JAVAWORLD

/A TUTORIALS

:or, JavaWorld

About

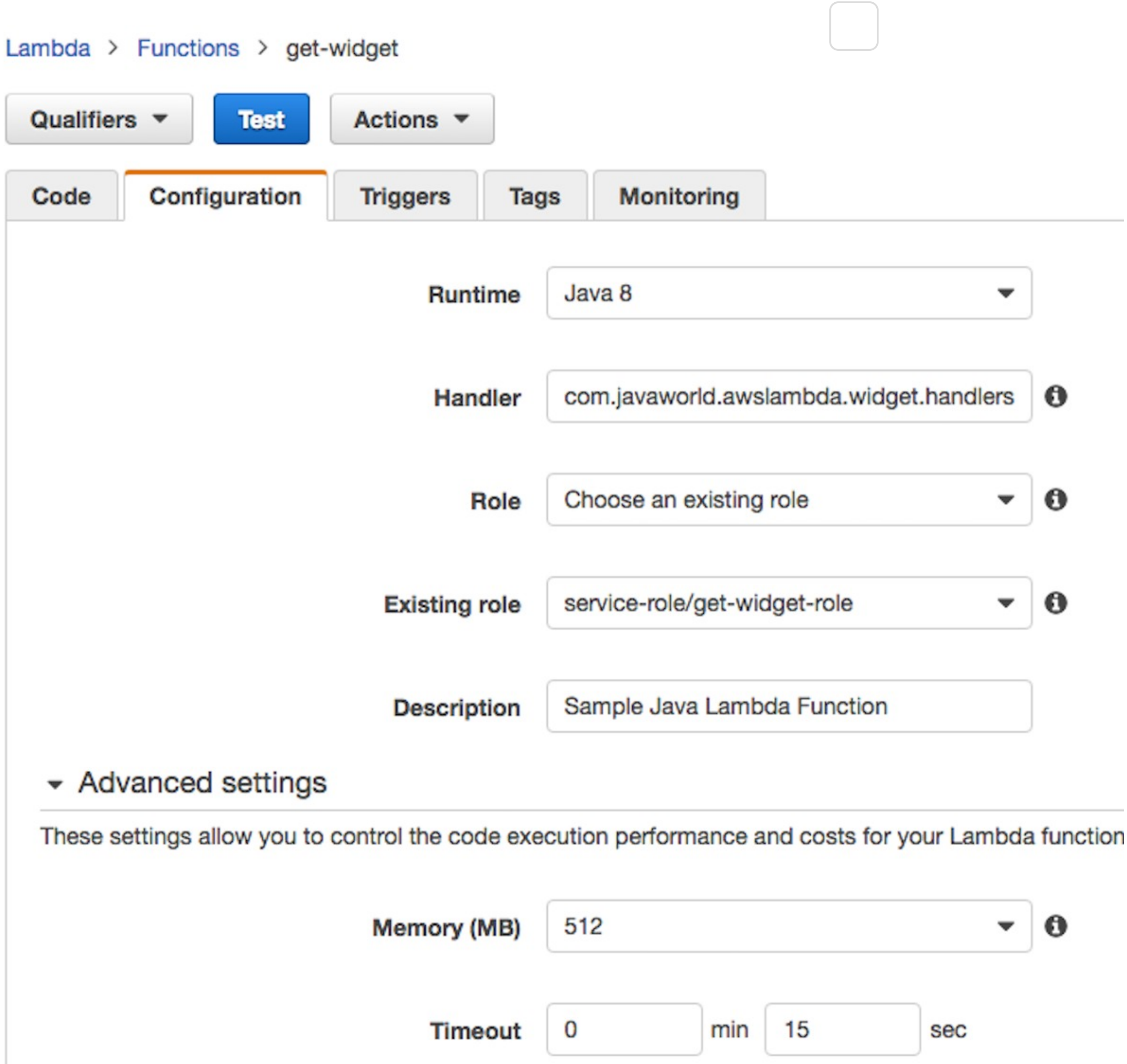
A working developer's guide to open source tools and frameworks for Java application development.

HOW-TO

Serverless computing with AWS Lambda, Part 2

Integrate AWS Lambda with DynamoDB, then call Lambda functions from a Java client

Page 2 of 2



Runtime: Java 8

Handler: com.javaworld.awslambda.widget.handlers

Role: Choose an existing role

Existing role: service-role/get-widget-role

Description: Sample Java Lambda Function

Advanced settings

These settings allow you to control the code execution performance and costs for your Lambda function

Memory (MB): 512

Timeout: 0 min 15 sec

Steven Haines

Figure 6. Changing memory settings

Calling AWS Lambda functions from a Java application

Now that we have a function running in AWS Lambda, we'll write a client application in Java that can call it. In order to execute a Lambda function from a Java application that isn't d to create an IAM user with permissions to invoke the Lambda ose credentials from our client application.

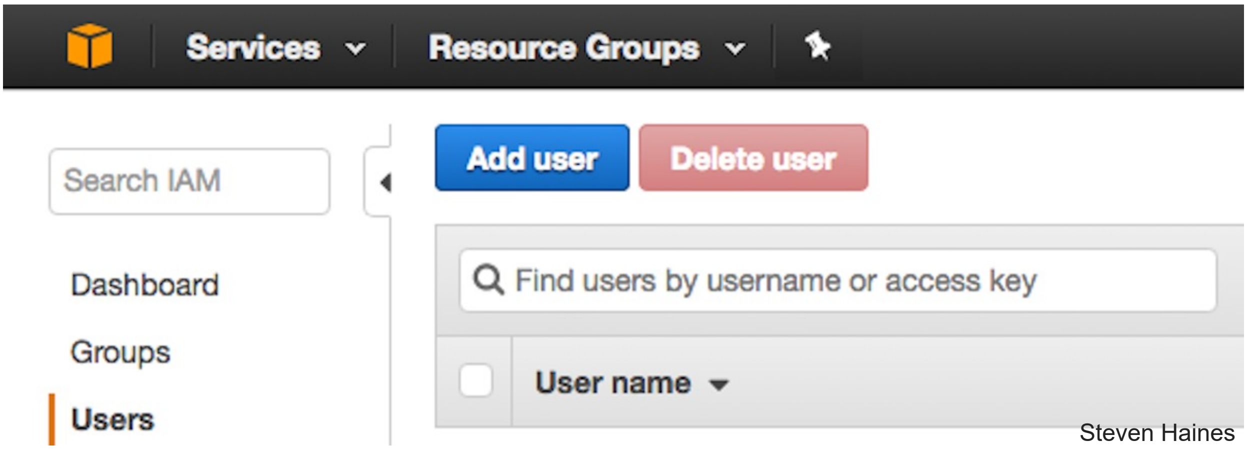
All of this is easy conceptually, but there are a lot of steps in the AWS console. I'll walk you through them with screenshots.

