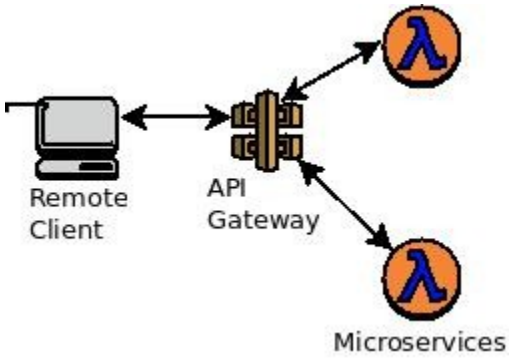
TCSS 562: Software Engineering for Cloud Computing
Fall 2018
http://faculty.washington.edu/wlloyd/courses/tcss562

School of Engineering and Technology
University of Washington – Tacoma

Tutorial 4 – Introduction to AWS Lambda

Disclaimer: Subject to updates as corrections are found
Version 0.10
Scoring: 20 pts maximum

The purpose of this tutorial is to introduce creating Function-as-a-Service functions on the AWS Lambda FaaS platform, and then to create a simple two-service application where the application flow control is managed by the client:



This tutorial will focus on developing Lambda functions in Java using the FaaS_Inspector framework. The FaaS_Inspector framework supports identifying the cloud infrastructure used to run FaaS functions to identify function performance implications resulting from the Freeze-Thaw cycle of serverless infrastructure.

1. Download the Faas_Inspector Lambda function template

To begin, using git, clone the FaaS_Inspector framework.

If you do not already have git installed, plus do so.

On ubuntu see the official documentation:
https://help.ubuntu.com/lts/serverguide/git.html.en

For a full tutorial on the use of git, here is an old tutorial for TCSS 360:
http://faculty.washington.edu/wlloyd/courses/tcss360/assignments/TCSS360_w2017_Tutorial_1.pdf

If you prefer using a GUI-based tool, on Windows/Mac check out the GitHub Desktop:
https://desktop.github.com/

Once having access to a git client, clone the source repository:

```
git clone https://github.com/wlloyduw/faas_inspector.git
```

The “FaaS inspector” for Lambda provides a Lambda function template as a maven project. If you’re familiar with Maven as a build environment, you can simply edit your Java Lambda function code using an editor such as vi, emacs, pico/nano. However, working with an IDE tends to be easier.

2. Build the faas_inspect Lambda function Hello World template

If you have a favorite Java IDE with maven support, feel free to try to open and work with the project. The project is provide as an Oracle NetBeans project, but has maven build files. It should be possible to import the project into Eclipse or IntelliJ.

If you are new to maven builds, it may be best to download the Netbeans IDE.
The present version is 8.2.
The “Java SE” version is probably sufficient. You may want the “Java EE” or “All” versions, but if you have limited disk space on your system, it’s probably best to use “Java SE”:

Download Netbeans 8.2
https://netbeans.org/downloads/

Once you’ve downloaded Netbeans, you’ll be able to open the project directly.

Select “File | Open Project”.

Then locate your clone git project, and drill down:

faas_inspector → lambda → java_template

Then on the left-hand side, expand the “Source Packages” folder.

You’ll see a “lambda” folder.
Expand this.

This contains a Hello World Lambda function as a starter template.

There are three class files.

Hello.java Provides the implementation of the AWS Lambda Function. The handleRequest() method is called by Lambda as the entry point of your Lambda function. You’ll see where in this function code should be inserted. Hello also contains a public static void main() traditional Java program entry point. This allows your Lambda function to be first tested on the command line before deployment to Lambda.

Request.java This class is a POJO. This stands for a plain-old java object. You will need to define getter and setter methods and private variables to capture data sent from the client to the Lambda function. Any JSON

object sent to your Lambda function is automatically marshalled into this Java class for easy consumption.

Response.java This class is also a POJO. Define getter and setter methods and private variables to match the desired JSON output of the Lambda function.

Now compile the project.
In NetBeans right click on the name of the project “java_template” in the left-hand list of Projects and click “Clean and Build”.

If not using Netbeans, perform a maven build in your IDE or cmd line environment.

