JAVAWORLD

OPEN SOURCE JAVA TUTORIALS

By Steven Haines, Contributor, JavaWorld

JAN 11, 2018 10:26 AM PST

About

A working developer's guide to open source tools and frameworks for Java application development.

UPDATED

Serverless computing with AWS Lambda, Part 1

Get an overview of AWS Lambda's nanoservices architecture and execution model, then build your first Lambda function in Java

Serverless computing may be the hottest thing in cloud computing today, but what, exactly, is it? This two-part tutorial starts with an overview of serverless computing--from what it is, to why it's considered disruptive to traditional cloud computing, and how you might use it in Java-based programming.

Following the overview, you'll get a hands-on introduction to AWS Lambda, which is considered by many the premiere Java-based solution for serverless computing today. In Part 1, you'll use AWS Lambda to build, deploy, and test your first Lambda function in Java. In Part 2, you'll integrate your Lambda function with DynamoDB, then use the AWS SDK to invoke Lambda functions in a Java application.

What is serverless computing?

Last year I was talking to a company intern about different architectural patterns and mentioned serverless architecture. He was quick to note that all applications require a server, and cannot run on thin air. The intern had a point, even if he was missing mine. Serverless computing is not a magical platform for running applications.

In fact, *serverless computing* simply means that you, the developer, do not have to *deal with* the server. A serverless computing platform like AWS Lambda allows you to build your code and deploy it without ever needing to configure or manage underlying servers. Your unit of deployment is your code; not the container that hosts the code, or the server that runs the code, but simply the code itself. From a productivity standpoint, there are obvious benefits to offloading the details of where code is stored and how the execution environment is managed. Serverless computing is also priced based on execution metrics, so there is a financial advantage, as well.

What does AWS Lambda cost?

At the time of this writing, AWS Lambda's price tier is based on number of executions and execution duration:

- Your first million executions per month are free, then you pay $0.20 per million executions thereafter ($0.0000002 per request).
- Duration is computed from the time your code starts executing until it returns a result, rounded to the nearest 100ms. The amount charged is based on the amount of RAM allocated to the function, where the cost is $0.00001667 for every GB-second.

Pricing details and free tier allocations are slightly more complicated than the overview implies. Visit the price tier to walk through a few pricing scenarios.

To get an idea for how serverless computing works, let's start with the serverless computing execution model, which is illustrated in Figure 1.