Principle of Least Privilege

Amazon strongly encourages practicing the Principle of Least Privilege, which means that you should only give users (both console users and API users, which we're creating here) the privileges required to perform their job, and nothing more. This principle minimizes potential damage if user credentials are ever compromised.

Step 1. Create the AWS user

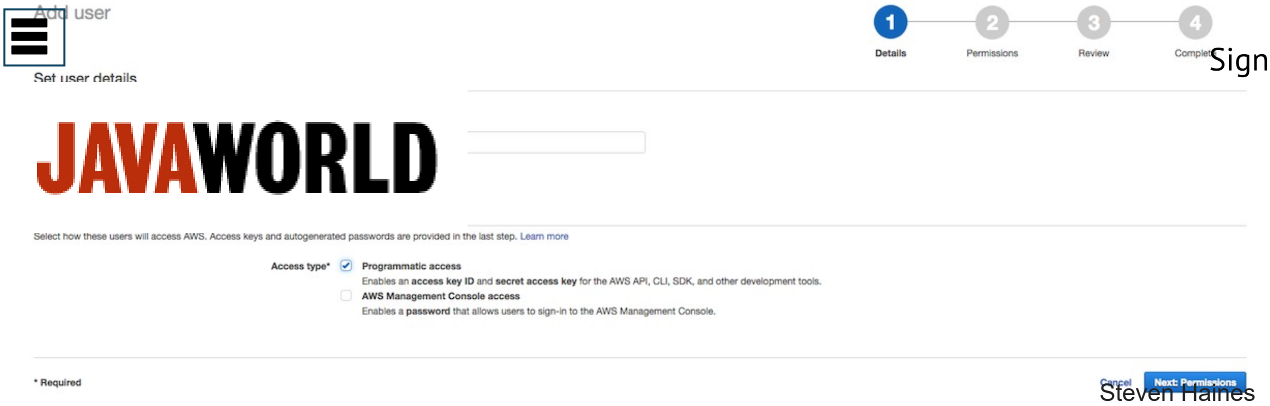
We don't want to embed our primary user credentials into a Java application, so we're going to create a new user with fewer privileges. Navigate to the AWS console, choose **Services**, and find **IAM** under **Security, Identity & Compliance**. Click on **Users** and then the **Add user** button, as shown below.



Steven Haines

Figure 7. Add user

Give your user a name, such as `get-widget-lambda-user` and check the **Programmatic access** checkbox and press **Next: Permissions**, as shown below.



Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

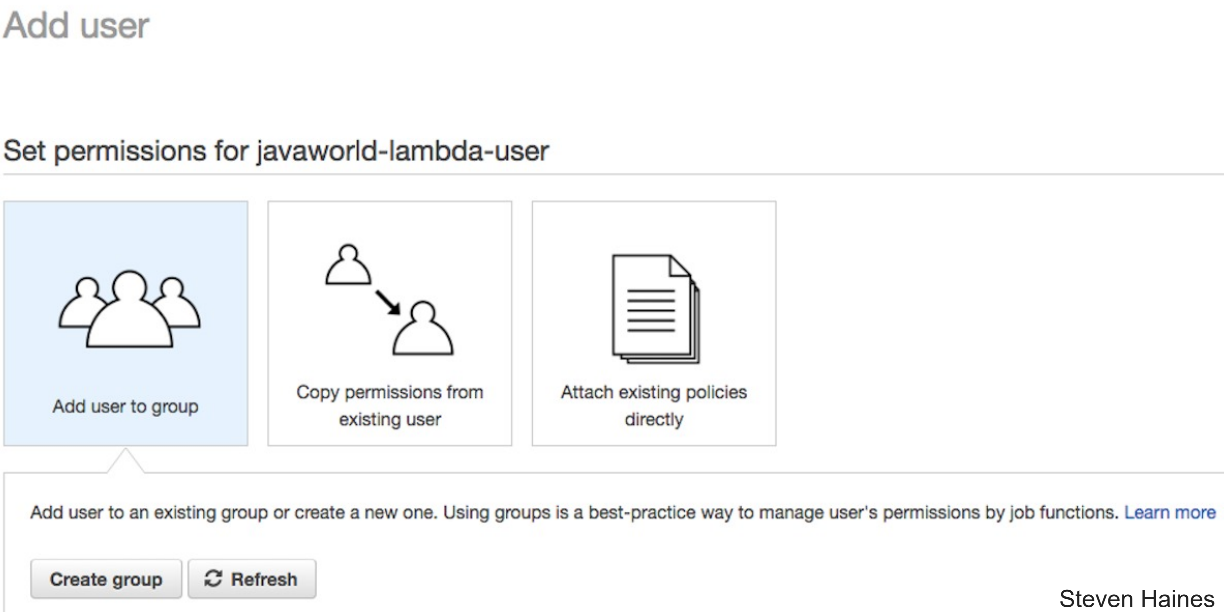
Access type* ☑ **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.
□ **AWS Management Console access**
Enables a **password** that allows users to sign-in to the AWS Management Console.

* Required

Steven Haines

Figure 8. Set user access

Next, click on the **Create Group** button.