3. Test Lambda function locally before deployment

Now, open a command shell and navigate to:

```
cd {base directory where project was cloned}/faas_inspector/lambda/java_template/test
```

Then run the command line test script to test your Lambda locally first:

```
./invoke_via_cmdline.sh
```

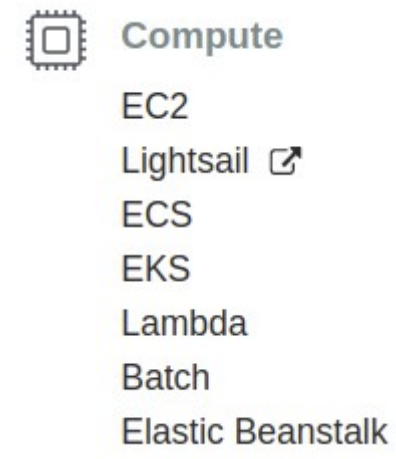
Output should be provided as follows:

```
$ ./invoke_via_cmdline.sh
cmd-line param name=fred
function result:value=Hello fred
uuid=b309f9e4-1535-4a7e-aa16-8a0070b8f2b0
vmuptime=1538676759
```

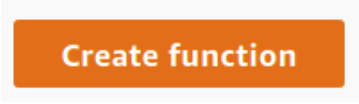
4. Deploy the function to AWS Lambda

If the Lambda function has worked locally, next deploy this to AWS Lambda.

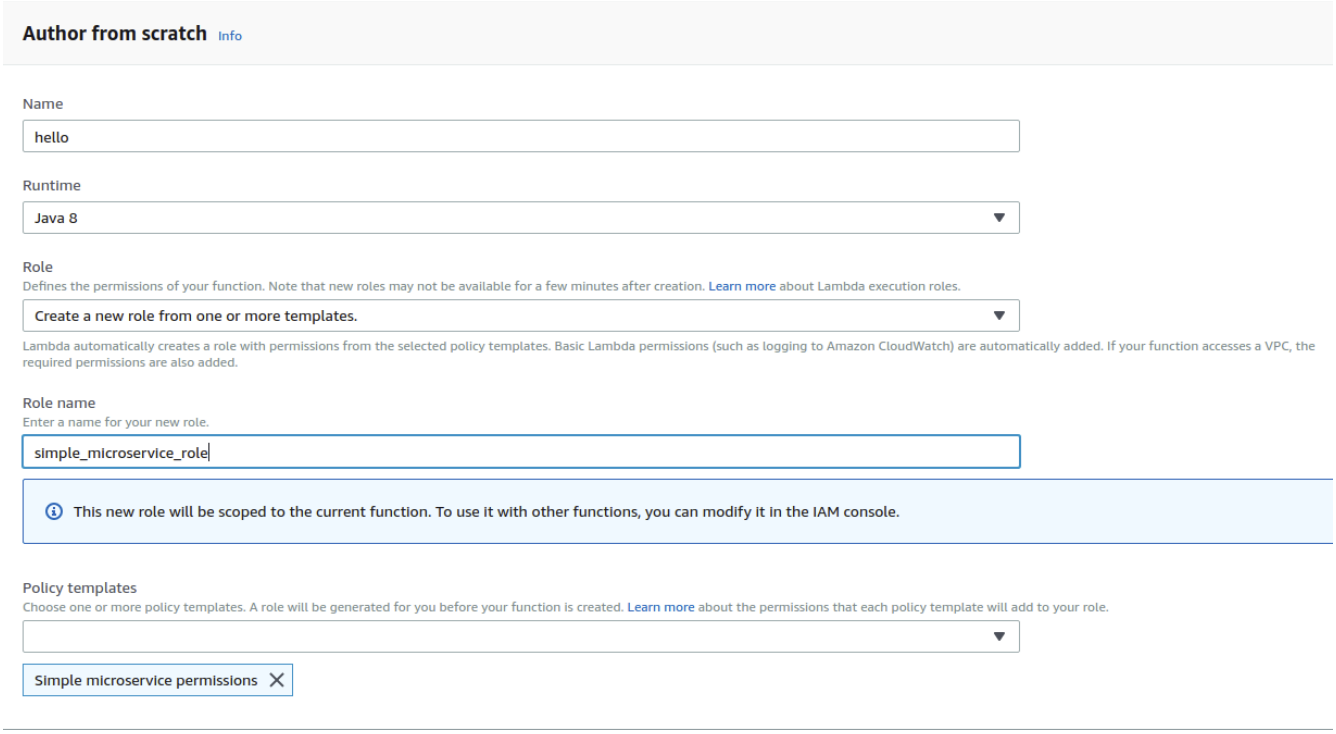
Log into your AWS account, and locate under “Compute” services, “Lambda”:



Click the button to create a new Function:

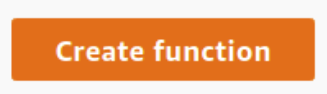


Using the wizard, provide the following values:

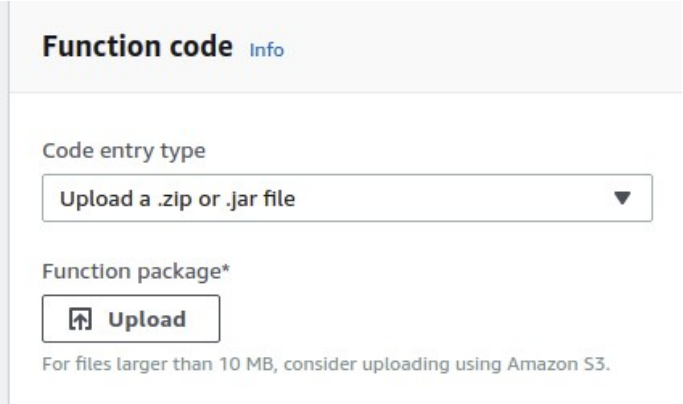


Name: hello
Runtime: Java 8
Role: “Create a new role from one or more templates”
Role name: simple_microservice_role
(Roles can be inspected under IAM | Roles in the AWS Management Console)
Policy templates: “Simple microservice permissions”

Once filling the form, click the button:



Next, upload your compiled Java JAR file to AWS Lambda:



Click the “Upload” button to navigate and locate your JAR file. The jar file is under the “target” directory. It should be called “lambda_test-1.0-SNAPSHOT.jar”.

Once selecting the file, change the “Handler”.

```
Handler: lambda.Hello::handleRequest
```

***** IF THE HANDLER IS NOT UPDATED, LAMBDA WILL NOT BE ABLE TO LOCATE THE ENTRY POINT TO YOUR CODE. THE LAMBDA FUNCTION WILL FAIL TO RUN *****

Then press the “Upload” button.

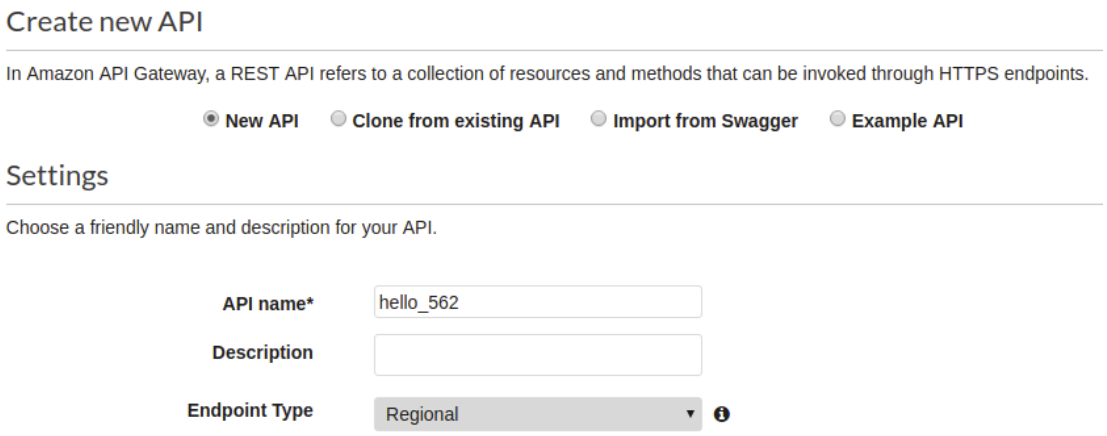
5. Create an API-Gateway REST URL

Next, in the AWS Management Console, navigate to the **API Gateway**.

This appears under the Network & Content Delivery services group:

Create a new API:

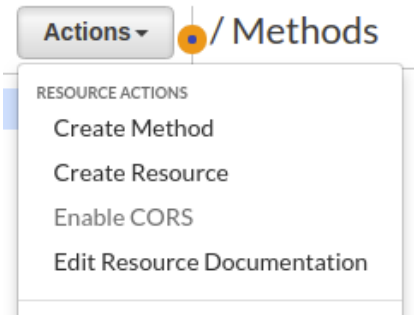
Next, complete the form:



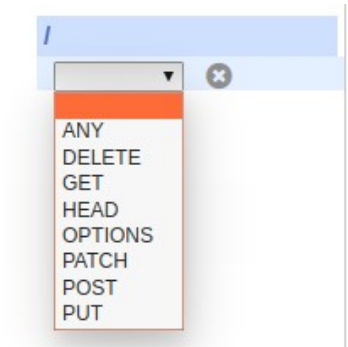
Only the API name needs to be specified. Use “hello” or “hello_562”.

