Prereq: Get disk image (OVA file) from
http://sww.sas.com/tst/tools/container-training/sgf-ws.ova

You will also need an installed version of VirtualBox to run it on .

Understanding Containers and Related Technologies

A Hands-On Approach

Workshop Labs

Instructor: Brent Laster

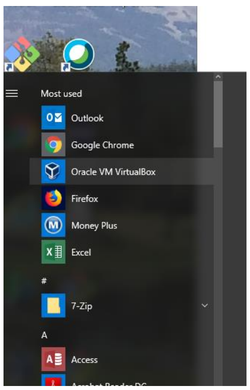
Understanding Containers and Related Technologies
A Hands-On Approach
Workshop Labs
Version 1.5 by Brent Laster

07/16/2020

Setup

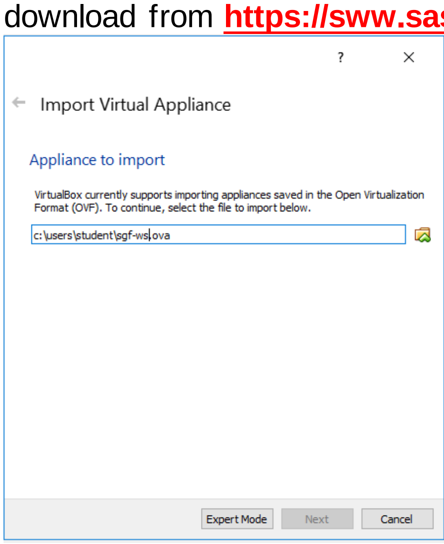
Purpose: This section guides you through getting the VM up and running.

1. Via the start menu or on your desktop, open up the VirtualBox application.



2. Once VirtualBox comes up, click on the **File** option in the menu bar, and select **Import Appliance…**

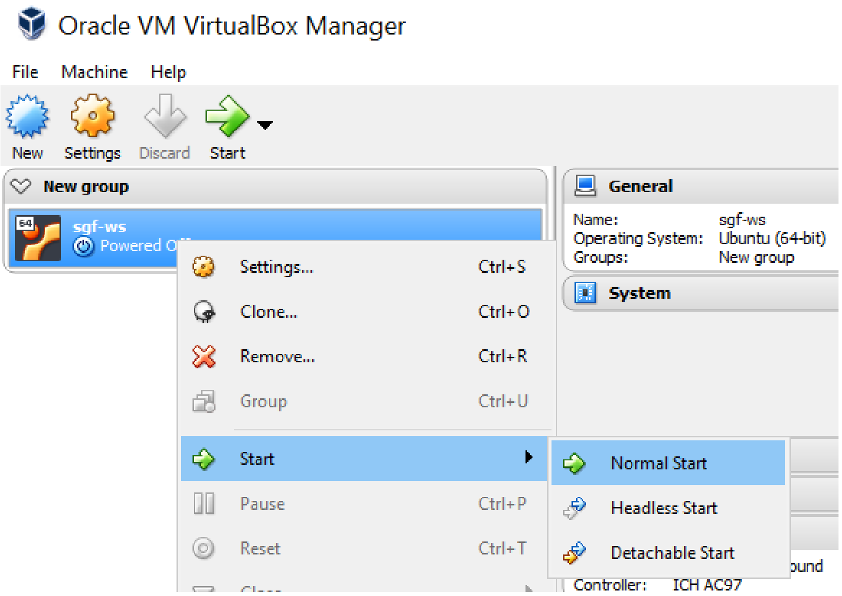
3. In the dialog box that comes up, enter the location of the sgf-ws.ova image that you can download from **https://sww.sas.com/tst/tools/container-training/sgf-ws.ova**



