Scripting Languages for AWS Lambda: Running PHP, Ruby, and Go

by Bryan Liston | on 09 DEC 2016 | in AWS Lambda | Permalink | Comments

Dimitrij Zub, Solutions Architect

Raphael Sack, Technical Trainer

In our daily work with partners and customers, we see a lot of different amazing skills, expertise and experience in many fields, and programming languages. From languages that have been around for a while to languages on the cutting edge, many teams have developed a deep understanding of concepts of each language; they want to apply these languages with and within the innovations coming from AWS, such as AWS Lambda.

Lambda provides native support for a wide array of languages, such as Java, Node.js, Python, and C#. In this post, we outline how you can use Lambda with different scripting languages.

For each language, you perform the following tasks:

- Prepare: Launch an instance from an AMI and log in via SSH
- Compile and package the language for Lambda
- Install: Create the Lambda package and test the code

The preparation and installation steps are similar between languages, but we provide step-by-step guides and examples for compiling and packaging PHP, Go, and Ruby.

Common steps to prepare

You can use the capabilities of Lambda to run arbitrary executables to prepare the binaries to be executed within the Lambda environment.

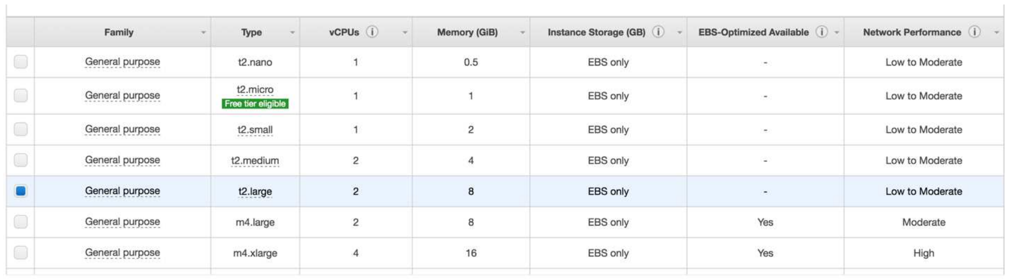
The following steps are only an overview on how to get PHP, Go, or Ruby up and running on Lambda; however, using this approach, you can add more specific libraries, extend the compilation scope, and leverage JSON to interconnect your Lambda function to Amazon API Gateway and other services.

After your binaries have been compiled and your basic folder structure is set up, you won't need to redo those steps for new projects or variations of your code. Simply write the code to accept inputs from STDIN and return to STDOUT and the written Node.js wrapper takes care of bridging the runtimes for you.

For the sake of simplicity, we demonstrate the preparation steps for PHP only, but these steps are also applicable for the other environments described later.

In the Amazon EC2 console, choose **Launch instance**. When you choose an AMI, use one of the AMIs in the Lambda Execution Environment and Available Libraries list, for the same region in which you will run the PHP code and launch an EC2 instance to have a compiler. For more information, see Step 1: Launch an Instance.

Pick **t2.large** as the EC2 instance type to have two cores and 8 GB of memory for faster PHP compilation times.



	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
☐	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate
☐	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate
☐	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate
■	General purpose	t2.large	2	8	EBS only	-	Low to Moderate
☐	General purpose	m4.large	2	8	EBS only	Yes	Moderate
☐	General purpose	m4.xlarge	4	16	EBS only	Yes	High

Choose **Review and Launch** to use the defaults for storage and add the instance to a default, SSH only, security group generated by the wizard.

Choose **Launch** to continue; in the launch dialog, you can select an existing key-pair value for your login or create a new one. In this case, create a new key pair called "php" and download it.