Then press [Create API].

Next, pull down the Actions button-menu, and select “Create Method”:

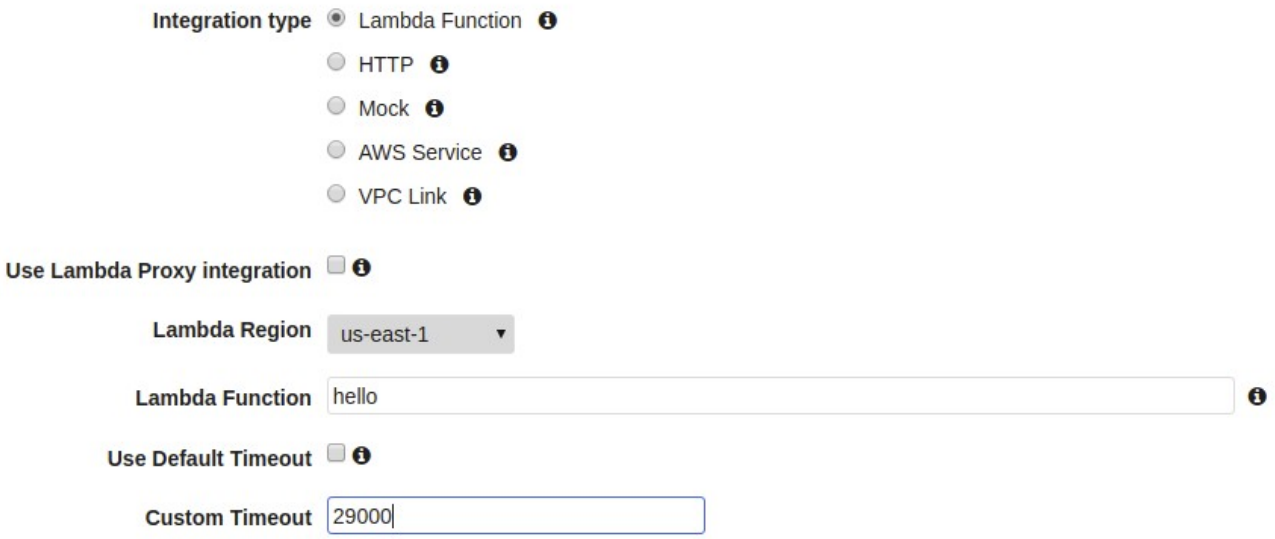


Select, drop down the menu and select “Post”:



Next, press the “checkmark” icon so it turns green:

Then complete the form.



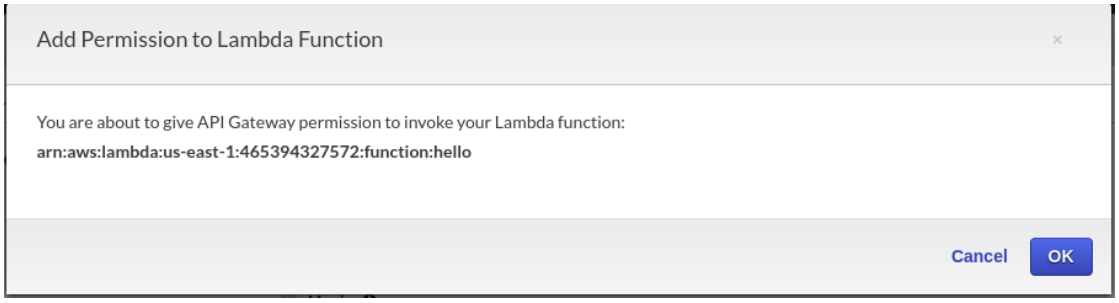
Fill in “Lambda Function” to match your function name “hello”.

Uncheck the “Use Default Timeout”.

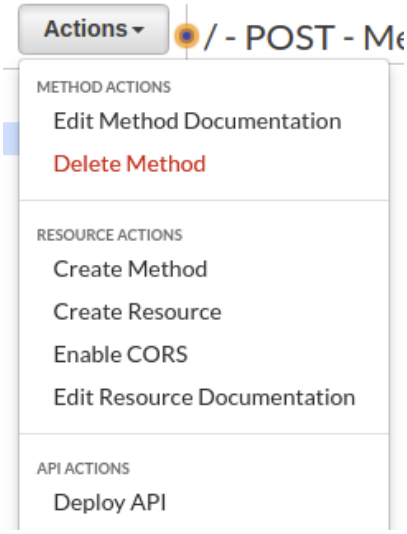
The API Gateway default time out for synchronous calls can be set between 50 and 29,000 milliseconds. Here provide the maximum synchronous timeout “29000”.

