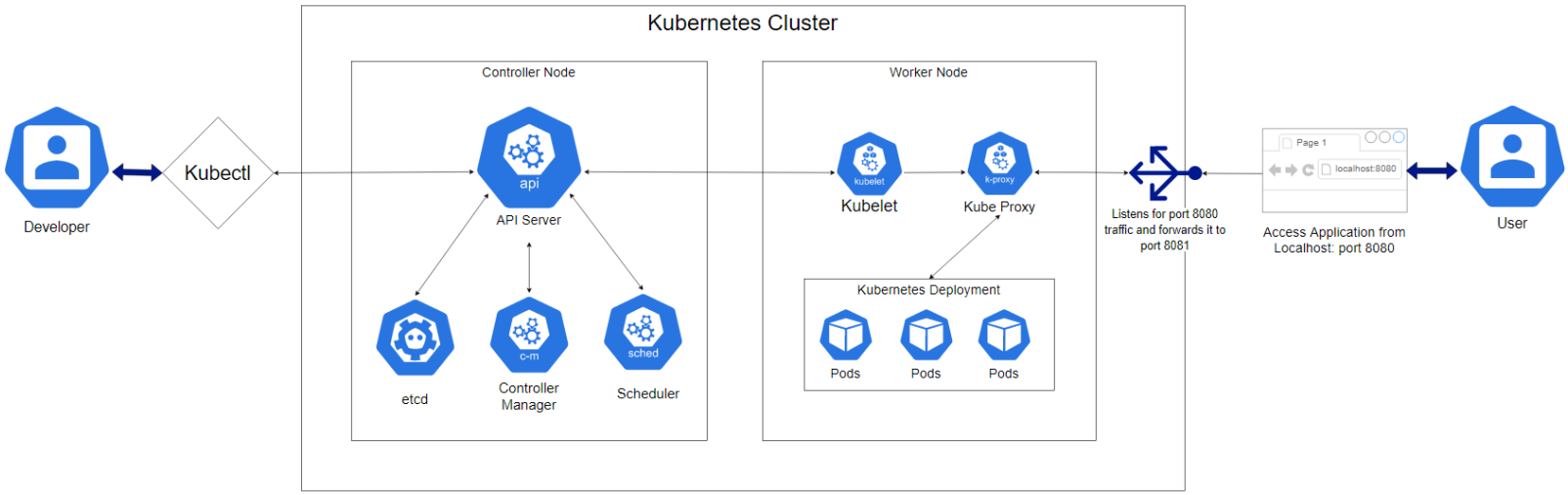
K8s Assignment



The objective of this assignment was to dockerize a Flask application then deploy it using Kubernetes. When dockerizing our Flask application, we used Docker to build an image out of our application and pushed it to Dockerhub. Once we remotely host our image on DockerHub, we use Kubernetes and set up a cluster that has a load balancer attached to it. Kubernetes is an open-source container orchestration system that allows us to ease the burden of configuring, deploying, managing, and monitoring containerized applications. This will allow us to scale our applications into microservices where we can scale each service inside its own instance, which will essentially give us application resilience on top of machine resilience when deployed. Kubernetes also uses hardware more efficiently which allows us to save money by optimizing our infractures resources.

Once the cluster was ready, we used Kubernetes to create a deployment and a service using a YAML file. A deployment allows us to specify what's inside your pod. The service is used to allow us to interact with our pods. Pods are simply packaged up Docker containers that contain our application. YAML files are essentially infrastructure as code for Kuberenetes. Once we have successfully created our YAML file, we can access our Flask application on our local machine. Deploying our application this way will give us resilience for our application. Inside our node, we will have multiple pods with our application. If one application goes down, there will be others running.

Task 1

The first part of this assignment requires that we package up our flask application using Docker. With Docker, we can create Dockerfiles. Dockerfiles are text documents that contain all the commands a user calls on the command line to assemble an image. A docker image is a file used to execute code in a docker container. It is sort of a template. An image can also be seen as a snapshot in a virtual machine. Once we create the Dockerfile, we will use a docker build command to build our image.

