SPLAB

transcript

type:
tutorial

distribution:
public

status:
final

initiative:
Service Tooling

Companion Guide to Distributed Service Prototyping with Cloud Functions – Tutorial @ ICDCS 2018

Josef Spillner
Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/)
8401 Winterthur, Switzerland
josef.spillner@zhaw.ch

June 30, 2018

Learning Objective

Learn on a practical level about different FaaS programming conventions, runtimes, service providers and tools. Use some of the tools by yourself to "FaaSify" simple Python or Java applications and to execute and debug functions.

Note: This transcript contains ready-to-repeat commands for the first successful steps with functions in cloud environments. It is based on a previous one initially used for PyParis'17 (around 2 hours) and UCC'17 (half-day). The transcript has been extended in order to capture a larger set of tools suitable for a full-day tutorial. It is by no means complete; additions for future derived or specialised tutorials are welcome.

Contents

1 Code Transformation: Using Lambada

Lambada extracts your functions and methods from Python code files and turns them into cloud-hosted functions. To learn more about it, read the Lambada preprint [2] and its README file.

Obtain Lambada directly from its source repository. You will furthermore need Python 3.5 or more recent installed for which the assumed command name is `python3`.

Listing 1: Obtaining Lambada

```
git clone https://gitlab.com/josefspillner/lambada.git
cd lambada
```

Start by creating a file `temperature.py` with an editor of your choice (e.g. `vim`, `joe`). It contains one function and one class which together form your «temperature application» (Listing 2).

Listing 2: Temperature application

```
def temperature():
    return 42.23
class CityTemperature:
    def temperaturecity(self, city):
        temp = {"Vienna": 20, "Zurich": 21}
        return temp[city]
```

Now use Lambada to transform the contents into FaaS units which can still be executed locally but are ready to be deployed in AWS Lambda and compatible function runtimes (Listing 3).

Listing 3: FaaSification process

```
./lambada --local temperature.py
```

Then verify the generated Python file and notice its structure. Use any editor or file viewer of your choice (e.g. `less`) on the newly generated file `temperature_lambdafied.py`.

Afterwards, check the functionality in the Python console. Run the following commands (Listing 4; the console is indicated by `>>>`) to import the generated code as module which contains the original methods and generated ones.

Note: The method `temperature_remote` is the Lambda signature-compliant function, and the method `netproxy_handler` is equally Lambda signature-compliant and wraps method calls on objects whose attributes are transferred between the local proxy object and the FaaS-hosted remote object. It exists as single method along with the single class `Netproxy` for an arbitrary number of

classes. Thus, functions are statically rewritten (along with generated invocation code) whereas methods are dynamically proxied in a FaaS context, offering a suitable drop-in module to offload code to a FaaS environment.

Listing 4: FaaSified temperature application

```
python3
>>> import temperature_lambdafied as temperature
>>> dir(temperature)

>>> temperature.temperature()
>>> temperature.temperature_remote(None, None)
>>> temperature.netproxy_handler({"d": "{}", "name": "
    temperaturecity", "args": ["Winti"], "classname": "
    CityTemperature"}, None)
```

Finally, it is time to deploy the code to AWS Lambda and execute it there. You can run the command twice to verify that in the second case no redeployment happens (Listing 5). In parallel, you can verify the deployments in the AWS Lambda Dashboard at `https://<orgname>.signin.aws.amazon.com/console`.

Listing 5: FaaSified temperature application

```
./lambada temperature.py
./lambada temperature.py # nothing happens, already deployed
```

As these operations require an AWS account, you can alternatively deploy to a running Snafu instance; see Section 3. The commands would be amended with an endpoint configuration as follows. Note that Lambada still requires the `aws` command-line tool (AWS CLI) to be present in this case and it does have to be configured a priori with a region name (arbitrary) and credentials either matching the `snafu-control` accounts or as arbitrary values in case no authentication is required (i.e. without Snafu's `-a` parameter).

Listing 6: FaaSified temperature application using Snafu

```
./lambada --endpoint http://localhost:10000 temperature.py
```

For a more advanced use case, consider annotating the functions. Function-level annotations are called decorators in Python. You will need to provide a stub decorator implementation in your code which is then interpreted accordingly by Lambada. Only annotated functions are then selectively transformed and published. Assume you only want to export the temperature function but not an associated secret function, use the code below (Listing 7). Run the same commands as before but now include the `-annotations` switch as well to filter out the secret function.

