Lab 3: Deploying a Circuit Breaking ambassador and a Function-as-a-Service (FaaS)

Objective:

1. Learn how to create a service using NodeJS.
2. Learn how to create a Docker image.
3. Learn how to configure and use a circuit breaker using Nginx.
4. Get Familiar with FaaS.
5. Learn how to configure and use OpenFaas on GCP
6. Get Familiar with Decorator Pattern

Repository: (Contains all the files created within the lab):

https://github.com/GeorgeDaoud3/SOFE4790U-lab3

Part 1.

Read the following document (Health Endpoint Monitoring pattern - Azure Architecture Center | Microsoft Learn). Focus on the problem being solved by the pattern, how is it solved? and the requirements needed for the solution.

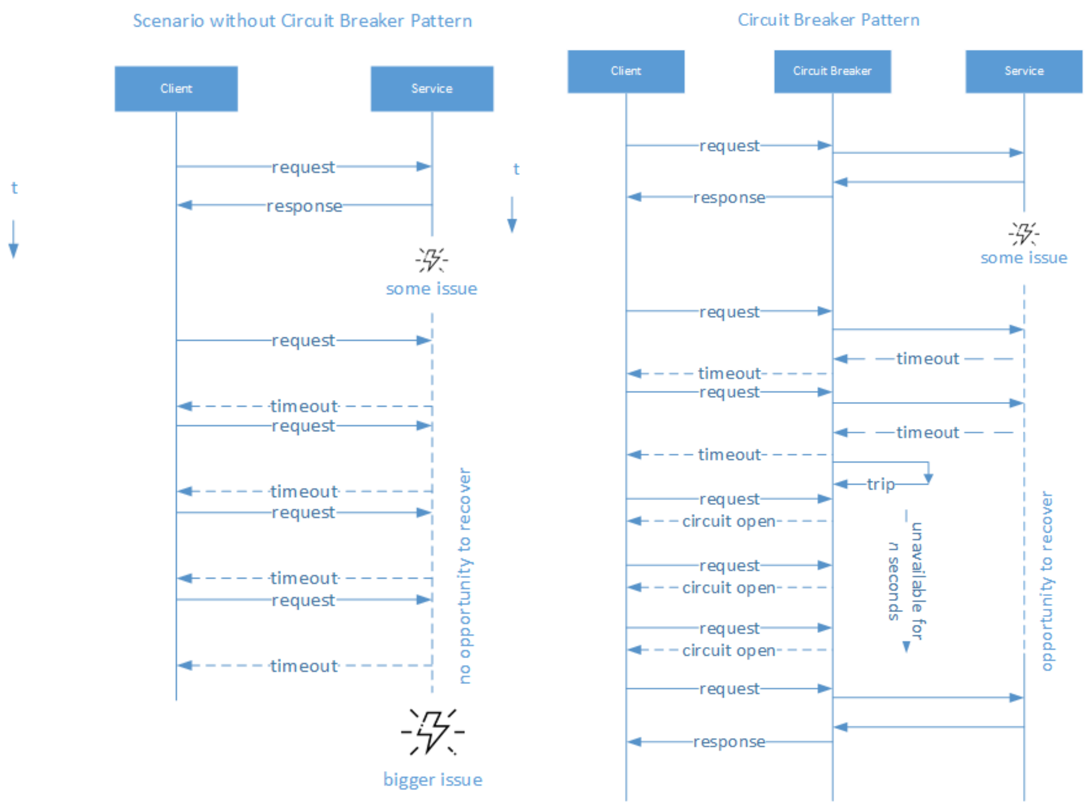
Part 2.

In this lab, we'll guide you through the steps to deploy a Circuit Breaker pattern as an ambassador.

