

## Containers A-Z

An overview of Containers, Docker, Kubernetes, Istio, Helm, Kubernetes Operators and GitOps

### Class Labs

Version 3.0 by Brent Laster

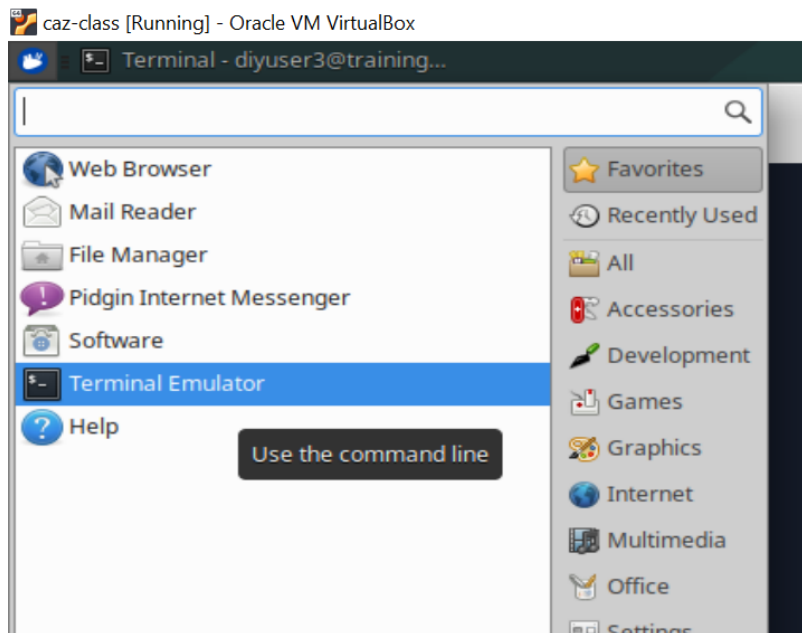
4/1/2021

**Important Prereq:** These labs assume you have already followed the instructions in the separate setup document and have VirtualBox up and running on your system and have downloaded the *caz-class.ova* file and loaded it into VirtualBox. If you have not done that, please refer to the setup document for the workshop and complete the steps in it before continuing!

### Lab 1- Building Docker Images

**Purpose:** In this lab, we'll see how to build Docker images from Dockerfiles.

1. Open a terminal session by using the one on your desktop or clicking on the little mouse icon in the upper left corner and selecting **Terminal Emulator** from the drop-down menu.



2. To start, we'll need to get the paused minikube environment up and running. Open a terminal window and enter the command below.

```
$ sudo minikube start --vm-driver=none
```

3. Switch into the working directory for the main class and update it with the latest versions of the files. Then switch into the directory for our docker work.

```
$ cd caz-class
```

```
$ git pull
```

```
$ cd roar-docker
```

4. Do an **ls** command and take a look at the files that we have in this directory.

```
$ ls
```

5. Take a moment and look at each of the files that start with “Dockerfile”. See if you can understand what’s happening in them.

```
$ cat Dockerfile_roar_db_image
```

```
$ cat Dockerfile_roar_web_image
```

6. Now let’s build our docker database image. Type (or copy/paste) the following command: (Note that there is a space followed by a dot at the end of the command that must be there.)

```
$ docker build -f Dockerfile_roar_db_image -t roar-db .
```

7. Next build the image for the web piece. This command is similar except it takes a build argument that is the war file in the directory that contains our previously built webapp.

(Note the space and dot at the end again.)

```
$ docker build -f Dockerfile_roar_web_image --build-arg warFile=roar.war -t roar-web .
```

- Now, let's tag our two images for our local registry (running on localhost, port 5000). We'll give them a tag of "v1" as opposed to the default tag that Docker provides of "latest".

```
$ docker tag roar-web localhost:5000/roar-web:v1
```

```
$ docker tag roar-db localhost:5000/roar-db:v1
```

- Do a docker images command to see the new images you've created.

```
$ docker images | grep roar
```

END OF LAB

## Lab 2 – Composing images together

**Purpose:** In this lab, we'll see how to make multiple containers execute together with docker compose and use the docker inspect command to get information to see our running app.

- Take a look at the docker compose file for our application and see if you can understand some of what it is doing.

```
$ cat docker-compose.yml
```

- Run the following command to compose the two images together that we built in lab 1.

```
$ docker-compose up
```

- You should see the different processes running to create the containers and start the application running. Take a look at the running containers that resulted from this command.

Note: We'll leave the processes running in the first session, so **open a second terminal emulator** and enter the command below.

```
$ docker ps | grep roar
```

4. Make a note of the first 3 characters of the container id (first column) for the web container (row with **roar-web** in it). You'll need those for the next step.
5. Let's find the web address so we can look at the running application. To do this, we will search for the information via a docker **inspect** command. Enter this command in the **second** terminal session, substituting in the characters from the container id from the step above for "<container id>" - the one for *roar-web*.

(For example, if the line from docker ps showed this:

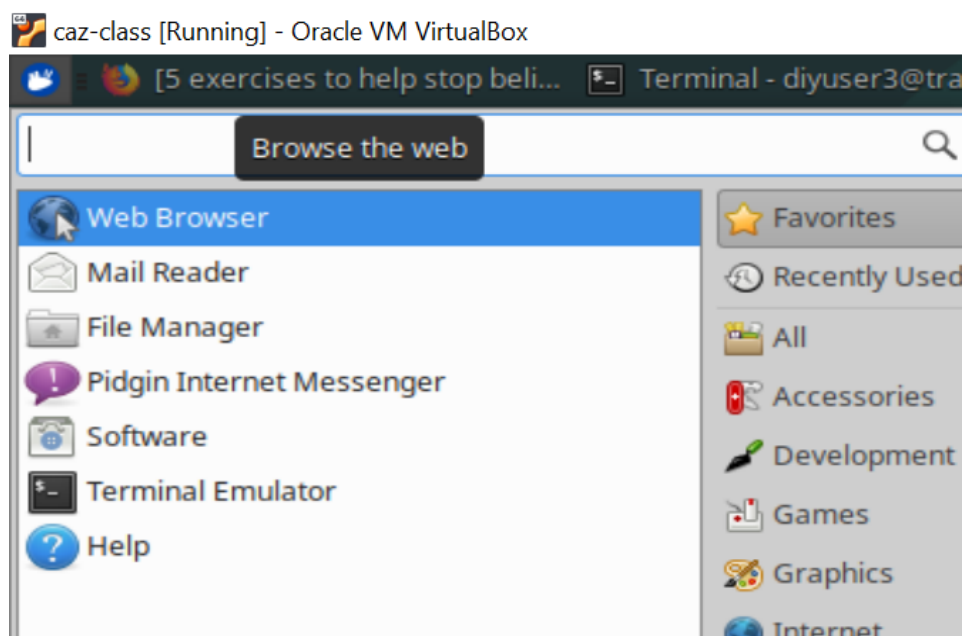
```
237a48a2aeb8    roar-web    "catalina.sh run"    About a minute
ago    Up About a minute    0.0.0.0:8089->8080/tcp
```

then <container id> could be "**237**". Also note that "IPAddress" is case-sensitive.)

Make a note of the url that is returned.

```
$ docker inspect <container id> | grep IPAddress
```

6. Open a web browser by clicking on the mouse icon in the upper left and then selecting the **Web Browser** menu item.



7. In the browser, go to the url below, substituting in the ip address from the step above for “<ip address>”. (Note the :8080 part added to the ip address)

<http://<ip address>:8080/roar/>

8. You should see the running app on a screen like the following:

Id	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
1	Road Runner	bird	1935-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
2	Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
3	Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	...inator
4	Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
5	Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun

END OF LAB

## Lab 3 – Debugging Docker Containers

**Purpose:** While our app runs fine here, it’s helpful to know about a few commands that we can use to learn more about our containers if there are problems.

1. Let’s get a description of all of the attributes of our containers. For these commands, use the same 3 character container id you used in step 2.