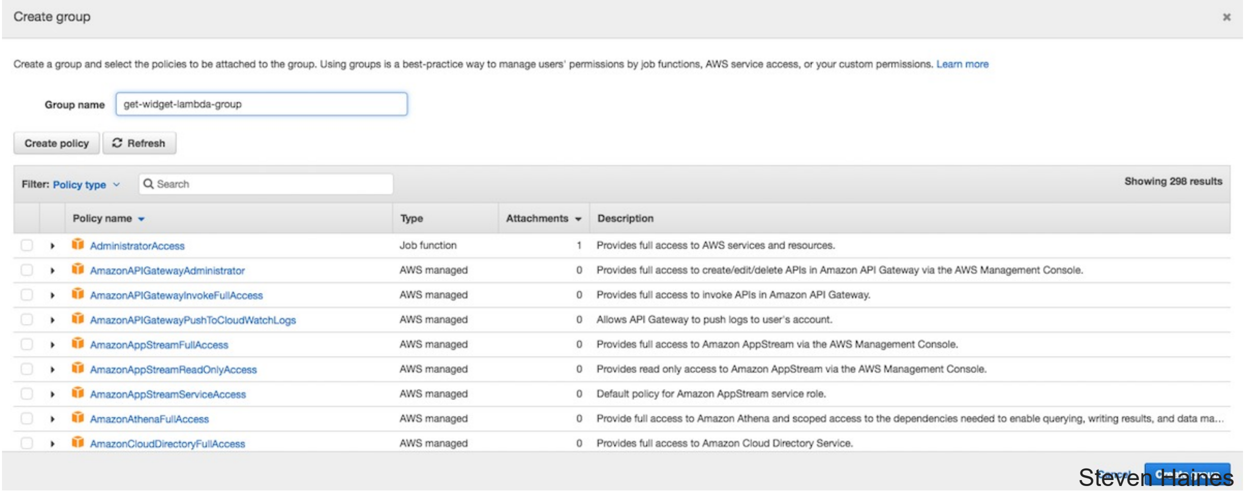
Add user
Set permissions for javaworld-lambda-user
Add user to group
Copy permissions from existing user
Attach existing policies directly
Create group
Refresh

Add user to an existing group or create a new one. Using groups is a best-practice way to manage user's permissions by job functions. Learn more

Steven Haines

Figure 9. Create a group for your user

While we can add inline policies to users, it is a better practice to create a group that manages policies for you, and then add the user to that group. Enter a name for your group, then click **Create policy**.



Create a group and select the policies to be attached to the group. Using groups is a best-practice way to manage users' permissions by job functions, AWS service access, or your custom permissions. Learn more

Group name: get-widget-lambda-group

Filter: Policy type · Search · Showing 298 results

		Policy name	Type	Attachments	Description
□	▸	AdministratorAccess	Job function	1	Provides full access to AWS services and resources.
□	▸	AmazonAPIGatewayAdministrator	AWS managed	0	Provides full access to create/edit/delete APIs in Amazon API Gateway via the AWS Management Console.
□	▸	AmazonAPIGatewayInvokeFullAccess	AWS managed	0	Provides full access to invoke APIs in Amazon API Gateway.
□	▸	AmazonAPIGatewayPushToCloudWatchLogs	AWS managed	0	Allows API Gateway to push logs to user's account.
□	▸	AmazonAppStreamFullAccess	AWS managed	0	Provides full access to Amazon AppStream via the AWS Management Console.
□	▸	AmazonAppStreamReadOnlyAccess	AWS managed	0	Provides read only access to Amazon AppStream via the AWS Management Console.
□	▸	AmazonAppStreamServiceAccess	AWS managed	0	Default policy for Amazon AppStream service role.
□	▸	AmazonAthenaFullAccess	AWS managed	0	Provide full access to Amazon Athena and scoped access to the dependencies needed to enable querying, writing results, and data ma...
□	▸	AmazonCloudDirectoryFullAccess	AWS managed	0	Provides full access to Amazon Cloud Directory Service.

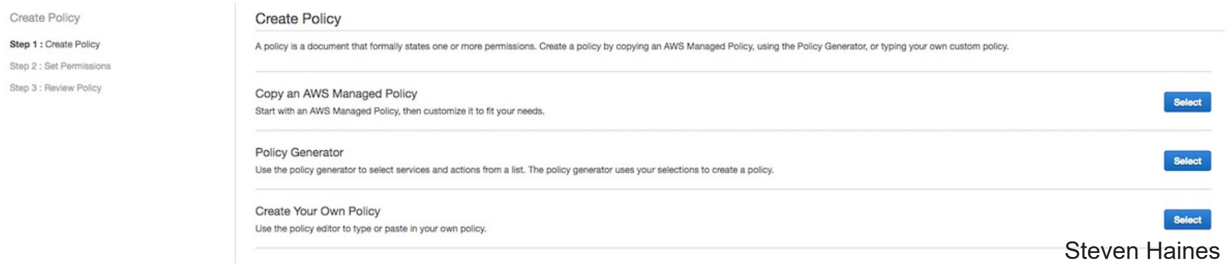
Steven Haines

Figure 10. Create a group for your user

Step 2. Create a custom access policy

licy but to try to make things easier I searched the existing he only existing policy for granting `lambda:InvokeFunction` permissions is the Lambda All Access policy. This basically gives the user root access to all Lambda functionality, which is not a good idea.

We'll need to create a custom policy to restrict access for a programatic user. First, we'll build a policy using the policy generator. To start, click on **Create policy**. You should see a screen like this one:



Create Policy

Step 1 : Create Policy

Step 2 : Set Permissions

Step 3 : Review Policy

Create Policy

A policy is a document that formally states one or more permissions. Create a policy by copying an AWS Managed Policy, using the Policy Generator, or typing your own custom policy.

Copy an AWS Managed Policy

Start with an AWS Managed Policy, then customize it to fit your needs.

Policy Generator

Use the policy generator to select services and actions from a list. The policy generator uses your selections to create a policy.