Blog

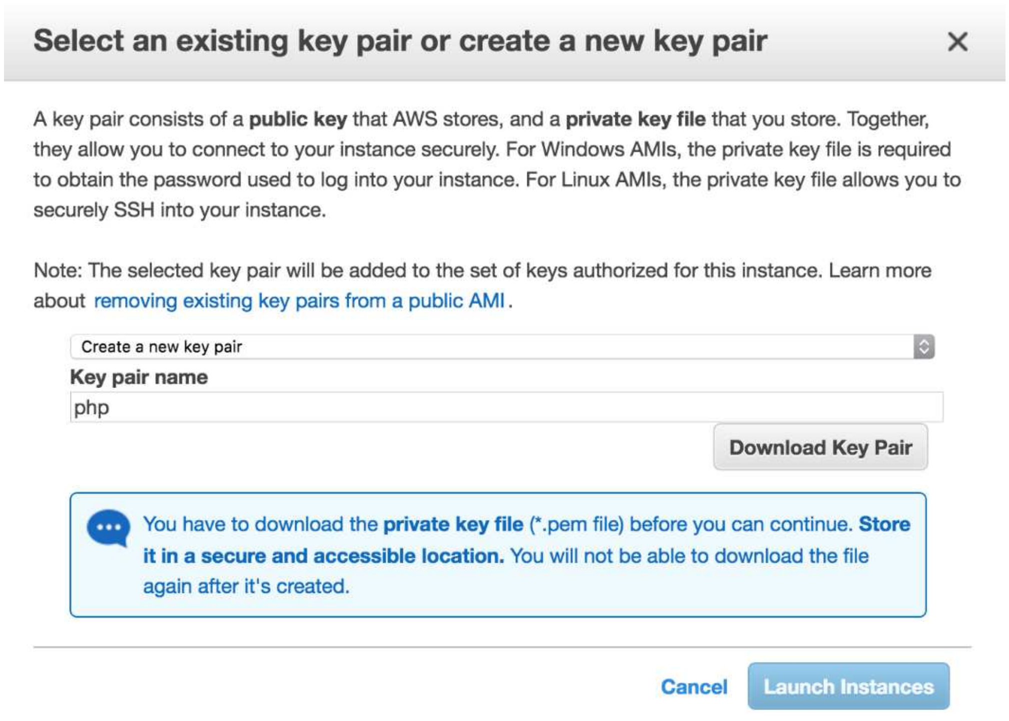
AWS Blog (Korea)

RSS Feed

Subscribe to this blog's feed

Contact Us

Comments? Questions? Talk to us



Select an existing key pair or create a new key pair

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about removing existing key pairs from a public AMI.

Create a new key pair

Key pair name

php

You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

After downloading the keys, navigate to the download folder and run the following command:

Bash

```bash
chmod 400 php.pem
```

This is required because of SSH security standards. You can now connect to the instance using the EC2 public DNS. Get the value by selecting the instance in the console and looking it up under **Public DNS** in the lower right part of the screen.

Bash

```bash
ssh -i php.pem ec2-user@[PUBLIC DNS]
```

You're done! With this instance up and running, you have the right AMI in the right region to be able to continue with all the other steps.

Getting ready for PHP

After you have logged in to your running AMI, you can start compiling and packaging your environment for Lambda. With PHP, you compile the PHP 7 environment from the source and make it ready to be packaged for the Lambda environment.

Setting up PHP on the instance

The next step is to prepare the instance to compile PHP 7, configure the PHP 7 compiler to output in a defined directory, and finally compile PHP 7 to the Lambda AMI.

Update the package manager by running the following command:

Bash

```bash
sudo yum update -y
```

Install the minimum necessary libraries to be able to compile PHP 7:

Bash

```bash
sudo yum install gcc gcc-c++ libxml2-devel -y
```

With the dependencies installed, you need to download the PHP 7 sources available from

PHP Downloads.

For this post, we were running the EC2 instance in Ireland, so we selected http://ie1.php.net/get/php-7.0.7.tar.bz2/from/this/mirror as our mirror. Run the following command to download the sources to the instance and choose your own mirror for the appropriate region.

Bash

```bash
cd ~

wget http://ie1.php.net/distributions/php-7.0.7.tar.bz2 .

Extract the files using the following command:

tar -jxvf php-7.0.7.tar.bz2
```

This creates the php-7.0.7 folder in your home directory. Next, create a dedicated folder for the php-7 binaries by running the following commands.

