A look into server-less containerized services

CPS831 - Software Engineering 2

Rudi Zhou
rudi.zhou@ryerson.ca
Ryerson University
Toronto, Ontario

Jeremy Sirois
jsirois@ryerson.ca
Ryerson University
Toronto, Ontario

1 ABSTRACT

The purpose of this research paper is to look into the world of containers and micro-services. For our project we have decided to use a mixture of containers such as Docker and cloud services such as AWS Lambda to create containerized services where token are required to run these services. We created a simple text-based hangman game where user guesses the mystery key word by guessing letters that fill out the characters of the word in the string with a set amount of tries. This program acts as the primary service and a user must "pay" for this service by requesting a token from AWS and validating it against another containerized service.

Resources found at: https://github.com/sirois/cps831

2 INTRODUCTION

The development of virtualization has been around since the 1960's. As research and hardware capabilities progressed, Virtual Machines (VM) became of of the standard tools used to isolate different workloads and environments. Two different types of Virtual Machine Managers were developed; Type 1 and Type 2. However one of the issues with the VM's were the hardware requirements and the time it took to start them up. As the concept and idea of micro-services began to dominate the way developers chose to build out their applications, the cons of VM's became a bigger issue when deploying these services. However at the same time, the development of a new tool called Docker began by Google in 2013 and it quickly began to rise in popularity. Docker provided the ability to package containers so that they could be moved from one environment to another. The concept of containers provided a much needed portability and light-weightiness to it that was needed for deploying a micro-service based architecture. Hence as of today, containers have now become an industry standard and are one of the most preferred options for virtualization in various tech shops across the world.

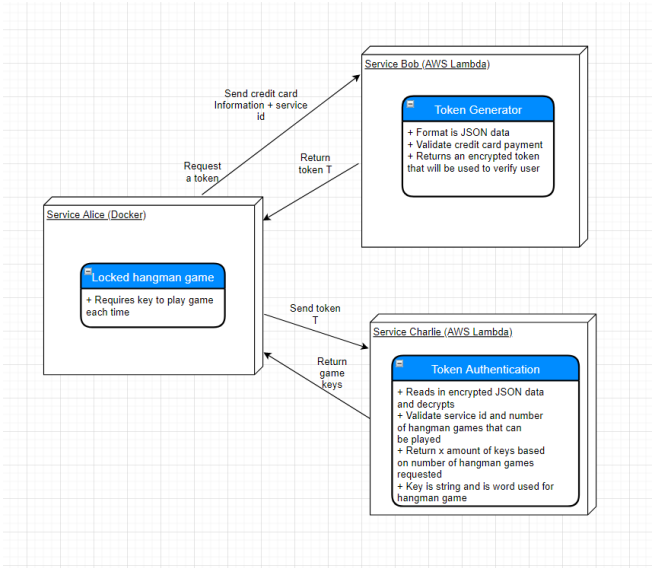
3 SET UP

Here are our initial setup used for our research:

- We used Docker containers services that will either request or provide the service which will be a text based hangman game
- AWS is cloud service used for the third party payment / token authentication service
- AWS Lambda is used for server less deployment for encrypting and decrypting token (separate service for each part)
- As proof of concept, we have used a simple text-based hangman game as the core service being offered

4 ARCHITECTURE

As explained in bits of the abstract and set up, the core architecture of this configuration is the use of Docker and AWS. There is a Docker containerised service that we will name Alice. Alice contains a locked service and requests access to it by the use of a authentication token. In this paper we created a text-based hangman game in Java to act as the primary service. For the AWS third party service, we have created a Lambda function that takes in a request message from Alice and returns an encrypted JSON data with one portion of the encrypted data being the access/authentication token. Finally the AWS service which also runs a Lambda function is a service named Charlie which is used to decrypt then validate the token and return String key(s) to start the hangman game.



As seen on the figure above there is process that must be followed before a user would be allowed to access the hangman game. The service Alice first make a payment with their credit card which requests a authentication token from another service Bob who is a third party payment/token authentication service (AWS Lambda in this case). Bob generates a token via JSON data then encrypts the body of the JSON (a key named token has authentication information) and returns it over to Alice. Next the service Alice needs

to provide this requested token to another AWS service Charlie. Charlie receives the encrypted JSON data and decrypts the data and validates that the token is a valid one and returns a String keyword that is to be used for the hangman game. This process repeats allowing the user to play the hangman game multiple times depending on how much was paid for by the user in the service Alice. As you can see this hangman game is simply a proxy service and can be substituted for any service that is needed in the software industry.

