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

Page 2 of 3

The `Context` object provides information about your function and the environment in which it is running, such as the function name, its memory limit, its logger, and the amount of time remaining, in milliseconds, that the function has to complete before AWS Lambda kills it.

With that overview out of the way, we'll spend the remainder of Part 1 building a simple Lambda function that returns a widget. For the purposes of our example application, the user requests a widget by ID (a `String`), which we'll wrap into a `WidgetRequest` object. The function will then return a `Widget` object as its response.

Building a Lambda function

Listing 2 shows the source code for `Widget`, which is a POJO with an `id` and `name`:

Listing 2. Widget.java

```
package com.javaworld.awslambda.widget.model;
public class Widget {
    private String id;
    private String name;
    public Widget() {
    }
    public Widget(String id, String name) {
        this.id = id;
        this.name = name;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Listing 3 shows the source code for a `WidgetRequest`, which is a POJO that contains an `id`:

Listing 3. WidgetRequest.java

```
package com.javaworld.awslambda.widget.model;
public class WidgetRequest {
    private String id;
    public WidgetRequest() {
    }
    public WidgetRequest(String id) {
        this.id = id;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
}
```

Next, we'll build our lambda request handler. Listing 4 shows the source code for the `GetWidgetHandler` class:

Listing 4. GetWidgetHandler.java