Run the inspect command. Take a moment to scroll around the output.

```
$ docker inspect <container id>
```

2. Now, let’s look at the logs from the running container. Scroll around again and look at the output.

```
$ docker logs <container id>
```

3. While we're at it, let's look at the history of the image (not the container).

```
$ docker history roar-web
```

4. Now, let's suppose we wanted to take a look at the actual database that is being used for the app. This is a mysql database but we don't have mysql installed on the VM. So how can we do that? Let's connect into the container and use the mysql version within the container. To do this we'll use the "docker exec" command. First find the container id of the db container.

```
$ docker ps | grep roar-db
```

5. Make a note of the first 3 characters of the container id (first column) for the db container (row with **roar-db** in it). You'll need those for the next step.

6. Now, let's exec inside the container so we can look at the actual database.

```
$ docker exec -it <container id> bash
```

Note that the last item on the command is the command we want to have running when we get inside the container – in this case the bash shell.

7. Now, you'll be inside the db container. Check where you are with the pwd command and then let's run the mysql command to connect to the database. (Type these at the **/#** prompt. Note no spaces between the options -u and -p and their arguments. You need only type the part in bold.)

```
root@container-id:/# pwd
root@container-id:/# mysql -uadmin -padmin registry
```

(Here -u and -p are the userid and password respectively and registry is the database name.)

8. You should now be at the “mysql>” prompt. Run a couple of commands to see what tables we have and what is in the database. (Just type the parts in **bold**.)

```
mysql> show tables;
mysql> select * from agents;
```

9. Exit out of mysql and then out of the container.

```
mysql> exit
root@container-id:/# exit
```

10. Let’s go ahead and push our images over to our local registry so they’ll be ready for Kubernetes to use.

```
$ docker push localhost:5000/roar-web:v1
$ docker push localhost:5000/roar-db:v1
```

11. Since we no longer need our docker containers running or the original images around, let’s go ahead and get rid of them with the commands below. (Hint: `docker ps | grep roar` will let you find the ids more easily)

Stop the containers

```
$ docker stop <container id for roar-web>
$ docker stop <container id for roar-db>
```

Remove the containers

```
$ docker rm <container id for roar-web>
$ docker rm <container id for roar-db>
```

Remove the images

```
$ docker rmi -f roar-web
$ docker rmi -f roar-db
```

END OF LAB

## Lab 4 - Exploring and Deploying into Kubernetes

**Purpose:** In this lab, we'll start to learn about Kubernetes and its object types, such as nodes and namespaces. We'll also deploy a version of our app that has had Kubernetes yaml files created for it.

1. Before we can deploy our application into Kubernetes, we need to have appropriate Kubernetes manifest yaml files for the different types of k8s objects we want to create. These can be separate files or they can be combined. For our project, there is a combined one (deployments and services for both the web and db pieces) already setup for you in the `caz-class/roar-k8s` directory. Change into that directory and take a look at the yaml file there for the Kubernetes deployments and services.

```
$ cd ~/caz-class/roar-k8s
```

```
$ cat roar-complete.yaml
```

See if you can identify the different services and deployments in the file.

2. We're going to deploy these into Kubernetes into a namespace. Take a look at the current list of namespaces and then let's create a new namespace to use.

```
$ kubectl get ns
```

```
$ kubectl create ns roar
```

3. Now, let's deploy our yaml specifications to Kubernetes. We will use the `apply` command and the `-f` option to specify the file. (Note the `-n` option to specify our new namespace.)

```
$ kubectl -n roar apply -f roar-complete.yaml
```

After you run these commands, you should see output like the following:

```
deployment.extensions/roar-web created
service/roar-web created
deployment.extensions/mysql created
service/mysql created
```

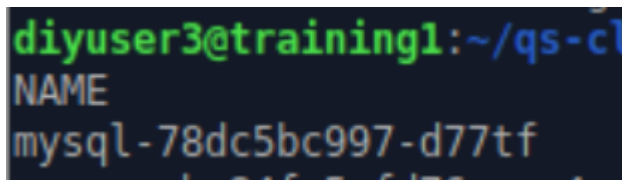


4. Now, let's look at the pods currently running in our "roar" namespace.

```
$ kubectl get pods -n roar
```

Notice the STATUS field. What does the "ImagePullBackOff" or "ErrImagePull" status mean?

5. Let's check the logs of the pod to learn more about what's going on. Highlight and copy the NAME of the db pod ( the one that starts with "mysql") to use in the next step.



```
diyuser3@training1:~/qs-cl
NAME
mysql-78dc5bc997-d77tf
```

6. Now run this command to see the logs (note again that we add the -n option to specify the namespace):

```
$ kubectl logs <paste pod name here> -n roar
```

(example: `kubectl logs mysql-78dc5bc997-d77tf -n roar`)

7. The output here confirms what is wrong – notice the part on "trying and failing to pull image". To get the overall view (description) of what's in the pod and what's happening with it, we'll use the "describe" command. Use the command below, pasting in the full name of the container that you copied in the previous step.

```
$ kubectl -n roar describe pod <paste pod name here>
```

(example: `kubectl -n roar describe pod mysql-78dc5bc997-d77tf`)

8. Near the bottom of this output, notice the "Events" messages:

## Events:

Type	Reason	Age	From	Message
Normal	Scheduled	7m24s	default-scheduler	Successfully assigned roar/mysql-78dc5bc997-d77tf to minikube
Normal	Pulling	5m48s (x4 over 7m20s)	kubelet, minikube	Pulling image "localhost:5000/roar-db-v1"
Warning	Failed	5m48s (x4 over 7m20s)	kubelet, minikube	Failed to pull image "localhost:5000/roar-db-v1": rpc error: code = Unknown desc = Error response from daemon: manifest for localhost:5000/roar-db-v1 not found
Warning	Failed	5m48s (x4 over 7m20s)	kubelet, minikube	Error: ErrImagePull
Warning	Failed	5m35s (x7 over 7m18s)	kubelet, minikube	Error: ImagePullBackOff
Normal	BackOff	2m17s (x21 over 7m18s)	kubelet, minikube	Back-off pulling image "localhost:5000/roar-db-v1"

9. Remember that we tagged the images for our local registry as **localhost:5000/roar-db:v1** and **localhost:5000/roar-web:v1**. But if you scroll back up and look at the “Image” property in the describe output, you’ll see that it actually specifies “localhost:5000/roar-db-v1”.

```
diyuser3@training1:~/qs-class/roar-k8s$ kubectl -n roar describe pod mysql-7f689478cd-wp6tg
Name:          mysql-7f689478cd-wp6tg
Namespace:     roar
Priority:       0
PriorityClassName:  <none>
Node:          minikube/10.0.2.15
Start Time:    Sun, 21 Jul 2019 00:59:40 -0400
Labels:        name=mysql
               pod-template-hash=7f689478cd
Annotations:   <none>
Status:        Pending
IP:            172.17.0.21
Controlled By: ReplicaSet/mysql-7f689478cd
Containers:
  mysql:
    Container ID:  (empty)
    Image:          localhost:5000/roar-db-v1
    Image ID:       (empty)
    Ports:          3306/TCP
```

10. It is looking for an image with the “-v1” as part of the name. But that’s not what we tagged ours as. To fix this, edit the roar-complete.yaml file and modify the “Image” properties to change the “-” to a “:” for the web image (only). Let’s see if this fixes the problem. Still in the caz-class/roar-k8s directory:

```
$ gedit roar-complete.yaml
```

In the editor, change line 17 from

**image: localhost:5000/roar-web-v1**

to

**image: localhost:5000/roar-web:v1**

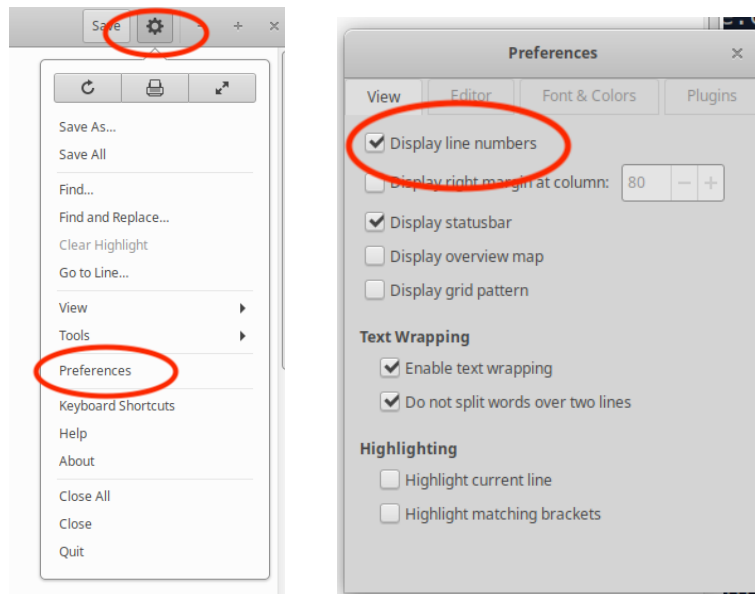
Also change line 54 from

**image: localhost:5000/roar-db-v1**

to

**image: localhost:5000/roar-db:v1**

(Note: You can turn on line numbers in gedit through the process below.)