Then click Save:

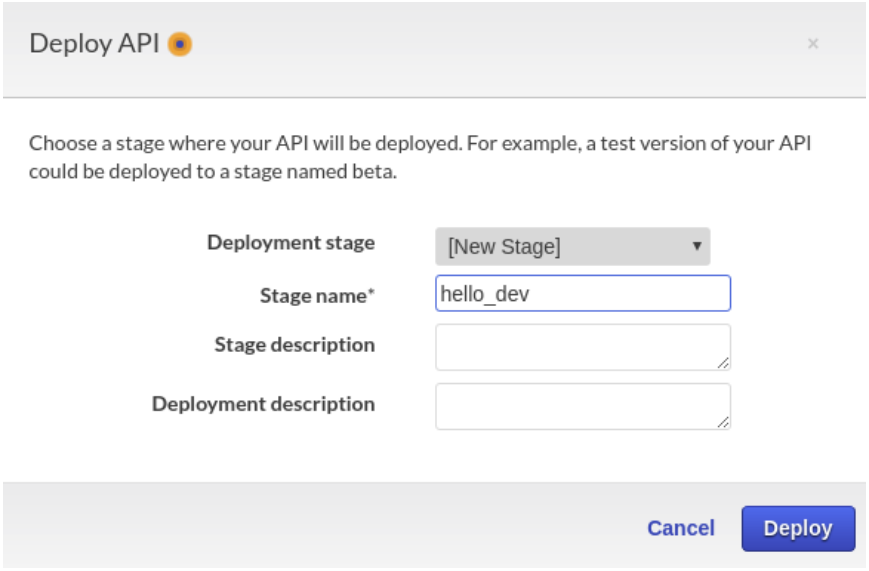
Next, acknowledge the permission popup:



Then, select the Actions drop-down and select Deploy API:



Next complete the form:



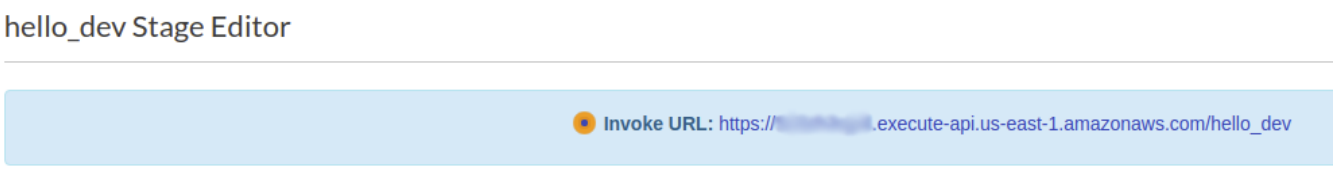
The API-Gateway allows many URLs to be configured as REST webservice backends. The API-Gateway is not limited to AWS Lambda functions. It can also point to other backends hosted by AWS. The most common is to specify a “HTTP” path. This causes the API-Gateway to provide a new AWS hosted URL that is a proxy to an existing one. This allows all traffic to be routed to the URL to go through the API-Gateway for logging and/or processing.

Using the API-Gateway it is also possible to host multiple implementations of a function to suppotr Agile software development processes. An organization may want to maintain multiple live version of a function in various tages of development such as : (dev)evelopment, test, staging and (prod)uction.

When complete, press the [Deploy] button.
The Stage name is appended to the end of the URL.

A stage editor should then appear with a REST URL to your AWS Lambda function.

COPY THIS URL TO THE CLIPBOARD:
Mouse over the URL, -right-click- and select “Copy link address”:



6. Add packages and configure your client to call Lambda

Next, return to the command prompt and navigate to the directory

```
cd {base directory where project was cloned}/faas_inspector/lambda/java_template/test
```

Using a text editor such as vi, pico, nano, vim, or gedit, edit the file “callservice.sh”

Locate the lines:

```
echo "Invoking Lambda function using API Gateway"
time output=`curl -s -H "Content-Type: application/json" -X POST -d  $json
{INSERT API GATEWAY URL HERE}`
```

Replace {INSERT API GATEWAY URL HERE} with your URL.

Be sure to include the small quote mark at the end: `

This quote mark is next to the number 1 on US keyboards.

Next, locate the lines:

```
echo "Invoking Lambda function using AWS CLI"
time output=`aws lambda invoke --invocation-type RequestResponse
--function-name   --region us-east-1 --payload $json /dev/stdout | head -n
1 | head -c -2 ; echo`
```

Replace {INSERT AWS FUNCTION NAME HERE} with your Lambda function name “hello”.