```java
package com.javaworld.awslambda.widget.handlers;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.javaworld.geekcap.awslambda.widget.model.Widget;
import com.javaworld.geekcap.awslambda.widget.model.WidgetRequest;
public class GetWidgetHandler implements RequestHandler<WidgetRequest, Widget> {
    @Override
    public Widget handleRequest(WidgetRequest widgetRequest, Context context) {
        return new Widget(widgetRequest.getId(), "My Widget " + widgetRequest.getId());
    }
}
```

The `GetWidgetHandler` class implements the `RequestHandler` interface, accepting a `WidgetRequest` and returning a `Widget`. The `WidgetRequest` includes an `id` parameter to indicate which `Widget` is being requested. For the purposes of this example, however, we won't load the `Widget` from a database. Instead, we'll build and return a new `Widget` instance with the specified ID and the name "My Widget ID." We'll do this on the fly.

Listing 5 shows the contents of the Maven POM file that can build a JAR file containing our Lambda function.

Listing 5. pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.javaworld.geekcap</groupId>
    <artifactId>aws-lambda-java</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>aws-lambda-java</name>
    <url>http://maven.apache.org</url>
    <properties>
        <java.version>1.8</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.1.0</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.3</version>
                <configuration>
                    <createDependencyReducedPom>false</createDependencyReducedPom>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

The POM file includes the `aws-lambda-java-core` dependency and builds against Java 8. The `maven-shade-plugin` packages all of our dependent JAR files inside of our JAR file so that it can run standalone. (See the AWS Lambda documentation to learn more about creating a standalone JAR file with `maven-shade-plugin` and using it with AWS Lambda.)

To run this build, execute the following command:

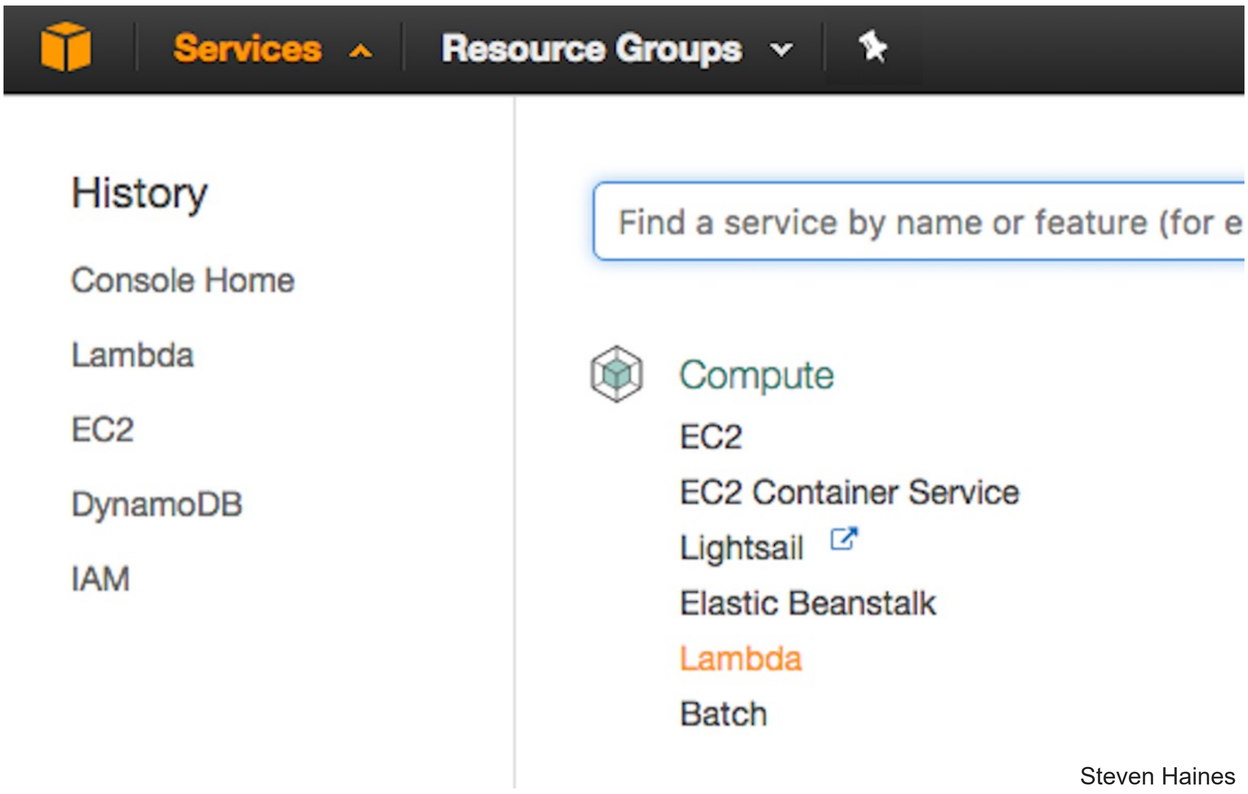
```
mvn clean install
```

The command creates a file named `aws-lambda-java-1.0-SNAPSHOT.jar` in your `target` directory. We'll upload this file to AWS Lambda in the next section.

Creating a Lambda function in the AWS console

Now that we have a Lambda function written and packaged into a standalone JAR file, let's set it up in AWS Lambda. Before proceeding with this step, you'll need to setup a free AWS account. Go ahead and do that now.

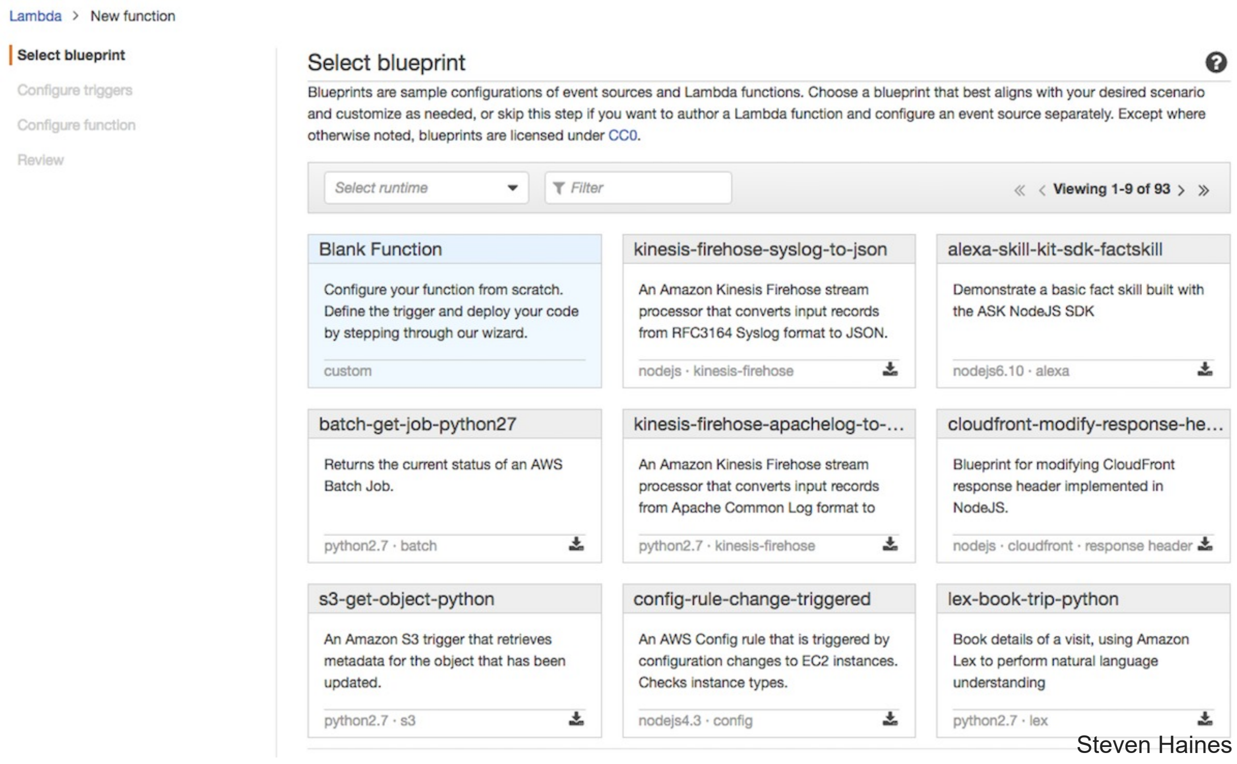
Once you have your account, login and open the AWS console. Click on Services, then choose Lambda in the Compute section, as shown in Figure 3.