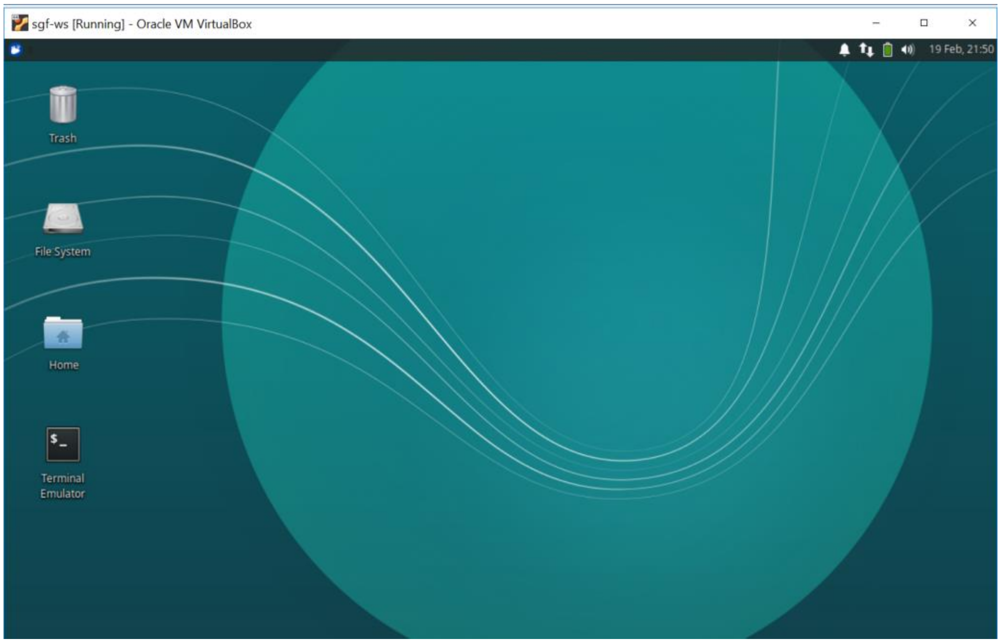
4. Click **Next** and proceed through the rest of the screens, taking the defaults.

5. After the machine is imported, you should see an entry for **sgf-ws** showing up in the list of machines.

6. Now, right click on the **sgf-ws** entry, select **Start**,and then click **Normal Start**.



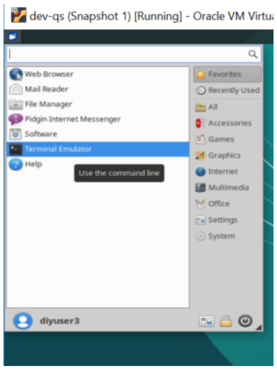
7. After the machine starts, you'll see the desktop as in the picture below.



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. Switch into the working directory for our docker work.

    cd sgf-ws/roar-docker

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

    ls

4. 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

5. Now let's build our docker database image. Type 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 .

6. 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 .

7. Now, let's tag our two images for our local registry (running on localhost on 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

8. 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.

1. 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

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

   docker-compose up

3. 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 another t(erminal emulator** (or use an existing one if you have a second available**)** 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:

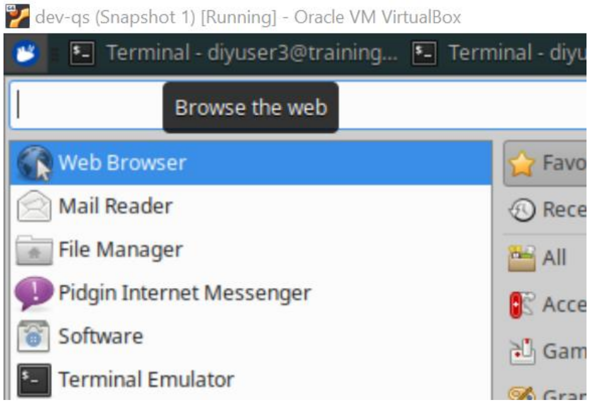
   237a48a2aeb8 roar-web "catalina.sh run" About a minute ago Up About a minute 0.0.0.0:8089->8080/tcp

   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	1955-01-20	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> <container id for roar-db>

    Remove the containers

    docker rm <container id for roar-web> <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 sgf-ws/roar-k8s directory. Change into that directory and take a look at the yaml file there for the Kubernetes deployments and services.

   cd ~/sgf-ws/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:

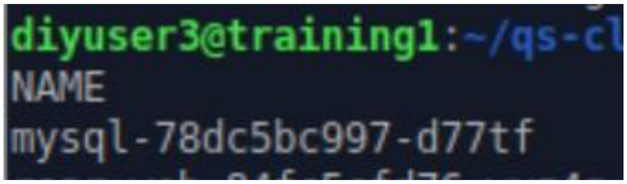
   deployment.extensions/roar-web created
   service/roar-web created
   deployment.extensions/mysql created
   service/mysql created

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.



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 (that begins with "Error from server") is the actual log. And it 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.

