Lab 2: Deploying a request splitting ambassador and a load balancer with Kubernetes

Objective:

1. Learn how to configure and run a request splitter using Nginx.
2. Be familiar with the ConfigMap tool in Kubernetes.
3. Learn how to use the curl command for requesting an HTTP method.
4. Learn how to configure load balancer services
5. Get Familiar with load balancing pattern

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

https://github.com/GeorgeDaoud3/SOFE4790U-lab2

Part 1.

Read the following document (Gateway Routing 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.

You will be guided through the steps to deploy a request-splitting ambassador that will split 10% of the incoming HTTP requests to an experimental server.

Procedure:

1. If you haven't yet created a GKE cluster, follow the instruction given in Lab 1 and create one.
2. Create the web server
   a) Create a YAML file with the name **web-deployment.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: web-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   run: web-deployment
 template:
  metadata:
   labels:
    run: web-deployment
  spec:
   containers:
   - name: web-depoyment
    image: mcr.microsoft.com/azuredocs/aci-helloworld
    ports:
    - containerPort: 80
```

It creates two replicas of an **aci-helloworld** image. You can read the documentation of **aci-helloworld** for more information.

b) Create a deployment using GKE

```
kubectl create -f web-deployment.yaml
```

You can check that there are two pods created with a name starting with **web-deployment**

c) Now, let's create a clusterIP service. clusterIP service creates a stable IP address that is accessible from nodes in the cluster.

```
kubectl expose deployment web-deployment --port=80 --type=ClusterIP --name web-deployment
```

3. Let's repeat those steps again but for an experimental server

a) Create a YAML file with the name **experiment-deployment.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: experiment-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   run: experiment-deployment
 template:
  metadata:
   labels:
    run: experiment-deployment
  spec:
   containers:
   - name: experiment-depoyment
    image: mcr.microsoft.com/azuredocs/aci-helloworld
    ports:
    - containerPort: 80
```

d) create a deployment using GKE

```
kubectl create -f experiment-deployment.yaml
```

You can check that there are two pods created with a name started with **web-deployment**

e) Now, let's create a clusterIP service. clusterIP service creates a stable IP address that is accessible from nodes in the cluster.

```
kubectl expose deployment experiment-deployment --port=80 --type=ClusterIP --name experiment-deployment
```

4. Create a request splitter

a) Generating a ConfigMap for the custom NGINX configuration

In a working folder of your choice, create a **conf.d** sub-folder. Then create an **nginx-ambassador.conf** file in the folder and copy the following configuration code block into the new **nginx-ambassador.conf** file:

```
upstream backend {
    server web-deployment:80 weight=9;
    server experiment-deployment:80;
```

```
}
server {
   location / {
      proxy_pass http://backend;
   }
}
```

Save the file. You should now have a **nginx-ambassador.conf** file in the **conf.d** folder.

The configuration gives the **web-deployment** server weight of 9 while the **experiment-deployment** server will have the default weight of 1. That's why the **web-deployment** will receive 90% of the requests while the **experiment-deployment** will receive only 10% of them.

Note: A **ConfigMap** is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables. It must be mounted as directories or as configuration files in a volume.

b) Now, run the command that generates a **ConfigMap** for the custom **NGINX ambassador configuration** file:

```
kubectl create configmap ambassador-config --from-file=conf.d
```

Note: Make sure you run this command from the working folder that contains the **conf.d** subfolder, you created earlier.

c) To deploy the request splitting ambassador, **nginx** image will be used to create two replicas. each pod will have a local path of **/etc/nginx/conf.d** mounted to the configMap named **ambassador-config** created before. This is configured using a file named **ambassador-deployment.yaml** with the following content.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: ambassador-deployment
spec:
 selector:
  matchLabels:
   app: ambassador
 replicas: 2 # tells deployment to run 2 pods matching the template
 template: # create pods using pod definition in this template
  metadata:
   labels:
    app: ambassador
  spec:
   containers:
   - image: nginx
    name: ambassador
    ports:
    - containerPort: 80
    volumeMounts:
    - name: config-volume
```