5 HANGMAN GAME

For our hangman game we decided to code it in Java and integrate it with AWS Lambda by making use of its HTTP request messages and sending/receiving data via REST API's using JSON as the data format. For the hangman game code we referenced some code from the following:

https://gist.github.com/SedaKunda/79e1d9ddc798aec3a366919f0c14a078

Below is a snippet of the code used for the hangman game. In this game we request a key/token from the AWS Lambda (service Bob) before the user is allowed to play the game. Once token is generated Alice needs to request another key from Service Charlie before the hangman game can start. It should return a String token and that string token is then used for the hangman game as the answer (ex. a string token could be "horse"). Once the token authentication is finished from AWS side and key has been received then the game starts for service Alice and this process repeats depending on how many "games" the user at service Alice has paid for.

```
//sample of payment code PrintWriter out = null;
BufferedReader in = null;
String result = "";
try {
URL realUrl = new URL("https://4lrr2rrik3.execute-api.us-
east-1.amazonaws.com/default/cps831");
// build connection
URLConnection conn = realUrl.openConnection();
// set request properties
conn.setRequestProperty("accept", "*/*");
conn.setRequestProperty("connection", "Keep-Alive");
conn.setRequestProperty("user-agent", "Mozilla/4.0
(compatible; MSIE 6.0; Windows NT 5.1; SV1)");
// enable output and input
conn.setDoOutput(true);
conn.setDoInput(true);
out = new PrintWriter(conn.getOutputStream());
// send POST DATA
out.print("id: " + name +", creditcard: " + cc + ", games: " +
games);
out.flush();
in = new BufferedReader(new InputStream-
Reader(conn.getInputStream()));
String line;
while ((line = in.readLine()) != null) {
result += "/n" + line;
}
catch (Exception e) {
e.printStackTrace();
}
```

```
//sample of code to start hangman game
try {
URL realUrl = new URL("https://tltga3b731.execute-api.us-
east-1.amazonaws.com/default/cps831Charlie");
// build connection
URLConnection conn = realUrl.openConnection();
// set request properties
conn.setRequestProperty("accept", "*/*");
conn.setRequestProperty("connection", "Keep-Alive");
conn.setRequestProperty("user-agent", "Mozilla/4.0 (com-
patible; MSIE 6.0; Windows NT 5.1; SV1)");
// enable output and input
conn.setDoOutput(true);
conn.setDoInput(true);
out = new PrintWriter(conn.getOutputStream());
// send POST DATA
out.print("token:" + token + "
");
out.flush();
in = new BufferedReader(new InputStream-
Reader(conn.getInputStream()));
String line;
while ((line = in.readLine()) != null) {
result += "/n" + line;
}
} catch (Exception e) {
e.printStackTrace();
}

int plays = Integer.parseInt(result.substring (re-
sult.indexOf("Games") +6,result.indexOf("headers")-9));

for(int x=0;x<plays;x++){
System.out.println("Game: " + (x+1));
Hangman h = new Hangman();
h.mainHang();
System.out.println();
}
```

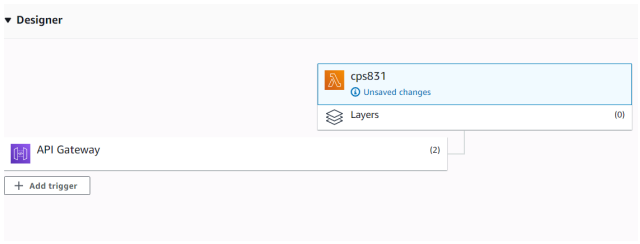
```
FROM alpine
WORKDIR /root/gameworld
COPY GameWorld.java /root/gameworld
COPY Hangman.java /root/gameworld
COPY Game.java /root/gameworld

#Install JDK
RUN apk add openjdk8
ENV JAVA_HOME /usr/lib/jvm/java-1.8-openjdk
ENV PATH $PATH:$JAVA_HOME/bin

#compile and run
RUN javac Game.java GameWorld.java Hangman.java

ENTRYPOINT java GameWorld
```

6 AWS AND DOCKER

Build and deployment process - Click Here

The game service was done in Java and packaged using Docker. The docker file specifies game files used, Java Runtime need, and compilation parameters. The end user does not need to have any of these runtimes or file on their system. When provided with a build, the end user will simply run the built container.