Steven Haines

Figure 3. Accessing the Lambda page in AWS Lambda

Click the **Create a Lambda function** button and select Blank Function, as shown in Figure 4.



Lambda > New function

Select blueprint
Configure triggers
Configure function
Review

Select blueprint

Blueprints are sample configurations of event sources and Lambda functions. Choose a blueprint that best aligns with your desired scenario and customize as needed, or skip this step if you want to author a Lambda function and configure an event source separately. Except where otherwise noted, blueprints are licensed under CC0.

Select runtime | Filter | Viewing 1-9 of 93

Blank Function Configure your function from scratch. Define the trigger and deploy your code by stepping through our wizard. custom	**kinesis-firehose-syslog-to-json** An Amazon Kinesis Firehose stream processor that converts input records from RFC3164 Syslog format to JSON. nodejs · kinesis-firehose	**alexa-skill-kit-sdk-factskill** Demonstrate a basic fact skill built with the ASK NodeJS SDK nodejs6.10 · alexa
batch-get-job-python27 Returns the current status of an AWS Batch Job. python2.7 · batch	**kinesis-firehose-apachelog-to-…** An Amazon Kinesis Firehose stream processor that converts input records from Apache Common Log format to python2.7 · kinesis-firehose	**cloudfront-modify-response-he…** Blueprint for modifying CloudFront response header implemented in NodeJS. nodejs · cloudfront · response header
s3-get-object-python An Amazon S3 trigger that retrieves metadata for the object that has been updated. python2.7 · s3	**config-rule-change-triggered** An AWS Config rule that is triggered by configuration changes to EC2 instances. Checks instance types. nodejs4.3 · config	**lex-book-trip-python** Book details of a visit, using Amazon Lex to perform natural language understanding python2.7 · lex

Steven Haines

Figure 4. Creating a blank function

Configuring triggers

At the time of this writing, there are no blueprints for creating Java Lambda functions, so you'll need to start the function from scratch. The first screen that you will see after selecting a blank function is the Configure Triggers page, shown in Figure 5.

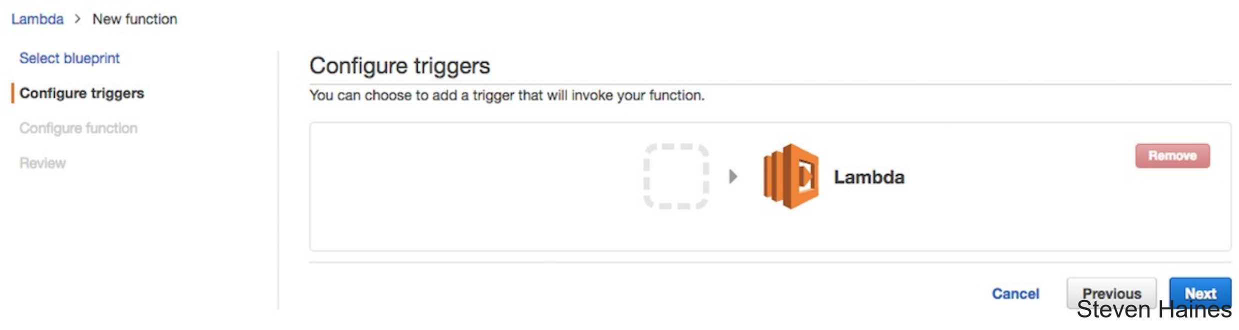


Figure 5. Configuring triggers

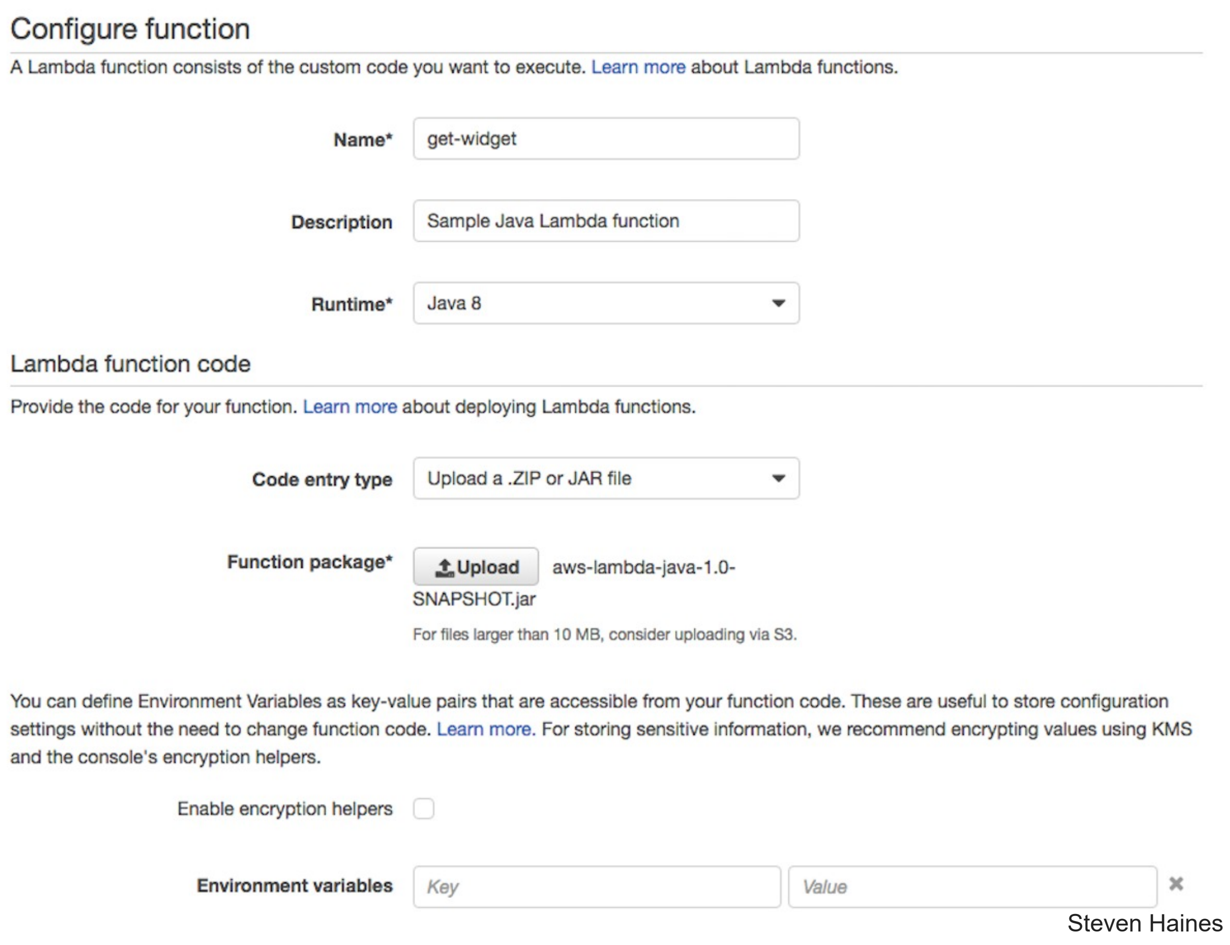
You will setup *triggers* to instruct AWS to call your Lambda function when something occurs. If you click on the rounded rectangle on the left side of this page, you'll see the types of things that can trigger Lambda functions, including:

- API Gateway: Allows a call to a RESTful resource to be forwarded to your function.
- AWS IoT: Calls your function in response to an IoT (Internet of Things) events.
