Assignment 3: Virtualization using Docker

Group 6

In both parts of the assignment we used provided VMs.

Part 1: Container Virtualization

Summary of the steps performed in order to deploy and run services in Docker on a VM:

1. Before starting, we followed the materials[1] about Docker listed in the assignment.
2. After copying the files prepared earlier in the 2nd assignment, we decided to reorganize the files, so that both services are in separate directories: auth-app and url-shortener. This way we obtained two goals:
    a. Default Dockerfiles could be created separately (Dockerfile in both directories)
    b. Additional decoupling of the services on the files level (already existing, just better marked in the directory structure).
3. We wrote the Dockerfiles that will automate building Docker images for the two services. Both files are in the base directories of both microservices. In principle, they are based on python:3 base image, expose port 5000 or 5001, copy the requirements text file, then import the requirements, copy the rest of the microservice files, set environmental variables for Flask and finally run the python scripts. The differences between the two files are only relating to the names of the files and numbers of ports that are used by them.
    a. Issue appeared while running pip install command during build. Apparently, it was caused by the DNS configuration, and the solution was to explicitly define DNS servers in the several additions to the `/lib/systemd/system/docker.service` file as explained in several comments found on the internet[2].
4. The new environment required some small changes in various elements. To be accessible from outside of the VM, the Flask needed to be configured to run on 0.0.0.0 address rather than localhost.
5. Finally, we tested if our Dockerfiles are working by building docker images using commands: "*docker build -t auth-app:1 .*" and *"docker build -t url-shortener:1 ."*, and then run the containers from images using commands "*docker run -p 5000:5000 url-shortener:1"* and *"docker run -p 5001:5001 auth-app:1"*.

Bonus part: Running Nginx proxy in a container

Additionally, we also deployed an Nginx server in a separate container. In order to do this, we had to prepare an additional directory ("nginx"), which consists of a Dockerfile (based on nginx image, only operation is copying the nginx configuration file inside the container). The configuration file is set in a way that provides upstreams to the auth-app and url-shorteners, identified by aliases in the Docker network. To have communication between the containers, we had to create a network ("*docker network create dev*"), add EXPOSE instructions to the Dockerfiles of the microservices, and, while running them, add two additional informations: the network and their aliases ("*docker run -d -p 5000:5000 --net dev --net-alias url-shortener url-shortener:1*" and "*docker run -d -p 5001:5001 --net dev --net-alias*

[1] Introduction to Docker: https://2020-03-qcon.container.training/intro-fullday.yml.html
[2] Github discussion for Failed to establish a new connection error, https://github.com/moby/moby/issues/30757#issuecomment-283304977

auth-app auth-app:1"). It would be useful to automate running them using docker-compose tools. Command to run nginx server: "*docker run -i -p 80:80 --net dev nginx-test*".

Comments

The creation and deployment of our microservices in Docker proved to be a relatively simple task. In a short time we were able to run our app having all of the power that is given by the Docker concept available. But it is also true that the majority of the features, e.g., the ones discussed in the tutorial [1], were not used here since the complexity of our system is rather low. In more advanced cases, other tools and approaches might prove to be troublesome, but perhaps thanks to this technology - still manageable.

Some minor technical obstacles and uncertainty in regard to the existing settings were hindering our testing process for a moment, but using the 'trial and error' method we finally managed to successfully test our solution using a Postman app running on a local machine and our microservices running on the provided VMs.

Part 2: Container Orchestrations

Before this part is started, both images must have been built and ready to use.

Step 1: Preparing and installing the Kubernetes cluster

These steps were based on the resources provided with the assignment[3]. First, we installed kubeadm, kubelet and kubectl tools. Then we used kubeadm to create the cluster for our worker VMs: on the master node, our control-plane node was initialized using *kubeadm init* command. Successful completion was followed with two actions: first, we followed the instructions at the end of the output to make kubectl work for non-root users, and second, we copied the line to add new slave nodes to the cluster (*kubeadm join).* This copied command was run on the worker node, which successfully joined our cluster (confirmed with kubectl get nodes). Additionally, we installed a pod network add-on (Calico) by *kubectl apply -f <add-on.yaml>.*

On the way to the final version of the assignment, we also had a chance to test the reset feature of the Kubernetes cluster, also closely following the tutorial [3].

Step 2: Deployment

To deploy our app made of two microservices, we followed two paths. First, we tested various command line kubectl commands, creating what was needed for our case:

- *kubectl create deployment auth-app --image=auth-app:1* to create a deployment that run one image and tested it with *kubectl get deployments*
- *kubectl scale deployments/auth-app --replicas=3* to create three replicas of the pod created earlier and tested with *kubectl get deployments*
- *kubectl expose deployment/auth-app --type="NodePort" --port 5000* to create a service and expose port 5000, tested with *kubectl get services*

[3] Creating a single control-plane cluster with kubeadm, https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/#tear-down

Next, we changed to a more replicable and comprehensive way to create the pods, deployments and services using YAML files. All three files (pod.yaml, deployment.yaml and service.yaml are in the root directory of the archive). Every single one of them is run with the command *kubectl create -f [filename]* and might be tested using the same commands listing the running elements as those mentioned above.

In **pod.yaml,** we define a single pod which is made of two containers. The file specifies the name of the pod (url-shortener-pod) and a label ("web"). For simplicity, each container is identified by the same name as the image and the container port is specified as earlier (5000 and 5001).

Deployment.yaml is defining the structure of the deployment. To take advantage of the variety of options that are available within the tool, we decided to redefine the pod explicitly inside this file (again). Service.yaml is using a selector ("web") to target all pods with this label.

Service.yaml is the last YAML file created in this assignment. The point of defining a service is to make our containers accessible from the outside - they essentially expose internal ports to the world outside the Kubernetes cluster. Here, we select all pods with a given label and set two mappings between ports in those pods (5000 to 9376 and 5001 to 9377).

Step 3: Testing

This is the final state of the running elements within Kubernetes cluster while the services were accessible outside (and were tested using a Postman app on a local machine).

```
student66@master-node:~/tmp/clouds-url-shortener$ kubectl get nodes
NAME           STATUS   ROLES    AGE     VERSION
master-node    Ready    master   3m15s   v1.18.2
slave-node1-g6   Ready    <none>   91s     v1.18.2
student66@master-node:~/tmp/clouds-url-shortener$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
url-shortener-deploy-75476b9689-4ftgf   2/2     Running   0          73s
url-shortener-deploy-75476b9689-kpksx   2/2     Running   0          73s
url-shortener-deploy-75476b9689-xlxst   2/2     Running   0          73s
student66@master-node:~/tmp/clouds-url-shortener$ kubectl get deployments
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
url-shortener-deploy   3/3     3            3           80s
student66@master-node:~/tmp/clouds-url-shortener$ kubectl get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)             AGE
kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP             3m33s
url-shortener   ClusterIP   10.96.78.250   <none>        5001/TCP,5000/TCP   76s
student66@master-node:~/tmp/clouds-url-shortener$
```