Listing 7: Decorated temperature application

```
def cloudfunction(f): return f

@cloudfunction
```

```
def temperature():
    return 42.23
def secretfunction():
    return "secret"
```

Even more advanced is the parameterisation of the decorators to customise the runtime behaviour of functions. Consider the following example which assigns more memory than usually to the temperature function and allows it to run more time than the default timeout of 3 seconds in AWS Lambda (Listing 8).

Listing 8: Parameterised decorated temperature application

```
def cloudfunction(**kwargs):
    def real_cloudfunction(f):
        def wrapper(*args, **kwargs):
            return f(*args, **kwargs)
        return wrapper
    return real_cloudfunction

@cloudfunction(memory=512, duration=10)
def temperature():
    return 42.23
```

2 Code Transformation: Using Termite

Termite extracts your functions from Java code files. It has been designed to overcome several flaws of Podilizer while working similarly in some aspects so the Podilizer preprint might be worth a read [3].

A Git checkout is necessary to bootstrap the usage of Termite.

Listing 9: Obtaining Termite

```
git clone https://github.com/serviceprototypinglab/termite
```

Then, follow the essential README file instructions on compiling the Termite library and integrating it into the Maven build system of the application whose methods should be cloud-hosted. A first step is to try the tool on the example application. Notice that in addition to Maven's INFO lines, it will output TERMITE lines which signal that the handler for the annoations has been triggered. For each annotated method, Termite will compile and deploy to AWS Lambda. To deploy to a Lambda-compatible service, the example will have to be modified as described below.

Listing 10: Termite compilation

```
cd termite
mvn clean install
cd example
mvn clean install
```

To use Termite for your Java code, import it as a library or run it as standalone tool as before and make use of the annotations. Termite offers a `@Lambda` method annotation which may optionally take parameters to configure the execution behaviour as well as the endpoint to deploy to. The following code serves as example.

Listing 11: Termite code example

```
package app;

import ch.zhaw.splab.podilizerproc.annotations.*;

public class Test MyApp {
    public static void main(String[] args){
        mycloudmethod();
    }
    @Lambda(timeOut = 120, memorySize = 1024, endPoint = "
        http://localhost:10000")
    public static void mycloudmethod(String in){
        System.out.println("mycloudmethod:" + in);
    }
}
```

The endpoint specification makes it possible to deploy to for instance Snafu; see Section 3. Please note however that Termite generates quite complex Java packages which are not entirely understood by Snafu. Hence, the deployment will work but execution will most likely not.

3 Controlled Execution: Using Snafu

Snafu runs your functions in code files. A preprint is available [1] as are studies about its runtime characteristics [4].

Start off with obtaining Snafu from its source repository, or for potentially outdated versions from the Python Package Index or Docker Hub. The following instructions assume an installation from the repository.

Listing 12: Obtaining Snafu

```
git clone https://github.com/serviceprototypinglab/snafu
cd snafu
# pip install snafu
# docker run -ti jszhaw/snafu bash
# ... for more installation options (OpenShift, Kubernetes,
    Helm) see README
```

You can directly run Snafu without any configuration. It ships with a number of default functions in various languages. Notice how on first startup, the interpreters for non-scripted functions are compiled first. This means that while the first startup is a bit slow, the second and subsequent ones are quote fast

(Listing 13). The behaviour depends on what is installed on the system. Sporadic error messages may appear for missing dependencies (e.g. Java compiler) but can be ignored unless the specific functions are needed.

Listing 13: Running Snafu

```
./snafu
# type Ctrl+D
./snafu # everything compiled
# type Ctrl+D again
```

Snafu chooses one parser and one executor per function implementation by default but this can be overridden. Now choose the C executor, which is anyway the default, to demonstrate the behaviour. Select the sample C Fibonacci function and execute it with a low parameter n (Listing 14).