Bash

```
mkdir /home/ec2-user/php-7-bin

./configure --prefix=/home/ec2-user/php-7-bin/
```

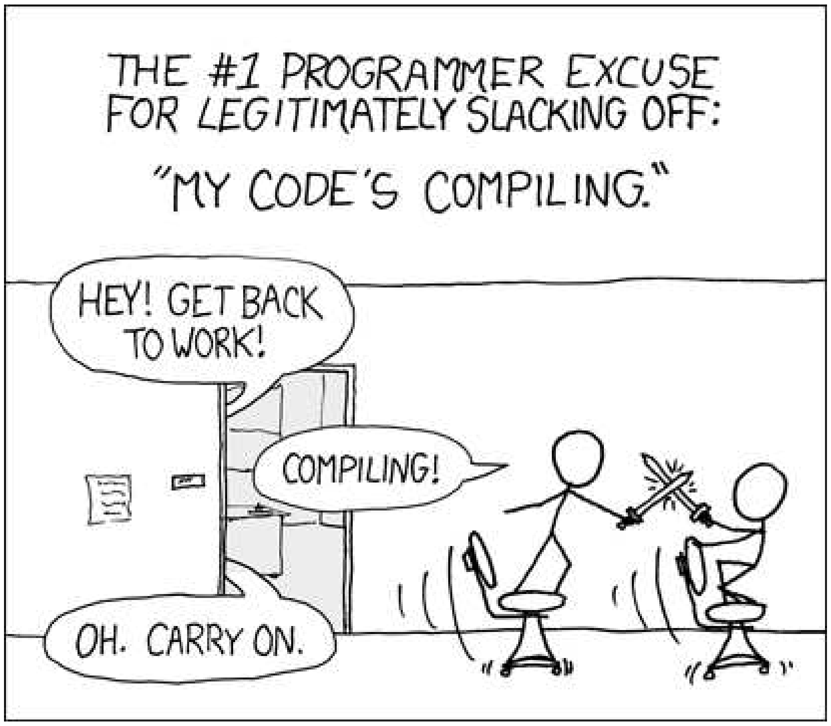
This makes sure the PHP compilation is nicely packaged into the php binaries folder you created in your home directory. Keep in mind, that you only compile the baseline PHP here to reduce the amount of dependencies required for your Lambda function.

You can add more dependencies and more compiler options to your PHP binaries using the options available in ./configure. Run ./configure –h for more information about what can be packaged into your PHP distribution to be used with Lambda, but also keep in mind that this will increase the overall binaries package.

Finally, run the following command to start the compilation:

Bash

```
make install
```



https://xkcd.com/303/

After the compilation is complete, you can quickly confirm that PHP is functional by running the following command:

Bash

```
cd ~/php-7-bin/bin/

./php -v

PHP 7.0.7 (cli) (built: Jun 16 2016 09:14:04) ( NTS )

Copyright (c) 1997-2016 The PHP Group

Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
```

Time to code

Using your favorite editor, you can create an entry point PHP file, which in this case reads input from a Linux pipe and provide its output to stdout. Take a simple JSON document and count the amounts of top-level attributes for this matter. Name the file HelloLambda.php.

PHP

```
<?php

$data = stream_get_contents(STDIN);

$json = json_decode($data, true);

$result = json_encode(array('result' => count($json)));

echo $result."n";

?>
```

Creating the Lambda package

With PHP compiled and ready to go, all you need to do now is to create your Lambda package with the Node.js wrapper as an entry point.

First, tar the php-7-bin folder where the binaries reside using the following command:

Bash

```
cd ~
```

```
tar -zcvf php-7-bin.tar.gz php-7-bin/
```

Download it to your local project folder where you can continue development, by logging out and running the following command from your local machine (Linux or OSX), or using tools like WinSCP on Windows:

Bash

```bash
scp -i php.pem ec2-user@[EC2_HOST]:~/php-7-bin.tar.gz .
```

With the package download, you create your Lambda project in a new folder, which you can call php-lambda for this specific example. Unpack all files into this folder, which should result in the following structure:

Bash

```bash
php-lambda

+-- php-7-bin
```

The next step is to create a Node.js wrapper file. The file takes the inputs of the Lambda invocations, invoke the PHP binary with helloLambda.php as a parameter, and provide the inputs via Linux pipe to PHP for processing. Call the file php.js and copy the following content:

JavaScript

```javascript
process.env['PATH'] = process.env['PATH'] + ':' + process.env

const spawn = require('child_process').spawn;

exports.handler = function(event, context) {

    //var php = spawn('php',['helloLambda.php']); //local debu
    var php = spawn('php-7-bin/bin/php',['helloLambda.php']);
    var output = "";

    //send the input event json as string via STDIN to php pro
    php.stdin.write(JSON.stringify(event));

    //close the php stream to unblock php process
    php.stdin.end();
```

```
        //dynamically collect php output
        php.stdout.on('data', function(data) {
              output+=data;
        });

        //react to potential errors
        php.stderr.on('data', function(data) {
                console.log("STDERR: "+data);
        });

        //finalize when php process is done.
        php.on('close', function(code) {
                context.succeed(JSON.parse(output));
        });
    }

    //local debug only
    //exports.handler(JSON.parse("{"hello":"world"}"));
```

With all the files finalized, the folder structure should look like the following:

php-lambda

+– php-7-bin

— helloLambda.php

— php.js

The final step before the deployment is to zip the package into an archive which can be uploaded to Lambda. Call the package LambdaPHP.zip. Feel free to remove unnecessary files, such as phpdebug, from the php-7-bin/bin folder to reduce the size of the archive.

Go, Lambda, go!

The following steps are an overview of how to compile and execute Go applications on Lambda. As with the PHP section, you are free to enhance and build upon the Lambda function with other AWS services and your application infrastructure. Though this example allows you to use your own Linux machine with a fitting distribution to work locally, it might still be useful to understand the Lambda AMIs for test and automation.

To further enhance your environment, you may want to create an automatic compilation pipeline and even deployment of the Go application to Lambda. Consider using versioning and aliases, as they help in managing new versions and dev/test/production code.

Setting up Go on the instance

The next step is to set up the Go binaries on the instance, so that you can compile the upcoming application.

First, make sure your packages are up to date (always):

Bash

```bash
sudo yum update -y
```

Next, visit the official Go site, check for the latest version, and download it to EC2 or to your local machine if using Linux:

Bash

```bash
cd ~

wget https://storage.googleapis.com/golang/go1.6.2.linux-amd64

Extract the files using the following command:

tar -xvf go1.6.2.linux-amd64.tar.
```

This creates a folder named “go” in your home directory.

Time to code

For this example, you create a very simple application that counts the amount of objects in the provided JSON element. Using your favorite editor, create a file named “HelloLambda.go” with the following code directly on the machine to which you have downloaded the Go package, which may be the EC2 instance you started in the beginning or your local environment, in which case you are not stuck with vi.

Go

```go
package main

import (
    "fmt"
    "os"
    "encoding/json"
)
```

```
func main() {
    var dat map[string]interface{}

    fmt.Printf( "Welcome to Lambda Go, now Go Go Go!n" )
    if len( os.Args ) < 2 {
        fmt.Println( "Missing args" )
        return
    }

    err := json.Unmarshal([]byte(os.Args[1]), &dat)

    if err == nil {
        fmt.Println( len( dat ) )
    } else {
        fmt.Println(err)
    }
}
```

Before compiling, configure an environment variable to tell the Go compiler where all the files are located:

Bash

```bash
export GOROOT=~/go/

You are now set to compile a nifty new application!

~/go/bin/go build ./HelloLambda.go

Start your application for the very first time:

./HelloLambda '{ "we" : "love", "using" : "Lambda" }'
```

You should see output similar to:

Bash

```bash
Welcome to Lambda Go, now Go Go Go!

2
```

Creating the Lambda package

You have already set up your machine to compile Go applications, written the code, and compiled it successfully; all that is left is to package it up and deploy it to Lambda.

If you used an EC2 instance, copy the binary from the compilation instance and prepare it for packaging. To copy out the binary, use the following command from your local machine (Linux or OSX), or using tools such as WinSCP on Windows.

Bash

```bash
scp -i GoLambdaGo.pem ec2-user@ec2-00-00-00-00.eu-west-1.compu
```

With the binary ready, create the Lambda project in a new folder, which you can call go-lambda.

The next step is to create a Node.js wrapper file to invoke the Go application; call it go.js. The file takes the inputs of the Lambda invocations and invokes the Go binary.