- Alexa Skills Kit: Allows you to create Alexa voice-activate skills.
- Alexa Smart Home: Handles Alexa Smart Home device events.
- CloudFront: Calls your function in response to CloudFront events, allowing you to customize content delivered by CloudFront (AWS's CDN solution).
- CloudWatch Events: Calls your function when a CloudWatch event or alert occurs.
- CloudWatch Logs: Calls your function when specific messages are logged to CloudWatch.
- CodeCommit: Calls your function when code is committed to CodeCommit, which is AWS's CI solution.
- DynamoDB: Calls your function when data is inserted into a DynamoDB table.
- Kinesis: Calls your function in response to analytics events.
- S3: Calls your function when a file is uploaded to S3.
- SNS: Calls your function when a notification is published to the Simple Notification Service.

We'll leave this blank for now, because we're going to call our Lambda function directly.

Configuring your function

Click **Next** and you'll see the Configure Function screen shown in Figure 6.


Configure function

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

Name*: get-widget

Description: Sample Java Lambda function

Runtime*: Java 8

Lambda function code

Provide the code for your function. Learn more about deploying Lambda functions.

Code entry type: Upload a .ZIP or JAR file

Function package*: Upload aws-lambda-java-1.0-SNAPSHOT.jar

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

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

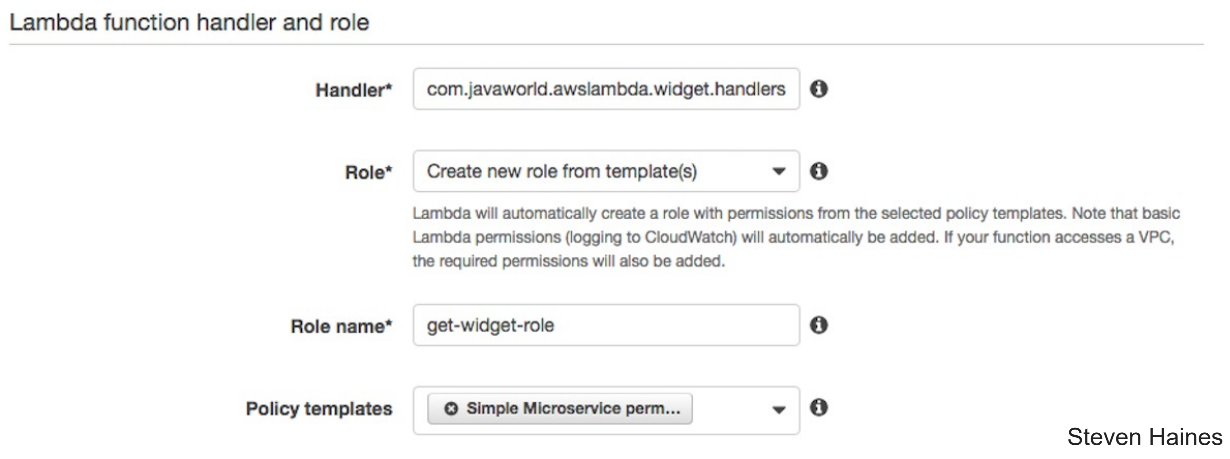
Enable encryption helpers

Environment variables: Key Value

Steven Haines

Figure 6. Configuring a function

Give your function a name (in this case it's "get-widget") and a description. Choose Java 8 as your runtime, which will add a dropdown that allows you to choose a JAR file to upload. Click the **Upload** button and find the JAR file that you built earlier in this tutorial. Below this, you'll see the page to configure your Lambda function handler and role, as shown in Figure 7.


Lambda function handler and role

Handler*: com.javaworld.awslambda.widget.handlers

Role*: Create new role from template(s)

Lambda will automatically create a role with permissions from the selected policy templates. Note that basic Lambda permissions (logging to CloudWatch) will automatically be added. If your function accesses a VPC, the required permissions will also be added.

Role name*: get-widget-role

Policy templates: Simple Microservice perm...

Steven Haines

Figure 7. Configuring the Lambda function handler and role

Fist, you'll configure the handler. The handler's format is as follows:

```
package.Class::handlerMethod
```

For our example, the handler is:

```
com.javaworld.awslambda.widget.handlers.GetWidgetHandler::handleRequest
```

Next, you'll specify a role for the function. Before doing that, let's make sure you understand roles in AWS Lambda.

Configuring roles in AWS Lambda

Roles define policies that grant the executor (in this case a Lambda function) permission to interact with AWS Lambda and other AWS resources. For example, if your Lambda function was going to query a DynamoDB instance, then it would need access permissions for `dynamodb:Scan` and probably `dynamodb:GetItem`. If your Lambda was going to add an object to S3, it would need access permission to `s3:PutObject` on your S3 resource.

The important thing about roles is that they grant or deny permissions for your Lambda function to interact with other AWS resources. Roles have nothing to do with application users or your application's internal configuration. Rather, they define how your application can or cannot interact with AWS resources.

For this example, we don't need any special permissions because our Lambda function doesn't actually do anything. We do need to be able to access DynamoDB for the examples in Part 2, however. To setup that permission, we'll start by creating a new role from a template. We'll name it get-widget-role and add the policy template "Simple Microservice Permissions." The Simple Microservice Permissions role is part of an existing Lambda blueprint, and provides full access to DynamoDB. Its contents are shown in Listing 6.

Listing 6. Simple Microservices Permissions

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:YOUR_ACCOUNT_NUMBER:table/*"
        }
    ]
}
```

