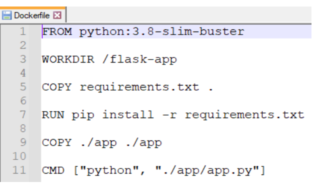
Lab 6

4.1. Docker Configuration files and Setup

In the Python Flask Github repository you have just cloned, notice that other than the App.py source file, in the directory "flask-app" there are also 2 other files "Dockerfile" and "requirements.txt".

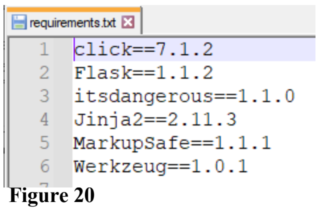
To containerize an application, Docker needs to know all the dependencies and external libraries the application would require to compile and run.

In the Dockerfile below, each line is an instruction for Docker to run when containerizing the application.



Dockerfile

```
FROM python:3.8-slim-buster

WORKDIR /flask-app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY ./app ./app

CMD ["python", "./app/app.py"]
```



requirements.txt

```
click==7.1.2
Flask==1.1.2
itsdangerous==1.1.0
Jinja2==2.11.3
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

Figure 20

The "requirements.txt" file is referenced by Docker when it runs the Python "pip" tool to install all the required Python libraries for the Flask application to run. "requirements.txt" contains the list of all the necessary Python libraries and their respective versions that should be installed with "pip"

4.2. Building a Docker Image

First step is to build a Docker Image from our "flask-app" application which contains all the dependencies needed by our Python code to run a Flask Web Server and Application.

- In the Command Prompt, change directory to **/home/devops-admin/ET0735/Lab6_flask_app**
- Run the following Docker command below **(Please note that there is a dot . character at the end of the command)**

  ```
  docker build -t flask-app .
  ```
- After running the command above, Docker will build and generate a Docker image called "flask-app"
- To check that Docker has successfully generated the Docker Imager "flask-app" run the command below which lists all the Docker images locally

  ```
  docker image ls
  ```

4.3. Running a Docker Container

Now that we created a Docker Image, the next step is to run the Docker container and start our Python "flask-app" Web Application.

- In the Command Prompt, enter the command below which runs the Docker Image "flask-app"

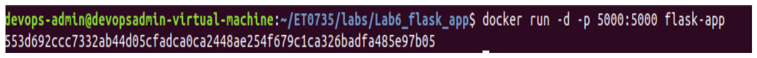
```
docker run flask-app
```

- The Docker command `docker run flask-app` runs our Docker Container for the "flask-app", however it runs the Web Application using the internal container HTTP Port 5000 which is not yet mapped to our local machine Port 5000

- To map the internal Docker port 5000 to our local machine's port 5000, we need to run the following Docker command

  ```
  docker run -d -p 5000:5000 flask-app
  ```

- In the console output after running the Docker Container "flask-app", notice now that after running the Docker Container it returns back to the Linux Bash Terminal and does not block other commands



```
devops-admin@devopsadmin-virtual-machine:~/ET0735/labs/Lab6_flask_app$ docker run -d -p 5000:5000 flask-app
553d692ccc7332ab44d05cfadca0ca2448ae254f679c1ca326badfa485e97b05
```

5.2. Log in to remote Docker Hub account in Linux Bash Terminal

Before we can "push" our locally created Docker image to our Docker Hub account, we first need to log in via Docker command line to Docker Hub.

Login to the Docker Hub account we have just created by running the following Docker command and enter your Docker Hub credentials for username and password when prompted,

```
docker login
```

5.3. Tagging Docker Image to Docker Hub namespace

In this step we will now use the Docker command line to "tag" our local Docker Image to an image that follows the namespace required by Docker Hub

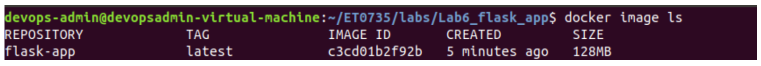
- Change to the directory of the Python "flask-app"

- Run the following Docker command to tag the "flask-app" docker image

  ```
  docker image tag flask-app {your namespace}/flask-app
  ```

  where {your namespace} is your Docker Hub user ID
- After creating the Docker image tag, run the command below to verify that the image tag was successfully created as shown in Figure 29

  ```
  docker image ls
  ```



```
devops-admin@devopsadmin-virtual-machine:~/ET0735/labs/Lab6_flask_app$ docker image ls
REPOSITORY          TAG                 IMAGE ID       CREATED         SIZE
flask-app           latest              c3cd01b2f92b   5 minutes ago   128MB
```

5.4. Pushing Docker Image to Docker Hub

- Run the following Docker command to push the "flask-app" Docker Image to Docker Hub

  ```
  docker push {your namespace}/flask-app
  ```

  Note: Replace {your namespace} above with your own Docker Hub Username
- After running the Docker push command, check that the push completes without any errors in the Linux Bash Terminal shown in Figure 30

Lab 7

5. Deploy the Python Flask Web Application from Docker Hub as a Pod in Kubernetes

- In the Terminal, enter the command below which will deploy the Docker Image "flask-app" from Docker Hub to your local Kubernetes:

  ```
  kubectl run my-flask-app --image={your namespace}/flask-app
  ```

- You should see the output like the following showing that the Python Flask Web Application named "my-flask-app" has been successfully deployed as a Pod in Kubernetes.

- To check that the Pod "my-flask-app" has been successfully running, run the command below which lists all the Pods running in Kubernetes.