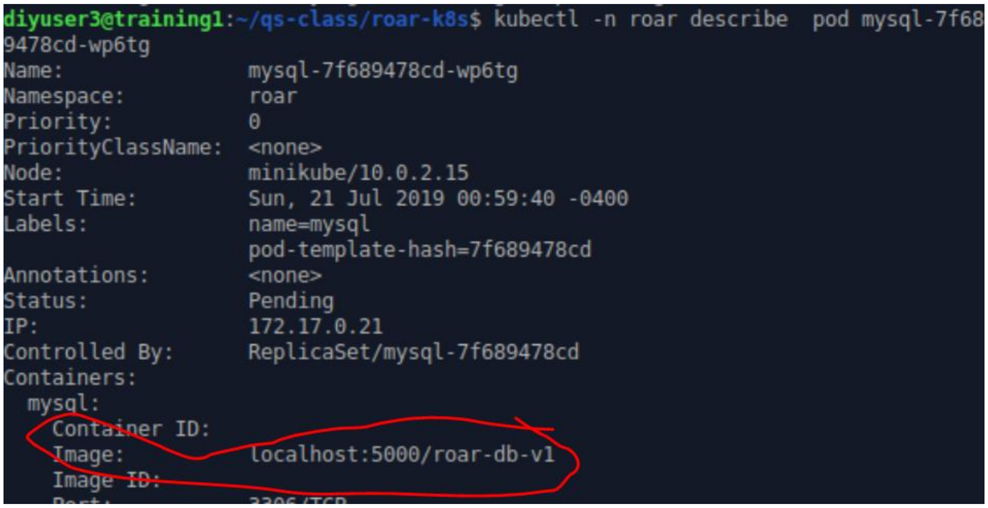
   kubectl -n roar describe pod <paste pod name here>

   (example: kubectl -n roar 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".

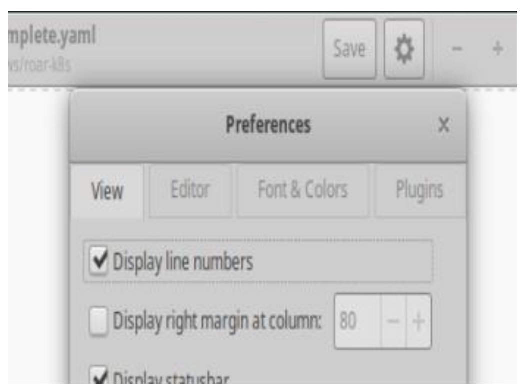


10. It is looking for an image with the "-v1" as part of the name. But that's not what we tagged ours as. To fix this, edit the roar-complete.yaml file and modify the "Image" properties to change the "-" to a ":" for the web image (only).

    Still in the sgf-ws/roar-k8s directory:

    gedit roar-complete.yaml

    (Note: In gedit, to display line numbers (for the next part), click on the gear icon (next to the Save button), then Preferences, then check the box for Display line numbers. See figure to right.)



    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

11. After you make your changes, save the file and close the editor. Now, in the original terminal window (the one that is probably still in *roar-docker*), start a command to watch the pods (the -w option) so we can see when changes occur.

    kubectl get pods -n roar -w

12. In the second emulator window (where you are 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 heading, after the service port (8089) and before the “/TCP”. For example, if we have **8089:31789/TCP** in that column, then the actual nodePort we need is **31789**.

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 – Using Helm

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

1. Switch to the sgf-ws subdirectory and use the tree command to look at the structure.

    cd ~/sgf-ws
    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.

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

10. 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.

11. Let's add a nameOverride setting to our values file for the database service chart. To keep this simple, there's already a file out there named "fixed-values.yaml" that you can just swap in. It is the same as the original file with the addition of the bolded line below.

```
# 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
```

Run the command below to copy it over and fix the issue.

cp extra/fixed-values.yaml roar-helm/charts/roar-db/values.yaml

12. Now, we'll do a helm upgrade to get our changes in for the service name. (You'll need to be in the ~sgf-ws directory.) 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

13. 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 6 – Using Kustomize

Purpose: In this lab, we'll look at how to deploy into Kubernetes with Kustomize, an alternative approach to Helm.

