

Kubernetes for Devs

An introduction to Kubernetes for Developers

Class Labs

Version 1.1 by Brent Laster for Tech Skills Transformations LLC

7/26/2022

Important Prereq: These labs assume you have already followed the instructions in the separate setup document and have either setup your own cluster and applications per those instructions or have VirtualBox up and running on your system and have downloaded the *k8s-dev.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!

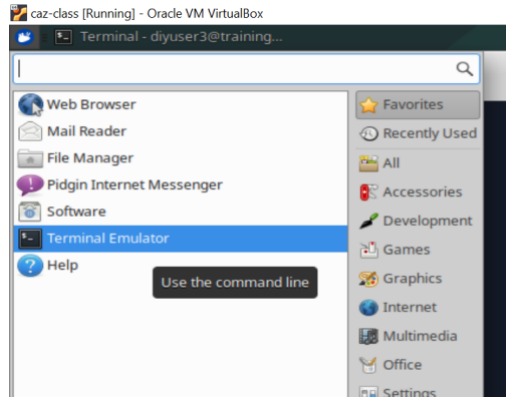
Throughout the labs, we will denote anything that is specific to the VirtualBox environment with a phrase like "**If running in the VM**".

Startup - to do before first lab

1. **If running in the VM**, enable networking. Enable networking by selecting the up/down arrow icon at top right and selecting the option to "Enable Networking". See screenshot below.



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. Get the latest files for the class. For this course, we will be using a main directory *k8s-ps* with subdirectories under it for the various labs.

If running in the VM

In the terminal window, `cd` into the main directory and update the files.

```
$ cd k8s-dev
```

```
$ git pull
```

If NOT running in the VM

```
$ git clone https://github.com/skillrepos/k8s-dev
```

```
$ cd beyond-k8s
```

3. **If not already done**, start up the paused Kubernetes (minikube) instance on this system using a script in the *extras* subdirectory. This will take several minutes to run.

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

4. If not already setup, install the monitoring pieces (Kubernetes dashboard, Prometheus, and Grafana pieces). Refer to the setup doc for details. You will need to have a token to use to connect to the dashboard and the initial password for Grafana. The setup doc references a script that will output these when it runs. Capture those and store them for use in lab 9.
5. Optional - setup alias. In these labs and on the VM, "k" is aliased to "kubectl". If you are not running in the VM, you can usually do this via the following command if you want:

```
$ alias k=kubectl
```

Lab 1 - 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. And we'll see how to do some simple debugging when Kubernetes deployments don't go as planned.

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 k8s-dev/roar-k8s directory. Change into that directory and take a look at the yaml file there for the Kubernetes deployments and services.

```
$ cd ~/k8s-dev/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. We need to investigate why this is happening. Let's do two things to make this easier. First, let's set the default namespace to be 'roar' instead of 'default' so we don't have to pass "-n roar" all of the time.

```
$ k config set-context --current --namespace=roar
```

6. Now let's get a list of the pods that shows their labels so we can access them by the label instead of having to try to copy and paste the pod name.(Note we don't have to supply the -n argument any longer.)

```
$ kubectl get pods --show-labels
```

7. Let's run a command to look at the logs for the web pod.

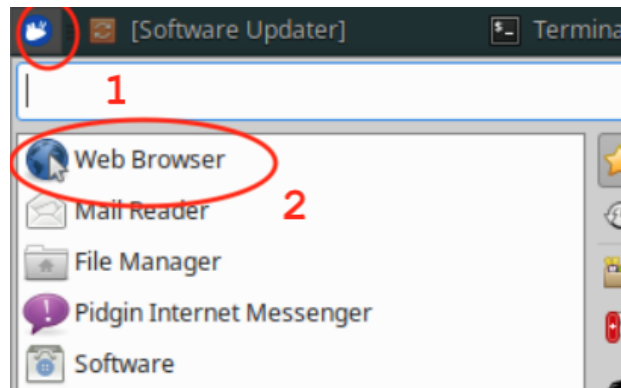
```
$ k logs -l app=roar-web
```

8. The output here confirms what is wrong – notice the part on "*trying and failing to pull image*" or "*image can't be pulled*". We need to get more detail though - such as the exact image name. We could use a describe command, but there's a shortcut using "get events" that we can do too.