Here's the content for another example of a Node.js wrapper:

JavaScript

```javascript
const exec = require('child_process').exec;
exports.handler = function(event, context) {
    const child = exec('./goLambdaGo ' + ''' + JSON.stringify(
        // Resolve with result of process
        context.done(error, 'Process complete!');
    });

    // Log process stdout and stderr
    child.stdout.on('data', console.log);
    child.stderr.on('data', console.error);

}
```

With all the files finalized and ready, your folder structure should look like the following:

go-lambda

— go.js

— goLambdaGo

The final step before deployment is to zip the package into an archive that can be uploaded to Lambda; call the package LambdaGo.zip.

On a Linux or OSX machine, run the following command:

Bash

```
zip -r go.zip ./goLambdaGo ./go.js
```

A gem in Lambda

For convenience, you can use the same previously used instance, but this time to compile Ruby for use with Lambda. You can also create a new instance using the same instructions.

Setting up Ruby on the instance

The next step is to set up the Ruby binaries and dependencies on the EC2 instance or local Linux environment, so that you can package the upcoming application.

First, make sure your packages are up to date (always):

Bash

```
sudo yum update -y
```

For this post, you use Traveling Ruby, a project that helps in creating “portable”, self-contained Ruby packages. You can download the latest version from Traveling Ruby linux-x86:

Bash

```
cd ~

wget http://d6r77u77i8pq3.cloudfront.net/releases/traveling-ru
```

Extract the files to a new folder using the following command:

Bash

```bash
mkdir LambdaRuby

tar -xvf traveling-ruby-20150715-2.2.2-linux-x86_64.tar.gz -C
```

This creates the “LambdaRuby” folder in your home directory.

Time to code

For this demonstration, you create a very simple application that counts the amount of objects in a provided JSON element. Using your favorite editor, create a file named “lambdaRuby.rb” with the following code:

Ruby

```ruby
#!./bin/ruby

require 'json'

# You can use this to check your Ruby version from within puts

if ARGV.length > 0
    puts JSON.parse( ARGV[0] ).length
else
    puts "0"
end
```

Now, start your application for the very first time, using the following command:

Bash

```bash
./lambdaRuby.rb '{ "we" : "love", "using" : "Lambda" }'
```

You should see the amount of fields in the JSON as output (2).

Creating the Lambda package

You have downloaded the Ruby gem, written the code, and tested it successfully… all that is left is to package it up and deploy it to Lambda. Because Ruby is an interpreter-based language, you create a Node.js wrapper and package it with the Ruby script and all the Ruby files.

The next step is to create a Node.js wrapper file to invoke your Ruby application; call it ruby.js. The file takes the inputs of the Lambda invocations and invoke your Ruby application. Here's the content for a sample Node.js wrapper:

JavaScript

```
const exec = require('child_process').exec;

exports.handler = function(event, context) {
    const child = exec('./lambdaRuby.rb ' + ''' + JSON.string:
        // Resolve with result of process
        context.done(result);
    });

    // Log process stdout and stderr
    child.stdout.on('data', console.log);
    child.stderr.on('data', console.error);
}
```

With all the files finalized and ready, your folder structure should look like this:

LambdaRuby

+– bin

+– bin.real

+– info

— lambdaRuby.rb

+– lib

— ruby.js

The final step before the deployment is to zip the package into an archive to be uploaded to Lambda. Call the package LambdaRuby.zip.

On a Linux or OSX machine, run the following command:

Bash

```
zip -r ruby.zip ./
```

Copy your zip file from the instance so you can upload it. To copy out the archive, use the following command from your local machine (Linux or OSX), or using tools such as WinSCP on Windows.