1. Switch to the roar-kz subdirectory and use the tree command to look at the structure.

   cd ~/sgf-ws/roar-kz
   tree

2. Notice how we have the base and overlay directories laid out and the occurrences of the *kustomization.yaml* files. Take a look at the kustomization.yaml files in the *base, overlays/stage,* and *overlays/prod* area to see what kind of changes these are layering on top of our specs. What is each one adding/modifying?

   cat base/kustomization.yaml

   cat overlays/stage/kustomization.yaml

   cat overlays/prod/kustomization.yaml

3. Let's do a kustomize build of the base area and capture the yaml that is produced for Kubernetes. Then let's also do a kustomize build of the overlays/staging area and capture that output.

   kz build base > base.out

   kz build overlays/stage > stage.out

4. Let's look at the difference between the kustomize build of base and the kustomize build of stage with the meld tool. What do you see as different?

   meld base.out stage.out

5. Now, let's go ahead and apply the yaml produced from the kustomize build output to the cluster. We do this by simply piping it through Kubernetes apply.

   kz build overlays/stage | k apply -f -

6. The preceding step should have created a new *roar-stage* namespace with a running instance of our app. Find the node for this as you have before and bring up the running instance.

   kubectl -n roar-stage get svc

Note that the type of service for roar-web is "NodePort". As before, 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 **8089:31789/TCP** in that column, then the actual nodePort we need is **31789**.

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/

7. In the kustomization.yaml file we used for *overlays/stage*, we created a *ConfigMapGenerator* which made a config map with a hash. The hash acts like a key that the deployment knows about. Take a quick look at how the configmap is setup.

   kubectl describe configmap -n roar-stage

8. Let's change the configmap to point to a different database for testing.

   gedit overlays/stage/kustomization.yaml

   Then add a "2" onto the last line – change =registry_test to =registry_test2

Save your changes and close the editor.

9. Now let's see what the new generated configmap difference that our change made compared to the previous one. (Note the space followed by a dash at the end of the line.)

   kz build overlays/stage | diff -c3 stage.out -

10. Build and apply the updated configmap.

    kz build overlays/stage | k apply -f -

11. OPTIONAL: Build and apply the kustomization from the prod area which will add in persistent storage.

    Take a look at what's mounted right now (should be nothing).

    ls /mnt

    Build and apply.

    kz build overlays/prod | k apply -f -

    Notice there's a "pv" (persistent volume) object created as well as a "pv-claim" (persistent volume claim) object created.

    You'll be able to see the persistent storage at /mnt/data.

    ls /mnt/data

END OF LAB

Lab 7 – 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.

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

   kubectl get pods -n istio-system

2. Change to the directory for this lab and take a look at the structure of files and directory under there. This should look similar to the structure we had in the last lab because we are again using Kustomize to deploy. (We could have also used Helm.)

cd ~/sgf-ws/roar-istiok

tree

We have separate overlays for each of the three istio scenarios we will be demoing – *fault injection, traffic-shifting*, and *delay injection*.

3. It's also worthwhile to take a quick look at the *namespace.yaml* file we're using for this one. In it, we're setting a special label to automatically inject sidecars into the pods (**istio-injection=enabled**).

cat base/namespace.yaml

4. After looking at that file, go ahead and use Kustomize to build the set of specs (kz build) for the traffic-shifting example and feed that to Kubernetes (pipe to k apply).

kz build overlays/traffic | k apply -f -

5. Finally, set the default namespace to be the new one we just created.

kubectl config set-context minikube --namespace roar-istiok

6. 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.

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

kubectl logs <name of one of the pods>

8. 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

9. 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.

10. 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.

overlays/common/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**.

11. Now, let's swap in another virtualservice spec that injects a delay of 3 seconds 25% of the time. To see how the delay spec differs from the traffic one, you can use the first command below. (Just close meld when done.) Then to actually make the change in the cluster, we'll just build and apply a separate overlay with the second command.

meld overlays/traffic/virtualservice.yaml overlays/delay/virtualservice.yaml

kz build overlays/delay | k apply -f -

12. Now, you can run the load again and notice the periodic delays.

overlays/common/load-roar.sh

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

13. Finally, let's swap in another virtualservice spec that injects a 500 http error 10% of the time. To see how the delay spec differs from the traffic one, you can use the first command below. (Just close meld when done.) Then to actually make the change in the cluster, we'll just build and apply a separate overlay with the second command.

meld overlays/delay/virtualservice.yaml overlays/fault/virtualservice.yaml

kz build overlays/fault | k apply -f -

14. Now, you can run the load again and notice the periodic fault aborts.

overlays/common/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 8 – 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 sgf-ws project. This directory contains the files we need for the lab.

   cd ~/sgf-ws/roar-operator

2. Update some images in our repository for the operator to work with.

   ./update-images.sh

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

   kubectl create ns op

4. 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_roarapp_crd.yaml

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

   kubectl apply -f crds/app_v1alpha1_roarapp_crd.yaml

   kubectl get crd

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

   kubectl get crd | grep roarapp

7. 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?

