SOFT8026 – Lab 05

The VM in the labs has MicroK8S installed. For more information, including how to install on your own machine, visit https://microk8s.io/docs/

You should prefix all the kubectl commands below with “microk8s.”, e.g.   
microk8s.kubectl exec -ti busybox -- nslookup redis

# Moving our Docker Compose app to Kubernetes

We will use Docker Hub to store our Docker images. Go to [http://hub.docker.com](http://hub.docker.com/) and create an account if you don’t have one. Log in.

Click on

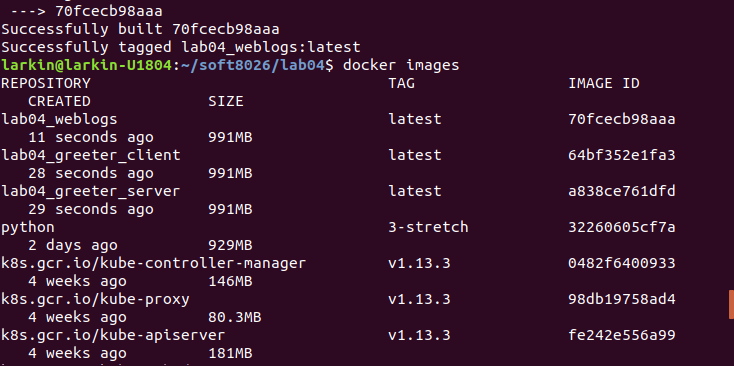
Fill in the details for the first of 3 repositories (one for each of our microservices). Enter the name “weblogs” and the description “The weblogs microservice” and click on “Create”. Create two more for “greeter\_server” and “greeter\_client”.

One little change we will make is in greeter\_client.py. Kubernetes doesn’t like underscore in its service names, so make the following highlighted change:



(Note: if you want your docker compose app to continue working, change all underscores to dashes.)

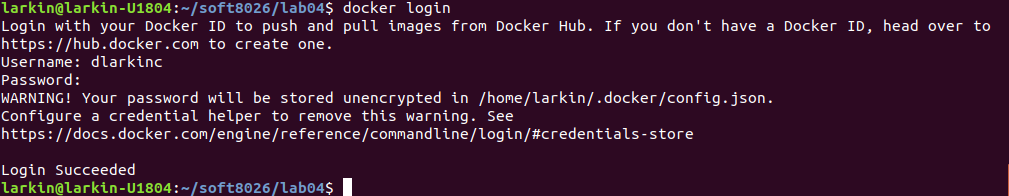
Run docker-compose build to rebuild all the images again.



Note the names, e.g. lab04\_weblogs. Then push these images to your Docker Hub repository. Begin by tagging, e.g.:



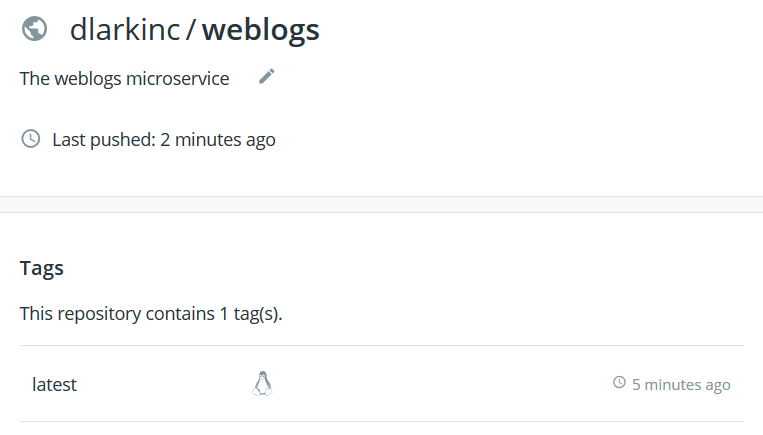
Login to Docker Hub:



Push each of the 3 to Docker Hub, e.g.:



Click on the Repositories link in Docker Hub and find your repositories and click into them. You should see something like this in each:



We are ready to create some Kubernetes manifests.

# Create Deployment and Service Manifests

Rather than create Pod manifests, we will use Deployment manifests. We need 4 services to cater for the 4 microservices from Lab 04 – greeter\_server, greeter\_client, weblogs and redis. A deployment for each will create the pods, but they will be individual pods with their own random ip addresses and hostnames. We need a consistent single hostname, so for each deployment manifest, we will create a service manifest (except for the greeter\_client, which does not receive incoming connections).

Create a new lab06 folder. In it, create a new file called greeter-server-deploy.yml as follows:

apiVersion: apps/v1

kind: Deployment

metadata:

name: greeter-server-deploy

spec:

replicas: 6

selector:

matchLabels:

app: greeter-server

minReadySeconds: 6

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 1

maxSurge: 1

template:

metadata:

labels:

app: greeter-server

spec:

containers:

- name: greeter-server

image: dlarkinc/greeter\_server:latest

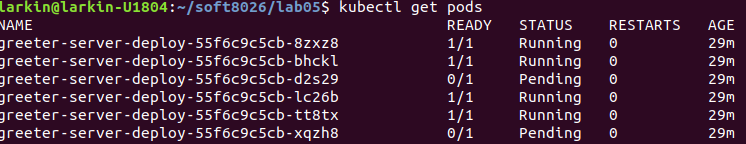
ports:

- containerPort: 50051

Issue the command: kubectl create -f greeter-server-deploy.yml

6 replicas will obviously be way more than enough, but it works for illustration purposes. We are opening up 1 port for grpc. We are also getting the image from Docker hub (or locally if it exists as a local image with that tag).