11. After you make your changes, save the file and close the editor. Now, in the other terminal window, start a command to watch the pods (the -w option) so we can see when changes occur.

**\$ kubectl get pods -n roar -w**

12. Now, in the second emulator window (the one in the roar-k8s directory), run a command to apply the changed file.

**\$ kubectl apply -n roar -f roar-complete.yaml**

13. Observe what happens in the window with the watched pods afterwards. You should be able to see Kubernetes terminating the old pod and starting up a new one. Eventually the new one should show as running.

```
diyuser3@training1:~/qs-class/roar-k8s$ kubectl get pods -n roar -w
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-78dc5bc997-d77tf	0/1	ImagePullBackOff	0	26m
roar-web-84fc5cfd76-wvz4g	0/1	ImagePullBackOff	0	26m
roar-web-84fc5cfd76-wvz4g	0/1	ErrImagePull	0	26m
mysql-78dc5bc997-d77tf	0/1	ErrImagePull	0	26m
roar-web-84fc5cfd76-wvz4g	0/1	ImagePullBackOff	0	26m
mysql-78dc5bc997-d77tf	0/1	ImagePullBackOff	0	26m
roar-web-556c44c588-q6vtf	0/1	Pending	0	0s
roar-web-556c44c588-q6vtf	0/1	Pending	0	0s
roar-web-84fc5cfd76-wvz4g	0/1	Terminating	0	31m
roar-web-556c44c588-q6vtf	0/1	ContainerCreating	0	1s
roar-web-84fc5cfd76-wvz4g	0/1	Terminating	0	31m
roar-web-84fc5cfd76-wvz4g	0/1	Terminating	0	31m
roar-web-84fc5cfd76-wvz4g	0/1	Terminating	0	31m
roar-web-556c44c588-q6vtf	1/1	Running	0	9s
mysql-78dc5bc997-d77tf	0/1	ErrImagePull	0	31m
mysql-78dc5bc997-d77tf	0/1	ImagePullBackOff	0	32m

14. Even though we did not directly change the deployment, this should have fixed that also. You can verify by looking at the deploy(ments) again.

```
$ kubectl get deploy -n roar
```

15. With everything running, we can now actually look at the application running (in Kubernetes). Get a list of services for our namespace.

```
$ kubectl -n roar get svc
```

16. Note that the type of service for roar-web is “NodePort”. This means we have a port open on the Kubernetes node that we can access the service through. Find the nodePort under the PORT(S) column and after the service port (8089) and before the “TCP”. For example, if we have **8089:31790/TCP** in that column, then the actual nodePort we need is **31790**.

17. In the web browser, go to the url below, substituting in the nodePort from the step above for “<nodePort>”. You should see the running application.

<http://localhost:<nodePort>/roar/>

END OF LAB

## Lab 5 - Working with Kubernetes secrets, and configmaps

**Purpose:** In this lab we'll get some practice storing secure and insecure information in a way that is accessible to k8s but not stored in the usual deployment files.

1. Cat the roar-complete.yaml and look at the “env” block that starts at line 58. We really shouldn't be exposing usernames and passwords in here.

```
$ cat -n roar-complete.yaml
```

2. Let's explore two ways of managing environment variables like this so they are not exposed - Kubernetes “secrets” and “configmaps”. First we'll look at what a default secret does by running the base64 encoding step on our two passwords that we'll put into a secret. **Change into roar-k8s directory** and then run these commands (the first encodes our base password and the second encodes our root password ).

```
$ echo -n 'admin' | base64
```

This should yield:

```
YWRtaW4=
```

Then do:

```
$ echo -n 'root+1' | base64
```

This should yield:

```
cm9vdCsx
```

3. Now we need to put those in the form of a secrets manifest (yaml file for Kubernetes). For convenience, there is already a “mysqlsecret.yaml” file in the same directory with this information. Take a quick look at it and then use the apply command to create the actual secret.

```
$ cat mysqlsecret.yaml
```

```
$ kubectl -n roar apply -f mysqlsecret.yaml
```

4. Now that we have the secret created in the namespace, we need to update our spec to use the values from it. The change will look like this:

from:

- name: MYSQL\_PASSWORD  
value: admin
- name: MYSQL\_ROOT\_PASSWORD  
value: root+1

to:

- name: MYSQL\_PASSWORD  
valueFrom:  
secretKeyRef:  
name: mysqlsecret  
key: mysqlpassword
- name: MYSQL\_ROOT\_PASSWORD  
valueFrom:  
secretKeyRef:  
name: mysqlsecret  
key: mysqlrootpassword

5. We also have the MYSQL\_DATABASE and MYSQL\_USER values that we probably shouldn't expose in here. Since these are not sensitive data, let's put these into a Kubernetes ConfigMap and update the spec to use that. For convenience, there is already a "mysql-configmap.yaml" file in the same directory with this information. Take a quick look at it and then use the apply command to create the actual secret.

```
$ cat mysql-configmap.yaml
```

```
$ kubectl -n roar create -f mysql-configmap.yaml
```

6. Similar to the changes to use the secret, we would need to change the main yaml file to use the new configmap. That change would look like this:

from:

- name: MYSQL\_DATABASE  
value: registry

to

- name: MYSQL\_DATABASE  
valueFrom:  
configMapKeyRef:  
name: mysql-configmap  
key: mysql.database

And from:

- name: MYSQL\_USER  
value: admin

to

- name: MYSQL\_USER  
valueFrom:  
configMapKeyRef:  
name: mysql-configmap  
key: mysql.user

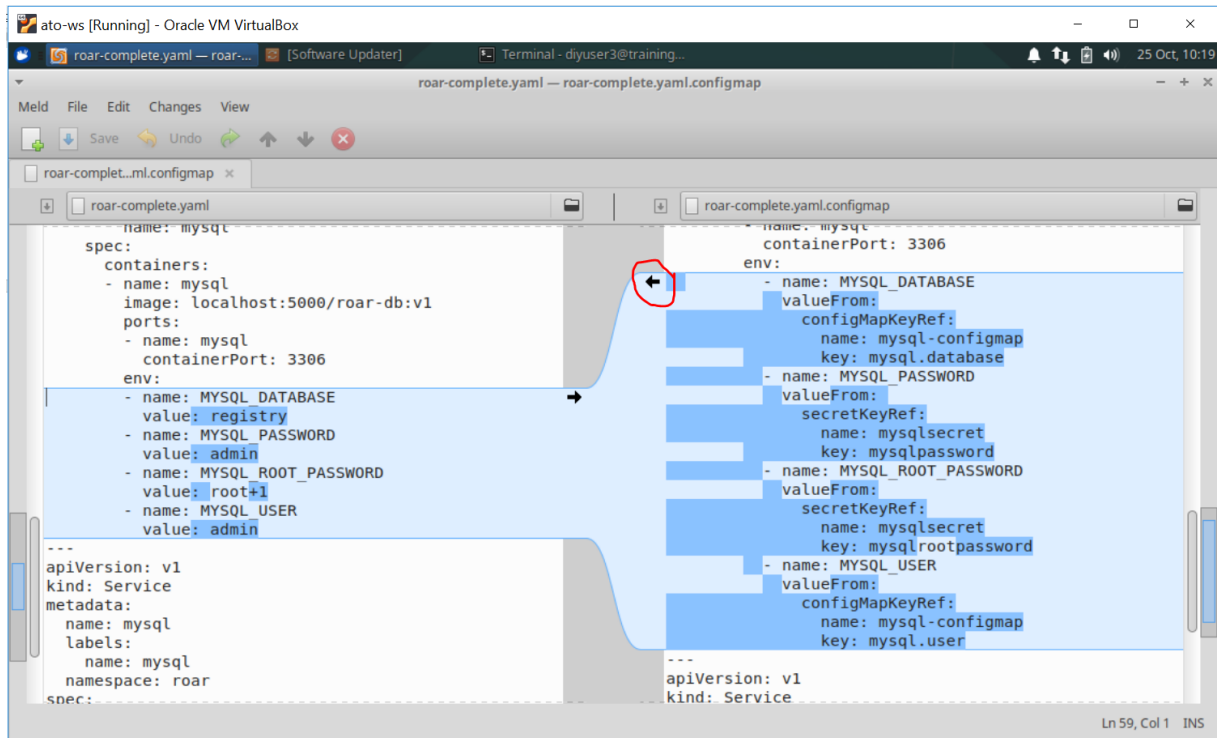
7. In the current directory, there's already a "roar-complete.yaml.configmap" file with the changes in it for accessing the secret and the configmap. Diff the two files with the visual diff tool "meld" to see the differences.