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

   kubectl apply -n op -f role.yaml -f role_binding.yaml -f service_account.yaml -f operator.yaml

9. 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

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

    cat replicas/roarapptest.yaml

11. 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 if you're interested. (There's also a bookmark to this file in the web browser on the VM.) 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.

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.

1. First, let's change the permissions on the minikube installation to make the remaining steps simpler.

    sudo chmod -R 755 /home/diyuser3/.minikube

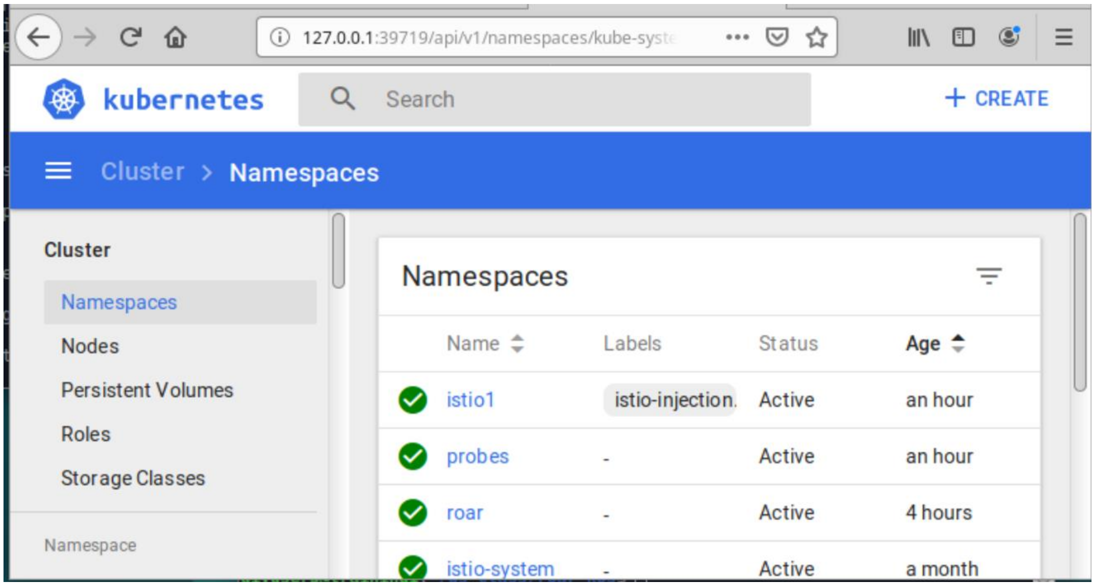
2. For most of our monitoring activities, we will need a Kubernetes "addon" named "Heapster" enabled. Go to one of your terminal sessions and enable Heapster with the following command.

