**TCSS 562: Software Engineering** School of Engineering and Technology

**for Cloud Computing** University of Washington – Tacoma

Fall 2020

[http://faculty.washington.edu/wlloyd/courses/tcss](http://faculty.washington.edu/wlloyd/courses/tcss360)562

**Tutorial 9 – Introduction to FaaS Runner***Disclaimer: Subject to updates as corrections are found*

Version 0.01

Scoring: 40 pts maximum

The purpose of this tutorial is to provide a comprehensive overview of many of FaaS Runners most important features. This tutorial will cover creating complex experiments, automating them, and creating pipelines of functions.

**1. Download the FaaS Runner tutorial functions.**

To begin, using git, clone the GitHub repository for this tutorial.

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, create a folder and clone the source repository:

|  |
| --- |
| **git clone https://github.com/RCordingly/faas\_runner\_tutorial** |

This tutorial builds upon Tutorial 4. If you have not completed that tutorial, please review it and install any dependencies (such as Maven and the AWS CLI). This tutorial does not require code changes but does require being able to deploy functions using the built in publish scripts.

**2. Deploy the Included Functions**

Included in the repository are four functions that need to be deployed to AWS Lambda. To simplify this process, SAAF's built in publish scripts can be used to deploy them automatically. The repository contains three 'Hello World' functions; pello\_world, jello\_world, and nello\_world, and the CalcsService function.

The Jello/Pello/Nello naming is because these are Hello World functions written in **J**ava, **P**ython, and **N**ode.js respectively. SAAF supports functions written in each of these languages.

To deploy these, we must first configure each config.json file with a role ARN. You should already have an ARN created from Tutorial 4 so you can retrieve that by visiting the AWS webpage, go to IAM -> Roles and select the role you would like to use.

Graphical user interface, application

Description automatically generated

Copy your ARN shown at the top of the page. All functions can share the same ARN. Next open the **config.json** files located in each function's **deploy** folder and paste the ARN into the JSON attribute called **lambdaRoleARN**. **No other attributes in the config files need to be changed.**

You may use any text editor to enter the ARN. The example below shows opening each file in Nano.

|  |
| --- |
| **cd {base directory where project was cloned}**  **nano pello\_world/deploy/config.json**  **nano jello\_world/deploy/config.json**  **nano nello\_world/deploy/config.json**  **nano calcService/deploy/config.json** |

Graphical user interface, text

Description automatically generated

**3. Deploy each Function.**

Once each configuration file has an ARN, each function should be able to be deployed using the publish scripts.

|  |
| --- |
| **cd {base directory where project was cloned}**  **sudo chmod -r u+x ./**  **# ./publish.sh AWS GCF IBM AZURE MEMORY**  **./pello\_world/deploy/publish.sh 1 0 0 0 1024**  **./jello\_world/deploy/publish.sh 1 0 0 0 1024**  **./nello\_world/deploy/publish.sh 1 0 0 0 1024**  **./calcs\_service/deploy/publish.sh 1 0 0 0 1024** |

The publish scripts automatically package functions and can deploy them to AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and Azure Functions. Here we are just deploying to AWS Lambda with a memory reservation setting of 1024 MBs. The publish scripts can be used to deploy new functions or update existing functions.

To verify that each deployment was successful, the publish script will automatically invoke the function with the **test** payload in the config file. Verify that each function was deployed and executed successfully. The output should look similar to the example below.

Text

Description automatically generated

Each function should now be visible on the AWS Lambda web page:

Graphical user interface, text, application

Description automatically generated

**4. Running an Experiment with FaaS Runner**

Now that we have all of our functions deployed, we will begin running some experiments with FaaS Runner. To work with FaaS Runner, open the **test** folder in a terminal and execute the **faas\_runner.py** script.

FaaS Runner uses two types of files. Function files, which define the endpoints needed to execute a function, and experiment files that define how to process an experiment. Let's execute the built-in calcsService experiment to get an understanding of what FaaS Runner is doing and how the output is recorded.

|  |
| --- |
| **cd ./test**  **./faas\_runner.py -f ./functions/calcsService.json -e ./experiments/calcsServiceExp1.json** |

The **-f** flag defines the path to the function file and **-e** defines the path to the experiment file. After executing this function FaaS Runner should execute the entire experiment and automatically open a spreadsheet on MacOS and Linux.

FaaS Runner produces a lot of output text to show what is going on. It is broken into section that will be explained here.

Text

Description automatically generated

The first section is where the function information and experiment data are loaded. Here you can see the list of loaded functions, and the list of loaded experiments. For this experiment we only have one function and one experiment. If an experiment or function file is missing attributes (such as in this example **parentPayload, payloadFolder, shufflePayloads, passPayloads,** and **transitions**) default values will be used instead.