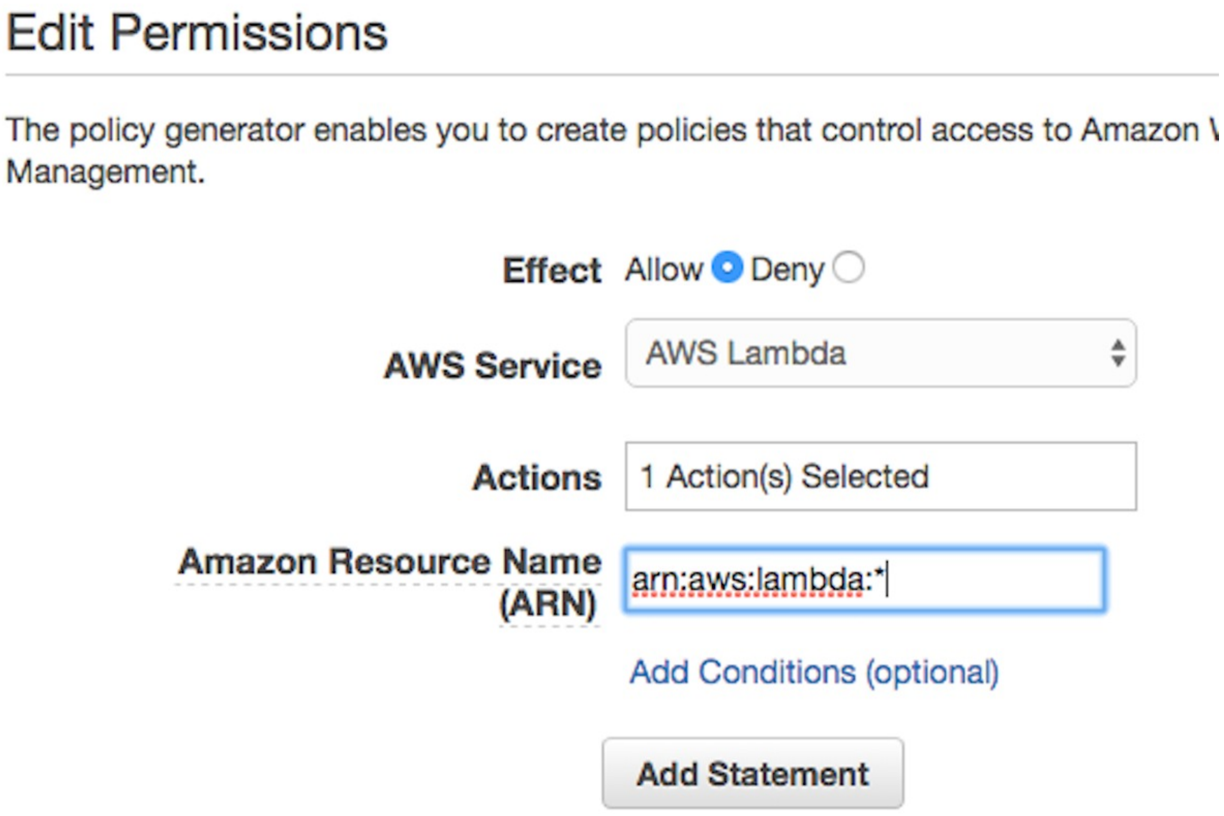
Create Your Own Policy

Use the policy editor to type or paste in your own policy.

Steven Haines

Figure 11. Create policy

Click **Select** at the end of the Policy Generator line. This will bring up the Edit Permissions page shown here:



Edit Permissions

The policy generator enables you to create policies that control access to Amazon \ Management.

Effect Allow ● Deny ○

AWS Service: AWS Lambda

Actions: 1 Action(s) Selected

Amazon Resource Name (ARN): arn:aws:lambda:*

Steven Haines

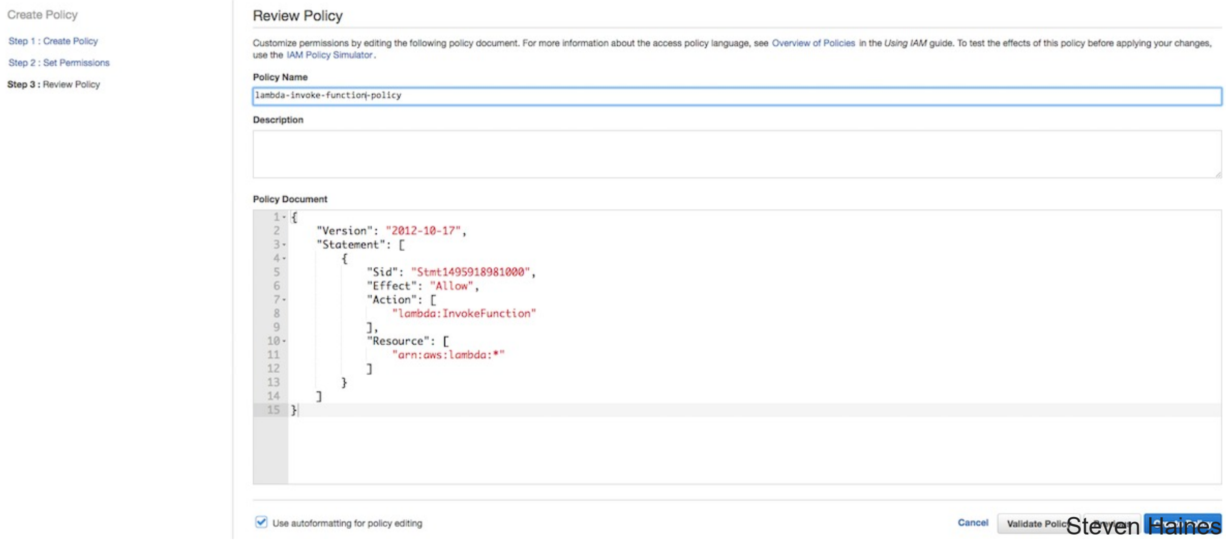
Figure 12. Edit Permissions

Choose AWS Lambda from the AWS Service dropdown, then click on **Actions** and choose InvokeFunction. Add the following for the ARN:

This policy allows the InvokeFunction action to be performed on any AWS Lambda function. You could further restrict this policy to limit invocation to a single AWS Lambda by copying the ARN for the AWS Lambda itself, which is on the top of the Lambda's dashboard page. Mine is:

```
arn:aws:lambda:us-east-1:YOUR_ACCOUNT_NUMBER:function:get-widget
```

When you are ready, press **Add Statement** to add this permission to your policy. Press **Next Step** to review the policy, as shown here.