Listing 14: Executing a C function

```
./snafu -e c
# type function: fib_so.handler
# type argument: 10
# type Ctrl+D
```

Now remember your previous example functions used with Lambada. Obviously, Snafu can interpret the original file as well by performing some (limited) FaaSification on its own. Execute Snafu on this file and explore the execution (Listing 15).

Listing 15: Executing a Python function

```
./snafu .../path-to-lambada/temperature.py
```

For advanced FaaSification, the Lambada parser can be used which requires Lambada to be installed as Python module. In this case, the classes are transformed as well.

Listing 16: Executing a Python function with Lambada parser

```
./snafu -f lambada .../path-to-lambada/temperature.py
```

For testing functions prior to deployment to commercial cloud providers (AWS, IBM, Google etc.) it is convenient to use Snafu's control plane mode. In this mode, the default calling convention is the same as in AWS Lambda (Listing 17) although the behaviour can be tuned to be closer aligned by adding authentication and container isolation (`-a aws -e docker`) which may require additional configuration upfront.

Listing 17: Using the control plane

```
./snafu-control .../path-to-lambada/temperature.py
# notice how no methods are selectable
# type Ctrl+D
```

```
# add to source: def lambda_handler(event, context):return
    str(temperature())
# & run snafu-control command again
# open a second terminal, and run:
aws --endpoint-url http://localhost:10000 lambda invoke --
    function-name test.lambda_handler --payload '{"event":
    ""}' /tmp/_out
cat /tmp/_out
```

4 Controlled Execution: Using OpenFaaS

Note: OpenFaaS works on the basis of Docker Swarm.

Listing 18: OpenFaaS script

```
# preparation
docker swarm init
curl -sL cli.openfaas.com | sh
git clone https://github.com/openfaas/faas
cd faas
./deploy_stack.sh
cd ..
# retrieve echo template function
./faas-cli template pull
./faas-cli new --lang python3 hello-openfaas
# customise function as needed here
# build + execute function
./faas-cli build -f hello-openfaas.yml
./faas-cli deploy -f hello-openfaas.yml
curl http://127.0.0.1:8080/function/hello-openfaas -d Test
```

5 Controlled Execution: Using Fission

Note: You will need to have Kubernetes installed with minikube + kubectl + helm for a local installation. Ensure that the minikube context is used by kubectl. Furthermore, ensure that your system user is in the group `docker`.

Listing 19: Fission script

```
# prepare for local installation
kubectl config use-context minikube
helm install --namespace fission https://github.com/fission/
    fission/releases/download/0.8.0/fission-core-0.8.0.tgz
curl -Lo fission https://github.com/fission/fission/releases
    /download/0.8.0/fission-cli-linux && chmod +x fission
# initiate
./fission env create --name python --image fission/python-
    env
```

```
curl https://raw.githubusercontent.com/fission/fission/
    master/examples/python/hello.py > hello.py
# explore & deploy
cat hello.py
./fission function create --name hello --env python --code
    hello.py
# output: function 'hello' created
./fission route create --method GET --url /hello --function
    hello
# output: trigger '35ad2fae-****-****-****-679d755fde8f'
    created
# curl http://`sudo minikube ip`:31314/hello # ceased
    working in new version
./fission function test --name hello
```

6 Controlled Execution: Using OpenLambda (tentative)

Listing 20: OpenLambda script

```
git clone https://github.com/open-lambda/open-lambda
cd open-lambda
sudo apt-get install libpython2.7-dev
sudo make
bin/admin new -cluster pyparis
# returns lots of JSON
bin/admin workers -cluster pyparis
# output: Started worker: pid ...
bin/admin status -cluster pyparis
# output: Worker Pings: ...
cat quickstart/handlers/hello/lambda_func.py
# ... minimal Lambda-signature Python function
cp -r quickstart/handlers/hello pyparis/registry
curl -X POST localhost:8080/runLambda/hello -d '{"name": "
    Alice"}'
# -> permission denied!?
```

7 Outsourced Execution: Using AWS Lambda

The Lambda service provided by Amazon Web Services allows for invoking function instances with various memory allocations between 128 MB and 1536 MB and with memory-proportional performance.

The prerequisite for using Lambda is an AWS account. You will need to register with a credit card here: `https://aws.amazon.com/de/lambda/`.