We will upload it to DockerHub, which is a library of container images. When we have our images hosted. We will then need to push our image to dockerhub from our local machine to the remote library.

First, we need to make a folder with the necessary Flask files and change directory to it
application.py
helper.py (optional)
requirements.txt
static/
template/

Inside your flask application (application.py), we need to add the following at the bottom of the code…

```
if __name__ = '__main__':
        app.run(debug=True, host='0.0.0.0')
```

Once everything is setup, we can create a Dockerfile
nano Dockerfile

Paste the following into the Dockerfile.

```
FROM python:3.10
COPY . /root/flask
WORKDIR ./root/flask
RUN pip install -r requirements.txt
ENV FLASK_APP=application.py
EXPOSE 5000
CMD flask run --host=0.0.0.0
```

These commands will essentially pull a Python image. The COPY command will then copy all the files in the current directory and move it to a path inside the image. The WORKDIR will change the current working directory into the specified path. The RUN command will be executed in layers and it will install all the dependencies needed for the flask application. To run a flask environment, the environmental variable needs to be set so the ENV command will set the variable. We will also need to expose a port so the flask

application can be accessed. Finally once everything is set up we will run a CMD command which will run once the container runs. This will run the flask application on a specific host address.

Build the Docker images using the following command..
(Change tag to your dockerhub username)…
We are tagging it with the correct DockerHub naming convention so we don't have to retag it in the future.
docker build -t **rixardo**/flask .

Once everything is successful, we can log into DockerHub and push the image to the remote library.
(Change tag to your dockerhub username)…
docker login
docker push **rixardo**/flask

(OPTIONAL) Run the docker images to test if your Flask application works…
(Change tag name to your dockerhub)
docker run -ti --rm -p 8080:5000 rixardo/flask

Task 2

Once everything is pushed up with the right configurations, we need to deploy our Flask application in Kubernetes. We have to install K3D, which is a lightweight tool that is used to interact with kubernetes locally. Once installed, we can run a command to create a cluster with a load balancer mapped to certain ports. The load balancer has an external and internal port. Users access our application using the external port and the load balancer uses port forwarding to send the traffic to the internal port.

We then need to create a YAML file that has a kubernetes deployment and service configured in one file. YAML files are simply configuration files. Deployments allow us to specify what is inside our pods. Pods are essentially a bundle of containers needed to run an application. The YAML file also creates a service which allows us to interact with our pods. Once we have set up our YAML file with the specific ports and configuration, we can build our YAML file. This will create all the configurations we specified. We can then simply access our application on our localhost browser.

Install k3d. K3D is a lightweight tool that is used to interact with Kubernetes locally…
curl -s https://raw.githubusercontent.com/rancher/k3d/master/install.sh | bash

Create a cluster with a load balancer attached. A cluster is a set of nodes that run containerized applications. The load balancer is a load distribution that's attached to this cluster with a specific external and internal port. When a user wants to access our application, they will go to the external port and be forwarded to the internal port.
(Be careful copying. Quotation marks get messed up)…
k3d cluster create flask-cluster -p “8080:8081@loadbalancer”

Once we have created a cluster, we should create a YAML file with out configuration of our flask application…
nano flask.yaml

The following YAML file will create a kubernetes deployment and service in one file. The deployment allows you to specify what's inside your pod. The service simply allows us to interact with our pods. It is important that we keep note of the ports and type in the service. Paste the following inside your YAML file
(Change image location and image to your docker hub username)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-deployment #name of the service once created. Use to access pod
spec:
  selector:
    matchLabels:
      app: flask #app labeled flask
  replicas: 1 # tells deployment to run 1 pods matching the template
  template: # create pods using pod definition in this template
    metadata:
      labels:
        app: flask
    spec: #task definition for each pod
      containers:
      - name: flask
        image: rixardo/flask:latest
        ports:
        - containerPort: 5000