```
$ k get events | grep web | grep image
```

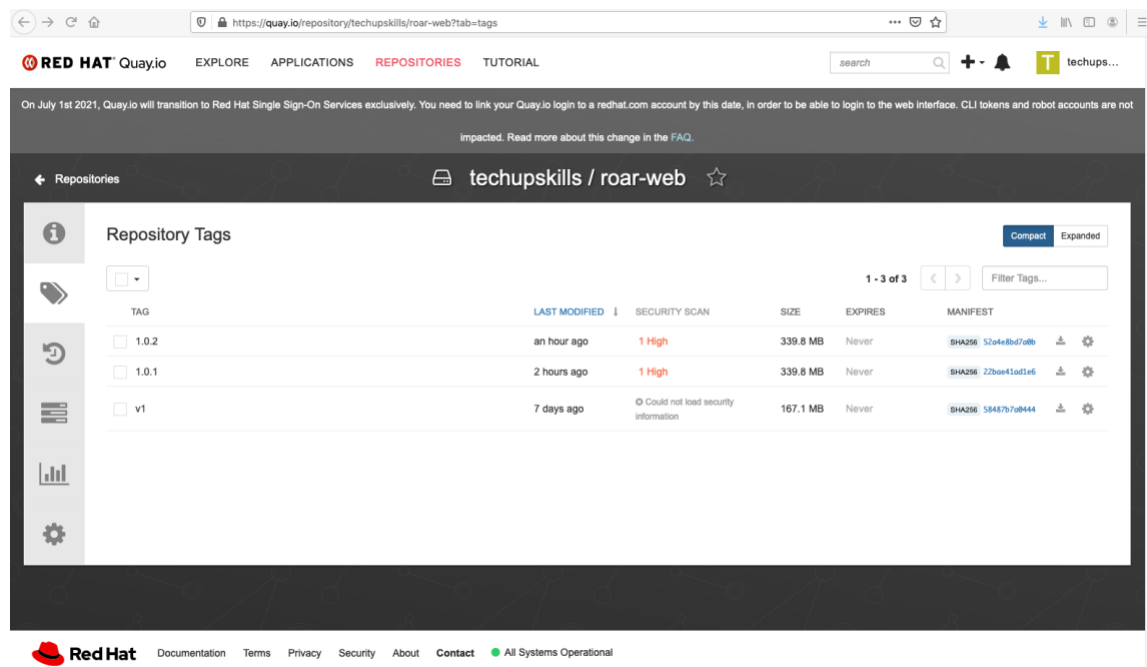
9. Notice that the output of the command from the step above gives us an image path and name: "*quay.io/techupskills/roar-web:1.10.1*". Since it says it can't pull it, let's check and see if it actually exists by going to the URL for it. Open the following URL in a web browser.

If running in the VM, open a browser by clicking on the "mouse head" button in the upper left part of the VM window and then selecting "Web Browser" from the list.



10. After the browser window opens, put the following in the address bar

`https://quay.io/repository/techupskills/roar-web?tab=tags`



11. You can see that we don't have an image with the tag "1.10.1". Instead we have a "1.0.1". So there's probably a typo.
12. We can change the existing deployment to see if this fixes things. But first, let's setup a watch in a separate window so we can see how Kubernetes changes things when we make a change to the configuration.

Do this one command in a **separate terminal session**:

```
$ k get pods -w
```

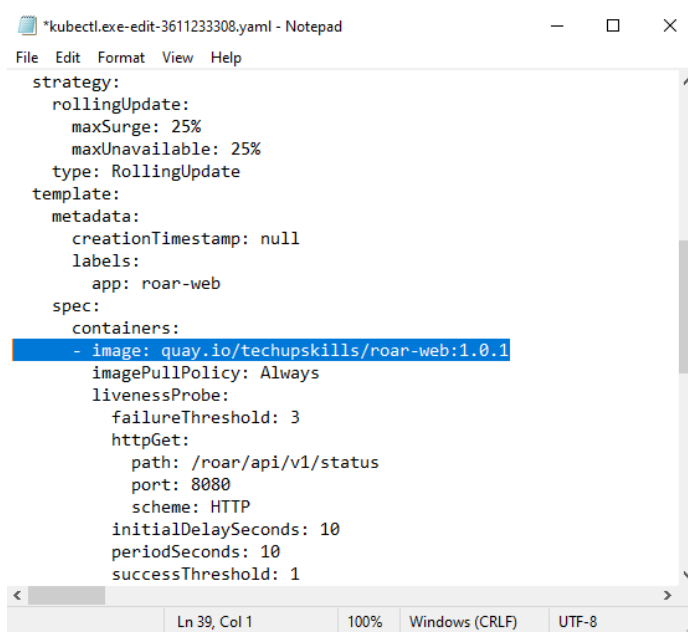
13. (Optional) Set your editor to a different one than the default one for text files if you want.