  ```
  kubectl get pods
  ```
- To view the Pod "my-flask-app" in more detail, run the command below:

  ```
  kubectl describe pod my-flask-app
  ```

6 Delete the Python Flask Web Application pod

- We will simulate the failure of the "my-flask-app" pod by deleting the pod. Run the command below:

  ```
  kubectl delete pod my-flask-app
  ```

7 Deploy the Python Flask Web Application as a Deployment

A Deployment manages the Pod and ensures the Pod is always running. We will deploy the Python Flask Web Application as a Deployment this time round in Kubernetes.

- In the Terminal, enter the command below which will deploy the Docker Image "flask-app" from Docker Hub to your local Kubernetes:

  ```
  kubectl create deployment my-flask-app --image={your
  namespace}/flask-app
  ```

- You should see the output like the following showing that the Python Flask Web Application named "my-flask-app" has been successfully deployed as a Deployment in Kubernetes.

- To check that the Deployment "my-flask-app" has been successfully running, run the command below which lists all the Deployments running in Kubernetes:

  ```
  kubectl get deployments
  ```

9 Describe the Python Flask Web Application Deployment

To look in more detail on the deployment configuration

- Run the command below to view the deployment:

  ```
  kubectl describe deployment my-flask-app
  ```

10 Scale the Python Flask Web Application with more Replicas

- We will scale our flask-app by increasing to 5 Replicas. Run the command below to increase our replicas to 5:

  ```
  kubectl scale deployment my-flask-app --replicas=5
  ```

- To check the updated deployment, run the command below:

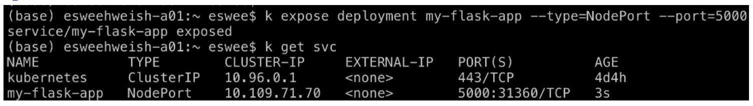
  ```
  kubectl get deployments
  ```

11 Expose the Python Flask Web Application Deployment as a Service

Pods are unreliable and are replaced with new IP addresses when new Pods are created by replica sets. A Service provides a stable network abstraction point with a reliable IP address which load-balances across the Pods.

- We will expose our flask-app deployment as a Service with the type NodePort by running the command below:

  ```
  kubectl expose deployment my-flask-app --type=NodePort --
  port=5000
  ```



```
(base) esweehweish-a01:~ eswee$ k expose deployment my-flask-app --type=NodePort --port=5000
service/my-flask-app exposed
(base) esweehweish-a01:~ eswee$ k get svc
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP          4d4h
my-flask-app   NodePort    10.109.71.70    <none>        5000:31360/TCP   3s
```

```
kubectl get svc
```

Open your web browser and go to the assigned ip address for your service. In this case it is http://10.109.71.70:5000

Lab 8

Stop and remove all Docker containers:

```
docker container stop $(docker container ls -aq)
docker container rm $(docker container ls -aq)
```
Remove all Docker images:

```
docker image rm $(docker image ls -aq)
```
Build and Run Docker Python "flask-app" Web Server:

Enable WiFi on Raspberry Pi: Connect to "eee-iot" SSID using VNC Viewer.

Clone GitHub Repository:
git clone --recurse-submodules https://github.com/ET0735-DevOps-AIoT/Lab8_flask_app_raspberry_pi.git

Modify Dockerfile if SSL Cert Failed Error Occurs:
Dockerfile
RUN pip install --trusted-host pypi.org --trusted-host pypi.python.org --trusted-host files.pythonhosted.org -r requirements.txt
Build Docker Image:
```
docker build -t flask-app .
```
Run Docker Container:
```
docker run -d -p 8000:5000 flask-app
```
Access Web Server: Open browser and navigate to Raspberry Pi's IP with port 8000.(to access go if command:ifconfig)
Interfacing Raspberry Pi IO Peripherals in Docker:

Build and Run Docker Container for LED Blinking:

```
docker build -t docker blink_led:v1 .
docker container run --privileged -d
docker_blink_led:v1
```
Use pipreqs to List Python Libraries:

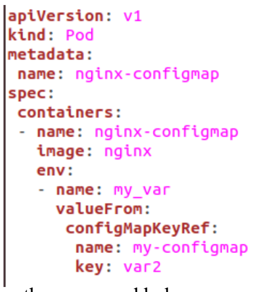
1.6. Creating Configmap using YAML file

Write the command to create a configmap named "**my-configmap**" with the following key/value pairs:

1. var2=val2