---
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
```

```
  type: LoadBalancer
  ports:
  - port: 8081
    protocol: TCP
    targetPort: 5000
  selector:
    app: flask
```

Once we created the YAML, we can create it using the following command…
kubectl create -f flask.yaml

We can check the status of our application by using the following command.(wait approximately 3 minutes for the application to be READY 1/1)...
kubectl get all

```
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flask-deployment    1/1     1            1           2m20s
```

Once our application is ready go to the following in your local browser…
localhost:8080

You will see your flask application

Helpful Commands

Delete a cluster using…
k3d cluster delete name

Delete a pod using…
kubectl delete pods podname

Delete a deployment using…
kubectl delete deploy <name>

Why are we using YAML vs YML?

Both are same in interpretation and syntax. In Older versions of windows, Extensions are restricted to 3 letters like .yml. Nowadays, There is no OS system level enforce to have 3 letters in extensions. Most of the yaml users are using .yaml as their preferred choice.

Task 3 - Wishlist (Optional)

For this optional task, we had to configure kubernetes to allow our cluster to be accessed on 3 different ports. Each port has a different part of a wishlist application. When attaching a load balancer, it opened up a port on the internal and external host system. The service created was configured to listen for a port and direct traffic to the container port that has our specific application. Each port represents a different application, wishlist, authentication, and catalog. The targetport in the YAML file refers to the containerport. This is equivalent to the expose command in Kubernetes. We are simply connecting the load balancer to our application with the service.

Create a cluster with 3 load balancer
(Be careful copying. Quotation marks get messed up)…
k3d cluster create wishlist-cluster -p “8081:8081@loadbalancer” -p “8082:8082@loadbalancer” -p “8083:8083@loadbalancer”

Once we have created a cluster, create a yaml file
nano wishlist-deployment.yaml

Paste the following inside your YAML file…

```
# Wishlist deployment yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: wishlist-deployment
  labels:
    app: wishlist
spec:
  replicas: 3 #We always want more than 1 replica for HA
  selector:
    matchLabels:
      app: wishlist
  template:
    metadata:
      labels:
        app: wishlist
    spec:
      containers:
      - name: wishlist #1st container
        image: karthequian/wishlist:1.0 #Dockerhub image
        ports:
        - containerPort: 8080 #Exposes the port 8080 of the container
        env:
        - name: PORT #Env variable key passed to container that is read by app
          value: "8080" # Value of the env port.
```