Before running this script, it is necessary to install some packages.

You should have curl installed from tutorial #2. If not, please install it:

```
sudo apt install curl
```

Next, install the AWS command line interface:

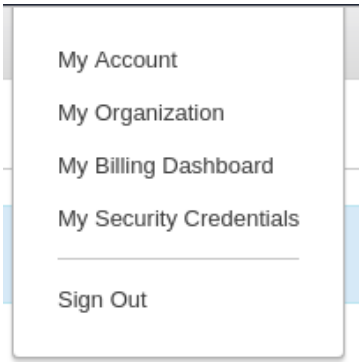
```
sudo apt install awscli
```

Next, configure the AWS CLI with your AWS account credentials:

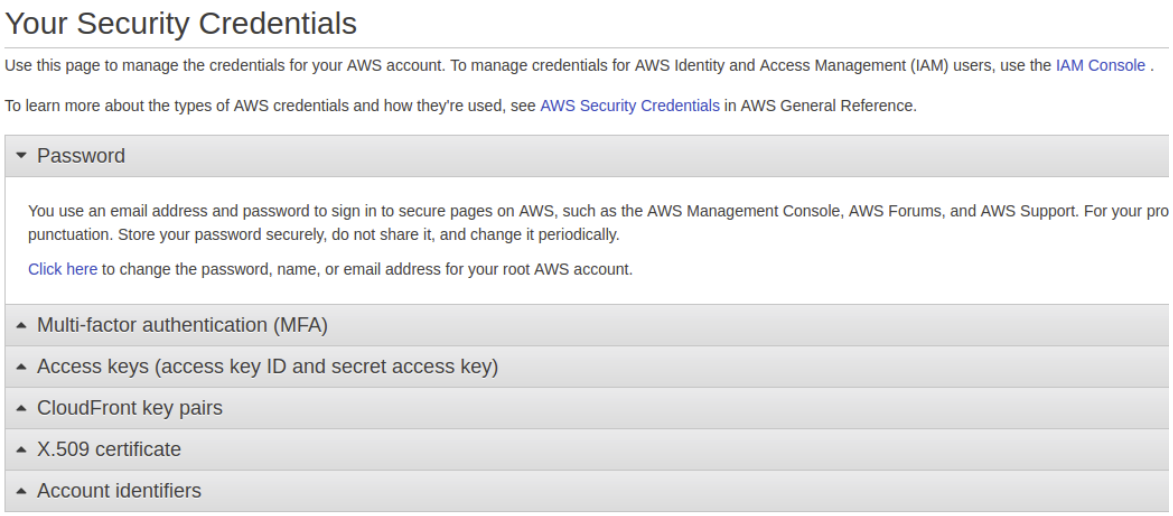
You will need to acquire a AWS Access Key and an AWS Secret Access Key to use the AWS CLI.

In the far upper right-hand corner, locate your Name, and drop-down the menu.

Select “My Security Credentials”:

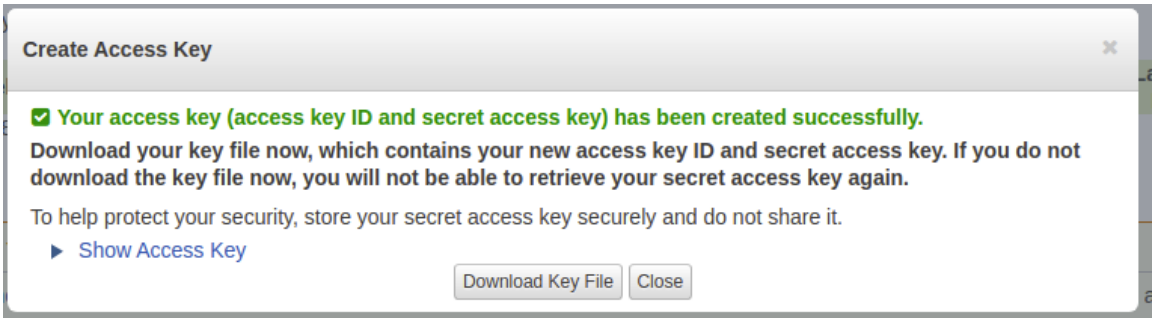


Then expand the menu option for “Access keys (access key ID and secret access key):



Click the blue button:

In the dialog, expand the “Show Access Key” section:



Copy and paste the Access Key ID and the Secret Access Key to a safe place:

The values are blurred-out above.

Next configure your AWS CLI.

