Assignment 3

Virtualization using Docker

3.1 Container Virtualization

Dockerfiles

We followed the official documentation and installed Docker on our VMs. We then created a Dockerfile for each of the two web services, by following this guide on how to dockerize Node.js applications: https://nodejs.org/fr/docs/guides/nodejs-docker-webapp/.

These files are identical, with the exception of which port is exposed when running the images as containers (line 19). These ports match the ones used by the web services.

user > Dockerfile > ...

```
1  FROM node:10
2
3  # Create app directory
4  WORKDIR /app
5
6  # Install app dependencies
7  # A wildcard is used to ensure both package.json
8  # AND package-lock.json are copied
9  # where available (npm@5+)
10 COPY package*.json ./
11
12 RUN npm install
13 # If you are building your code for production
14 # RUN npm ci --only=production
15
16 # Bundle app source
17 COPY . .
18
19 EXPOSE 2000
20 CMD [ "node", "index.js" ]
```

url-shortener > Dockerfile > ...

```
1  FROM node:10
2
3  # Create app directory
4  WORKDIR /app
5
6  # Install app dependencies
7  # A wildcard is used to ensure both package.json
8  # AND package-lock.json are copied
9  # where available (npm@5+)
10 COPY package*.json ./
11
12 RUN npm install
13 # If you are building your code for production
14 # RUN npm ci --only=production
15
16 # Bundle app source
17 COPY . .
18
19 EXPOSE 4000
20 CMD [ "node", "index.js" ]
```

Figure 1 & 2: Dockerfiles for both web services

Docker Compose

Our first container deployments were done using the *docker build* and *docker run* commands, but we discovered a useful tool that made this process easier for us - Docker Compose. To use Compose you just need to specify the configuration of your web services in a YAML file, and using one single command you then build images and run containers for all the services at once.

We didn't have any trouble getting the containers up and running, but one challenge we faced was making the dockerized applications be able to communicate with each other. This was a major problem, since the URL shortener is dependent on making a HTTP request to the User service in order to verify if a user is logged in or not. After doing some research, we

found out that Docker containers use their own networks to communicate with each other, and that each container had its own internal IP address on this network.
The IP addresses were prone to change every time a container was run from an image, so we had to make sure they remained static and consistent between each deployment. We also wanted to isolate the two web service containers on a separate network from the default one. For this we added a new “networks” block to the *docker-compose.yml* file, where we defined a new network called “static-network” with its own subnet. We then configured each of the services to use the new network, while also specifying their static IP addresses.

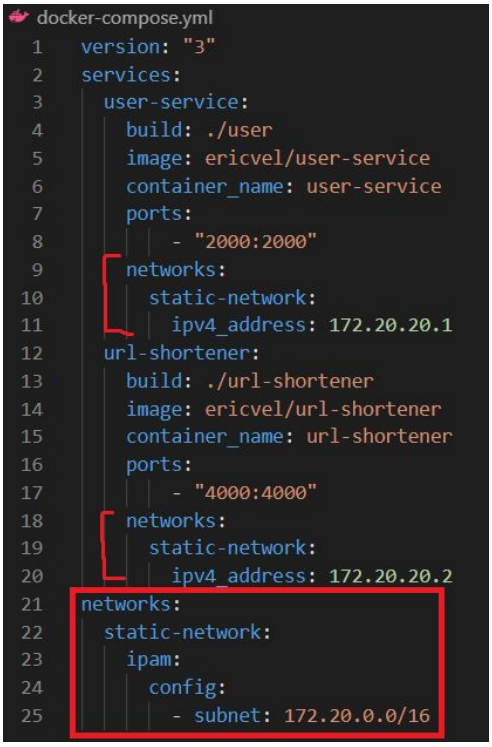


Figure 3: Docker Compose YAML file

Finally, we updated the code that verified whether a user is logged in to use the User service’s new static IP address.

Figure 4: Updated IP address for JWT verification

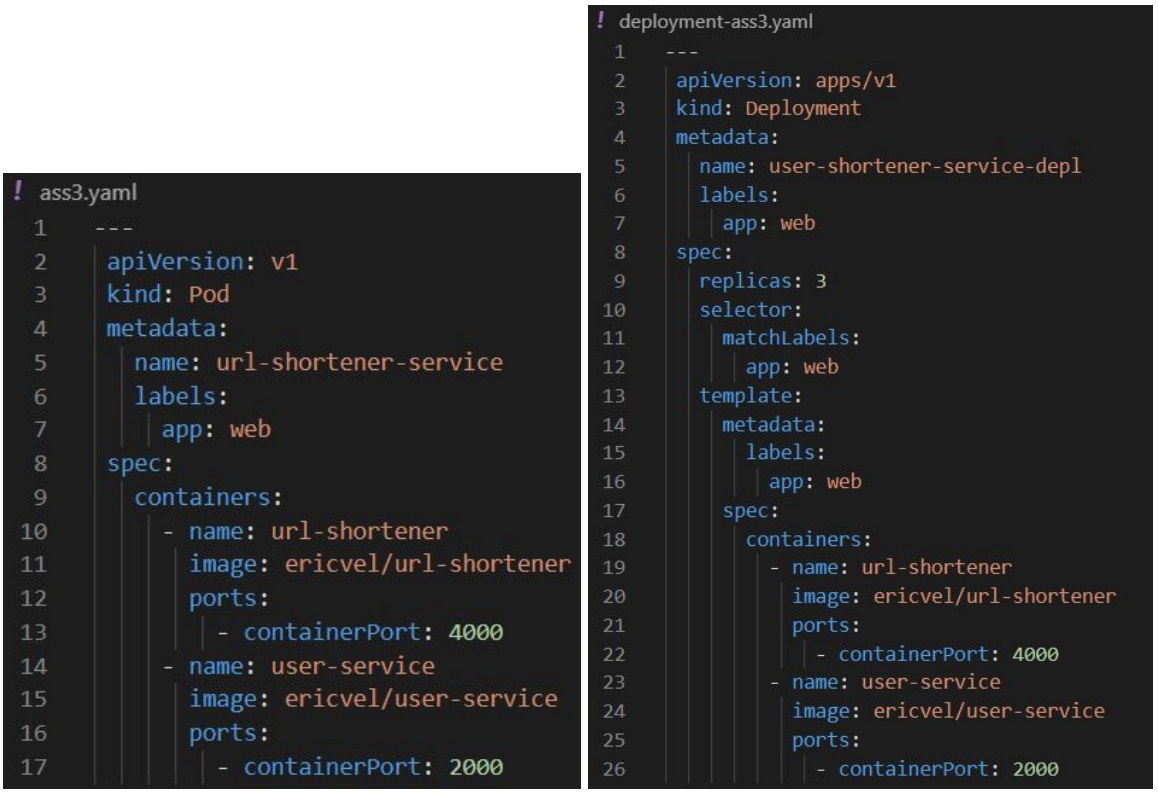
All the files were now ready, so we ran *docker-compose up* to build the images and run the containers. We also pushed the Docker images to Docker Hub using the *docker push* command:

- https://hub.docker.com/r/ericvel/user-service
- https://hub.docker.com/r/ericvel/url-shortener

3.2 Container Orchestrations

We started by installing kubeadm, kubelet and kubectl on the three VMs assigned to us by the TA. In order to avoid communication issues we disabled the firewall on each of the machines and configured the IP tables on master node to allow traffic to and from the worker nodes. We then ran *kubeadm init* on the master node, followed by a command to install Calico as our Pod network (overlay network). Finally we ran *kubeadm join* on the worker nodes. We now had successfully created a Kubernetes cluster.

Moving on we created a Pod which we configured using a YAML file. The Pod was set up to pull the images from Docker Hub that we had uploaded earlier. The Pod was then deployed using a similar YAML file and similar command: *kubectl create*.



ass3.yaml

```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: url-shortener-service
  labels:
    app: web
spec:
  containers:
    - name: url-shortener
      image: ericvel/url-shortener
      ports:
        - containerPort: 4000
    - name: user-service
      image: ericvel/user-service
      ports:
        - containerPort: 2000
```

deployment-ass3.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-shortener-service-depl
  labels:
    app: web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: url-shortener
          image: ericvel/url-shortener
          ports:
            - containerPort: 4000
        - name: user-service
          image: ericvel/user-service
          ports:
            - containerPort: 2000
```

Figure 5 & 6: YAML files for creating and deploying Pods

We then ran *kubectl get pods* and saw that all three replicas were running as they should.

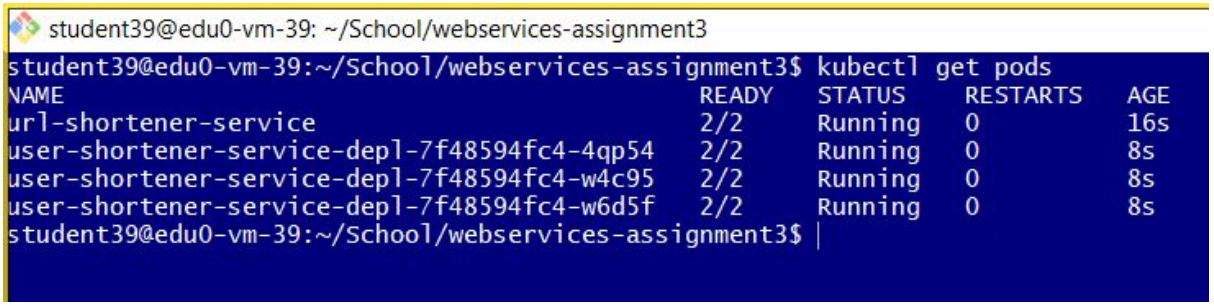


Figure 7: Pods are running

After a short while, though, they all received the status “CrashLoopBackOff”. After inspecting the pods using *kubectl logs*, we saw that it was caused by the following error: “connect: no route to host”. We couldn’t figure out the exact reason for this error, but we think that it might be caused by some iptables related settings.

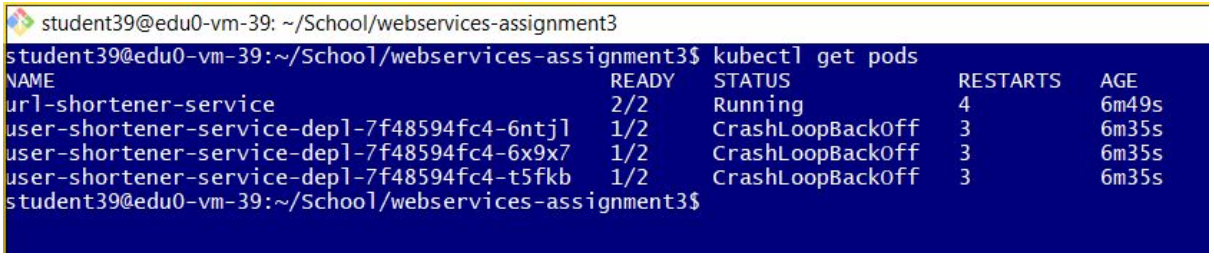


Figure 8: Pods are not running