Bash

```
scp -i RubyLambda.pem ec2-user@ec2-00-00-00-00.eu-west-1.compu
```

Common steps to install

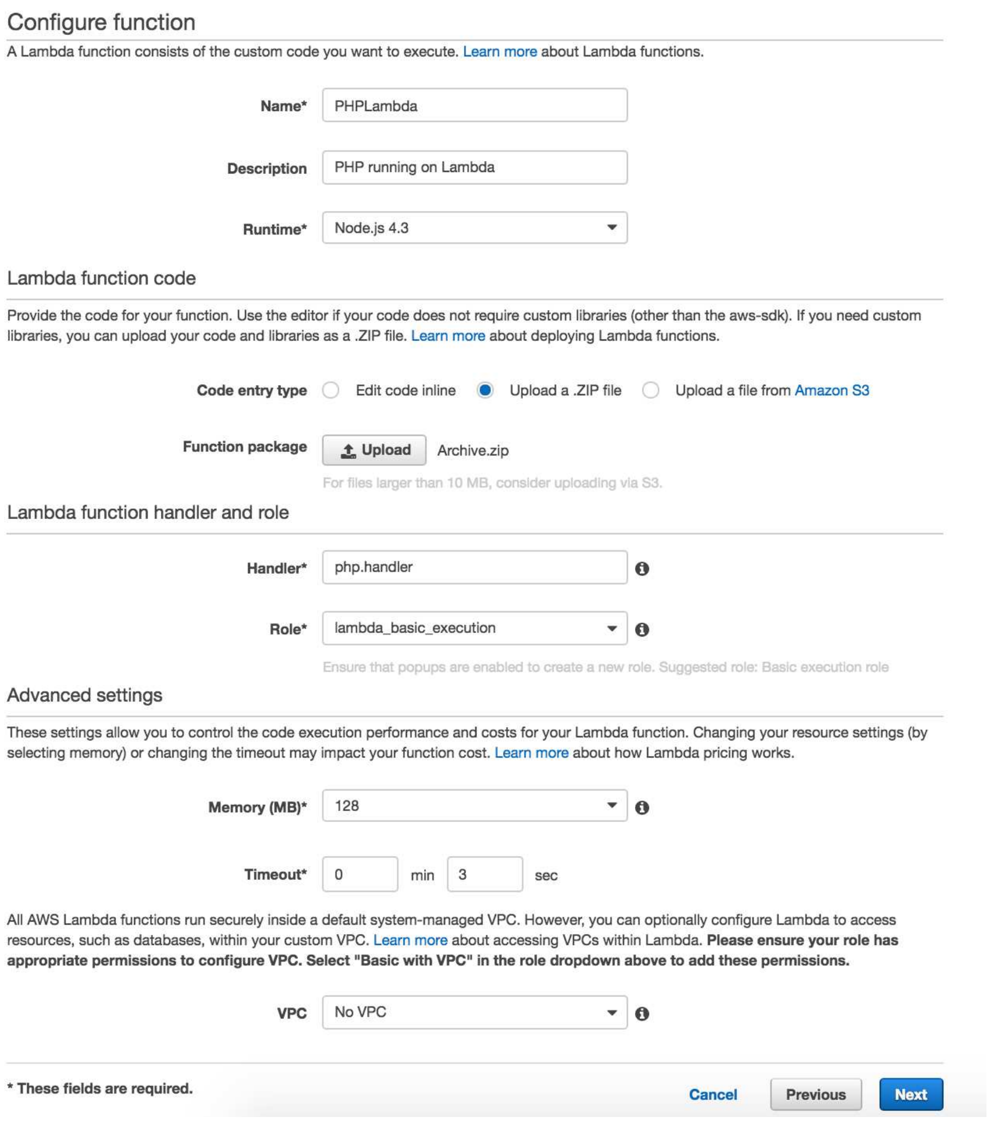
With the package done, you are ready to deploy the PHP, Go, or Ruby runtime into Lambda.

Log in to the AWS Management Console and navigate to Lambda; make sure that the region matches the one which you selected the AMI for in the preparation step.

For simplicity, I've used PHP as an example for the deployment; however, the steps below are the same for Go and Ruby.

Creating the Lambda function

Choose **Create a Lambda function**, **Skip**. Select the following fields and upload your previously created archive.


Configure function

A Lambda function consists of the custom code you want to execute. Learn more about Lambda functions.

Name*: PHPLambda

Description: PHP running on Lambda

Runtime*: Node.js 4.3

Lambda function code

Provide the code for your function. Use the editor if your code does not require custom libraries (other than the aws-sdk). If you need custom libraries, you can upload your code and libraries as a .ZIP file. Learn more about deploying Lambda functions.

Code entry type: ○ Edit code inline [✓ Upload a .ZIP file] ○ Upload a file from Amazon S3

Function package: Upload Archive.zip

For files larger than 10 MB, consider uploading via S3.