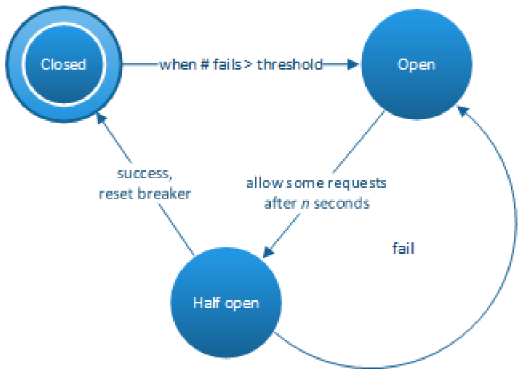
A key tenet of modern application design is that failure will occur. You need to assume that one or more parts of your application will fail in some manner at some point. The **Circuit Breaker** pattern can help you prevent, mitigate, and manage failures by:

- allowing failing services to recover, before sending them requests again
- re-routing traffic to alternative data sources
- rate limiting

Without a **Circuit Breaker**, the **Client** will keep trying to send a request to the **Service**, even after something went wrong and the **Service** isn't able to respond in time. Because the **Client** keeps retrying, possibly with many other clients simultaneously, the **Service** will have no opportunity to recover, assuming the issue at hand is only transient.



With a Circuit Breaker, the **Client** will no longer be able to overload the **Service** with requests after the **Circuit Breaker** was "tripped" after the second failure. The **Circuit Breaker** will avoid sending the **Service** any new requests for the next *n* seconds. This allows the **Service** to recover. Once a certain time has passed, the **Circuit Breaker** will slowly start passing on requests to the **Service** again until it has fully regained strength, as illustrated in the following State Diagram:



Check out these pages if you want to read more about the Circuit Breaker Pattern:

- Circuit Breaker - Martin Fowler
- NGINX Circuit Breaker Pattern

Procedure:

1. If you haven't yet created a GKE cluster, follow the instruction given in Lab 1 and create one.
2. Clone the lab repository

```
cd ~
git clone https://github.com/GeorgeDaoud3/SOFE4790U-lab3.git
```

   where **~** means the home directory within the GCP local terminal

3. Let’s start by creating an image for the Dummy Service. In this lab, we will not use a pre-made image, but we will our own.
   a) Use the GCP text editor to read `server.js` from the following path **~/SOFE4790U-lab3/part2/DummyServiceContainer**

   It’s a NodeJS service that consists of five endpoints. Each is handled by a function. The first is /alive endpoint which is used by Kubernetes to check the status of the container and rebuild it whenever fails. Thus, it always returns **OK** (code: 200). The second function responds to /ready endpoint, and according to a Boolean variable, it may return **Service Unavailable** (code: 503) or **OK**. This Boolean variable that mimics an error can be set or removed using two other endpoints: /fakeerrormodeoff and /fakeerrormodeon. While the fifth endpoint is the main (/) one and returns **OK** or **Service Unavailable** after a certain delay according to the same Boolean variable.

   b) Now, read **Dockerfile** from the path **~/SOFE4790U-lab3/part2/DummyServiceContainer.**

   Dockerfile is the standard name of the file used to create Docker images. It contains instructions on how the image is created. By skipping the comments which are the line started with **#**. It contains 7 instructions. The first one will inform it to use an already existing image that contains a Linux OS with NodeJS installed on it as a base image. The second one sets the current working directory to a certain path within the base image. The third command copies all **JSON** files whose names started with the prefix **package** from the local directory (**~/SOFE4790U-lab3/part2/DummyServiceContainer**) to the container active directory (**/usr/src/app**). While the fourth one installed all required dependencies as a list in **package.json.** The service code and all other files will be copied from the local directory to the container according to the fifth command. The second instruction sets the active port that should be exported from the container to the outside world. The last line contains the command to start the service.

   c) To execute those instructions and create the first image we have to navigate to the **DummyServiceContainer** directory and run a Docker build command