Once registered, create a new function via the graphical web interface or via the command-line interface. This tutorial guides through the second path. Ensure to have the command-line interface (AWS CLI) installed and properly configured for your account; verify the instructions on `https://aws.amazon.com/de/cli/` if in doubt.

Then, write your first function. Note that depending on the programming language, Lambda requires a certain convention for functions, methods and parameters. The following example shows a Python function which adheres to the conventions.

Listing 21: AWS Lambda function example

```
def lambda_handler(event, context):
  return 99
```

Save the function to a Python file. Assuming its name is `first.py`, the following command uploads it to Lambda. For Python, both version 2.7 and 3.6 are available.

Listing 22: AWS Lambda function upload

```
zip first.zip first.py
role=‘aws sts get-caller-identity --output text --query '
    Account’‘
aws lambda create-function --function-name First --
    description 'my first function' --runtime 'python2.7' --
    role $role --handler 'first.lambda_handler' --zip-file '
    fileb://first.zip'
```

Afterwards, the function execution can be tested.

Listing 23: AWS Lambda function execution

```
aws lambda invoke --function-name First --payload '{}'
    _logfile
cat _logfile
rm _logfile
```

8 Outsourced Execution: Using IBM Cloud Functions

Based on Apache OpenWhisk (formerly IBM OpenWhisk), the Cloud Functions service is the IBM Cloud (formerly Bluemix)-integrated facility to execute functions offered by IBM.

The prerequisite for using Cloud Functions is an IBM Cloud account which is an IBMid account. A 30-day trial is offered without requiring a credit card. The registration is possible here: `https://console-regional.ng.bluemix.net/registration/?target=%2Fopenwhisk`.

Alternatively, an existing account can send invites for collaborators by e-mail. Accept the invite, sign up with your full name and a password, and sign in. Select the Offerings menu and choose Functions. You may only have access to a space in a certain region and therefore need to change the region setting first. Afterwards, you can choose Create Action to get started. Example: Location - Germany, space - ICDCS18-Tutorial.

In IBM Cloud Functions, functions can be deployed and executed programmatically or through the graphical web interface. Functions adhere to the conventions set by OpenWhisk. A typical function may look like the following example given in Python 3:

Listing 24: OpenWhisk Python function example

```
def main(dictionary):
  return {"response": 99}
```

Listing 25: OpenWhisk JavaScript function example

```
function main(input) {
    return {"upper": input["text"].toUpperCase(),
            "lower": input["text"].toLowerCase()}
}
```

Using the `bx` or `wsk` tools, OpenWhisk as offered by IBM Cloud Functions can be used programmatically while vendor support for the latter has ceased in March 2018, effectively diverging from stock OpenWhisk tooling in particular for the login. The `wsk` tool can still be used after login.

Listing 26: Working with OpenWhisk

```
# Installation of bx tool and login
wget https://clis.ng.bluemix.net/download/bluemix-cli/latest
    /linux64
tar xvf linux64
cd Bluemix_CLI/ && ./install_bluemix_cli
bx plugin install cloud-functions -r Bluemix
bx login -a api.eu-de.bluemix.net -o "ZHAW ISPROT" -s "
    ICDCS18-Tutorial"
# fill out e-mail and password, check for OK and system
    information
bx wsk action invoke /whisk.system/utils/echo -p message
    hello --result
# Installation of wsk tool
wget https://github.com/apache/incubator-openwhisk-cli/
    releases/download/latest/OpenWhisk_CLI-latest-linux-amd64
    .tgz
tar xf OpenWhisk_CLI-latest-linux-amd64.tgz
./wsk action invoke /whisk.system/utils/echo -p message
    hello --result
```

References

[1] J. Spillner. Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation. arχiv:1703.07562, March 2017.

[2] J. Spillner. Transformation of Python Applications into Function-as-a-Service Deployments. arχiv:1705.08169, May 2017.

[3] J. Spillner and S. Dorodko. Java Code Analysis and Transformation into AWS Lambda Functions. arχiv:1702.05510, February 2017.

[4] J. Spillner, C. Mateos, and D. A. Monge. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. In *4th Latin American Conference on High Performance Computing (CARLA)*, September 2017. To appear.