Lambda function handler and role

Handler*: php.handler

Role*: lambda_basic_execution

Ensure that popups are enabled to create a new role. Suggested role: Basic execution role

Advanced settings

These settings allow you to control the code execution performance and costs for your Lambda function. Changing your resource settings (by selecting memory) or changing the timeout may impact your function cost. Learn more about how Lambda pricing works.

Memory (MB)*: 128

Timeout*: 0 min 3 sec

All AWS Lambda functions run securely inside a default system-managed VPC. However, you can optionally configure Lambda to access resources, such as databases, within your custom VPC. Learn more about accessing VPCs within Lambda. **Please ensure your role has appropriate permissions to configure VPC. Select "Basic with VPC" in the role dropdown above to add these permissions.**

VPC: No VPC

*** These fields are required.**

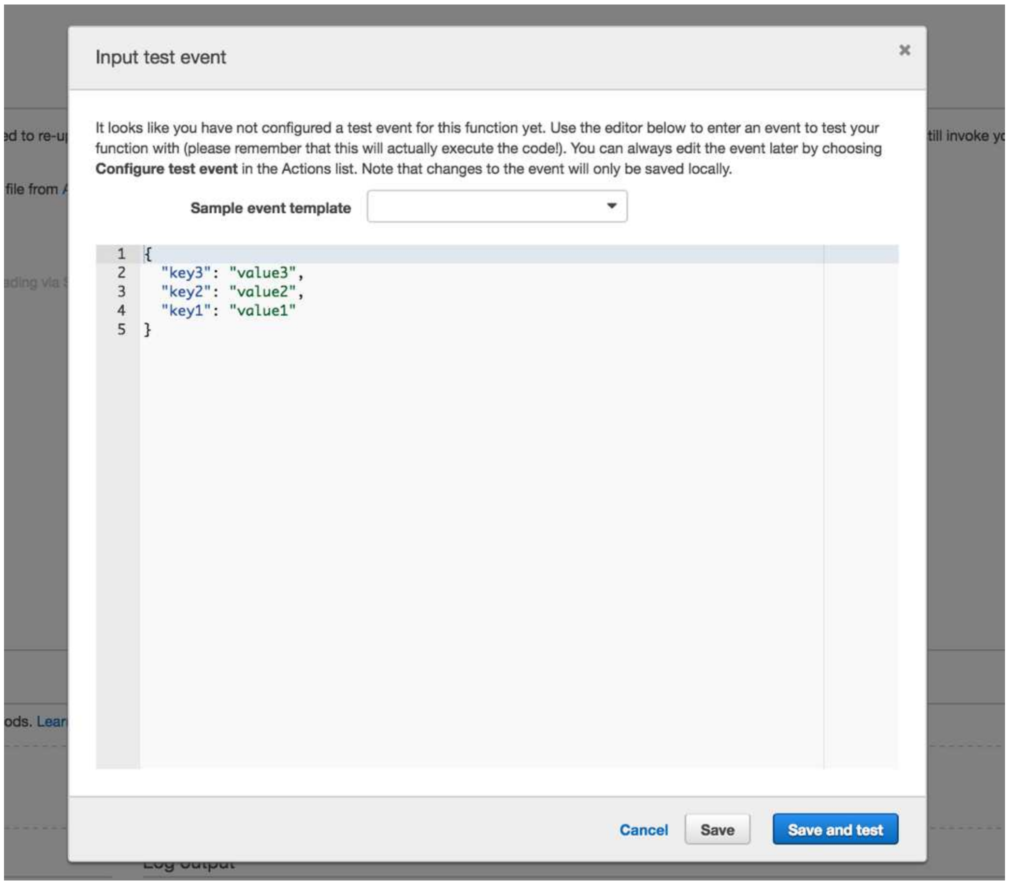
The most important areas are:

- **Name**: The name to give your Lambda function
- **Runtime**: Node.js
- **Lambda function code**: Select the zip file created in the PHP, Go, or Ruby section, such as php.zip, go.zip, or ruby.zip
- **Handler**: php.handler (as in the code, the entry function is called handler and the file is php.js. If you have used the file names from the Go and Ruby sections use the following format: [js file name without .js].handler, i.e., go.handler)
- **Role**: Choose **Basic Role** if you have not yet created one, and create a role for your Lambda function execution

Choose **Next**, **Create function** to continue to testing.