If you are not using Virginia, you'll need to specify the region identifier.
Oregon is "us-west-2".

```
$ aws configure
AWS Access Key ID [None]: <enter your access key>
AWS Secret Access Key [None]: <enter your secret key>
Default region name [None]: us-east-1
Default output format [None]:
```

This creates two hidden files at:
/home/ubuntu/.aws/config
/home/ubuntu/.aws/credentials

Use "ls -alt /home/ubuntu/.aws" to see them.

At any time, if needing to update the configuration, these files can be edited manually, or "aws configure" can be re-run. Amazon suggests changing the access key and secret access key every 90 days.

Never upload your access keys to a git repository. Avoid hard coding these keys directly in source code where feasible.

7. Test your Lambda function using the API-Gateway and AWS CLI

It should now be possible to test your Lambda function using the callservice.sh script.

Run the script:

./callservice.sh
Output should be provided:

```
# ./callservice.sh
Invoking Lambda function using API Gateway

real  0m0.767s
user  0m0.108s
sys   0m0.020s

CURL RESULT:
{"value":"Hello Fred Smith","uuid":"f0f945fc-9bb6-4e18-9f5d-
2d1da8f78be4","error":"","vmuptime":1539930217,"newcontainer":0}


Invoking Lambda function using AWS CLI

real  0m0.884s
user  0m0.408s
sys   0m0.052s

AWS CLI RESULT:
{"value":"Hello Fred Smith","uuid":"f0f945fc-9bb6-4e18-9f5d-
2d1da8f78be4","error":"","vmuptime":1539930217,"newcontainer":0}
```

Check out the callservice.sh bash script code:

cat callservice.sh

The script calls Lambda twice. The first instance uses the API gateway. As a synchronous call the curl connection is limited to 29 seconds.

The second instance uses the AWS command line interface. This runtime is limited by the AWS Lambda function configuration. It can be set to a maximum of 5 minutes. The default is 15 seconds.

8. Parallel Client Testing of AWS Lambda

Next, configure two files to try out the partest parallel test client script.

Continue to work in the “test” directory:

```
cd {base directory where project was cloned}/faas_inspector/lambda/java_template/test
```

Three more packages are required:

apt install parallel bc jq

Next edit and replace the contents of two files as follows:

parurl:
replace {AWS API Gateway function URL}
with your API-Gateway URL

parfunction:
replace {AWS Lambda Function name for partest script}
with “hello” or the name of your AWS Lambda function-name

Now try the partest parallel test script.

./partest.sh 10 10 1 1

This creates 10 threads to call Lambda in parallel. Each call is synchronous and the execution is traced.

Output describes the results of all ten threads:

```
Fri Oct 19 07:12:35 UTC 2018
Setting up test: runsperthread=1 threads=10 totalruns=10
run_id,thread_id,uuid,cputype,cpusteal,vmuptime,pid,cpuusr,cpukrn,elapsed_time,sleep_time_ms,new_container
1,1,"53bcbfa7-1569-47f2-aeff-98c59aaa2db1",unknown,null,1539930217,null,null,null,751,.24900000000000000000,0
1,2,"e3d3f526-3f38-45dd-81e3-d5e34a41b72a",unknown,null,1539930217,null,null,null,741,.25900000000000000000,0
1,3,"e3d3f526-3f38-45dd-81e3-d5e34a41b72a",unknown,null,1539930217,null,null,null,732,.26800000000000000000,0
1,4,"53bcbfa7-1569-47f2-aeff-98c59aaa2db1",unknown,null,1539930217,null,null,null,736,.26400000000000000000,0
1,5,"ac8b6dfa-f404-46a9-9f56-82217290ad7b",unknown,null,1539930217,null,null,null,742,.25800000000000000000,0
1,6,"1f592ecd-613a-4ac3-a0ba-752f3c8b94ae",unknown,null,1539930217,null,null,null,732,.26800000000000000000,0
1,7,"ac8b6dfa-f404-46a9-9f56-82217290ad7b",unknown,null,1539930217,null,null,null,721,.27900000000000000000,0
1,8,"8846c27a-0c11-44f5-b430-ea3f06274bf2",unknown,null,1539925335,null,null,null,741,.25900000000000000000,0
1,9,"8846c27a-0c11-44f5-b430-ea3f06274bf2",unknown,null,1539925335,null,null,null,743,.25700000000000000000,0
1,10,"1f592ecd-613a-4ac3-a0ba-752f3c8b94ae",unknown,null,1539930217,null,null,null,715,.28500000000000000000,0
uuid,host,uses,totaltime,avgruntime_cont,uses_minus_avguses_sq
"e3d3f526-3f38-45dd-81e3-d5e34a41b72a",1539930217,2,1473,736.50000000000000000000,0
"ac8b6dfa-f404-46a9-9f56-82217290ad7b",1539930217,2,1463,731.50000000000000000000,0
"1f592ecd-613a-4ac3-a0ba-752f3c8b94ae",1539930217,2,1447,723.50000000000000000000,0
"53bcbfa7-1569-47f2-aeff-98c59aaa2db1",1539930217,2,1487,743.50000000000000000000,0
"8846c27a-0c11-44f5-b430-ea3f06274bf2",1539925335,2,1484,742.00000000000000000000,0
Current time of test=1539933157
host,host_up_time,uses,containers,totaltime,avgruntime_host,uses_minus_avguses_sq
1539930217,2940,8,4,5870,733.75000000000000000000,9.00000000000000000000
1539925335,7822,2,1,1484,742.00000000000000000000,9.00000000000000000000
containers,newcontainers,recycont,hosts,recyvms,avgruntime,runs_per_container,runs_per_cont_stdev,runs_per_host,runs
_per_host_stdev
5,0,0,2,0,735.40000000000000000000,2.00000000000000000000,0,5.00000000000000000000,9.00000000000000000000
```