```
kubectl create configmap my-configmap --from-
literal=var2=val2
```
Create new nginx pod named "**nginx-configmap**" using the image "**nginx**" and inject the value from variable "**var2**" as an environment variable within the container. The key of the environment variable within the container is named "**my_var**". Refer to the template below and fill up the blanks accordingly.
Create a file named **lab9-configmap.yml** with the text below:

```
apiVersion: v1
kind: Pod
metadata:
 name: nginx-configmap
spec:
 containers:
  - name: nginx-configmap
   image: nginx
   env:
    # Define the environment variable
    - name: __my_var
     valueFrom:
      configMapKeyRef:
       name: my-configmap
       # Specify the key associated with the value

       key: __var2
```



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-configmap
spec:
  containers:
  - name: nginx-configmap
    image: nginx
    env:
    - name: my_var
      valueFrom:
        configMapKeyRef:
          name: my-configmap
          key: var2
```

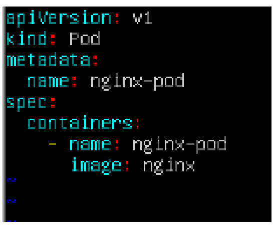
Run the YML file "lab9-configmap.yml" using the command below:

```
kubectl apply -f lab9-configmap.yml
```

You should see the system return the message:

pod/nginx-configmap created.

After that, verify that a new pod "nginx-configmap" has been created using the command :
```
kubectl get pod
```



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx-pod
      image: nginx
~
~
```

Lab 9

1.1. Kubernetes Namespaces

Kubernetes provides Namespaces to add additional scope to the naming of the pods, deployments that must be unique within a specific namespace but can be reused with similar names across other different namespaces.

- Create a new namespace called "myflask-app-nspace" and write down the command

```
Kubectl create namespace myflask-app-nspace
```

- Check that the Kubernetes namespace is successful created by using the appropriate

```
Kubectl get namespace
```

1.1. Creating a Pod via Kubernetes CLI with Docker images

Create a new pod called "**nginx-test**" on the **namespace** that was created in previous exercise with image "**nginx**".

```
kubectl run nginx-test --image=nginx -n myflask-app-nspace
```

1.2. Creating a Deployment via Kubernetes CLI

In the previous lab we created Kubernetes Deployments using the command below,

```
kubectl create deployment my-flask-app --image={your namespace}/
flask-app
```

- Execute the kubectl command to expose the port 5000 in the Kubernetes Deployment "my-flask-app" to the host machine and write the command used in the box below.

```
kubectl expose deployment nginx-test --type=NodePort --port=5000
```

```
kubectl delete deployment  my-flask-app
```

- Check if the Kubernetes service performing the port forwarding is still running.
- Write the "kubectl" command used in the box below.

```
kubectl get svc
```

- You will notice that the Kubernetes service is still running although the deployment has been deleted.

- Delete the Kubernetes service created to run in the port 5000 and write the "kubectl" command used in the box below.

```
kubectl delete svc my-flask-app
```

1.3. Kubernetes Pod Environment Variables

Environment variables allow pods to be configured from the host operating system.

- Using kubectl, run a command that will create and run a pod named "env-pod"

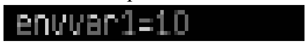
from image "nginx" which maps the environment variables envar1 to a value 10.
Write the command used in the box below.

```
kubectl run env-pod --image=nginx --env=envar1=10
```

- Verify that the Pod and the associated Environment Variables are created and mapped.

```
kubectl exec env-pod -- env
```

In the terminal output below, you should see the list of environment variables inside the Pod "env-pod".



```
envvar1=10
```

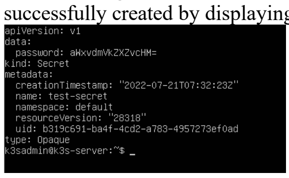
1.4. Creating Secrets in Kubernetes

Secrets are used in Kubernetes to create key/value pair data based on the base64 encoding.

Using "kubectl", create a Kubernetes secret "test-secret" based on the following key/value pair "password=ilovedevops"

```
kubectl create secret generic test-secret --from-
literal=password=ilovedevops
```

After creating the secret, run a "kubectl" command to check if the secret has been successfully created by displaying the following console output.



```
apiVersion: v1
data:
  password: aWxvdmVkZXZvcHM=
kind: Secret
metadata:
  creationTimestamp: "2022-07-21T07:32:23Z"
  name: test-secret
  namespace: default
  resourceVersion: "28318"
  uid: b319c691-ba4f-4cd2-a783-4957273ef0ad
type: Opaque
k3sadmin@k3s-server:~$ _
```

Write the command used to check if the secret "test-secret" was correctly created in the box below.

```
kubectl get secret test-secret -o yaml
```
To decode the secret value to plaintext, you can use the Linux command below:
echo {encoded password} | base64 -d

1.5. Creating a Pod using YAML file

- Create a new directory "yml' in the directory path "/home/devops-admin/ET0735/lab9"

- Using the text editor, create a new YAML file "**pod.yml**". Enter the YAML code below.

  Save the file in the newly created "yml" directory.

  pod.yml

```
kubectl apply -f pod.yml
kubectl get pods
```