Review Policy

Customize permissions by editing the following policy document. For more information about the access policy language, see Overview of Policies in the *Using IAM* guide. To test the effects of this policy before applying your changes, use the IAM Policy Simulator.

Policy Name

lambda-invoke-function-policy

Description

Policy Document

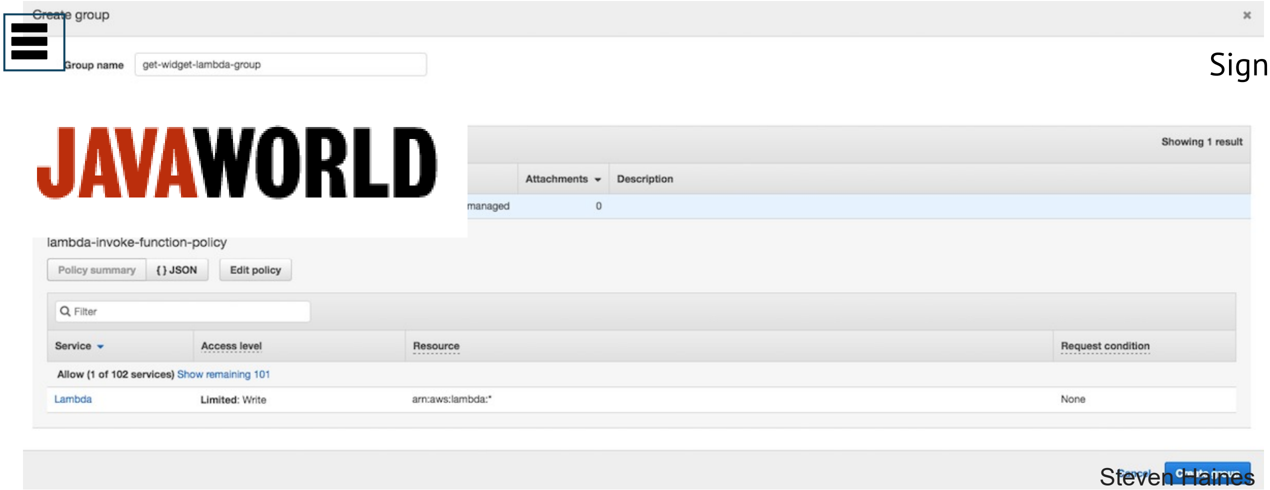
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1495918981000",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:*"
            ]
        }
    ]
}
```

☑ Use autoformatting for policy editing

Steven Haines

Figure 13. Review policy

The important thing on this page is to set a policy name, otherwise your policy will be named "policygen" followed by some random numbers. Press the **Create Policy** button. Because you had to create the policy in a new window, go back to the user-creation screen, press the **Refresh** button, and find and select your policy, as shown:



Group name: get-widget-lambda-group

Showing 1 result

	Attachments	Description
managed	0	

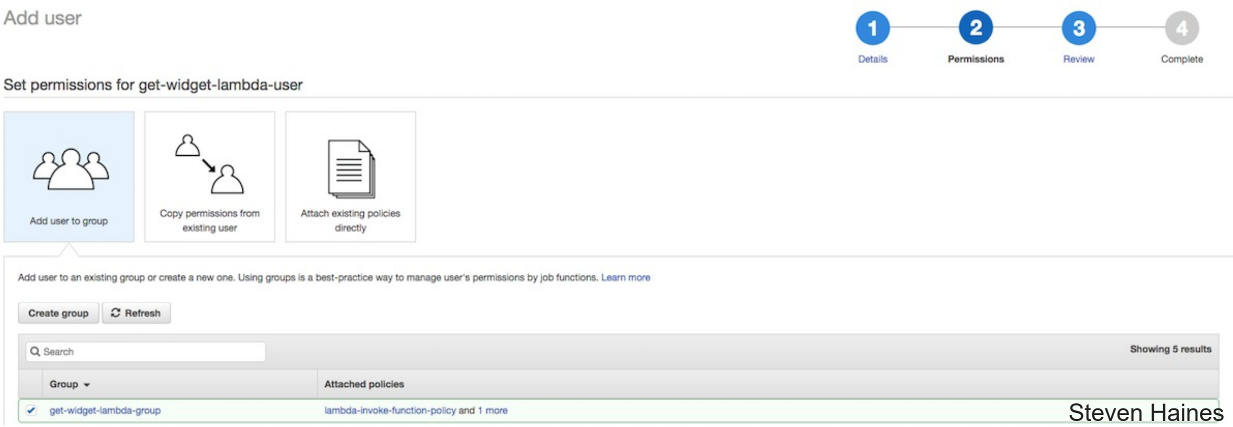
lambda-invoke-function-policy

Service	Access level	Resource	Request condition
Allow (1 of 102 services) Show remaining 101			
Lambda	Limited: Write	arn:aws:lambda:*	None

Steven Haines

Figure 14. Refresh the Create Group page

If it's difficult to find your newly created policy, you can use filters to help. As an example, try selecting customer managed policies and enter the name "`lambda-invoke`". Press **Create Group** and it will return you to your user-creation workflow with your new group:



Set permissions for get-widget-lambda-user

Add user to an existing group or create a new one. Using groups is a best-practice way to manage user's permissions by job functions. Learn more

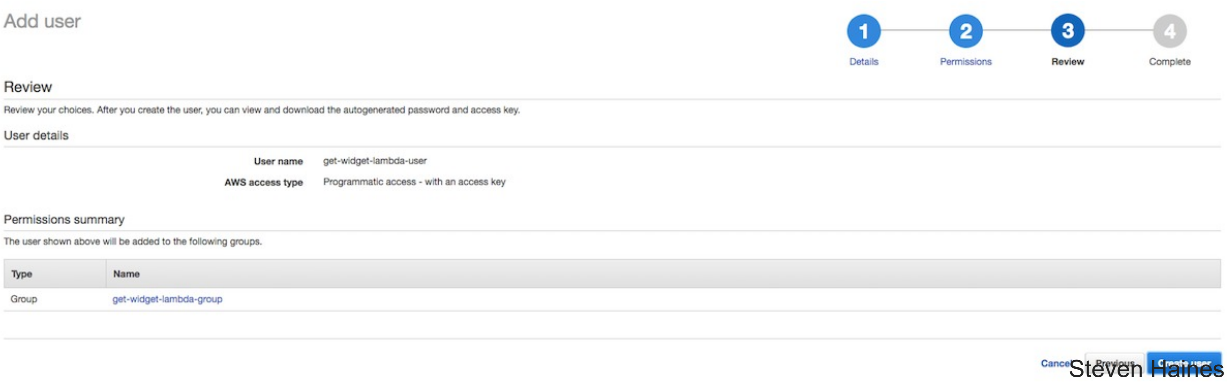
Showing 5 results

	Group	Attached policies
☑	get-widget-lambda-group	lambda-invoke-function-policy and 1 more

Steven Haines

Figure 15. Select group

Finally, review your user and press **Create User**:



Review

Review your choices. After you create the user, you can view and download the autogenerated password and access key.

User details

User name: get-widget-lambda-user

AWS access type: Programmatic access - with an access key

Permissions summary

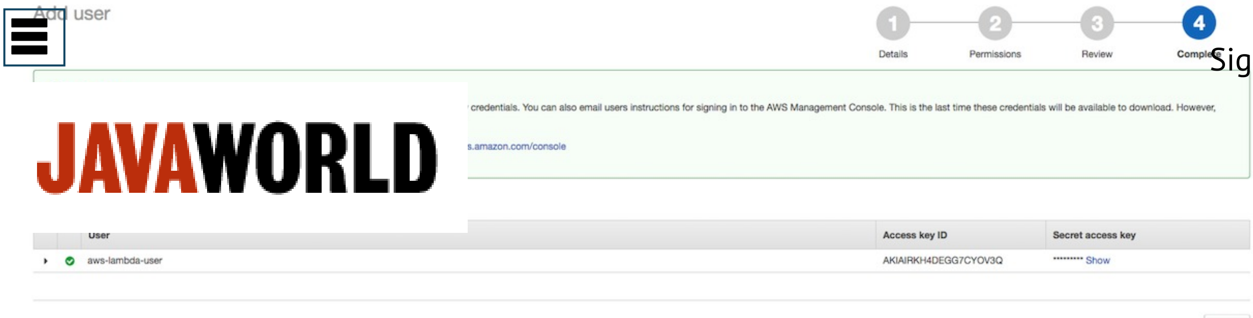
The user shown above will be added to the following groups.

Type	Name
Group	get-widget-lambda-group

Steven Haines

Figure 16. Review user

The last page is important because this is where you can see your access key ID and secret access key. If you press the **Show** link, it will reveal your secret access key. You'll need to add this information to your Java code in the next section, so be sure to preserve it.



credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However,

s.amazon.com/console

User	Access key ID	Secret access key
aws-lambda-user	AKIAIRKH4DEGG7CYOV3Q	********* Show

Steven Haines

Figure 17. New user

That completes the policy setup. So far you have

1. Created a new policy allowing the invocation of any Lambda function in your account.
2. Created a group that has this single policy.
3. Created a user and added it to this group.
4. Saved your access key ID and secret access key, which you'll add to your Java code in the next section.

Step 3. Create a Lambda client class

Next we'll write the code to connect to AWS and invoke our function. Listing 3 shows the source code for the `WidgetLambdaClient` class.

Listing 3. WidgetLambdaClient.java