    minikube addons enable heapster

3. 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

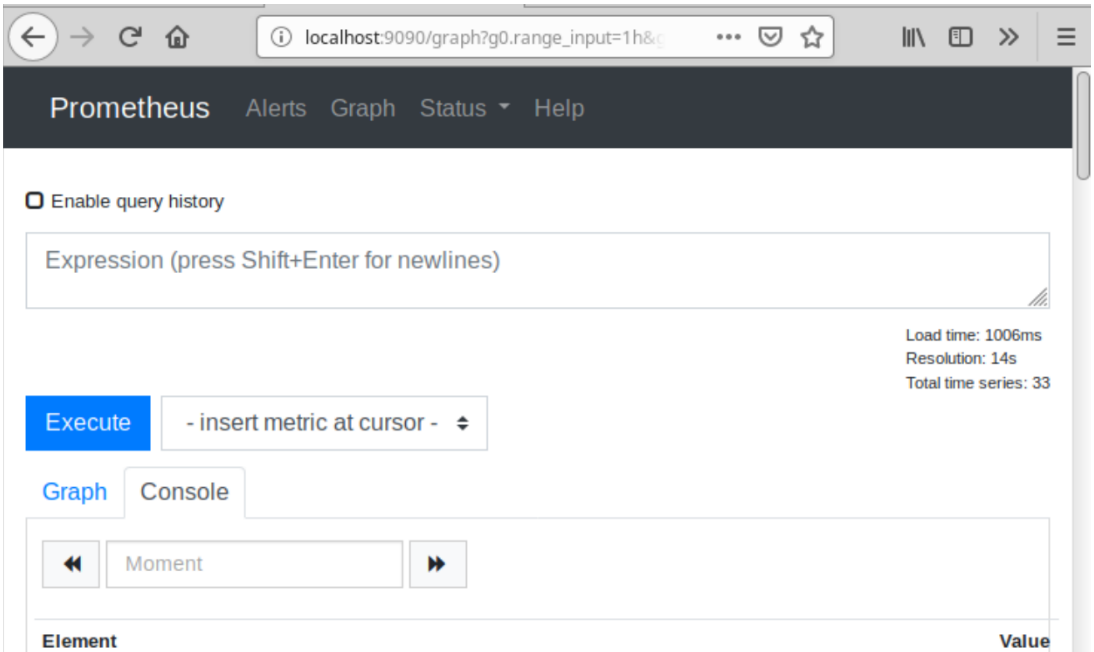
4. The dashboard for our cluster will open up in a browser. You can choose K8S objects on the left and get a list of them, explore them, etc.



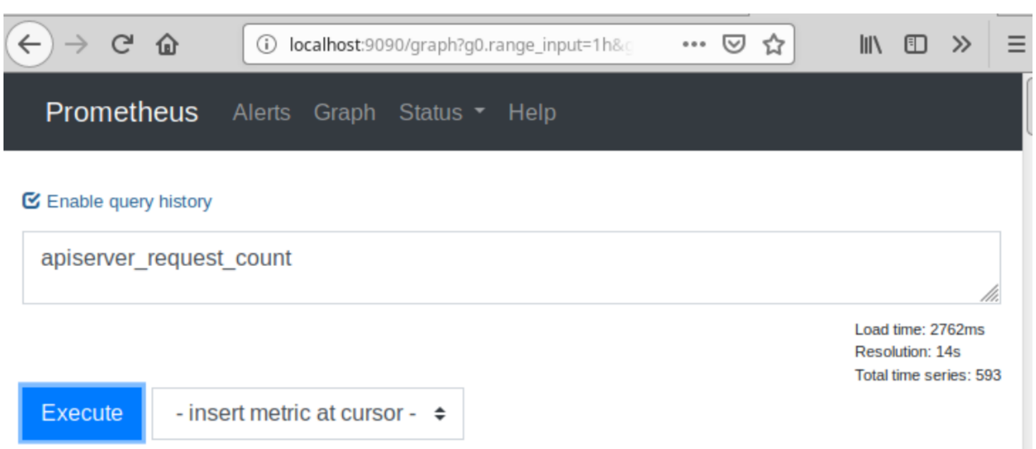
5. Now let's look at some metrics gathering with a tool called Prometheus. To be able to access it, we need to port-forward it from our localhost to the port on the pod running in the istio-system namespace. To do that, find the name of the Prometheus pod in the istio-system namespace and enter the command below in a terminal window:

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

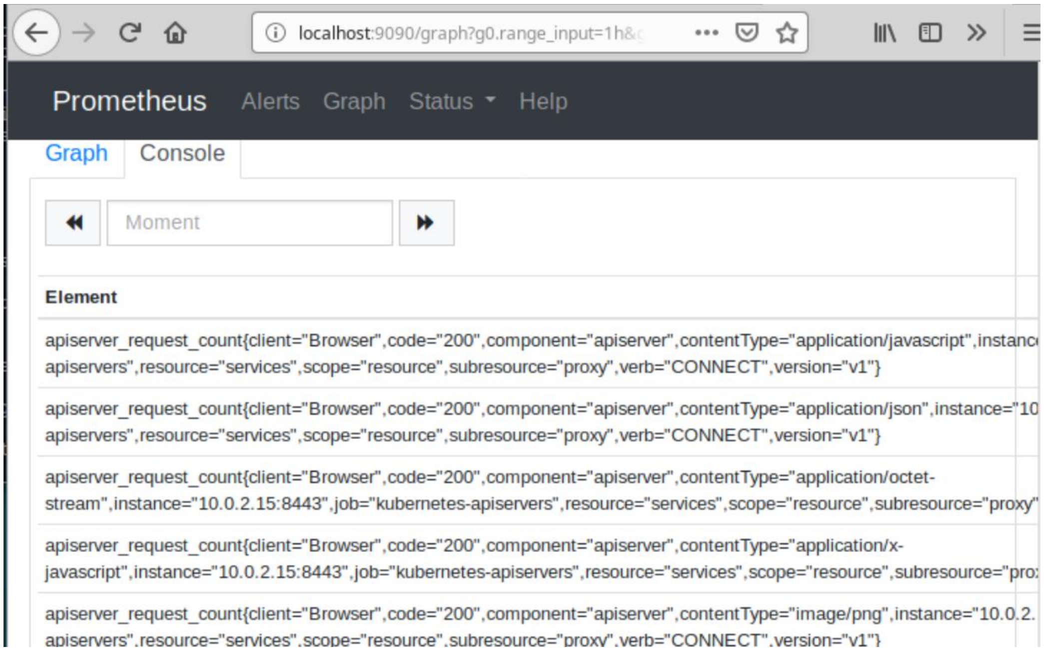
6. Now, in a new browser tab, go to http://localhost:9090 . This may take a while, but eventually you should see a screen like the one below:



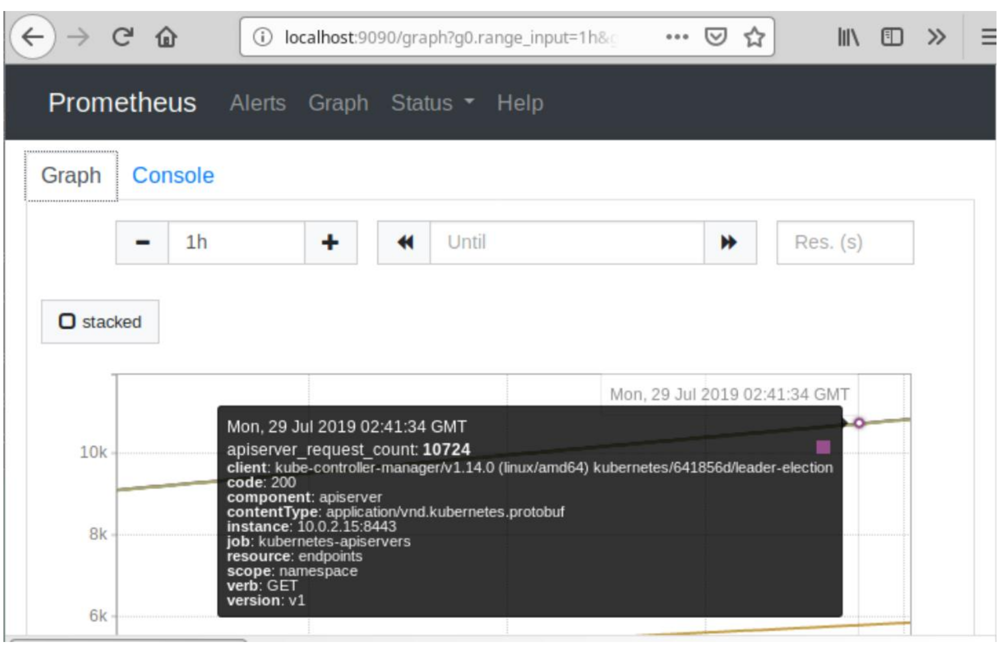
7. Prometheus comes with a set of built-in metrics. Just start typing in the "Expression" box. For example, let's look at one called "apiserver_request_count". Just start typing that in the Expression box. After you begin typing, you can select it in the list that pops up. After you have got it in the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



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



9. 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.