```
$ meld roar-complete.yaml roar-complete.yaml.configmap
```

(You may need to stretch the meld window to be able to see the differences.)

8. Now we'll update our **roar-complete.yaml** file with the needed changes. To save trying to get the yaml all correct in a regular editor, we'll just use the meld tool's merging ability.

In the meld window, on the right pane (the one with roar-complete.yaml.configmap), click the arrow that points left to replace the code in our roar-complete.yaml file with the new code from the roar-complete.yaml.configmap file. (In the figure below, this is the arrow that is circled.)



9. You should then see messages pop up that the files are identical. **Click on the Save button at the top to save the changes.** Then you can **close** the meld application.
10. Apply the new version of the yaml file to make sure it is syntactically correct.

```
$ kubectl apply -f roar-complete.yaml
```

END OF LAB



## Lab 6 – Working with persistent storage – Kubernetes Persistent volumes and Persistent volume claims

**Purpose:** In this lab, we'll see how to connect pods with external storage resources via persistent volumes and persistent volume claims.

1. While we can modify the containers in pods running in the Kubernetes namespaces, we need to be able to persist data outside of them. This is because we don't want the data to go away when something happens to the pod. Let's take a quick look at how volatile data is when just stored in the pod. First, open a browser with the instance that you're running in the "roar" namespace.

Get the endpoint for the instance.

```
$ kubectl get ep -n roar
```

This should return an ip address and port for the container for the NodePort. Take that ip address and port (the one for **roar-web**), go to a browser and plug in the url below. This should bring up a running instance of the app.

```
http://<ip address and port>/roar/
```

2. There is a very simple script in our roar-k8s directory that we can run to insert a record into the database in our mysql pod. If you want, you can take a look at the file *update-db.sh* to see what it's doing. Run it, refresh the browser, and see if the additional record shows up. (Make sure to pass in the namespace – "roar" and don't forget to refresh the browser afterwards.)

```
$ ./update-db.sh <namespace> (such as ./update-db.sh roar)
```

3. After you refresh your browser, you should see a record for "Woody Woodpecker" in the table. Now, what happens if we delete the mysql pod and let Kubernetes recreate it? Get the pod name from the first command and then plug that into the second.

```
$ kubectl get pods -n roar
```

```
$ kubectl delete pod -n roar < mysql pod name >
```

4. After a moment, a new mysql pod will be started up. When that happens, refresh the browser and notice that the record we added for “Woody Woodpecker” is no longer there. It disappeared when the pod went away.
5. This happened because the data was all contained within the pod’s filesystem. In order to make this work better, we need to define a persistent volume (PV) and persistent volume claim (PVC) for the deployment to use/mount that is outside of the pod. As with other objects in Kubernetes, we first define the yaml that defines the PV and PVC. The file storage.yaml defines these for us. Take a look at it now.

```
$ cat storage.yaml
```

6. Now create the objects specified here.

```
$ kubectl create -n roar -f storage.yaml
```

7. Now that we have the storage objects instantiated in the namespace, we need to update our spec to use the values from it. In the file the change would be to add the lines in bold in the container’s spec area:

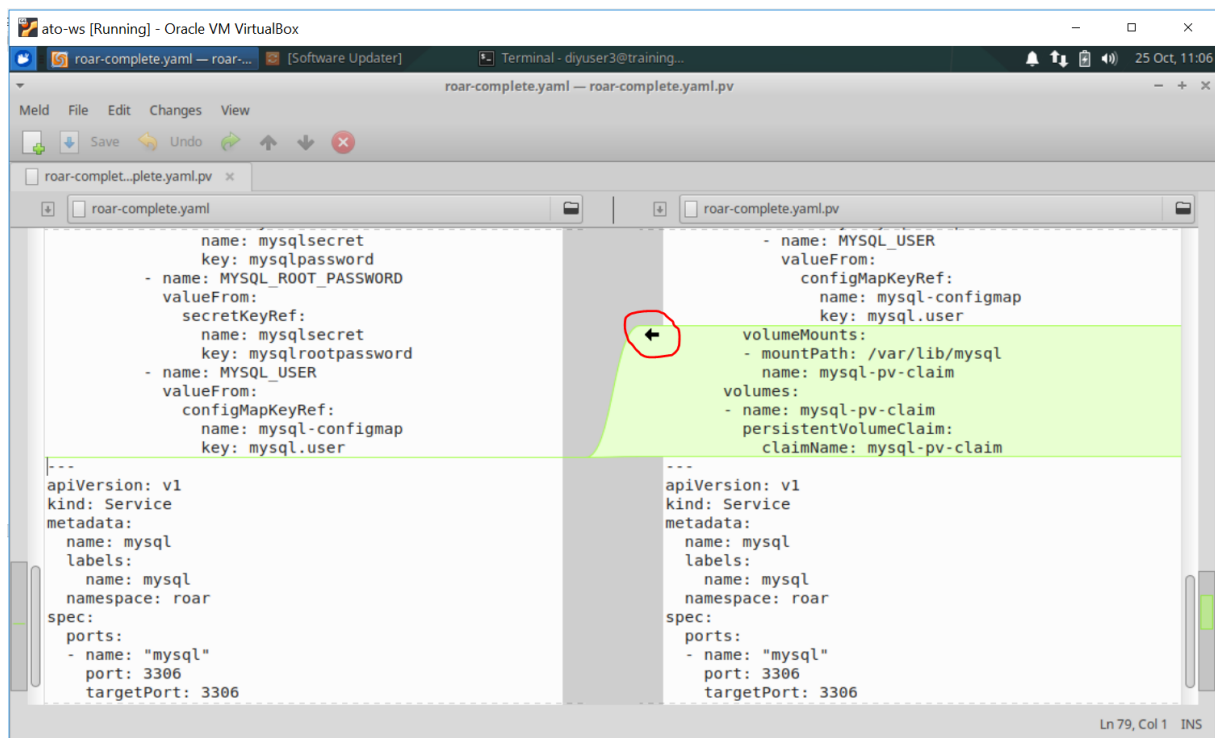
```
spec:
  containers:
    - name: mysql
      ...
      - name: MYSQL_USER
        valueFrom:
          configMapKeyRef:
            name: mysql-configmap
            key: mysql.user
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: mysql-pv-claim
    volumes:
    - name: mysql-pv-claim
      persistentVolumeClaim:
        claimName: mysql-pv-claim
```

8. In the current directory, there’s already a “roar-complete.yaml.pv” file with the changes in it for accessing the storage objects. Diff the two files with the visual diff tool “meld” to see the differences.