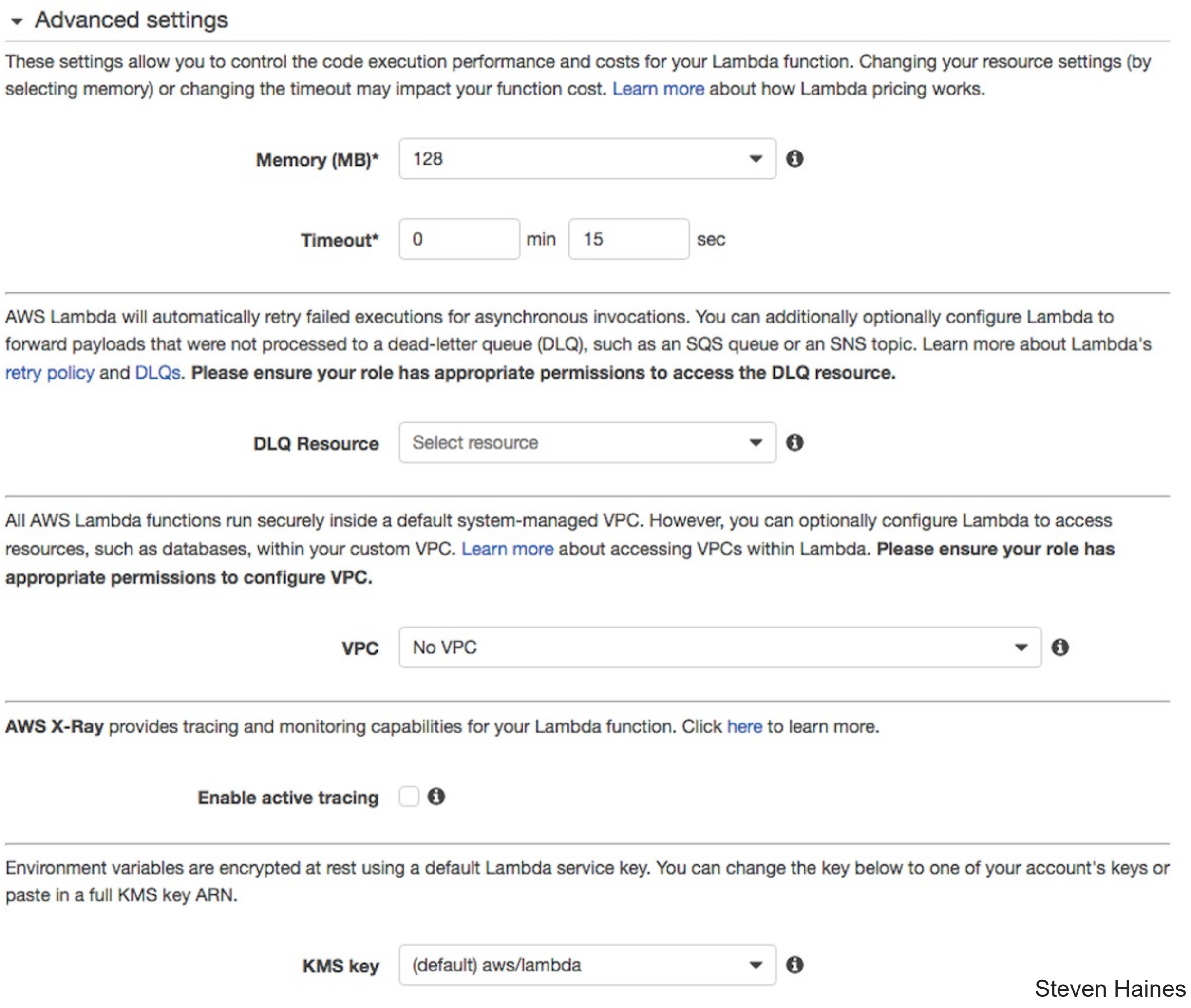
This policy reads as follows:

> *Allow the specified actions (DeleteItem, GetItem, etc) on resource `arn:aws:dynamodb:us-east-1:YOUR_ACCOUNT_NUMBER:table/*`, where `YOUR_ACCOUNT_NUMBER` will be your account number and `table/*` means all of your tables in DynomDB.*

If you wanted to refine the policy further, you could specify a single table name instead of all tables; or you could change the list of allowable actions. For example, a function that simply retrieves an item from DynamoDB does not need permissions to delete, put, or update items, so you might remove those.

Configuring Lambda's advanced settings

Finally, expand the Advanced Settings section, which is shown in Figure 8.



▾ 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 15 sec

AWS Lambda will automatically retry failed executions for asynchronous invocations. You can additionally optionally configure Lambda to forward payloads that were not processed to a dead-letter queue (DLQ), such as an SQS queue or an SNS topic. Learn more about Lambda's retry policy and DLQs. **Please ensure your role has appropriate permissions to access the DLQ resource.**

DLQ Resource Select resource

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.**

VPC No VPC

AWS X-Ray provides tracing and monitoring capabilities for your Lambda function. Click here to learn more.

Enable active tracing ☐

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

KMS key (default) aws/lambda

Steven Haines

Figure 8. Advanced Settings for Lambda function

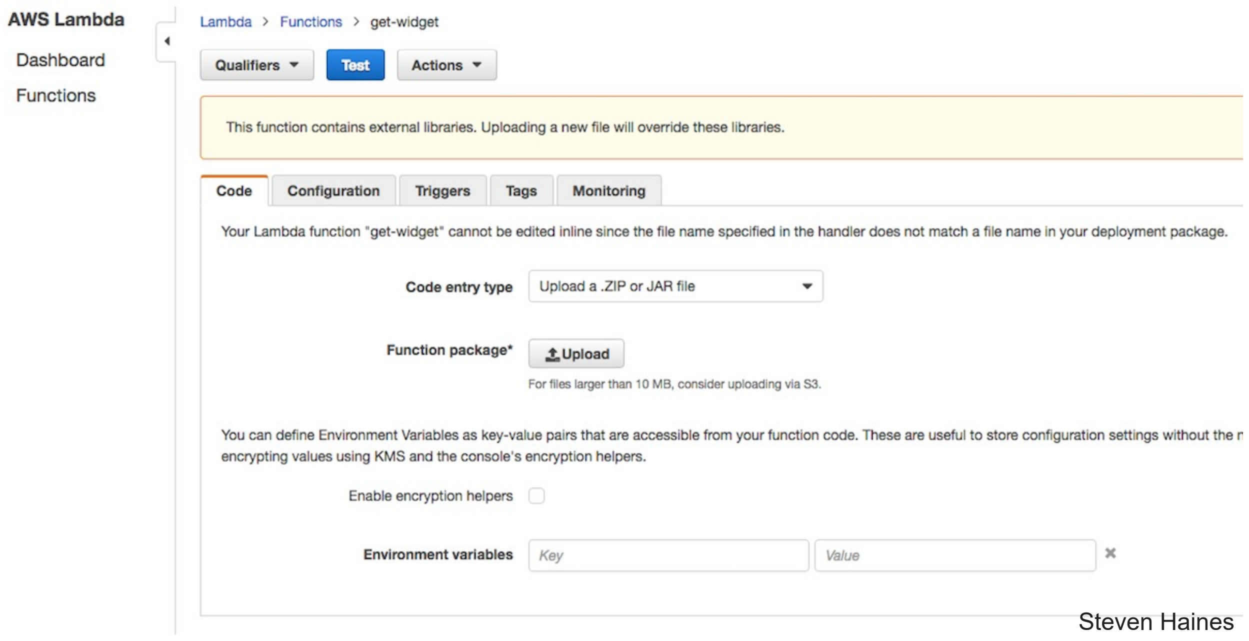
Once again, we can leave most of this alone. Because we pay for Lambda executions both by execution time and by the amount of RAM that we use, we can reduce the memory requirement from 512MB to 128MB. You may be thinking that 128MB is not a lot of memory for a Java application, but recall the Lambda functions are small and simple. A function is not like a servlet container that starts a web application and leaves it running for hours or days on end, so for most purposes 128MB is just fine.

Another important thing to notice on this screen is the timeout. Because Lambda functions are meant to be short-lived and you pay for execution time, AWS Lambda allows you to specify a timeout. If your function does not complete within the allotted time, it will be killed. This protects your cloud bill in case things go wrong.

When you're ready, press **Next**. Review the summary of your function, then press **Create Function**.

Testing your AWS Lambda function

Now that you have your function deployed in AWS Lambda, let's use the AWS console to test it out. After creating your function, you should have been dropped off on your function's page (**Lambda > Functions > get-widget**). If you need to, you can always navigate through Services to Lambda, then click on **Functions** and choose your function name (in this case, get-widget). Either way, you'll land on the page shown in Figure 9.



This function contains external libraries. Uploading a new file will override these libraries.

Your Lambda function "get-widget" cannot be edited inline since the file name specified in the handler does not match a file name in your deployment package.