```
package com.javaworld.awslambda.widget.client;

                        StaticCredentialsProvider;
                        icAWSCredentials;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.lambda.AWSLambda;
import com.amazonaws.services.lambda.AWSLambdaClientBuilder;
import com.amazonaws.services.lambda.model.InvokeRequest;
import com.amazonaws.services.lambda.model.InvokeResult;

public class WidgetLambdaClient {

    public static void main(String[] args) {
        // Setup credentials
        BasicAWSCredentials awsCreds = new BasicAWSCredentials("YOUR_ACCESS_KEY_ID", "YOUR_SEC

        // Create an AWSLambda client
        AWSLambda lambda = AWSLambdaClientBuilder.standard().withCredentials(new AWSStaticCr

        // Create an InvokeRequest
        InvokeRequest request = new InvokeRequest()
                .withFunctionName("get-widget")
                .withPayload("{ \"id\": \"1\"}");

        try {
            // Execute the InvokeRequest
            InvokeResult result = lambda.invoke(request);

            // We should validate the response
            System.out.println("Status Code: " + result.getStatusCode());

            // Get the response as JSON
            String json = new String(result.getPayload().array(), "UTF-8");

            // Show the response; we could use a library like Jackson to convert this to an object
            System.out.println(json);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

We begin by creating a `BasicAWSCredentials` instance with our access key and secret key, which you preserved in Step 2. Our main interface into AWS Lambda is the `AWSLambda` class, which can be created using the `AWSLambdaClientBuilder`. We invoke `standard()` to create it and then we add our user credentials, which we've wrapped in an AWSStaticCredentialsProvider instance.

d in specific regions, so you'll need to set the region. If you aren't sure where you created your Lambda function, navigate back to your Lambda page and look at the ARN in the upper-right side of the page. Or you can look at the upper-right side of the top toolbar next to your name on the Lambda page in the AWS console. As an example, I am running in "N. Virginia", which is "us-east-1". Use this website to translate the physical location to the region. Once you've set the region, call `build()` to create the `AWSLambda` instance.

Step 4. Invoke the Lambda function

There are different ways of invoking Lambdas. In this case we'll opt for the most straightforward (and manual) way, of sending an `InvokeRequest` to the function and receiving an `InvokeResult` back. To start, create a new `InvokeRequest` instance, call `withFunction()` to tell it the name of the function you want to invoke, and then pass the payload you want to send via the `withPayload()` method. This payload should look just like the one we used for testing in the AWS Lambda dashboard.