```
$ meld roar-complete.yaml roar-complete.yaml.pv
```

9. Now we'll update our `roar-complete.yaml` file with the needed changes. To save trying to get the yaml all correct in a regular editor, we'll just use the meld tool's merging ability. In the meld window, on the right pane (the one with `roar-complete.yaml.pv`), click the arrow that points left to replace the code in our `roar-complete.yaml` file with the new code from the `roar-complete.yaml.pv` file. (In the figure below, this is the arrow that is circled.)

(NOTE: Ignore any other changes, such as the NodePort)



10. You should then see messages pop up that the files are identical. **Click on the Save button at the top to save the changes.** Then you can **close** the meld application.
11. Apply the new version of the yaml file to make sure it is syntactically correct.

```
$ kubectl apply -f roar-complete.yaml
```

12. Force a refresh in the running instance of the app in the browser. Look at the local area for the mount. You should see data from mysql.

```
$ ls -la /mnt/data
```

13. Add the extra record again into the database.

```
$ ./update-db.sh <namespace> (such as ./update-db.sh roar)
```

14. Refresh the browser to force data to be written out the disk location.

15. Repeat step 3 to kill off the current **mysql** pod. After it is recreated, get the ip address again from the new pod and go to that in the browser (may be different!). Once there, refresh the screen and notice that the new record is still there!

END OF LAB

## Lab 7 – Using Helm

**Purpose:** In this lab, we'll start to get familiar with Helm – an orchestration engine for Kubernetes.

1. Switch to the caz-class subdirectory and use the tree command to look at the structure.

```
$ cd ~/caz-class
$ tree roar-helm
```

2. Let's look at how things map from values to templates to instantiated objects. Take a look at the template for the roar-web service and then use the template command to see how the rendered template looks.

```
$ cat roar-helm/charts/roar-web/templates/service.yaml
```

```
$ helm template roar-helm/charts/roar-web -x templates/service.yaml
```

3. Finally, let's look at the values.yaml file for the roar-web charts.

```
$ cat roar-helm/charts/roar-web/values.yaml
```

4. Next, let's deploy the full set of charts (note double hyphens in options)

```
$ helm install --name roar2 --namespace roar2 roar-helm
```

5. Get a list of the existing helm deployments and then the status of our current one with the commands below.

```
$ helm list
```

```
$ helm status roar2
```

6. We want to look at our app running from the helm deployment. Get the NodePort info from the web-roar service via helm status.

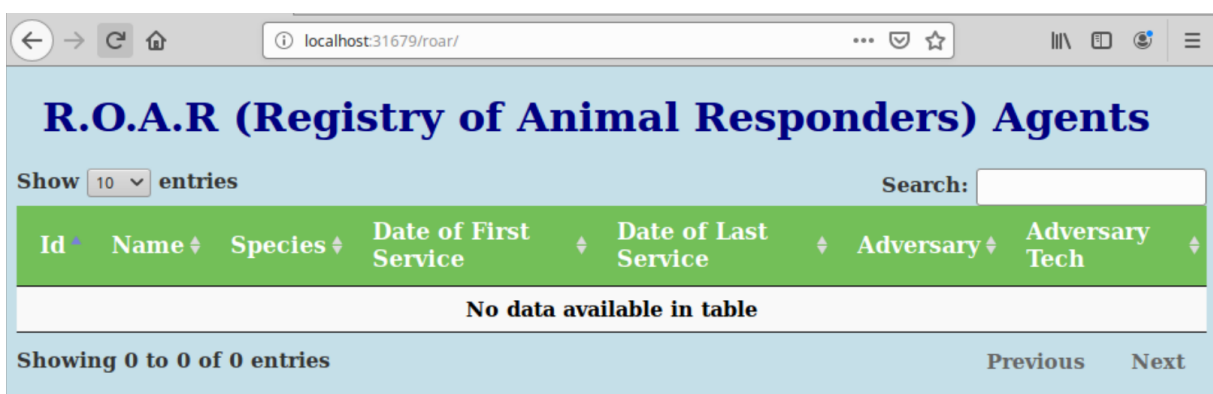
```
helm status roar2 | grep NodePort
```

7. Go to the URL for the webapp.

<http://localhost:<nodeport>/roar/>

(This will be the port like "3####")

You will probably notice that while you have the web interface up, there is no data in the table. We'll fix this next.



8. The problem with our Helm deployment is that the name of the service for the database pod is different than what the web pod expects. To see this, compare the database service name from the roar namespace with the one in the roar2 namespace.

```
$ kubectl get svc -n roar
```

```
$ kubectl get svc -n roar2
```

9. You can see where the name gets set in the “roar-db.name” function in the `_helpers.template`. Use the command below to look at the code.

```
$ cat roar-helm/charts/roar-db/templates/_helpers.tpl
```

11. You don’t have to understand all of this, but notice that there is this line in there:

```
{{- default .Chart.Name .Values.nameOverride -}}
```

We can interpret this line to say that the default value is `Chart.Name`, but we also can have an override specified via a “`nameOverride`” field.

12. Let’s add a `nameOverride` setting to our values file for the database service chart. Edit the file below and then add the line in bold after the initial comments (or anywhere that is not indented). NOTE that there IS A SPACE between “`nameOverride:`” and “`mysql`”.

```
$ gedit roar-helm/charts/roar-db/values.yaml
```

```
# Default values for roar-db-chart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
nameOverride: mysql
replicaCount: 1
image:
  repository: localhost:5000/roar-db
  tag: v1
```

Since the format of the value was `.Values.nameOverride`, that indicates that it should be set at the top level of the chart. (If it were something like `.Values.service.nameOverride`, that would indicate it should be set in the “service” section of the chart.)

13. **Save** your changes and quit the editor. Now, we’ll do a helm upgrade to get our changes in for the service name. (You’ll need to be in the `~/caz-class` directory.)

Now let's run the upgrade. Then check the overall status of the helm release with the helm status command until it shows that things are ready.

```
$ helm upgrade --recreate-pods roar2 roar-helm
```

```
$ helm status roar2
```

14. After a few moments, you should be able to do a helm status, see that things are ready, refresh the browser and see the data showing up in the app. You can also see the list of helm releases with the command below.

```
$ helm history roar2
```

END OF LAB

## Lab 8 – Working with Istio

**Purpose:** In this lab, we'll look at istio and see how we can leverage some of its functionality with the sidecar containers.

**NOTE:** For this lab, we need to turn off some "noise" from minikube, so run the command below first.

```
$ minikube config set WantUpdateNotification false
```

1. Take a look at the pods running in the istio namespace on our system.

```
$ kubectl get pods -n istio-system
```

2. Let's setup a new namespace to run in. We'll then set the default context to it. And finally, we'll set a label to tell Istio to automatically inject sidecars into the pods.

```
$ kubectl create ns istio1
```

```
$ kubectl config set-context minikube --namespace istio1
```

```
$ kubectl label namespace istio1 istio-injection=enabled --overwrite
```

3. To keep things simple, we'll be creating a combined pod for working with istio – one pod with both the db and web containers in it. As well, we'll use helm to deploy. Our helm charts will also include gateway, virtualservice, and destinationrule specs. Change into the class directory for roar-istio and use helm to deploy this. (Note the period on the end of the helm command since we are already in the helm chart location.)

```
$ cd ~/caz-class/roar-istio

$ helm install --name istio1 .
```

4. While waiting on things to get ready, take a look at the pods we have here. Notice that we have 2 pods – one named “current” and one named “new”. These are two deployed versions of our app so we can compare with the various istio features. Also notice there are 3 containers in our pods (3/3). Take a look at one of the pods with the describe to see what is in one.