Testing the Lambda function

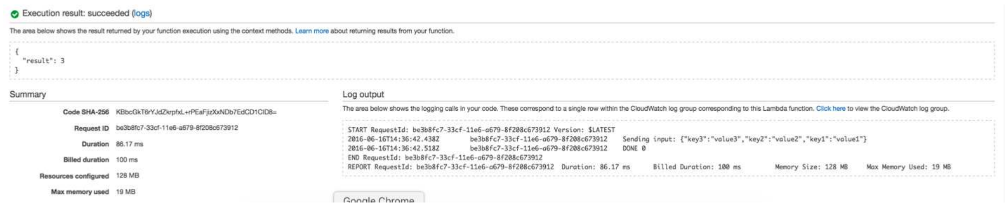
To test the Lambda function, choose **Test** in the upper right corner, which displays a sample event with three top-level attributes.



It looks like you have not configured a test event for this function yet. Use the editor below to enter an event to test your function with (please remember that this will actually execute the code!). You can always edit the event later by choosing **Configure test event** in the Actions list. Note that changes to the event will only be saved locally.

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

Feel free to add more, or simply choose **Save and test** to see that your function has executed properly.



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

```
{
  "result": 3
}
```

Code SHA-256	KBbcGkT6rYJdZkrpfxL+rPEaFjizXxNDb7EdCD1ClD8=
Request ID	be3b8fc7-33cf-11e6-a679-8f208c673912
Duration	86.17 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	19 MB

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

```
START RequestId: be3b8fc7-33cf-11e6-a679-8f208c673912 Version: $LATEST
2016-06-16T14:36:42.438Z        be3b8fc7-33cf-11e6-a679-8f208c673912    Sending input: {"key3":"value3","key2":"value2","key1":"value1"}
2016-06-16T14:36:42.518Z        be3b8fc7-33cf-11e6-a679-8f208c673912    DONE 0
END RequestId: be3b8fc7-33cf-11e6-a679-8f208c673912
REPORT RequestId: be3b8fc7-33cf-11e6-a679-8f208c673912  Duration: 86.17 ms      Billed Duration: 100 ms         Memory Size: 128 MB     Max Memory Used: 19 MB
```

Conclusion

In this post, we outlined three different ways to create scripting language runtimes for Lambda, from compiling against the Lambda runtime for PHP and being able to run scripts, compiling the actuals as in Go, or using packaged binaries as much as possible with Ruby. We hope you enjoyed the ideas, found the hidden gems, and are now ready to go to create some pretty hefty projects in your favorite language, enjoying serverless, Amazon Kinesis, and API Gateway along the way.

If you have questions or suggestions, please comment below.

TAGS: go, php, ruby

View Comments

Create a Free Account

AWS on Twitter AWS on Facebook AWS on Google+ AWS Blog What's New? RSS

AWS & Cloud Computing

What is Cloud Computing?
Products & Services
Customer Success
Economics Center
Architecture Center
Security Center
What's New
Whitepapers
AWS Blog
Events
Sustainable Energy
Press Releases
AWS in the News
Analyst Reports
Legal

Solutions

Websites & Website Hosting
Business Applications
Backup & Recovery
Disaster Recovery
Data Archive
DevOps
Big Data
High Performance Computing
Mobile Services
Digital Marketing
Game Development

Digital Media
Government & Education
Health
Financial Services
Windows on AWS

Resources & Training
Developers
Java on AWS
JavaScript on AWS
Mobile on AWS
PHP on AWS
Python on AWS
Ruby on AWS
Windows & .NET on AWS
SDKs & Tools
AWS Marketplace
User Groups
Support Plans
Service Health Dashboard
Discussion Forums
FAQs
Documentation
Articles & Tutorials
Test Drives
AWS Business Builder

Manage Your Account
Management Console
Billing & Cost Management
Subscribe to Updates
Personal Information
Payment Method
AWS Identity & Access Management
Security Credentials
Request Service Limit Increases
Contact Us

Language Deutsch | English | Español | Français | Italiano | Português | Русский | 日本語 | 한국어 | 中文 (简体) | 中文 (繁體)

Site Terms | Privacy