```
      - name: catalog #2nd container
        image: karthequian/wishlist-catalog:1.0
        ports:
        - containerPort: 8081
        env:
        - name: PORT
          value: "8081"
      - name: auth #3rd container
        image: karthequian/wishlist-auth:1.0
        ports:
        - containerPort: 8082
        env:
        - name: PORT
          value: "8082"
---
kind: Service
apiVersion: v1
metadata:
  name: wishlist-service
  namespace: default
spec:
  type: LoadBalancer
  selector:
    app: wishlist
  ports:
  - name: wishlist-port
    protocol: TCP
    port: 8081
    targetPort: 8080
  - name: wishlist-auth-port
    protocol: TCP
    port: 8082
    targetPort: 8081
  - name: wishlist-catalog-port
    protocol: TCP
    port: 8083
    targetPort: 8082
```

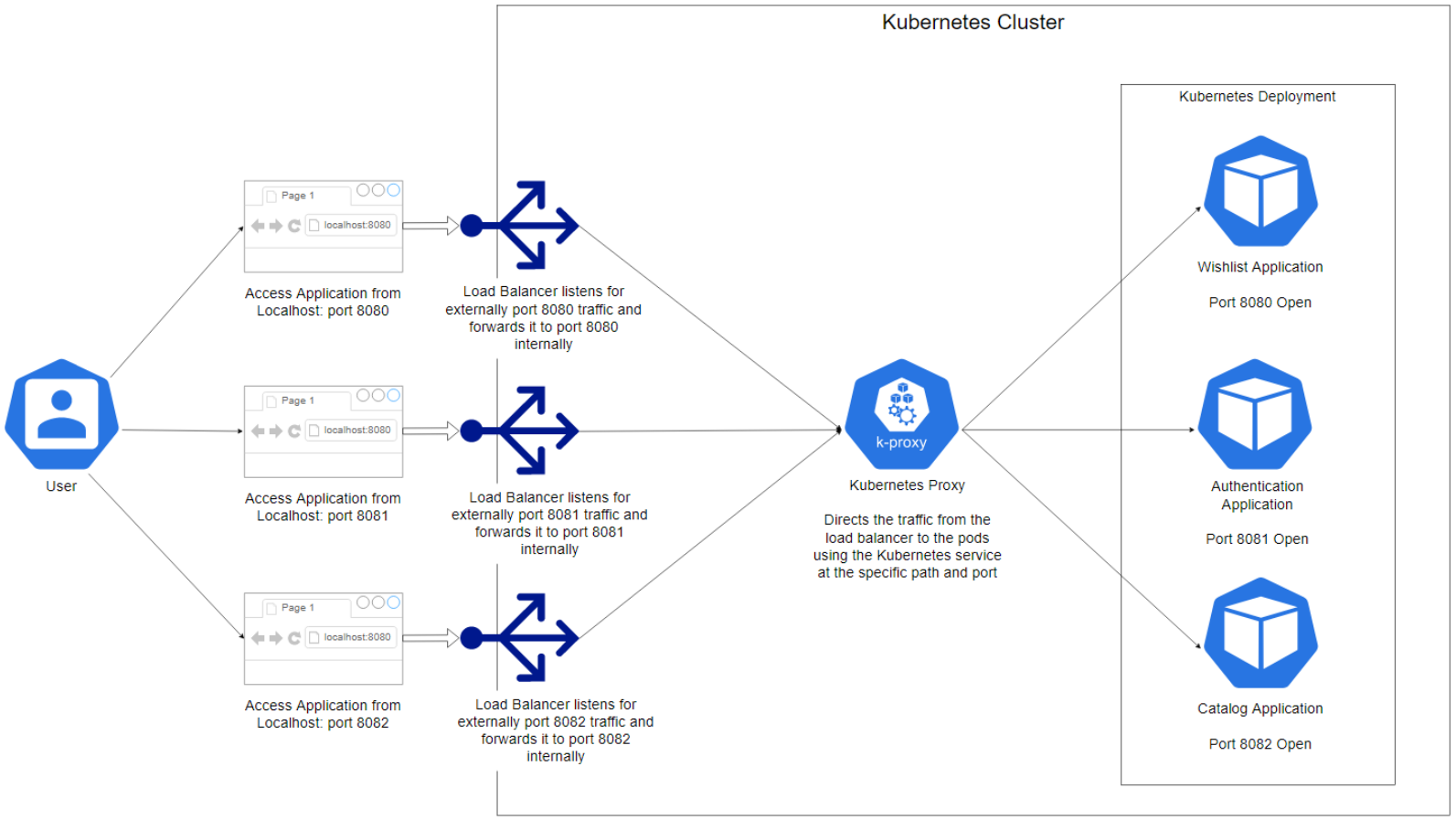
Run the YAML file using…
kubectl create -f wishlist-deployment.yaml

Run the following command…
kubectl get all

Wait until the PODS are 3/3…

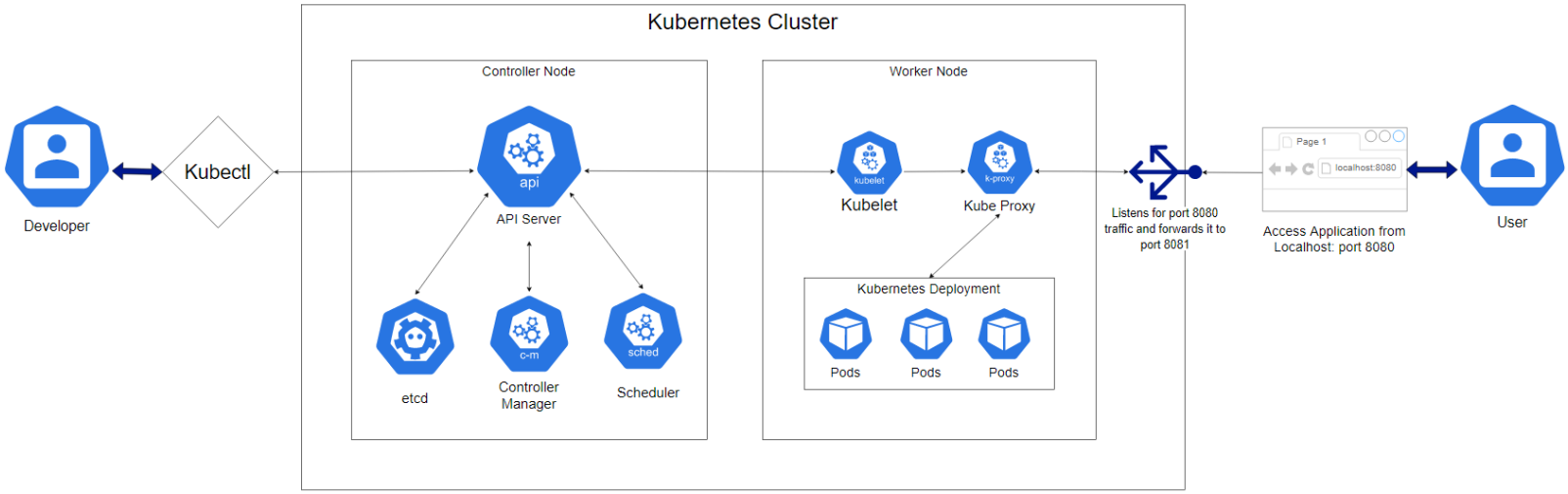
```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/wishlist-deployment   3/3     3            3           2m47s
```

Go to the following inside your browser…
localhost:8080/metrics
localhost:8081/metrics
localhost:8082/metrics



If a user wants to access a specific application, they will go on their browser and go to “localhost:8080”, “localhost:8081”, or “localhost:8082”. There are 3 load balancers attached to the kubernetes cluster that will listen for external traffic and redirect it to an internal port. The load balancer talks with the kubernetes proxy (service) that will direct traffic to the pods and interact with it. Each pod has a port that is open.

K8s Assignment Topology



This system design shows two different paths of traffic from Developer and User. Developers will talk to Kubectl locally. Kubectl is used to interact with the controller node. Inside of the controller node is the API server which is responsible for handling user requests, storing data in etcd, and talking to Kubelets to implement. The etcd is a replicated datastore that holds on to our pods/deployment/other specifications. The controller manager is a daemon that embeds the core control loops shipped with kubernetes. In other words, it implements abstractions like deployments. And finally, inside the controller node is the scheduler that decides which pod runs on which node.

Traffic is sent from the controller node to the worker node that has the Kubelet, Kube Proxy, and pods. The kubelet is responsible for talking to the controller node. It manages pods and containers. Kubelet also passes information to the API server which allows us to see information about the nodes such as status. The Kube-proxy is a key component of any kubernetes deployment. The role of kube proxy is to load balancer traffic that is destined for services to the correct container pods. When a regular user accesses our application on their browser as a specific port, the load balancer that is attached to the cluster will forward the port to another port. The kube proxy will balance traffic to the correct container pods at the redirected port.

Traffic Flow: User to Application