   ```
   cd ~/SOFE4790U-lab3/part2/DummyServiceContainer
   docker build . -t us.gcr.io/<Project-ID>/dummyservice
   ```
   The project ID is usually highlighted in the terminal prompt and can be listed by using the command.
   ```
   gcloud projects list
   ```
   d) Now, we have an image but it’s still in the local machine provided by the GCP terminal and can’t yet be used by Kubernetes. So, we have to push it first to the GCP container hub
   ```
   docker push us.gcr.io/<Project-ID>/dummyservice
   ```
   Now, we can use it in Kubernetes deployment.
4. To deploy the Dummy service via GKE,
   a) We have to edit the **~/SOFE4790U-lab3/part2/dummy-deployment.yaml** file first by setting the image name in line 20 to **us.gcr.io/<Project-ID>/dummyservice.**
   b) After saving the changes, let’s browse the file and try to highlight important parts.
   It starts with the deployment instructions of the image into a single pod as there is only one replica. Lines 23 to 30 include the script that configured Kubernetes how to check the status of the service whether it’s alive or not. This is done through the /alive endpoint

through the service via a GET HTTP request via port 80. It starts after 5 seconds of the deployment and is repeated every 10 seconds after that.

c) Change the directory to the YAML file location. Then, deploy the service
```
cd ~/SOFE4790U-lab3/part2/
kubectl create -f dummy-deployment.yaml
```
d) Then, expose it via a load-balancing GKE service
```
kubectl expose deployment dummy-deployment --port=80 --type=LoadBalancer --name
dummy-deployment
```
e) Verify that the pod, the deployment, and the service are available and running

5. To deploy the backup service via GKE,
   f) edit the **~/SOFE4790U-lab3/part2/backup-deployment.yaml** file by setting the image name in line 20 to **us.gcr.io/<Project-ID>/dummyservice.**
   g) Save the changes, Then change the directory and deploy the service
   ```
   cd ~/SOFE4790U-lab3/part2/
   kubectl create -f backup-deployment.yaml
   ```
   h) Then, expose it via a load-balancing GKE service
   ```
   kubectl expose deployment backup-deployment --port=80 --type=LoadBalancer --name
   backup-deployment
   ```
   i) Verify that the pod, the deployment, and the service are available and running
6. Configure the circuit breaker
   a) We will use **~/SOFE4790U-lab3/part2/nginx-configmap.yaml** to create a ConfigMap. It has extra configurations more than those used in the previous lab. In line 13, one of the servers is marked as a backup. While lines 15 through 17 sets the range of returned status from 200 to 399 to be considered OK. The last component (lines 18: 23) configures the main endpoint (/) to be the health-check endpoint that is checked every 3 seconds and considers it a failure after a repeated failure threshold of 1. The passing threshold is also set to 1.
   ```
   cd ~/SOFE4790U-lab3/part2/
   kubectl create -f nginx-configmap.yaml
   ```
   b) The deployment and service are configured via ~/SOFE4790U-lab3/circuitbreaker.yaml
   ```
   cd ~/SOFE4790U-lab3/part2/
   kubectl create -f circuitbreaker.yaml
   ```
   c) Get the external IP address of the circuit breaker
7. Test your Circuit Breaker
   a) Execute the following command 3 times
   ```
   curl -v http://<circuit-breaker-IP>
   ```
   Check the IP of the local machine in the last line in the response.
   b) Mimic an error by running
   ```
   curl -d "" -s -D - http://<dummy-service-IP>/fakeerrormodeon
   ```
   c) Execute the following command
   ```
   curl -v http://<circuit-breaker-IP>
   ```
   Check the IP of the local machine in the last line in the response.
   d) Reset the error by running
   ```
   curl -d "" -s -D - http://<dummy-service-IP>/fakeerrormodeoff
   ```
   e) Execute the following command
   ```
   curl -v http://<circuit-breaker-IP>
   ```
   Check the IP of the local machine in the last line in the response.

Part 3.

In this lab, we will use the Decorator Pattern to implement a Function that adds default values and performs transformations to the input of an HTTP RESTful API.