We pass the `InvokeRequest` to `AWSLambda`'s `invoke()` method and it returns an `InvokeResult` instance. The `getStatusCode()` method will tell us if it succeeded (returning a `200` response) or failed (returning a non-`2xx` response.) In a production application you should examine the status code after every Lambda invocation and respond accordingly.

We can retrieve the body of the response by calling the `InvokeResult`'s `getPayload()` method, which returns a `ByteBuffer`. We can convert this to a raw JSON string by converting the `ByteBuffer` to an array, by calling its `array()` method, and then passing that array to the `String`'s constructor. If we wanted to convert this into an object, we could use a tool like Jackson or Gson to deserialize the JSON into an object.

Make an executable JAR file

Finally, to make running this code easier, we'll add the `maven-jar-plugin` to the POM file, referencing our `WidgetLambdaClient` in the `mainClass`:

Listing 8. Add the maven-jar plugin to the Maven POM file

```
            <plugin>
                          apache.maven.plugins</groupId>
                          maven-jar-plugin</artifactId>
                          on>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>lib/</classpathPrefix>
                            <mainClass>com.javaworld.awslambda.widget.client.WidgetLambdaClient</m
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
```

Step 5. Build and monitor the Lambda client class

Build with `mvn clean install` and then you can execute from the `target` directory with the following command:

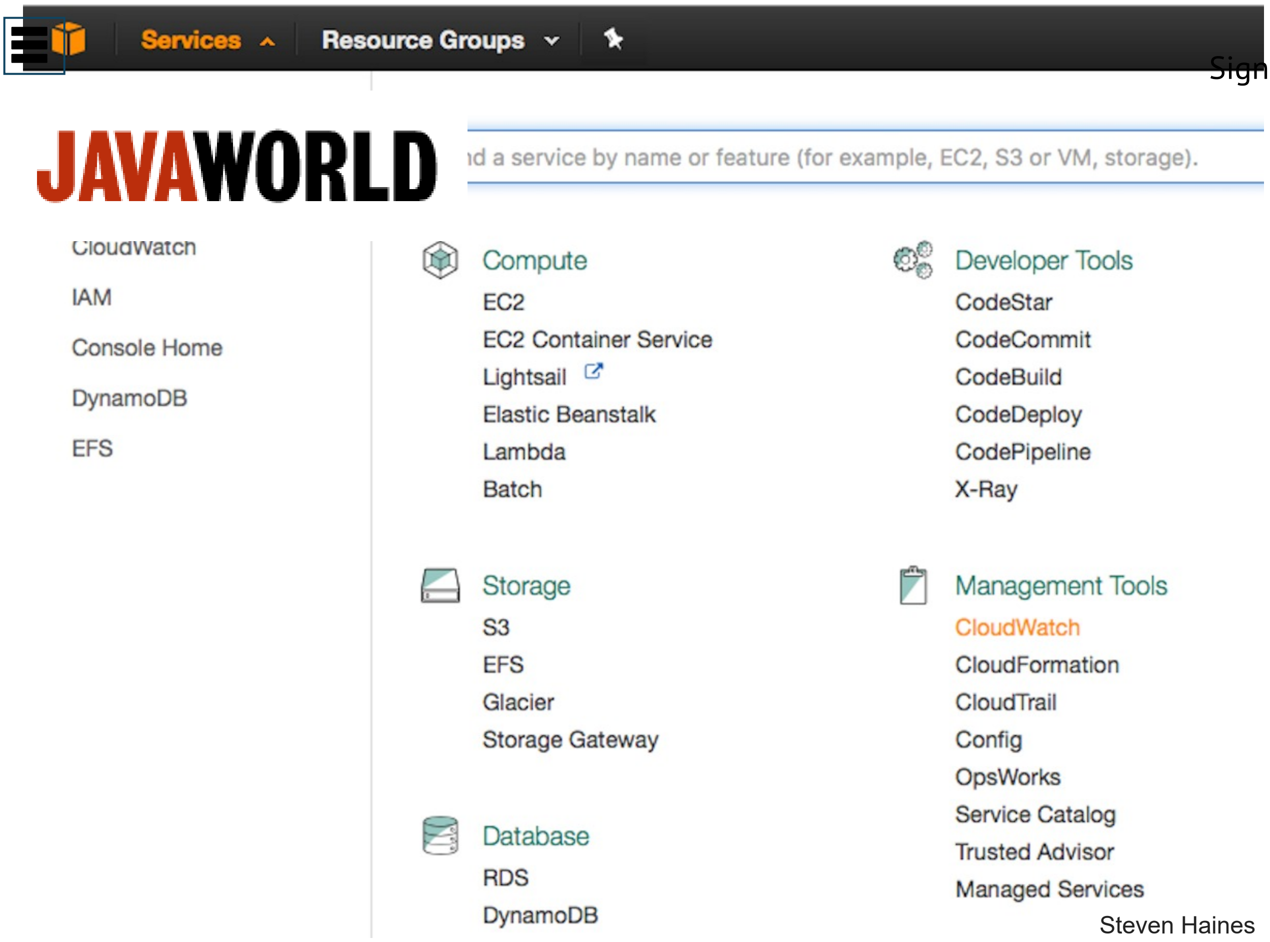
```
java -jar aws-lambda-java-1.0-SNAPSHOT.jar
```

When I run this code, I see the following output:

```
Status Code: 200
{"id":"1","name":"Widget 1"}
```

If you run into problems, check that you have the proper credentials configured and that your IAM user has the proper permissions to execute your function. You could also check the AWS Lambda logs, which you may access either through the Lambda dashboard or directly from CloudWatch:

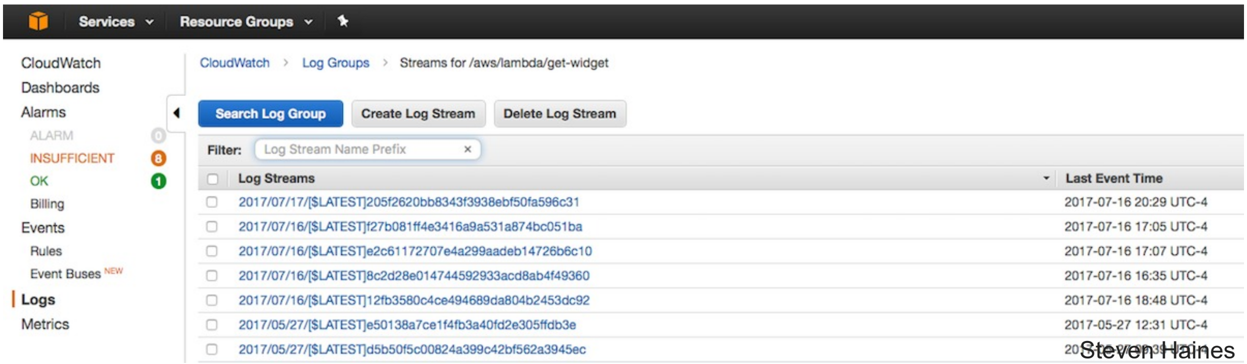
1. From your Lambda page, click on the Monitoring tab and then choose View Logs in CloudWatch in the upper-right corner
2. From the Services menu, choose CloudWatch, under Management Tools, as shown in Figure 18. Select Logs from the left panel and then choose the `aws/lambda/get-widget` log group.



Steven Haines

Figure 18. Opening CloudWatch

Accessing logs through your Lambda's page will automatically filter to your Lambda's filter group, but regardless of how you got there, you should see something similar to Figure 19.

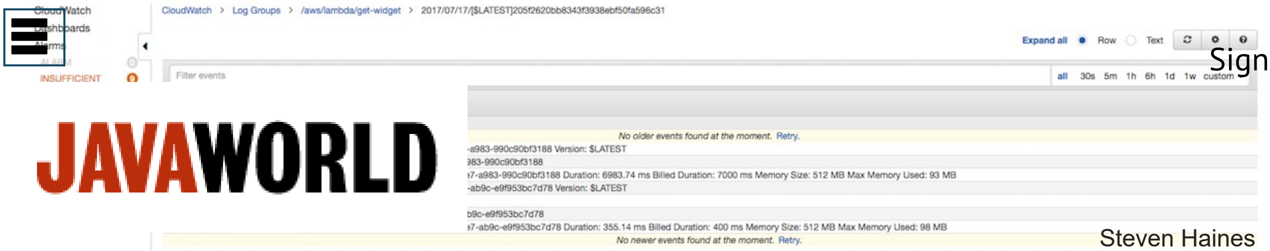


□	Log Streams	Last Event Time
□	2017/07/17/[$LATEST]205f2620bb8343f3938ebf50fa596c31	2017-07-16 20:29 UTC-4
□	2017/07/16/[$LATEST]f27b081ff4e3416a9a531a874bc051ba	2017-07-16 17:05 UTC-4
□	2017/07/16/[$LATEST]e2c61172707e4a299aadeb14726b6c10	2017-07-16 17:07 UTC-4
□	2017/07/16/[$LATEST]8c2d28e014744592933acd8ab4f49360	2017-07-16 16:35 UTC-4
□	2017/07/16/[$LATEST]12fb3580c4ce494689da804b2453dc92	2017-07-16 18:48 UTC-4
□	2017/05/27/[$LATEST]e50138a7ce1f4fb3a40fd2e305ffdb3e	2017-05-27 12:31 UTC-4
□	2017/05/27/[$LATEST]d5b50f5c00824a399c42bf562a3945ec	20[illegible]

Steven Haines

Figure 19. CloudWatch logs for your Lambda's log group

Each Log Stream represents the periodic rollup of logs for a specific Lambda. You can click on one of the entries to see the contents of the log; for example, I updated the client to request the Widget with ID "2", which does not exist. Figure 20 shows the contents of the logs for that execution.


CloudWatch
Dashboards
Alarms
ALARM
INSUFFICIENT
CloudWatch > Log Groups > /aws/lambda/get-widget > 2017/07/17/[$LATEST]205f2620bb8343f3938ebf50fa596c31
Expand all
Row
Text
Filter events
all 30s 5m 1h 6h 1d 1w custom

```
No older events found at the moment. Retry.
-a983-990c90bf3188 Version: $LATEST
983-990c90bf3188
7-a983-990c90bf3188 Duration: 6983.74 ms Billed Duration: 7000 ms Memory Size: 512 MB Max Memory Used: 93 MB
-ab9c-e9f953bc7d78 Version: $LATEST
b9c-e9f953bc7d78
7-ab9c-e9f953bc7d78 Duration: 355.14 ms Billed Duration: 400 ms Memory Size: 512 MB Max Memory Used: 98 MB
No newer events found at the moment. Retry.
```

Steven Haines

Figure 20. CloudWatch logs showing an error

In this example you can see a successful execution (start and end without any custom logging) followed by an error ("No widget found with ID: 2"). Logs are a good tool to help you troubleshoot Lambdas, especially ones executed asynchronously.

Conclusion

This two-part tutorial has introduced you to serverless computing with AWS Lambda. In Part 1 we answered the question, "What is serverless computing, anyway?" and I explained the relationship between serverless computing, microservices, and nanoservices architectures. You got your first look at AWS Lambda and we built, deployed, and tested our first Lambda function in Java.

In Part 2, we've added support for Amazon's DynamoDB, enabling our Lambda function to retrieve a live Widget from DynamoDB instead of creating one on-the-fly. You learned how to interact with DynamoDB using the `DynamoDBMapper` API. We created a Java client application that could invoke our Lambda function, and you learned how to create a new IAM user, group, and custom policy. Finally, we built the updated Lambda function and reviewed logs captured and sent to CloudWatch for troubleshooting.

Steven Haines is an author, educator, architect and cloud expert with a passion for designing and architecting large-scale cloud-based applications. He is a principal software architect at Turbonomic, working with the team responsible for their cloud initiatives.

Follow

Page 2 of 2