However, this will just create 6 pods:



Which one does our client connect to? For example  is a valid hostname, but it is unstable – a crash will result in a new pod with a different name in its place.

We need a service to expose our deployment. Create the file greeter-service-svc.yml:

apiVersion: v1

kind: Service

metadata:

name: greeter-server

labels:

app: greeter-server

spec:

ports:

- port: 50051

protocol: TCP

selector:

app: greeter-server

It will pick up the 6 pods because they were labelled with app=greeter-server.

Issue the command: kubectl create -f greeter-server-svc.yml

Then…



That server microservice needs a redis database to store its log messages. Create the following…

redis-deploy.yml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: redis-deploy

spec:

selector:

matchLabels:

app: redis

template:

metadata:

labels:

app: redis

spec:

containers:

- name: redis

image: redis:alpine

ports:

- containerPort: 6379

redis-svc.yml:

apiVersion: v1

kind: Service

metadata:

labels:

app: redis

name: redis

spec:

selector:

app: redis

type: NodePort

ports:

- port: 6379

nodePort: 30001

protocol: TCP

Notice that for this we have created a NodePort with external host port 30001. This was not needed, but it does allow us to run redis-cli -p 30001 back on the host to query our redis database.

Run the kubectl create command with each.

Let’s create the weblogs service next…

weblogs-deploy.yml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: weblogs-deploy

spec:

replicas: 4

selector:

matchLabels:

app: weblogs

minReadySeconds: 10

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 1

maxSurge: 1

template:

metadata:

labels:

app: weblogs

spec:

containers:

- name: weblogs

image: dlarkinc/weblogs:latest

ports:

- containerPort: 5000

weblogs-svc.yml:

apiVersion: v1

kind: Service

metadata:

labels:

app: weblogs

name: weblogs

spec:

selector:

app: weblogs

type: NodePort

ports:

- port: 5000

nodePort: 30000

protocol: TCP

This will allow us to connect to http://127.0.0.1:30000 from the host to access the web page. Use kubectl create with both files to create our weblogs microservice.

You can try accessing [http://127.0.0.1:30000](http://127.0.0.1:30000/) to see what happens. Hopefully now when you visit [http://127.0.0.1:30000](http://127.0.0.1:30000/) you will see nothing worse than a blank page (no guarantees though, this is a fairly unstable non-production environment).

Lastly, we need to generate some log messages by creating the client. This might be best done using a Job (see slide set #13), but we will stick with a deployment for now. Note: we don’t need a service this time because we don’t need to receive any connections to the client.

greeter-client-deploy.yml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: greeter-client-deploy

spec:

replicas: 10

selector:

matchLabels:

app: greeter-client

minReadySeconds: 10

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 1

maxSurge: 1

template:

metadata:

labels:

app: greeter-client

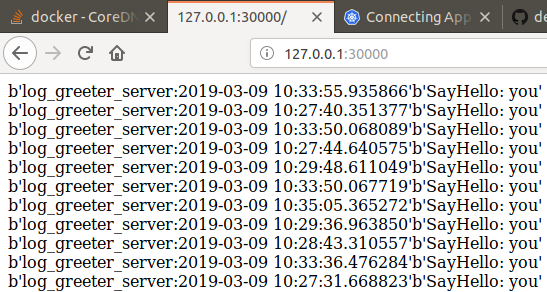
spec:

containers:

- name: greeter-client

image: dlarkinc/greeter\_client:latest

kubectl create the deployment and the logs in redis should start filling up. There may be a few misfires with name resolution, but there should be a steady stream of log messages. Keep refreshing [http://127.0.0.1:30000](http://127.0.0.1:30000/).



# Debugging

Busybox is a useful tool for debugging. Create the file busybox-test.yml with the following:

apiVersion: v1

kind: Pod

metadata:

name: busybox

namespace: default

spec:

containers:

- name: busybox

image: busybox:1.28

command:

- sleep

- "3600"

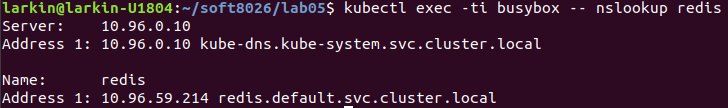
imagePullPolicy: IfNotPresent

restartPolicy: Always

Kubectl create the pod and then enter the following:

kubectl exec -ti busybox -- nslookup redis

You should see output like this:



But as has been said before, the DNS can be a bit unstable, so you might need to kill those dns pods to get them restarted.

You can also regularly check on the status of your pods, services and deployments using the correct kubectl get commands.