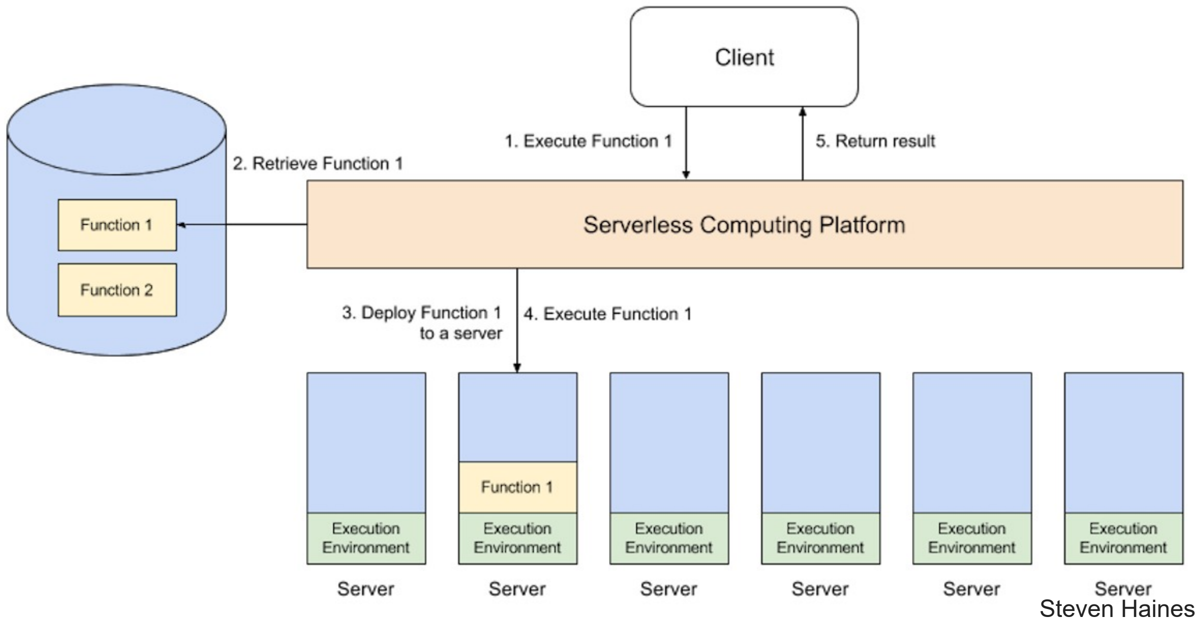

Figure 1. Serverless computing execution model

Here's the serverless execution model in a nutshell:

1. A client makes a request to the serverless computing platform to execute a specific function.

2. The serverless computing platform first checks to see if the function is running on any of its servers. If the function isn't already running, then the platform loads the function from a data store.
3. The platform then deploys the function to one of its servers, which are preconfigured with an execution environment that can run the function.
4. It executes the function and captures the result.
5. It returns the result back to the client.

Sometimes serverless computing is called Function as a Service (FaaS), because the granularity of the code that you build is a *function*. The platform executes your function on its own server and orchestrates the process between function requests and function responses.

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

Nanoservices, scalability, and price

Three things really matter about serverless computing: its nanoservice architecture; the fact that it's practically infinitely scalable; and the pricing model associated with that near infinite scalability. We'll dig into each of those factors.

Nanoservices

You've heard of microservices, and you probably know about 12-factor applications, but serverless functions take the paradigm of breaking a component down to its constituent parts to a whole new level. The term "nanoservices" is not an industry recognized term, but the idea is simple: each nanoservice should implement a single action or responsibility. For example, if you wanted to create a widget, the act of creation would be its own nanoservice; if you wanted to retrieve a widget, the act of retrieval would also be a nanoservice; and if you wanted to place an order for a widget, that order would be yet another nanoservice.

A nanoservices architecture allows you to define your application at a very fine-grained level. Similar to test-driven development (which helps you avoid unwanted side-effects by writing your code at the level of individual tests), a nanoservices architecture encourages defining your application in terms of very fine-grained and specific functions. This approach increases clarity about what you're building and reduces unwanted side-effects from new code.

Microservices vs nanoservices

Microservices encourages us to break an application down into a collection of services that each accomplish a specific task. The challenge is that no one has really quantified the *scope* of a microservice. As a result, we end up defining microservices as a collection of related services, all interacting with the same data model. Conceptually, if you have low-level functionality interacting with a given data model, then the functionality should go into one of its related services. High-level interactions should make calls to the service rather than querying the database directly.

There is an ongoing debate in serverless computing about whether to build Lambda functions at the level of microservices or nanoservices. The good news is that you can pretty easily build your functions at either granularity, but a microservices strategy will require a bit of extra routing logic in your request handler.

From a design perspective, serverless applications should be very well-defined and clean. From a deployment perspective you will need to manage significantly more deployments, but you will also have the ability to deploy new versions of your functions individually, without impacting other functions. Serverless computing is especially well suited to development in large teams, where it can help make the development process easier and the code less error-prone.

Scalability

In addition to introducing a new architectural paradigm, serverless computing platforms provide practically infinite scalability. I say "practically" because there is no such thing as *truly* infinite scalability. For all practical purposes, however, serverless computing providers like Amazon can handle more load than you could possibly throw at them. If you were to manage scaling up your own servers (or cloud-based virtual machines) to meet increased demand, you would need to monitor usage, identify when to start more servers, and add more servers to your cluster at the right time. Likewise, when demand decreased you would need to manually scale down. With serverless computing, you tell your serverless computing platform the maximum number of simultaneous function requests you want to run and the platform does the scaling for you.