```
        mountPath: /etc/nginx/conf.d
      volumes:
      - name: config-volume
        configMap:
          name: ambassador-config
```

d) Now, let's deploy the ambassador via the following command

```
kubectl create -f ambassador-deployment.yaml
```

e) And associate a load balancer service to it

```
kubectl expose deployment ambassador-deployment --port=80 --type=LoadBalancer
```

f) Finally, check that all pods, deployments, and services are running with no issues. What commands should you use? What's the external IP address associated with the **ambassador-deployment** service?

5. Test the request splitter
   a) Get the external Ip address for the **ambassador-deployment** service using

```
kubectl get services
```

   b) Go to a web browser and navigate to that IP **http://<ambassador-IP>** or use **CURL** command to see the output of the load balanced custom NGINX implementation:

```
curl http://<ambassador-IP>
```

   c) Repeat b) many times or run the following shell script that iteratively call the server

```
for _ in {1..20}; do curl http://<ambassador-IP>  -s > output.txt; done
```

   d) To check how many times each server was called, we can read their logs

```
kubectl logs -l run=web-deployment
kubectl logs -l run=experiment-deployment
```

6. (optional) after creating a video for submission, you can delete the generated deployments and services if you wish to save the free credits.

Part 3.

You will be guided through the steps to deploy a replicated load balancing service that will process requests for the definition of English words. The requests will be processed by a small NodeJS application that we will fire up in Kubernetes using a pre-existing Docker image

Procedure:

1. Deploy 3 replicas of a dictionary-server
   This dictionary-server is a small NodeJS application that takes HTTP request paths with an English word and responds with the definition of that word. E.g. a request to **http://someserver/dog** returns **A quadruped of the genus Canis, esp. the domestic dog (C.familiaris)**.
   a) Create a YAML file with the name **loadbalancer-deployment.yaml** and the following content

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: loadbalancer-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: loadbalancer-deployment
```

```
template:
  metadata:
    labels:
      app: loadbalancer-deployment
  spec:
    containers:
    - name: server
      image: brendanburns/dictionary-server
      ports:
      - containerPort: 8080
      readinessProbe:
        httpGet:
          path: /ready
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 5
```

the deployment looks similar to deployments in previous labs except we are deploying 3 pods of the image and there is an extra **readinessProbe** option that is used by GKE to check the readiness of each pod. It uses a ready endpoint and is first checked after 5 seconds of deployment and then every 5 seconds. You can check the image implementation for more details.

b) Create the Deployment in Kubernetes

```
kubectl create -f loadbalancer-deployment.yaml
```

c) Check the status of the created pods

```
kubectl get pods --output=wide
```

2. Expose our Replicated Load Balancing Service:
   a) By executing the following command

   ```
   kubectl expose deployment loadbalancer-deployment --port=8080 --type=LoadBalancer
   ```

   b) Get the external IP address for the server

   ```
   kubectl get services --watch
   ```

3. Confirm our Replicated Load Balancing Service is working
   By running the following command

   ```
   curl http://<Server-IP>:8080/dog
   curl http://<Server-IP>:8080/storey
   ```

4. (optional) after creating a video for submission, you can delete the generated deployments and services if you wish to save the free credits.

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:

Autoscaling is another pattern that can be implemented by GKE. Your task is to configure a Yaml file that autoscaling the deployment of a given Pod. Why autoscaling is usually used? How autoscaling is implemented? How autoscaling is different than load balancing and request splitter?

Deliverables:

1. A report that includes the discussion and the design parts.

2. An **audible** video of about 5 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 (step 6 in part 2, step 3 in part 3).
3. Another **audible** video of 3 minutes maximum shows the configuration and implementation of autoscaling using GKE.