```
$ export EDITOR=<path-to-editor-program>
```

14. Edit the existing object.

```
$ k edit deploy/roar-web
```

Change line 39 to use 1.0.1 instead of 1.10.1.



```
*kubectl.exe-edit-3611233308.yaml - Notepad
File Edit Format View Help
strategy:
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: roar-web
    spec:
      containers:
        - image: quay.io/techupskills/roar-web:1.0.1
          imagePullPolicy: Always
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /roar/api/v1/status
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 10
            periodSeconds: 10
            successThreshold: 1
          ...

```

15. Save your changes to the deployment and close the editor. Look back to the terminal session where you have the watch running. Eventually, you should see a new pod finished creating and start running. The previous web pod will be terminated and removed. Leave the watch running in the other window for the next lab.

END OF LAB

Lab 2 - Working with services and ports

1. Our app is now running as we can see at the end of lab 1. Let's take a look at the services that we have.

```
$ k get svc
```

2. The service for the webapp (roar-web) is the one we would access in the browser to see the application. But notice that it is of type ClusterIP. This type of service is intended for access within the cluster, meaning we can't access it directly. To access it, we need to forward the port to a port on the host machine. Find the port that the svc is using internally by looking under "PORT(S)" column in the output from step 1. Should be "8089".
3. We can forward this port to the host system with a "kubectl port-forward" command. In a different terminal session (you can stop the watch in the other terminal with a Ctrl-C and use that one), run the command below to forward the port from the service to a port on the host system. The ":" syntax will let Kubernetes find an unused port. Alternatively, we could supply a specific port to forward to.

```
$ kubectl port-forward svc/roar-web :8089
```

4. Take note of what host port the service port gets forwarded to (will be a very high number). In a browser on the host system, open up the web application at the url below.

<http://localhost:<port-from-above>/roar>

5. You should see a page like below. Notice that while we have the web app showing, there is no data being displayed. This suggests that there is something wrong with being able to get data from the database.

R.O.A.R (Registry of Animal Responders) Agents						
Show 10 entries			Search: <input type="text"/>			
Id	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
No data available in table						

Showing 0 to 0 of 0 entries Previous Next

- Let's take a quick look at the logs for the current mysql pod to see if there's any issues showing there.

```
$ k logs -l app=roar-db
```

- Things should look ok in the logs. Let's use exec to run a query from the database. If you are on Windows, you will need the "winpty" command in front. You'll need the pod name of the mysql pod name (which you can get from 'k get pods' and then copy just the NAME part for the mysql pod). Also use "kubectl" here, not the alias "k".

```
$ [winpty (if on windows)] kubectl exec -it <mysql-pod-name> --
mysql -uadmin -padmin -e 'select * from registry.agents'
```

- This should return a set of data. Since that works, let's move on to check the endpoints - to see if there are pods actually connected to the service. You can use the get endpoints command to do this.

```
$ k get ep
```

- This shows no endpoints for the mysql service. Endpoints are connected through matching labels. Let's see what labels the service is looking for.

```
$ k get svc/mysql -o yaml | grep -A1 selector
```

- From this we can see that the service is looking to select pods to talk to that have a label of "name: roar-db". So let's see what labels the pod for the database has.

```
$ k get pods --show-labels | grep mysql
```


11. From the output here, we can see that the pod does not have the label "name: roar-db" that the service is trying to use to select a pod to connect to. There are a couple of different ways to fix this, but the most simple may be just to update the label to be the one that is expected via the command below. Note that the first -l is a selector via an existing label that we then overwrite.

```
$ k label pod -l name=mysql --overwrite name=roar-db
```

12. After the command above is run, you should be able to get the list of endpoints again and see that there is a pod now matched to the mysql service. Then you can refresh your browser session and you should see data in the app as below.