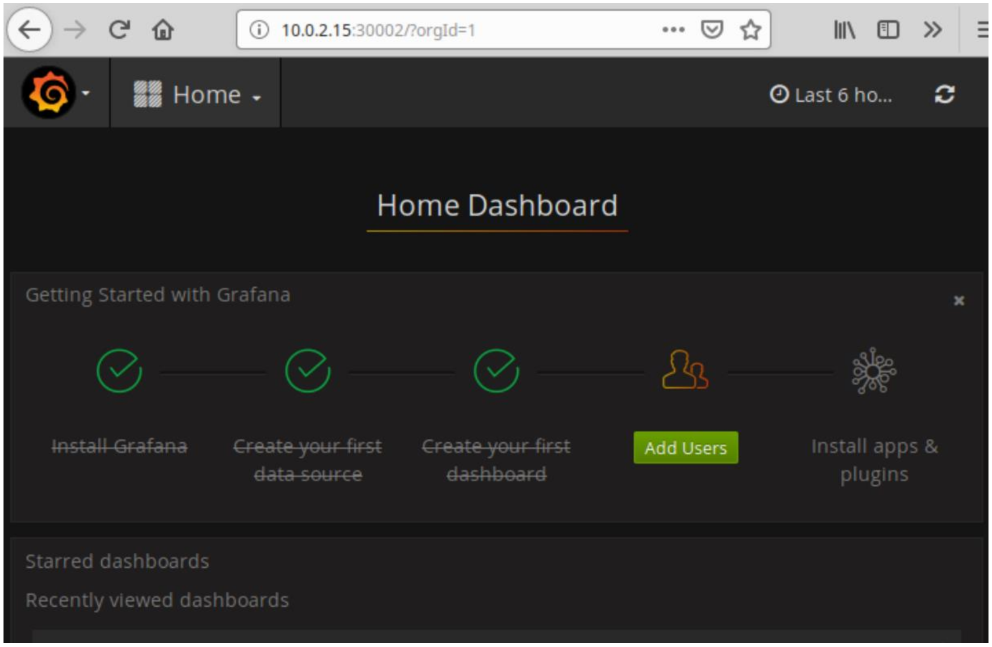


10. Finally, let's take a look at Grafana. Grafana is already running as a pod and service in our kube-system namespace. See if you can figure out how to access it based on the service type and port. (Hint: "get" the service info in namespace kube-system)

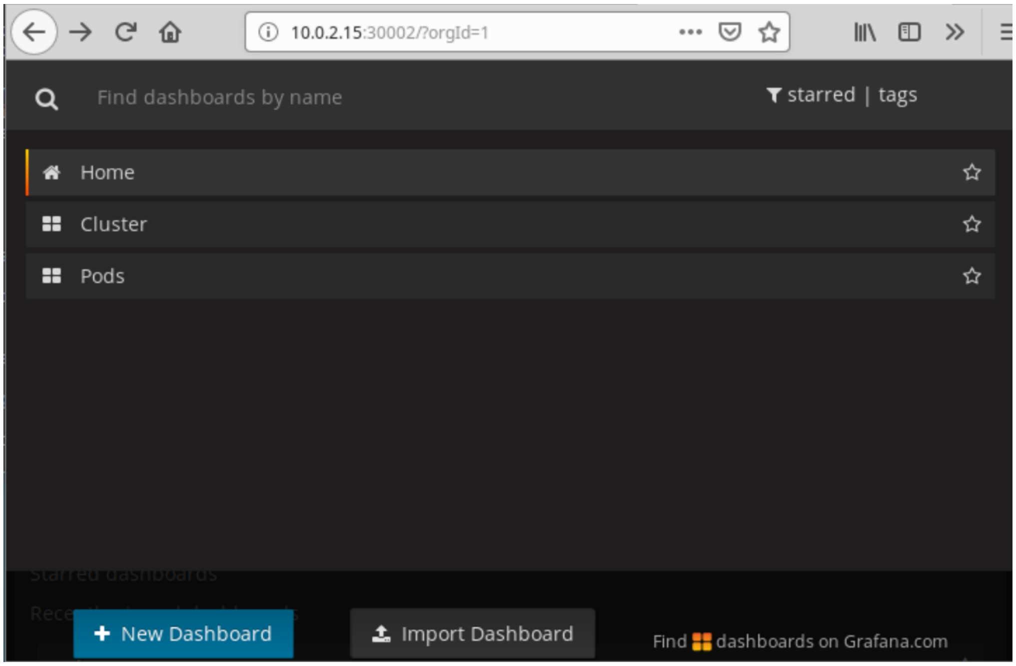
11. Since it's running as a NodePort service and we only have the one node in our cluster, we just need to get the ip address of the node and add the NodePort to open it up in a browser. Open up the url below (Remember you can use "minikube ip" to get the ip address.)

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

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



13. 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.


10.0.2.15:30002/?orgId=1
Find dashboards by name
starred | tags
Home
Cluster
Pods
New Dashboard
Import Dashboard
Find dashboards on Grafana.com

END OF LAB