```
$ kubectl get pods

$ kubectl describe pod <name of one of the pods>
```

In the output, you'll see the containers started for our web one, the db one, and the istio proxy.

5. While we're here, let's get the logs for the same pod.

```
$ kubectl logs <name of one of the pods>
```

6. What does the error message say? When we have multiple containers in a single pod, some commands have to have the container name to know which one we want. Let's do the one for the web container. To specify a particular container, we can use the “-c” option. Try the command again like this:

```
$ kubectl logs <name of one of the pods> -c roar-web
```

7. We have a gateway item that is setup to allow for istio requests through an ingress, a virtualservice that defines how requests map to services, and a destinationrule that allows for subsetting which pods things go to. Take a look at each of these and see if you can start to get an idea of how they work.



```
$ kubectl get gateway -o yaml
```

```
$ kubectl get destinationrule -o yaml
```

```
$ kubectl get virtualservice -o yaml
```

(Why didn't we have to specify a namespace or actual object name for these?)

Notice in the virtualservice that we are providing "weights" to each destination service. This describes how much of the traffic we want to go to each pod. The pods are selected by the labels specified in the destinationrule.

```
route:
  - destination:
      host: roar-web
      port:
        number: 8089
      subset: version-1
    weight: 80
  - destination:
      host: roar-web
      port:
        number: 8089
      subset: version-2
    weight: 20
```

8. Let's send traffic to the pods and services with the "load-roar.sh" script. Running it figures out the host and port for the Istio ingress and then sends queries to the rest api of our web service that are funneled through the conditions and route specified in the virtualservice.

```
$ ./load-roar.sh
```

The idea here is that with the weights defined in the virtualservice, we should see about 80 percent of the traffic going to our first pod (version 00.01.00) and 20 percent going to our second pod (version 00.02.00).

When you're done with this, stop the job with **Ctrl-C**.

9. Now, let's swap in another virtualservice spec that injects a delay of 3 seconds 25% of the time. We'll do this by copying in the virtualservice spec and then using helm to upgrade. To see how this is done, take a look at the file and notice the part about "fault" and "delay".

```
$ cp virtualservices/virtualservice.yaml.delay templates/virtualservice.yaml
$ cat templates/virtualservice.yaml
```

10. Upgrade the helm instance. Then run the load again and notice the periodic delays.

```
$ helm upgrade istio1 .
$ ./load-roar.sh
```

When you're done with this, stop the job with **Ctrl-C**.

11. Now, let's swap in another virtualservice spec that injects a 500 http error 10% of the time. We'll do this by copying in the virtualservice spec and then using helm to upgrade. To see how this is done, take a look at the file and notice the part about "fault" and "abort".

```
$ cp virtualservices/virtualservice.yaml.fault templates/virtualservice.yaml
$ cat templates/virtualservice.yaml
```

12. Upgrade the helm instance. Then run the load again and notice the periodic faults.

```
$ helm upgrade istio1 .
$ ./load-roar.sh
```

(Since we have this set to only happen 10% of the time, it may take a bit before you see the first "fault filter abort" message indicating the error.)

When you are done with this, you can kill the load job with **Ctrl-C**.

END OF LAB

## Lab 9 – Kubernetes Operators

**Purpose:** In this lab, we'll get to install and work with a simple Kubernetes operator for our ROAR app. We'll create a custom resource (CR) in k8s via a custom resource definition (CRD) and then use an operator to scale the number of instances of that CR.

1. Change to the roar-operator directory of the qs-class project. This directory contains the files we need for the lab.

```
$ cd ~/caz-class/roar-operator
```

2. Create a new namespace to run the operator content in.

```
$ kubectl create ns op
```

3. First we want to deploy our CRD into the cluster. Take a look at the first few lines of our app\_v1alpha1\_roarpp\_crd.yaml file. What are the various names for (ways we can refer to) our CRD?

```
$ head -n 12 crds/app_v1alpha1_roarpp_crd.yaml
```

4. Go ahead and deploy the CRD and verify that it is in there.

```
$ kubectl create -f crds/app_v1alpha1_roarpp_crd.yaml
```

```
$ kubectl get crd
```

5. That's a lot of CRD's . Let's look for just yours.

```
$ kubectl get crd | grep roarapp
```

6. Take a look at the remaining yaml files in the "roar-operator" directory and see if you can figure out what they do. How do the role\* ones relate to each other? Take a look at the operator.yaml one. Where does the image come from?

7. Go ahead and deploy these files to create the objects.

```
$ kubectl create -n op -f role.yaml -f role_binding.yaml -f
  service_account.yaml -f operator.yaml
```

8. This should set up the operator running as a container on your system. Verify that you see the pod for it in the operator namespace.

```
$ kubectl get pods -n op
```

9. Take a look at the replicas/roarapptest.yaml file (in the qs-class/roar-operator directory). This specifies how many replicaset we want for our CR.

```
$ cat replicas/roarapptest.yaml
```

10. The operator works by reconciling what's requested for the replicas with the custom resource definitions. The main part of that work is done in the "Reconcile" handler function in the code. You can see this at [https://github.com/brentlaster/roarv2-operator/blob/master/pkg/controller/roarapp/roarapp\\_controller.go](https://github.com/brentlaster/roarv2-operator/blob/master/pkg/controller/roarapp/roarapp_controller.go) if you're interested. The Reconcile function starts around line 80. NOTE: This is not intended to represent coding best practices – just a quick and simple (and contrived) example.

11. We need to do an extra step here to get a copy of our local images pushed for the names the operator expects.

```
$ docker tag localhost:5000/roar-db:v1 localhost:5000/roar-db-image:v1
```

```
$ docker tag localhost:5000/roar-web:v1 localhost:5000/roar-web-image:v1
```

```
$ docker push localhost:5000/roar-db-image:v1
```

```
$ docker push localhost:5000/roar-web-image:v1
```

12. Now let's put the operator to work. After you're done looking at it, go ahead and deploy it.

```
$ kubectl apply -n op -f replicas/roarapptest.yaml
```

13. Take a look at the (non-operator) pods we have running in the namespace. How many are there?

```
$ kubectl get pods -n op
```

14. So we've been able to scale up to 5 instances of the pod with our app in it. You can look at the app running in any of these by getting the IP address from the command below and then plugging it into a browser in the format after the command.

```
$ kubectl describe -n op pod <example roarapp pod name> | grep IP  
  
http://<IP address>:8080/roar/
```

15. We can also work with our CRD just as with any other type of native object in Kubernetes. Try the commands below:

```
$ kubectl get RoarApp -n op
```

```
$ kubectl describe -n op RoarApp
```

16. Finally, let's scale our number of pods back to 3. Edit the RoarApp object and change the replicas line from 5 to 3. (Change the editor to use gedit first to be easier than default vi/vim.)

```
export EDITOR=gedit
```

```
kubectl edit -n op RoarApp
```

change the line

```
replicas: 5
```

to be

```
replicas: 3
```

Save your changes and check the number of example pods now running in op.

**END OF LAB**

## Bonus Lab - Monitoring

**Purpose:** This lab will introduce you to a few of the ways we can monitor what is happening in our Kubernetes cluster and objects.

**Setup:** Open up permissions for .minikube area:

```
$ sudo chown -R diyuser3:diyuser3 ~/.minikube
```

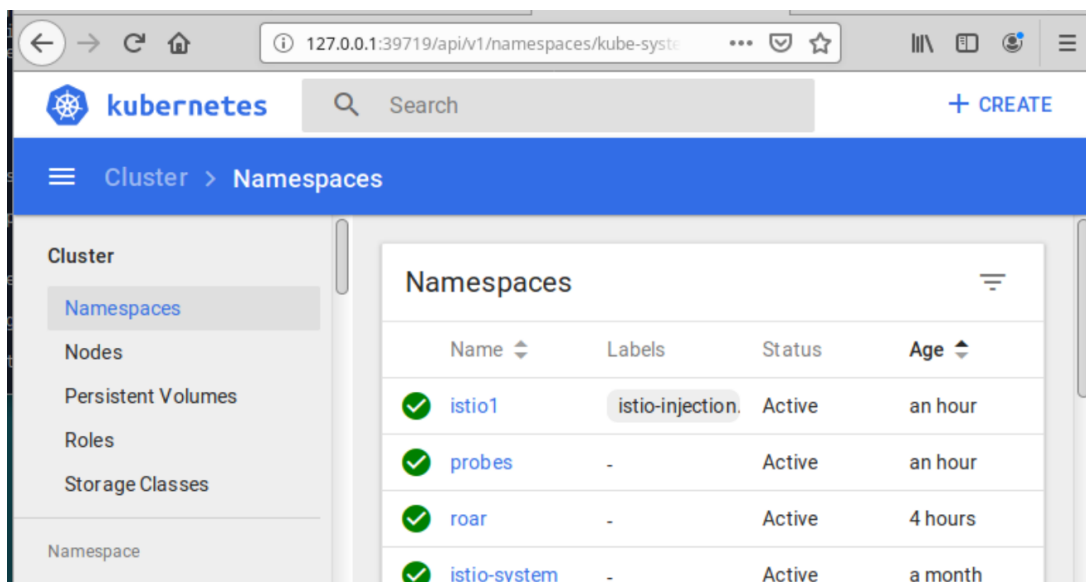
1. For most of our monitoring activities, we will need a Kubernetes “addon” named “Heapster” enabled. Go to one of your terminal sessions and enable Heapster with the following command.

```
$ sudo minikube addons enable heapster
```

2. First, let’s look at the built-in Kubernetes dashboard. We can invoke it most easily by using minikube again. In a terminal session, enter:

```
minikube dashboard
```

3. The dashboard for our cluster will open up in a browser. You can choose K8S objects on the left and get a list of them, explore them, etc.



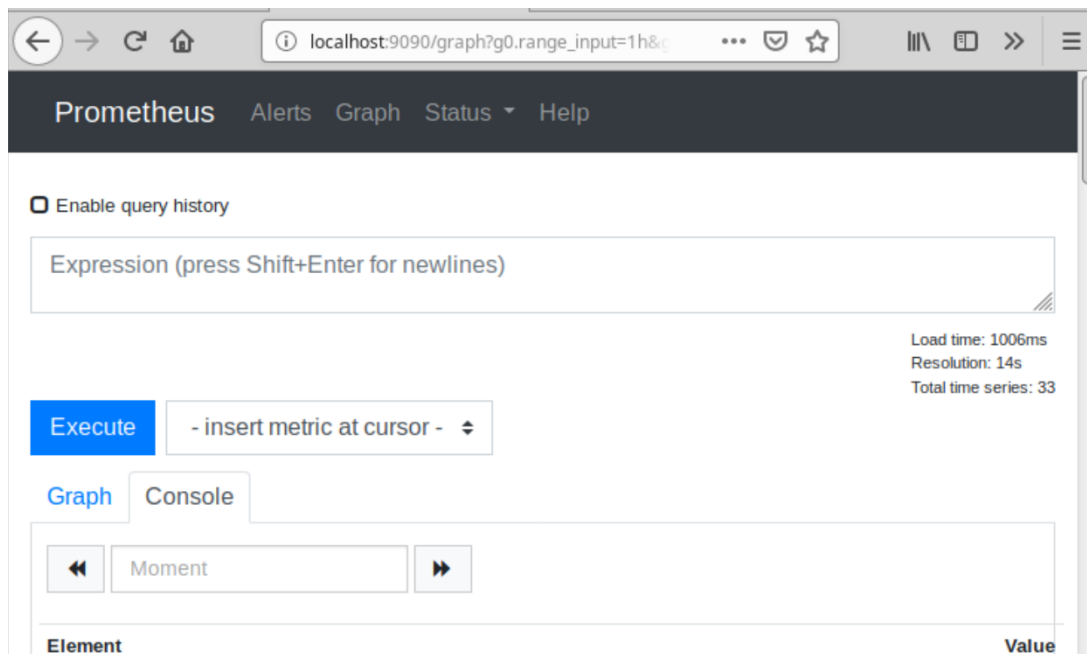
4. Now let’s look at some metrics gathering with a tool called Prometheus. To be able to access it, we need to port-forward it from our localhost to the port on the pod running in the **istio-system** namespace. To do that, find

the name of the Prometheus pod in the istio-system namespace and enter the command below in a terminal window:

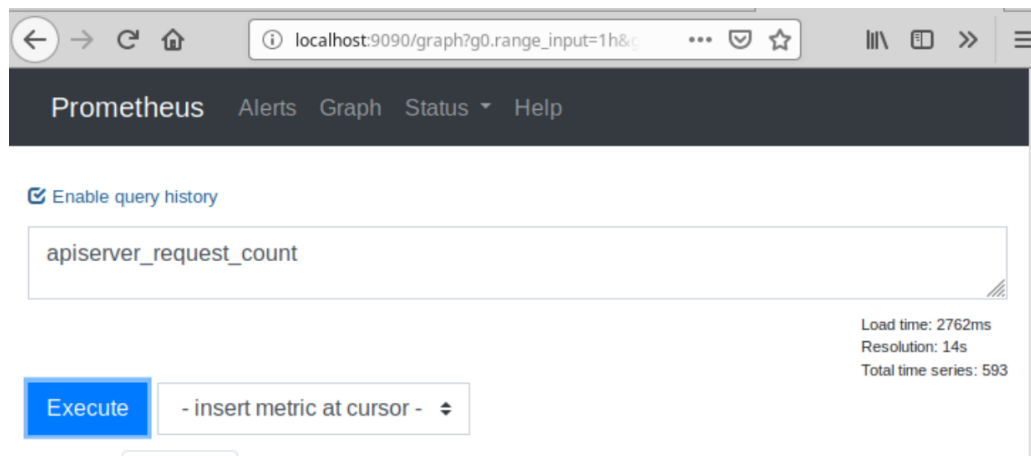
```
$ kubectl get pods -n istio-system | grep prom
```