```
$ k get ep
```

After refresh...

R.O.A.R (Registry of Animal Responders) Agents						
Show 10 entries		Search: <input type="text"/>				
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

Showing 1 to 5 of 5 entries Previous 1 Next

END OF LAB

Lab 3 - 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 69. 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 mysql-secret.yaml
```

```
$ kubectl apply -f mysql-secret.yaml
```

- Now that we have the secret created in the namespace, we need to update our spec to use the values from it. You don't need to make any changes in this step, but 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

- 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
```

```
$ k apply -f mysql-configmap.yaml
```

- Similar to the changes to use the secret, we would need to change the main yaml file to use the new configmap. Again, you don't need to make any changes in this step, but 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.

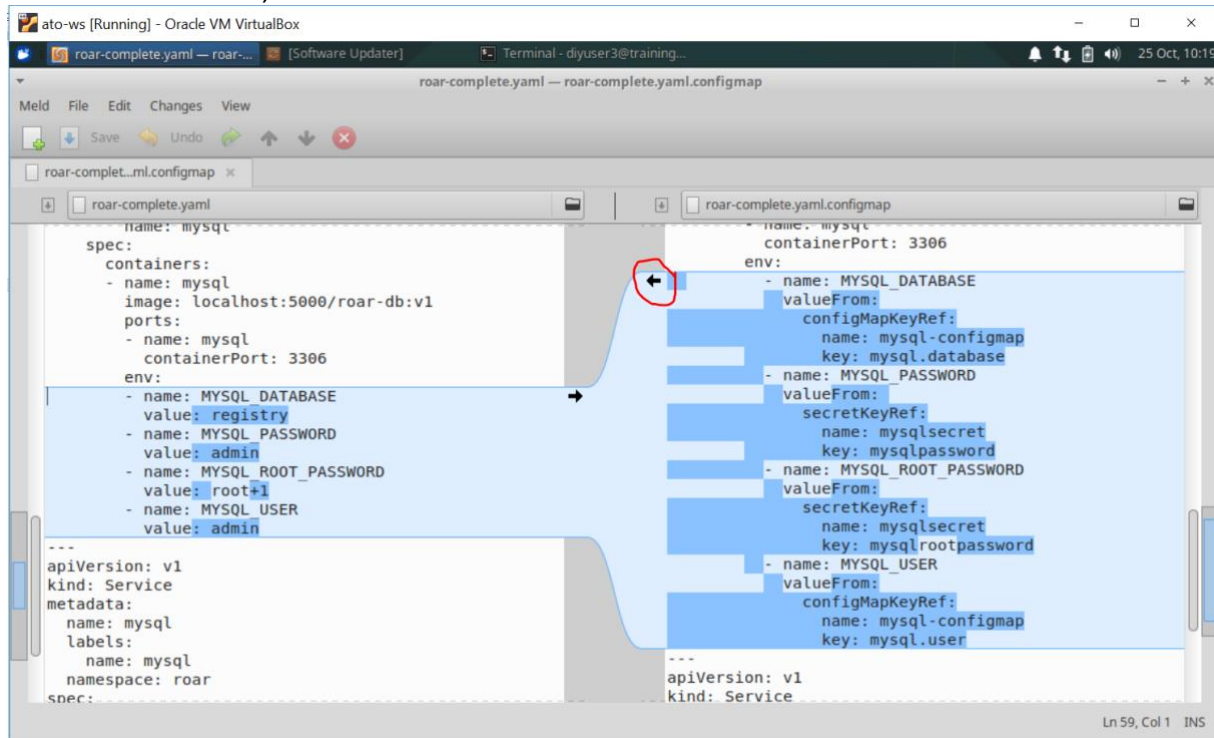
```
$ meld roar-complete.yaml roar-complete.yaml.configmap
```

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

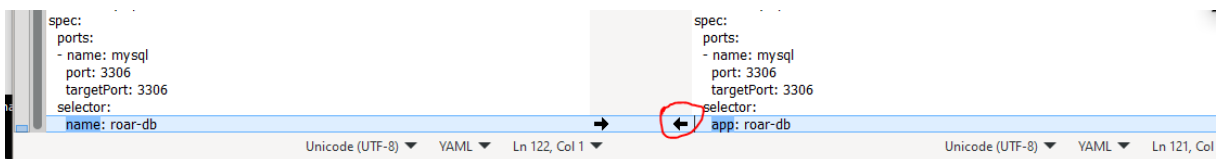
8. Now we'll update our **roar-complete.yaml** file with the needed changes. To save trying to get the yaml all correct in a regular editor, we'll just use the meld tool's merging ability.

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. Also, there will be one more change at the bottom to update the selector that the service is using. This will fix the problem permanently that we encountered in lab 2 with the endpoints so it won't be an issue for future labs. Again, just click on the arrow circled in red.



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
```