Procedure:

1. Watch this video to learn what Function as a Service (FaaS) https://www.youtube.com/watch?v=EOIja7yFScs
2. The framework that will be used in that lab to deploy FaaS, is OpenFaaS. The first step is to setup it up in GKE.
   a) First, create a cluster admin role binding
   ```
   kubectl create clusterrolebinding "cluster-admin-$(whoami)" \
           --clusterrole=cluster-admin \
           --user="$(gcloud config get-value core/account)"
   ```
   b) Finally, deploy OpenFaas to GKE
   ```
   curl -SLsf https://dl.get-arkade.dev/ | sudo sh
   arkade install openfaas --load-balancer
   ```
   c) Install the OpenFaaS-cli
   ```
   curl -SLsf https://cli.openfaas.com | sudo sh
   ```
   d) To verify that OpenFaaS has started, run:
   ```
   kubectl -n openfaas get deployments -l "release=openfaas, app=openfaas"
   ```
   The option **-n** is used to specify a certain namespace which is a way to organize Kubernetes components. All OpenFaas-related pods, deployments, services,… are within **openfaas** namespace.
3. Log in to the OpenFaaS gateway
   a) First, check that OpenFaaS is ready
   ```
   kubectl rollout status -n openfaas deploy/gateway
   ```
   b) Then get its IP address
   ```
   kubectl get svc -o wide gateway-external -n openfaas
   ```
   The IP address will be used to log to the OpenFaaS gateway (within port 8080) and also to invoke deployed functions.
   c) Let's get the password and login into the GateWay
   ```
   export OPENFAAS_URL="<OpenFaaS-GateWay-IP>:8080"
   PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode; echo)
   echo $PASSWORD
   echo -n $PASSWORD | faas-cli login --username admin --password-stdin
   ```
   The first line will save the gateway address in a shell variable called **OPENFAAS_URL**. Whenever you need to access this variable, it would be in the form **$OPENFAAS_URL**. Like printing its value by **echo $OPENFAAS_URL** command. The next command will store the gateway password in another shell variable named **PASSWORD**. While the third command will display the password to you. The last line will save the credentials within faas-cli, Thus you will not have to provide it again.
   d) You still can logged into the gateway using its UI interface by navigating into **http://<OpenFaaS-GateWay-IP>:8080/ui/**. The username is **admin** and the password will be the same as you got in c).

   Congratulations, you have successfully installed OpenFaaS!!!
4. **Note:** The terminal provided by GCP is a temporary virtual machine. Any installed software like **faas-cli** will be lost when the connection is lost. So, every time you logged in or reconnected to the terminal you have to repeat the following commands

```
curl -SLsf https://cli.openfaas.com | sudo sh
export OPENFAAS_URL="<OpenFaaS-GateWay-IP>:8080"
PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode; echo)
echo $PASSWORD
echo -n $PASSWORD | faas-cli login --username admin --password-stdin
```

5. Now let's start deploying the first function.
   It would be the Main function that will store shapes in a database. A shape will be defined by a JSON object with the attributes: Name, Color, and Number of dimensions. You should be able to call the Main function with a parameter like one of these JSON objects:
   {"Name":"Square","Dimensions":2,"Color":"Red"}
   {"Name":"Triangle","Dimensions":2,"Color":"Green"}
   Which will store in some persistence layer, although the implementation of actually storing the object is out of scope for this lab. Thus, the function only will return the received JSON object.
   a) First, let's create a directory for that function and pull all the available templates and list them

```
mkdir ~/OpenFaaS
cd ~/OpenFaaS
faas-cli template pull
faas-cli new --list
```

   b) Now, using the template already downloaded in the folder, we will create an empty NodeJS function

```
faas-cli new --lang node12 --prefix us.gcr.io/<project-id> main
```

   This will create a blank NodeJs function named **main** using NodeJS v12. The prefix is used in creating a YML file for deploying onto OpenFaas. The image name within the YAML file would be us.gcr.io/<project-id>/main:latest. This command will create the following files

   - main.Yaml: yml file for deploying on OpenFaas
   - main/handler.js : function source code
   - main/package.son: NodeJS metadata which includes required dependencies (libraries) to be installed.

   c) Update the main/handler.js file to

```
'use strict'

module.exports = async (event, context) => {
  var parameters=JSON.stringify(event.body)
 return context
  .status(200)
  .succeed(parameters)
}
```