```
$ kubectl port-forward -n istio-system <Prometheus pod name> 9090:9090
```

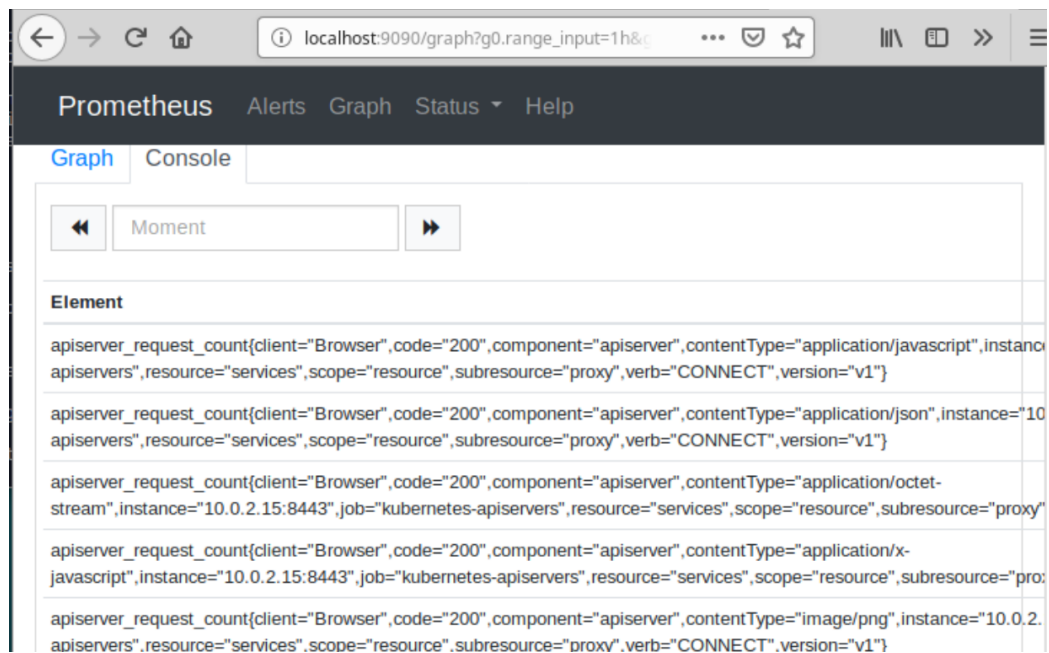
- Now, in a new browser tab, go to <http://localhost:9090> . This may take a while, but eventually you should see a screen like the one below:



- Prometheus comes with a set of built-in metrics. Just start typing in the "Expression" box. For example, let's look at one called **"apiserver\_request\_count"**. Just start typing that in the Expression box. After you begin typing, you can select it in the list that pops up. After you have got it in the box, click on the blue "Execute" button.

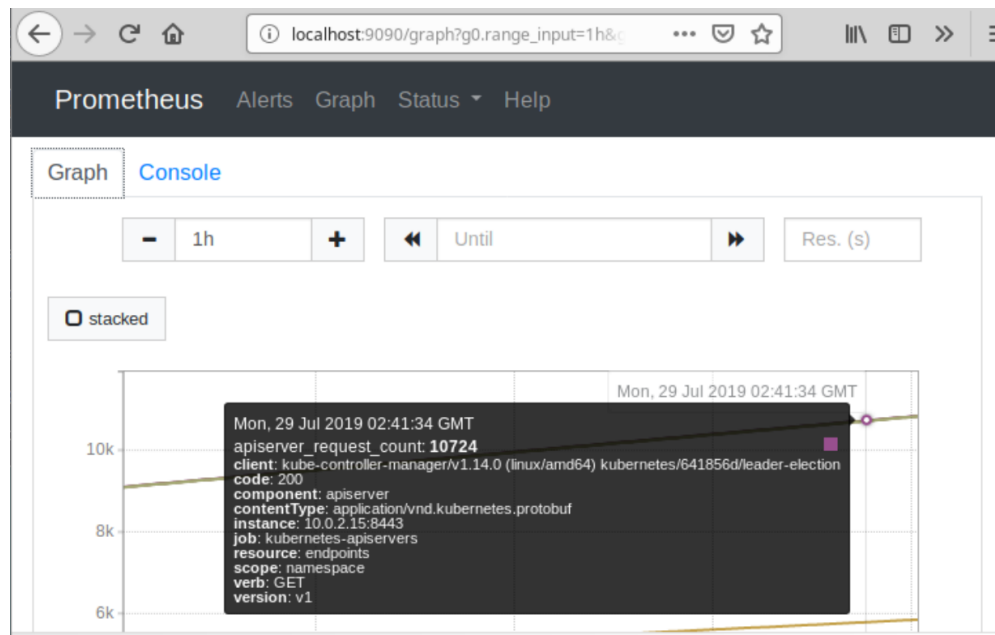


7. Now, scroll down and look at the console output (assuming you have the Console tab selected).



8. Now, click on the blue “Graph” link next to “Console” and take a look at the graph of responses. Note that you can hover over points on the graph to get more details.

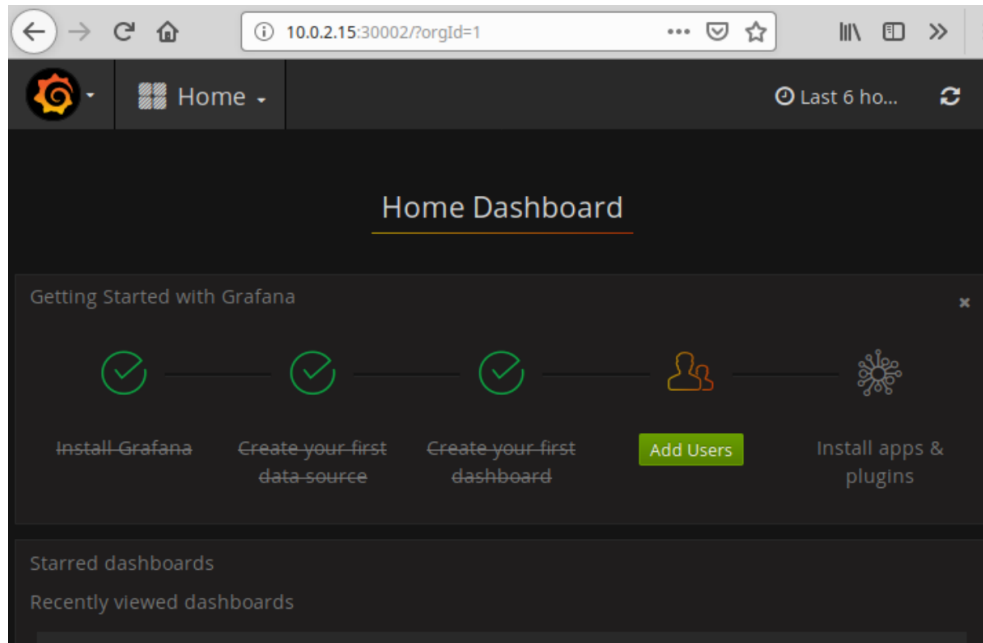




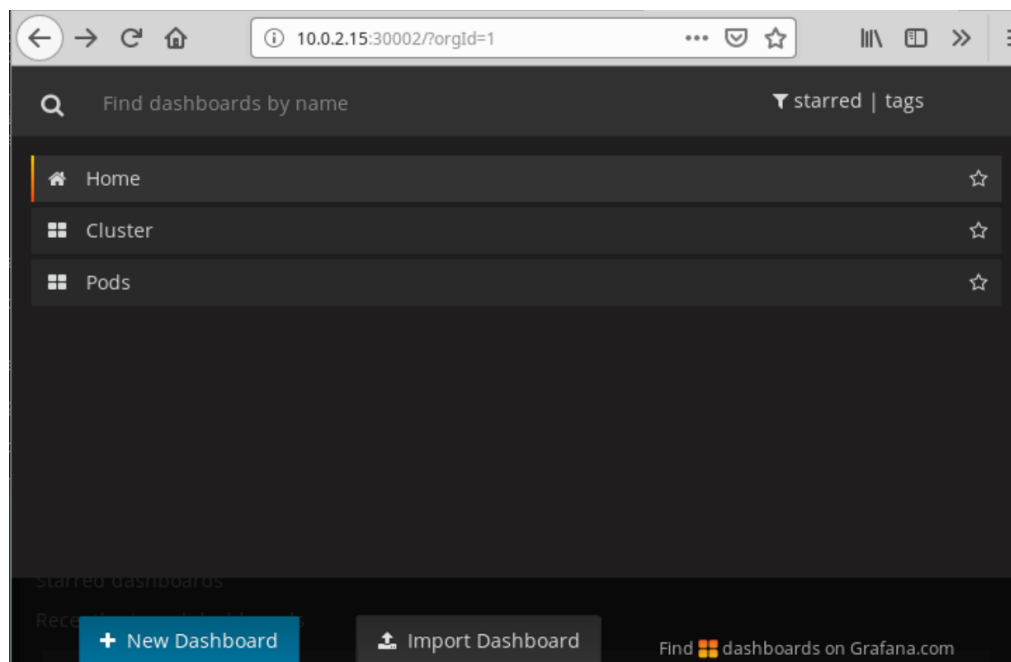
9. Finally, let's take a look at Grafana. Grafana is already running as a pod and service in our kube-system namespace. See if you can figure out how to access it based on the service type and port. (Hint: "get" the service info in namespace kube-system and look for "**monitoring-grafana**")
10. Since it's running as a NodePort service and we only have the one node in our cluster, we just need to get the ip address of the node and add the NodePort to open it up in a browser. Open up the url below (Remember you can use "**minikube ip**" to get the ip address.)

**http://<node ip>:<nodeport of Grafana service from kube-system>**

11. You should now be on the Grafana Home Dashboard.



12. Click on the down-arrow next to “Home”. You’ll see built-in dashboards for “Cluster” and “Pods”. Pick one and explore the different information in it. Then go back and select the other one and do it. Note in the Pods one you can select different namespaces, etc.



## END OF LAB