Code entry type: Upload a .ZIP or JAR file

Function package*: Upload

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

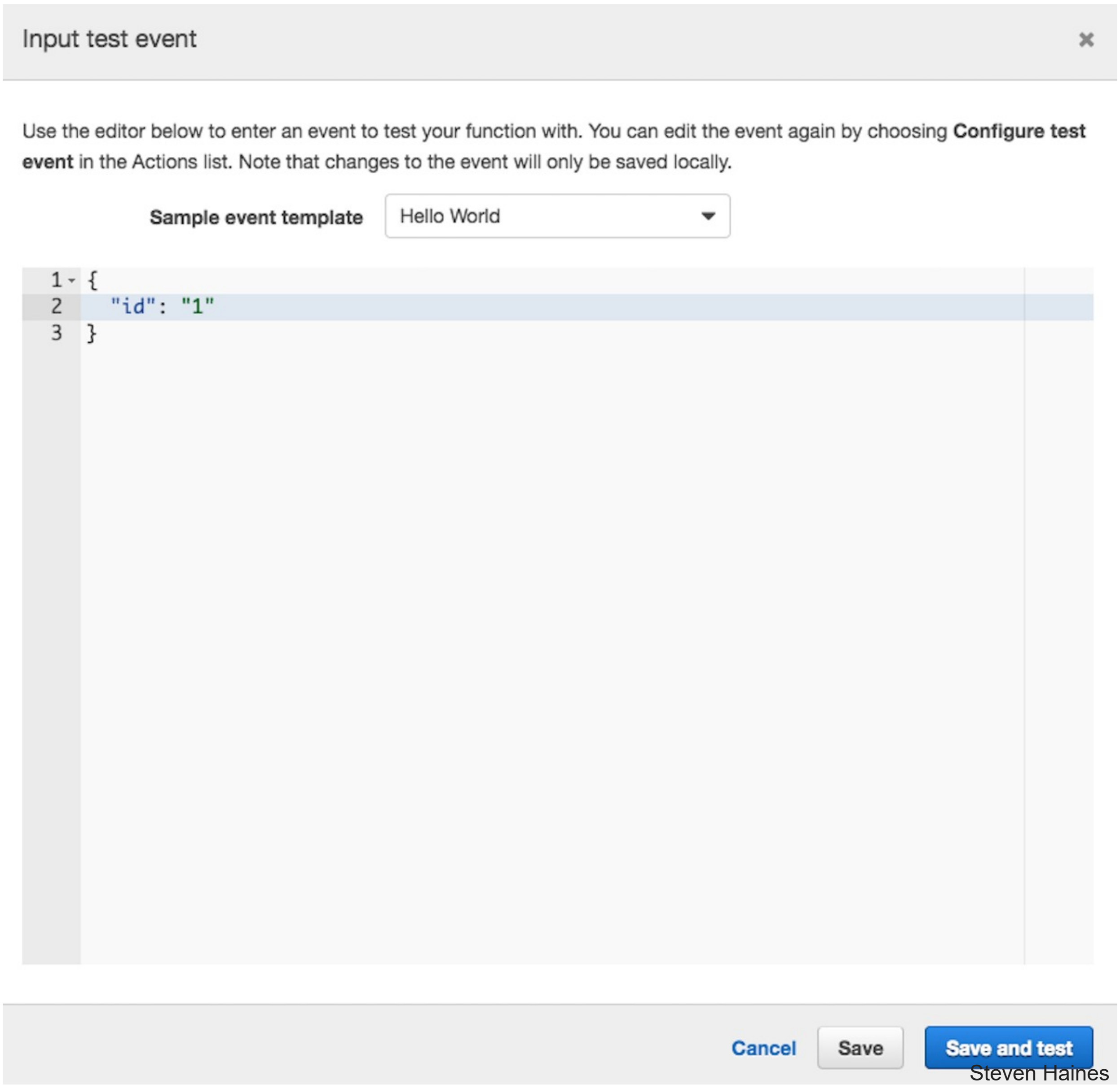
You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the r
encrypting values using KMS and the console's encryption helpers.

Enable encryption helpers

Environment variables

Figure 9. Lambda function page

To test your function, click the blue **Test** button and you will be prompted to define a new test event. Recall that the Lambda function accepts a `WidgetRequest`, which has a single `id` parameter. Because you implemented the `RequestHandler` interface, AWS Lambda will deserialize the JSON that you pass in your test event into a `WidgetRequest` object. All of this is to say that to test the function you only need to specify an `id` field, as shown in Figure 10.



Use the editor below to enter an event to test your function with. You can edit the event again by choosing **Configure test event** in the Actions list. Note that changes to the event will only be saved locally.

```
{
  "id": "1"
}
```

Figure 10. Lambda test event

Page 2 of 3