END OF LAB

Lab 4 – 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. If you don't already have a browser session open browser with the instance that you're running in the "roar" namespace, open it again. (Ref lab 2, steps 1-5, if you need to do this again.)
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.) You can ignore the warnings.

```
$ ./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?

```
$ kubectl delete pod -l app=roar-db
```

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 apply -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:

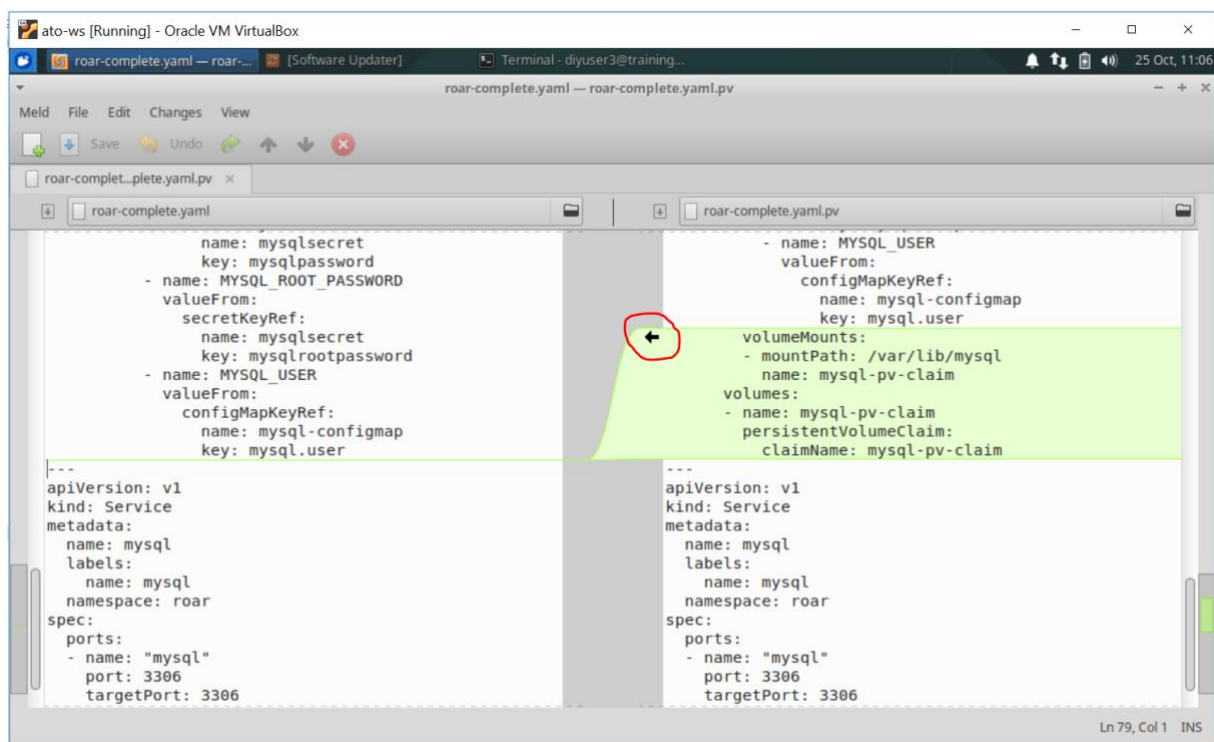
```
spec:
  containers:
    - name: mysql
      ...
      - name: MYSQL_USER
        valueFrom:
          configMapKeyRef:
            name: mysql-configmap
            key: mysql.user
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: mysql-pv-claim
    volumes:
    - name: mysql-pv-claim
      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, refresh the screen and notice that the new record is still there!