Graphical user interface, text

Description automatically generated

The second section applies any modifications to payloads if you choose to use inheritance. FaaS Runner has the ability to define parent payloads that children can inherit values from. This can be useful if you have an experiment but want to override some attribute instead of recreating the entire experiment file. In this first example we are not using this feature.

A picture containing graphical user interface, text

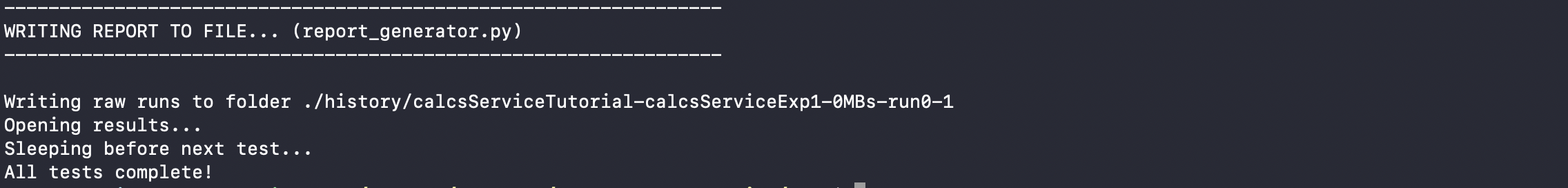
Description automatically generated

Next we have the section where functions are actually being invoked. At the start you can see the payloads of each function invocation and then shortly later you begin seeing the results of each run denoted by **STDOUT**. For long running experiments this section can be useful to make sure an experiment is executing properly.

Text

Description automatically generated

The next section is where the report is generated. The text shown here is the raw text of the CSV data that will be opened as a spreadsheet.



The final section is where files are written to disk. If an output path is not defined, FaaS Runner automatically saves data to the **history** folder. Navigate to that folder and view its contents. After running this experiment, you should see the CSV report alongside a folder with the same name. The folder will contain the JSON response payloads of each run in the experiment. This data can be used to regenerate a report.

Text

Description automatically generated

**5. Overriding Attributes with Command Line Arguments**

Next, let's create a more complex experiment with CalcsService. We will use the same experiment and function files but override attributes using command line arguments. Any attribute in function or experiment files can be defined through command line arguments.

For this experiment we are going to use the same workload but repeat it with different memory settings. FaaS Runner can automatically reconfigure memory settings on all supported platforms. This experiment will take a couple minutes.

|  |
| --- |
| **mkdir memorySettingExperiment**  **./faas\_runner.py -f ./functions/calcsService.json -e ./experiments/calcsServiceExp1.json --memorySettings [256, 512, 1024, 2048] --openCSV false -o memorySettingExperiment** |

This is the most complex experiment yet so let's see what is going on. We are defining the same function and experiment files (denoted with the **-f** and **-e** flags). Then we are overriding the experiment file's **memorySettings** attribute. Overriding attributes can be done by simply using the attribute name as a flag with '--' at the start. The memorySettings attribute is expected to be a list of memory settings you want to use. In this case we are using 256 MBs, 512 MBs, 1024 MBs, and 2048 MBs. Next we are overriding the **openCSV** attribute to be false. For larger experiments it can be annoying having many CSV files automatically opened so we will retrieve this information later. Finally, we define the output path by using the **-o** flag to be our newly created **memorySettingExperiment** folder. The order of command line arguments does not matter.

Text

Description automatically generated

Just like with the first experiment, if we open the output folder, we can now see CSV reports and folders of JSON files for each memory setting.

**6. Creating a Unified Report**

Instead of having 4 different reports for each memory setting, lets combine all the runs into one report. To do this we must first create a folder will all of the json files. This can be easily done through the command line.

|  |
| --- |
| **cd memorySettingExperiment**  **mkdir combined**  **cp -r \*\*/\*.json ./combined**  **cd ..** |

Next we can use the **compile\_results.py** script to create a single report with all 40 runs. Simply supply the path to the folder of json files (**./memorySettingExperiment/combined**) and then the path to an experiment file (**./experiments/calcsServiceExp1.json**).

|  |
| --- |
| **# ./compile\_results.py {FOLDER PATH} {PATH TO EXPERIMENT JSON}**  **./compile\_results.py ./memorySettingExperiment/combined ./experiments/calcsServiceExp1.json** |

This should generate a report such as the one shown below.

A picture containing window, large

Description automatically generated

Now that we can regenerate reports, this gives us the ability to create experiment files dedicated to formatting a report. Let's create a new experiment file to categorize this data.

|  |
| --- |
| **cd ./experiments**  **cp calcsServiceExp1.json report.json**  **nano report.json**  **cd ..** |

Edit the report.json file so that the ReportGenerator will create groups based on the **functionMemory** attribute.