   The first statement within the function converts the input parameters **event.body** from the JSON object into a string. Then, the function ends by returning **OK** (200) with the parameters. As there are no dependencies needed

   d) To build the docker image, use the following command

```
cd ~/OpenFaaS
faas-cli build -f main.yml
```

   e) Then push it to Google Container Registry

```
docker push us.gcr.io/<project-id>/main
```

f) Finally, deploy it to OpenFaaS

```
faas-cli deploy -f main.yml
```

Now, the function is deployed. You can check its status to be ready from the UI of the gateway

g) Invoke it and send the JSON object {"Name":"Square", "Color": "Red", "Dimensions": 2 }

```
curl http://<OpenFaaS-GateWay-IP>:8080/function/main -H 'Content-Type: application/json'
-d '{ "Name": "Square", "Color": "Red", "Dimensions": 2 }'
```

It should return the same JSON object

6. Defining the Decorator logic

Now that you have the Main function working and returning its input as output, create the Decorator function. A decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

For this lab, you will create a Decorator function that will add missing attributes to the JSON objects and set its default values so that the Main function always receives a JSON object with at least the following attributes: Name, Color, and Dimensions.

For example, if you call the Decorator function with the following JSON object as input:
{"Name":"Square","Dimensions":2}
It adds the missing Color attribute to the JSON object before sending it on to the Main function. And since it has no color, it will set the Color value to Transparent. Thus, the Decorator function will return this after it calls the Main function:
{"Name":"Square","Dimensions":2,"Color":"Transparent"}

a) Now, using the templated already downloaded in the OpenFaaS folder, we will create a new empty NodeJS function

```
cd ~/OpenFaaS
faas-cli new --lang node12 --prefix us.gcr.io/<project-id> decorator
```

b) Update the **decorator/handler.js** file to

```
'use strict'
const request = require('sync-request');

module.exports = async (event, context) => {
   var obj = event.body;
   if (obj['Name'] === undefined) {
      obj['Name'] = 'Nameless';
   }
   if (obj['Color'] === undefined) {
      obj['Color'] = 'Transparent';
   }
   var res = request('POST', 'http:// <OpenFaaS-GateWay-IP>:8080/function/main', {
      body: JSON.stringify(obj)
    });
    console.log(res["body"].toString())
   return context.status(200).succeed(res["body"].toString('utf8', 1, res["body"].length-
1).replace(/\\\"/g,'\"'));
}
```

This script fills any missing key with the default value and then sends it to the main function using a package **sync-request**. Then return the JSON object returned by the main function.

c) Add **sync-request** to the dependencies in **decorator/package.json** as

```
{
  "name": "openfaas-function",
  "version": "1.0.0",
  "description": "OpenFaaS Function",
  "main": "handler.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 0"
  },
  "keywords": [],
  "author": "OpenFaaS Ltd",
  "license": "MIT",
  "dependencies": {
    "sync-request": "^6.1.0"
  }
}
```

d) To build the docker image, use the following command

```
cd ~/OpenFaaS
faas-cli build -f decorator.yml
```

e) Then push it to Google Container Registry

```
docker push us.gcr.io/<project-id>/ decorator
```

f) Finally deploy it to OpenFaas

```
faas-cli deploy -f decorator.yml
```

Now, the function is deployed. You can check its status to be ready from the UI of the gateway

g) Invoke it and send the JSON object {"Name":"Square", "Color": "Red", "Dimensions": 2 }

```
curl http://<OpenFaaS-GateWay-IP>:8080/function/decorator -H 'Content-Type: application/json' -d '{ "Name": "Square", "Dimensions": 2 }'
```

It should return {"Name":"Square","Dimensions":2,"Color":"Transparent"}

Discussion:

Summarize the problem, the solution, and the requirements for the pattern given in part 1. Which of these requirements can be achieved by the procedures shown in parts 2 and 3?

Design:

Kubernetes provides persistent volumes. Why such a feature can be important? How to implement it? Provide an example in which persistent volumes are needed. Configure a YAML file to implement the example. Run it and test the creation of persistent volume and its ability to provide the required functionality within the example.

Deliverables:

1. A report that includes the discussion and the design parts.
2. An **audible** video of about 7 minutes maximum showing the final results of following the lab steps. It should include showing the deployments, services, and pods created in the lab with their test cases.

3. Another **audible** video of 3 minutes maximum shows the configuration and implementation of autoscaling using GKE.