A
B
C
D

For the output above:

Section A	Results of each individual Lambda call. Runtime is 3rd to last param.
Section B	Results grouped by unique container. Shows container reuse.
Section C	Results grouped by unique host (VM). Shows VM reuse.
Section D	Summary results including average performance for all calls.

This concludes the instructional portion of the tutorial.

9. Simple Caesar Cipher Two-Function Serverless Application

To complete tutorial #4, use the resources provided to construct a two-function serverless application.

To get started, create a new directory under /home/ubuntu
Then clone the faas_inspector repository twice to have two separate empty Lambdas:

```
$ cd ~
:~$ mkdir tcss562
:~$ cd tcss562
:~/tcss562$ mkdir func_a
:~/tcss562$ mkdir func_b
:~/tcss562$ cd func_a
:~/tcss562/func_a$ git clone https://github.com/wlloyduw/faas_inspector.git
Cloning into 'faas_inspector'...
remote: Enumerating objects: 61, done.
remote: Counting objects: 100% (61/61), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 61 (delta 12), reused 39 (delta 3), pack-reused 0
Unpacking objects: 100% (61/61), done.
Checking connectivity... done.
:~/tcss562/func_a$ cd ..
:~/tcss562$ cd func_b
:~/tcss562/func_b$ git clone https://github.com/wlloyduw/faas_inspector.git
Cloning into 'faas_inspector'...
remote: Enumerating objects: 61, done.
remote: Counting objects: 100% (61/61), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 61 (delta 12), reused 39 (delta 3), pack-reused 0
Unpacking objects: 100% (61/61), done.
Checking connectivity... done.
```

Next, implement two lambda functions.

One called “Encode”, and another “Decode” that implement the simple Caesar cipher.

The message will be passed for encoding/decoding as follows:

```
{
      "msg": "ServerlessComputingWithFaaS",
      "shift": 22,
}
```

The first service shifts the letters of an ASCII string forward to disguise the contents as follows:

```
{
      "value": "OanranhaooYkilqpejcSepdBwwO",
      "uuid": "036c9df1-4a1d-4993-bb69-f9fd0ab29816",
      "error": "",
      "vmuptime": 1539943078,
      "newcontainer": 0
}
```

The second service shifts the letters back to decode the contents:

```
{
      "value": "ServerlessComputingWithFaaS",
      "uuid": "f047b513-e611-4cac-8370-713fb2771db4",
      "error": "",
      "vmuptime": 1539943078,
      "newcontainer": 0
}
```

Notice that the two services have different uuids (container IDs) but the same vmuptime (VM/host ID).

Both services should accept two inputs:

integer	shift	number of characters to shift
String	msg	ASCII text message

There internet has many examples of implementing the ceaser cipher in Java:
https://stackoverflow.com/questions/21412148/simple-caesar-cipher-in-java

Modidy the call_service.sh script to create a JSON object to pass to the encode service. The output should be captured, parsed with jq, and sent to the decode service.

The result should be a simple pair of services for applying and removing the cipher runnable using the bash scripts. Host functions in your account to support testing.

For the submission, submit a working bash client script **(call_services.sh)** that can be used to test both ciphers. Also include a zip or tar.gz file that includes all source code for your Lambda functions.