Text

Description automatically generated

Once edited and saved, run the **report\_compiler.py** script again with the newly created **report.json** file.

|  |
| --- |
| **./compile\_results.py ./memorySettingExperiment/combined ./experiments/report.json** |

In the report you should now see aggregated categories for functionMemory. Alongside that, the results of each run should also be consolidated together in the report.

Table

Description automatically generated

**7. Creating Complex Experiments with Scripts**

Now that you have the ability to run multiple experiments and combine the results together into one report, we can create even more complex experiments with FaaS Runner. For the most complex experiments it is best to create a script that then invokes FaaS Runner. We can leverage many features of FaaS Runner to improve this process. Here is the process of what we want to experiment to do:

1. Vary the number of calculations (calcs) calcsService does between 1,000 and 100,000 in steps of 1,000 in each experiment run.

2. Repeat the experiment a second time to verify infrastructure is warm.

3. Change memory setting between 256 MBs, 512 MBs, 1024 MBs and 2048 MBs.

4. Repeat all the steps once again with memory stress (arraySize = 1,000).

We can create a bash script to easily create FaaS Runner arguments and execute this experiment. The script is included below. Review comments to see what arguments are being defined. Save and execute this script as **complexTest.sh** in the **test** directory. This experiment will take a few minutes to complete.

#!/bin/bash

# FaaS Runner Complex Experiment Example

# @author Robert Cordingly

# Define Experiment Arguments

args="--function calcsServiceTutorial --runs 100 --threads 100 --warmupBuffer 0 --combineSheets 0 --sleepTime 0 --openCSV 0 --iterations 2 --memorySettings [256, 512, 1024, 2048]"

# Create parent payload.

parentPayloadNoMemory="{\"threads\":2,\"sleep\":0,\"loops\":1000,\"arraySize\":1}"

parentPayloadMemory="{\"threads\":2,\"sleep\":0,\"loops\":1000,\"arraySize\":1000}"

# Generate scaling number of calcs payloads.

#

# This creates a list of payloads like this:

# [{"calcs":1000},{"calcs":2000},...,{"calcs":99000},{"calcs":100000}]

start=1000

step=1000

end=100000

payloads="["

for calcs in $(seq $start $step $end)

do

payloads="$payloads{\"calcs\":$calcs}"

if [ "$calcs" -lt "$end" ]

then

payloads="$payloads,"

else

payloads="$payloads]"

fi

done

# Create Output Folders

mkdir complexExperiment

mkdir complexExperiment/NoMemory

mkdir complexExperiment/Memory

# Run Experiments with and without Memory Stress

./faas\_runner.py -o ./complexExperiment/NoMemory --payloads $payloads --parentPayload $parentPayloadNoMemory $args

./faas\_runner.py -o ./complexExperiment/Memory --payloads $payloads --parentPayload $parentPayloadMemory $args

echo "Experiments Done!"

This script leverages FaaS Runner's payload inheritance. We first create a **parentPayload** that contains attributes that all function invocations in an experiment will use. In this case we create two parents, one with memory stress and one without. Then we create the **payloads** attribute to vary the number of calculations. This list of payloads will be distributed randomly between the threads. Finally, we define all other attributes in the **args** variable. This script also creates a few folders to keep our output organized. Unlike previous experiments, this experiment does not use any experiment or function files. Everything is defined through command line arguments and makes use of FaaS Runner's default parameters. For example, by default FaaS Runner assumes you are using AWS Lambda.

**Bonus:** Create a single report with all data from the complex experiment. In your **report.json** file add "**newcontainer"** and **"arraySize"** to the **outputGroups** list. Copy all json files from both the NoMemory and Memory folders into one combined folder. Run the **report\_compiler.py** script on the folder to generate the report.

**8. Using FaaS Runner with Function Pipelines**

Alongside running individual functions, FaaS Runner can execute complex pipelines of functions. To begin we must first explain the syntax. To execute a pipeline, you must define lists of functions and experiments. Like with single function calls, both functions and experiments can be defined through either files or command line arguments. For these examples we will use both.

Using the included function and experiment files. Try executing this experiment:

|  |
| --- |
| **./faas\_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json -e ./experiments/jello.json ./experiments/pello.json ./experiments/nello.json** |

Now let's explain what happened. The first experiment, in this case **jello.json**,is considered our parent experiment. This experiment file defines how many runs are going to be executed, the number of threads, and will be used to generate the report. In this case, this experiment file says that there will be 3 runs with 1 thread. In our output we saw a total of 9 function calls. For pipelines, the number of runs is runs of the entire pipeline. 1 Threads means that the pipeline was called sequentially so we saw responses come back in the expected order of Jello, Pello, Nello, Jello, Pello, Nello, etc. If we chose 3 threads, then 3 instances of the pipeline would run concurrently.