For the AWS portion, as stated before we had used the Lamda service for server less deployment and created a function to generate authentication using tokens. For the AWS configuration, we needed to add a API Gateway to communicate with the application in Docker. In AWS this is handled by adding a trigger to the lambda configuration and requires separate triggers and API gateways for different Lambda services. For the purpose of this research paper, security was not a primary concern and therefore there wasn't a custom built authentication and we simply just used the auto generated AWS API keys to access HTTP requests. The great thing about AWS API Gateway is that there are many authentication options that can be used if security is a bigger issue in your application. The function to generate the API token can be referenced below and was written in Node.JS as it's the default supported language in AWS Lambda. In service Alice we request a token by sending in a HTTP POST with some JSON data over to the API Gateway. If the formatted JSON data contains valid and correct information then the function creates a new authentication token, restructures the JSON data and finally encrypts the JSON data using the crypto package found in the NodeJS library. One of the interesting things about AWS Lambda is that it does not support HTTP requests from applications outside the AWS eco-system such as Postman or our own created Docker services and manual configuration is needed. In order to make this configuration happen, we needed to access the AWS API Gateway service its self and add custom configurations on how to handle POST/GET responses. This involved changing the integration request and adding a mapping template for a particular datatype (we used JSON but other data formats like XML work too) since AWS Lambda uses the Event Tests as data format by default.



Here is a snippet of code for the AWS Lambda function we used to generate the token and encrypt the final JSON data to be returned back to the Docker service. This service would represent service Bob.

```
let body;
let statusCode = '200';
const headers = {
    'Content-Type': 'application/json',
};

try {
    switch (event.httpMethod) {
        case 'POST':
            //body =  "ID:" + JSON.parse(JSON.stringify(event.id)) + ", " + "GAMES:" +  JSON.parse(JSON.stringify(event.games))
            body =  "ID:" + event.id + "," + "Games:" + event.games;
            break;

        case 'GET':
            body =  "ID:" + event.id + "," + "Games:" + event.games;
            break;

        default:
            body =  "ID:" + event.id + "," + "Games:" + event.games;

    }
}

catch (err) {
    statusCode = '400';
    body = err.message;
}

finally {
    body = JSON.stringify(body);
    var cipher = crypto.createCipher('aes-256-cbc','d6F3Efeq')
    var crypted = cipher.update(body,'utf8','hex')
    crypted += cipher.final('hex');
    body = crypted;
}
```

Here is some sample code for the AWS Lambda functions we used to decrypt the JSON data sent by the Docker service and validate the token. If validation is successful then it will return JSON data with a String key that contains the word for the hangman game. This service would present service Charlie.

```
loadToken = event.token;
var decipher = crypto.createDecipher('aes-256-cbc','d6F3Efeq')
var dec = decipher.update(loadToken,'hex','utf8')
dec += decipher.final('utf8');


try {

    data = dec.split(',');
    id = data[0].split(':');
    games = data[1].split(':');

    switch (event.httpMethod) {
        case 'POST':
            //body =  "ID:" + JSON.parse(JSON.stringify(event.id)) + ", " + "GAMES:" +  JSON.parse(JSON.stringify(event.games))
            body =  "ID:" + id[1] + "," + "Games:" + games[1];
            break;

        case 'GET':
            body =  "ID:" + id[1] + "," + "Games:" + games[1];
            break;

        default:
            body =  "ID:" + id[1] + "," + "Games:" + games[1];

    }
}
```

7 CONCLUSION

In conclusion, we have successfully created a architecture where multiple containerized services can be interact with each via tokens generated by a third party via AWS. We had 2 dockerized services in Alice and Charlie where one micro service would request to play a text-based hangman game by paying a certain amount. Payment validation was conducted through service Bob via AWS Lambda functions and it spit back parsed JSON objects to authenticate the tokens. Charlie would finally send back a certain amount of hangman games by returning a string to be used for the hangman game depending on the amount paid for by service Alice. As seen in this architecture, using Docker containers along with public cloud service offerings is very easy and we can clearly see why the industry is shifting towards cloud and micro-based architectures using containerized services. Most of the challenging work is handled

by services such as AWS and Docker and once its set up, one can easily just code and then deploy automatically with these services!

8 FURTHER IMPROVEMENTS

As stated in the original abstraction we decided to use Docker and AWS Lambda as the primary tools for our experiments. If we had the resources and time, we would have preferred to look into more tools especially in the field on containers such as Kubernetes and how it could be implemented into this project.