Pricing

Finally, the serverless computing pricing model allows you to scale your cloud bill based on usage. When you have light usage, your bill will be low (or nil if you stay in the free range). Of course, your bill will increase with usage, but hopefully you will also have new revenue to support your higher cloud bill. For contrast, if you were to manage your own servers, you would have to pay a base cost to run the minimum number of servers required. As usage increased, you would scale up in increments of entire servers, rather than increments of individual function calls. The serverless computing pricing model is directly proportional to your usage.

AWS Lambda for serverless computing

AWS Lambda is a serverless computing platform implemented on top of Amazon Web Services platforms like EC2 and S3. AWS Lambda encrypts and stores your code in S3. When a function is requested to run, it creates a "container" using your runtime specifications, deploys it to one of the EC2 instances in its compute farm, and executes that function. The process is shown in Figure 2.

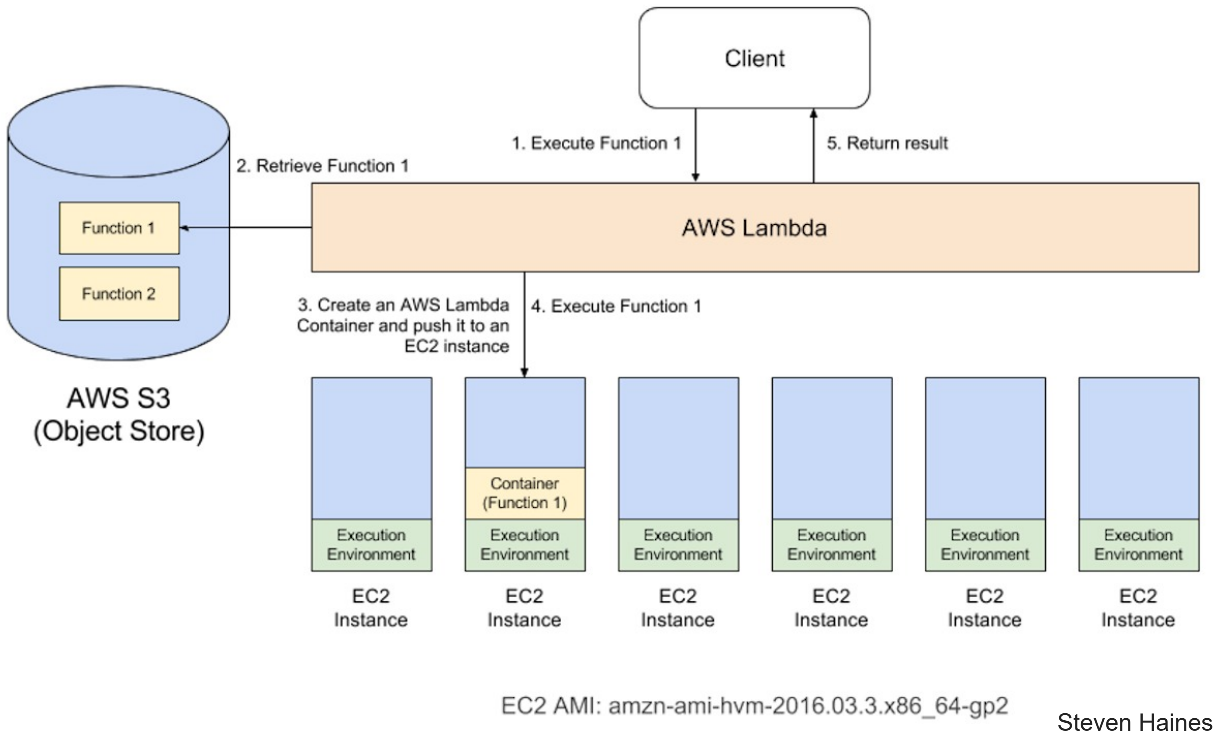

Figure 2. Execution process in AWS Lambda

When you create a Lambda function, you configure it in AWS Lambda, specifying things like the runtime environment (we'll use Java 8 for this article), how much memory to allocate to it, identity and access management roles, and the method to execute. AWS Lambda uses your configuration to setup a container and deploy the container to an EC2 instance. It then executes the method that you've specified, in the order of package, class, and method.

At the time of this writing, you can build Lambda functions in Node, Java, Python, and most recently, C#. For the purposes of this article we will use Java.

What is a Lambda function?

When you write code designed to run in AWS Lambda, you are writing *functions*. The term *functions* comes from functional programming, which originated in lambda calculus. The basic idea is to compose an application as a collection of functions, which are methods that accept arguments, compute a result, and have no unwanted side-effects. Functional programming takes a mathematical approach to writing code that can be proven to be correct. While it's good to keep functional programming in mind when you are writing code for AWS Lambda, all you really need to understand is that the function is a single-method entry-point that accepts an input object and returns an output object.

Serverless execution modes

While Lambda functions can run synchronously, as described above, they can also run asynchronously and in response to events. For example, you could configure a Lambda to run whenever a file was uploaded to an S3 bucket. This configuration is sometimes used for image or video processing: when a new image is uploaded to an S3 bucket, a Lambda function is invoked with a reference to the image to process it.

