
```

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` file 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 (Optional) Create a Lambda Function Authored in Java (p. 49). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-log4j` dependency for Log4j 1.2 dependency.
- When you create the Lambda function, specify `example.Hello::myHandler` (*package*`.`*class*`::`*method*) as the handler value.

(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. 30). 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).
- You can use the Lambda 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. 40).

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 creating a deployment package, see Creating a Deployment Package (Java) (p. 89).

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) (p. 50)
- The Context Object (Python) (p. 51)
- Logging (Python) (p. 54)
- Function Errors (Python) (p. 56)

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 returns `NONE`, 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. 96).
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 Create a Simple Lambda Function (p. 9). In this example, the handler is `hello_python.my_handler` (*`file-name`*.*`function-name`*). Note that the Getting Started (p. 3) 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. Note that the `--runtime` parameter specifies `python3.6`. You can also use `python2.7`. For a complete description of the `create-function` command and its parameters, see CreateFunction (p. 429)

   ```
   aws lambda create-function \
   --region region \
   --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 python3.6 \
   --timeout 15 \
   --memory-size 512
   ```

The Context Object (Python)

Topics