Table

Description automatically generated

Now take a look at the messages and payloads supplied to each function:

Since each experiment file defined payloads for the function, those payloads were used in the function invocation.

Instead of supplying a specific set of payloads to each function in the pipeline it may be necessary to pass the results from one function invocation to another.

Let's try and pass the response message from each function to the next, resulting in a final message of "Nello Pello Jello Jello"

**9. Command Line Arguments and Passing Attributes in a Pipeline**

FaaS Runner has the built-in attribute **passPayloads** that does just that! By default, this attribute is false so we can override that with command line arguments just like with single function experiments. Run the same experiment again but add the **"--passPayloads true"** flag.

|  |
| --- |
| **./faas\_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json -e ./experiments/jello.json ./experiments/pello.json ./experiments/nello.json --passPayloads true** |

Table

Description automatically generated

As we can see now ALL attributes returned by previous functions are passed onto the payload of the next function invocation. But our message response is still unchanged.

This is because the Hello World functions expects an attribute called "name" as the input and returns the response in the "message" attribute. Between function invocations we need to rename "message" to "name" to get the desired output we want.

To do this, we can use FaaS Runner's **transitions** attribute. This attribute expects a JSON object of key value pairs that will rename one attribute to another between function invocations. Like we did with passPayloads we can define this through command line arguments:

|  |
| --- |
| **./faas\_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json -e ./experiments/jello.json ./experiments/pello.json ./experiments/nello.json --passPayloads true --transitions {\"message\":\"name\"}** |

FaaS Runner is passing all attributes from the response of one function into the request of the next. While it does that, it renames the "message" attribute to "name" as defined by the transition attribute.

By default, when a command line argument is used to override something it applies it to ALL experiment/function files. If you want to only apply an argument to one specific function in the pipeline you can add array-style indexes to the argument (starting at 0). For example, if we want to do the same experiment but only pass arguments from the first function to the second, we can apply **passPayloads** only to the second function:

|  |
| --- |
| **./faas\_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json -e ./experiments/jello.json ./experiments/pello.json ./experiments/nello.json --passPayloads[1] true --transitions {\"message\":\"name\"}** |

This syntax can be applied to any attribute. If you want to have specific transitions between functions in a pipeline you can define that this way. This also allows complete pipelines to be entirely defined through command line arguments. For example, the same pipeline can be executed without using function or experiment files:

|  |
| --- |
| **./faas\_runner.py --function[0] jelloWorld --function[1] pelloWorld --function[2] nelloWorld --runs 3 --threads 1 --payloads [{\"name\":\"Jello\"}] --passPayloads true --transitions {\"message\":\"name\"}** |

**10. Dynamic Pipelines and State Machines**

For the most complex pipelines, FaaS Runner can be used to orchestrate function execution by modifying **test/tools/pipeline\_transition.py**.

def transition\_function(*index*, *functions*, *experiments*, *payloads*, *lastPayload*):

return (index + 1, functions, experiments, payloads, lastPayload)

This is the default transition function, after each execution increment the index to go to the next function; leaving the functions, experiments, and payloads unchanged. To better understand what data is being pass through here, add a few comments and rerun the previous pipeline:

def transition\_function(*index*, *functions*, *experiments*, *payloads*, *lastPayload*):

print("------------------ INDEX ------------------")

print(*str*(index))

print("------------------ FUNCTIONS ------------------")

print(*str*(functions))

print("------------------ EXPERIMENTS ------------------")

print(*str*(experiments))

print("------------------ PAYLOADS ------------------")

print(*str*(payloads))

print("------------------ LAST PAYLOAD ------------------")

print(*str*(lastPayload))

print("------------------------------------")

return (index + 1, functions, experiments, payloads, lastPayload)

**Bonus:** Using the experiment defined below, create a transition function that skips the 2nd function (pelloWorld) if the first function returns a message of "Jello End" otherwise execute the pipeline normally.

|  |
| --- |
| **./faas\_runner.py --function[0] jelloWorld --function[1] pelloWorld --function[2] nelloWorld --runs 10 --threads 1 --payloads [{\"name\":\"Jello\"},{\"name\":\"End\"}] --passPayloads true --transitions {\"message\":\"name\"} --shufflePayloads true** |

**Spoiler:** Solution. Copy and Paste text from box below to reveal.

def transition\_function(*index*, *functions*, *experiments*, *payloads*, *lastPayload*):

if (lastPayload["message"] == "Jello End"):

index += 1

return (index + 1, functions, experiments, payloads, lastPayload)