I worked with a very large company that leveraged this solution for photographers covering a marathon. The photographers were on the course taking photographs. Once their memory cards were full, they loaded the images onto a laptop and uploaded the files to S3. As images were uploaded, Lambda functions were executed to resize, watermark, and add a reference for each image to its runner in the database.

All of this would take a lot of work to accomplish manually, but in this case the work not only processed faster because of AWS Lambda's horizontal scalability, but also seamlessly scaled up and back down, thus optimizing the company's cloud bill.

In addition to responding to files uploaded to S3, lambdas can be triggered by other sources, such as records being inserted into a DynamoDB database and analytic information streaming from Amazon Kinesis. We'll look at an example featuring DynamoDB in Part 2.

AWS Lambda functions in Java

Now that you know a little bit about serverless computing and AWS Lambda, I'lll walk you through building an AWS Lambda function in Java.

Get the code

Source code for the example application for this tutorial, "Serverless computing with AWS Lambda." *Created by Steven Haines for JavaWorld.*

Implementing Lambda functions

You can write a Lambda function in one of two ways:

- The function can receive an input stream to the client and write to an output stream back to the client.
- The function can use a predefined interface, in which case AWS Lambda will automatically deserialize the input stream to an object, pass it to your function, and serialize your function's response before returning it to the client.

The easiest way to implement an AWS Lambda function is to use a predefined interface. For Java, you first need to include the following AWS Lambda core library in your project (note that this example uses Maven):

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

Next, have your class implement the following interface:

Listing 1. RequestHandler.java

```

public interface RequestHandler<I, O> {
    /**
     * Handles a Lambda function request
     * @param input The Lambda function input
     * @param context The Lambda execution environment context object.
     * @return The Lambda function output
     */
    public O handleRequest(I input, Context context);
}
```

The RequestHandler interface defines a single method: handleRequest(), which is passed an input object and a Context object, and returns an output object. For example, if you were to define a Request class and a Response class, you could implement your lambda as follows:

```
public class MyHandler implements RequestHandler<Request, Response> {
  public Response handleRequest(Request request, Context context) {
    ...
  }
}
```

Alternatively, if you wanted to bypass the predefined interface, you could manually handle the InputStream and OutputStream yourself, by implementing a method with the following signature:

```
public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context)
      ...
}
```

Page 1 of 3