END OF LAB

Lab 5 – Working with Helm

Purpose: In this lab, we'll compare a Helm chart against standard Kubernetes manifests and then deploy the Helm chart into Kubernetes

1. For this lab, reset the default namespace.

```
$ k config set-context --current --namespace=default
```

2. In the *manifests* subdir, we have the “regular” Kubernetes manifests for our app, with the database pieces in a sub area under the web app pieces. Then in the *helm* subdir, we have a similar structure with the charts for the two apps.

To get a better idea of how Helm structures content, do a diff of the two areas. If you are running in the VM, or have meld installed, you can use it and you will see a picture similar to below. Otherwise, you can use any suitable diffing tool.

```
$ cd ~/k8s-dev (if not already there)
```

```
$ meld manifests helm
```

manifests			helm		
Name	Size	Modification time	Name	Size	Modification time
manifests	0 B	Sun 19 Jun 2022 13:49:57	helm	0 B	Tue 21 Jun 2022 07:26:21
extra			extra	4.1 kB	Tue 21 Jun 2022 00:11:23
lab2-deployment.yaml			lab2-deployment.yaml	1.5 kB	Mon 20 Jun 2022 23:46:44
lab2-values.yaml			lab2-values.yaml	377 B	Mon 20 Jun 2022 23:54:35
roar-port.sh			roar-port.sh	170 B	Tue 21 Jun 2022 00:00:48
test-db.yaml			test-db.yaml	83 B	Tue 21 Jun 2022 00:11:23
roar-web	4.1 kB	Sun 19 Jun 2022 14:03:46	roar-web	4.1 kB	Sun 19 Jun 2022 12:20:37
charts			charts	0 B	Sun 19 Jun 2022 12:20:37
roar-db			roar-db	0 B	Sun 19 Jun 2022 12:20:37
templates			templates	4.1 kB	Sun 19 Jun 2022 14:11:08
_helpers.tpl			_helpers.tpl	778 B	Sun 19 Jun 2022 12:20:37
configmap.yaml			configmap.yaml	163 B	Sun 19 Jun 2022 14:12:13
deployment.yaml			deployment.yaml	1.5 kB	Sun 19 Jun 2022 14:42:06
secret.yaml			secret.yaml	179 B	Sun 19 Jun 2022 14:11:57
service.yaml			service.yaml	544 B	Sun 19 Jun 2022 12:20:37
Chart.yaml			Chart.yaml	95 B	Sun 19 Jun 2022 12:20:37
values.yaml			values.yaml	314 B	Sun 19 Jun 2022 14:07:16
roar-db	4.1 kB	Sun 19 Jun 2022 14:03:46	roar-db		
configmap.yaml	144 B	Sun 19 Jun 2022 13:49:39	configmap.yaml		
deployment.yaml	1.1 kB	Sun 19 Jun 2022 14:42:30	deployment.yaml		
secret.yaml	160 B	Sun 19 Jun 2022 13:49:39	secret.yaml		
service.yaml	208 B	Sun 19 Jun 2022 13:49:39	service.yaml		
templates			templates	0 B	Sun 19 Jun 2022 12:20:37
_helpers.tpl			_helpers.tpl	507 B	Sun 19 Jun 2022 12:20:37
deployment.yaml			deployment.yaml	910 B	Sun 19 Jun 2022 12:20:37
service.yaml			service.yaml	595 B	Sun 19 Jun 2022 12:20:37
Chart.yaml			Chart.yaml	99 B	Sun 19 Jun 2022 12:20:37
deployment.yaml	473 B	Sun 19 Jun 2022 13:49:39	deployment.yaml		
service.yaml	235 B	Sun 19 Jun 2022 13:49:39	service.yaml		
values.yaml			values.yaml	402 B	Sun 19 Jun 2022 14:22:55

3. Notice the format of the two area is similar, but the helm one is organized as chart structures.

Let's take a closer look at the differences between a regular K8s manifest and one for Helm. We'll use the deployment one from the web app. Again, you can use meld or a suitable diffing/comparison tool. Notice the differences in the two formats, particularly the placeholders in the Helm chart (with the `{{ }}` pairs instead of the hard-coded values.

```
$ meld manifests/roar-web/deployment.yaml helm/roar-web/templates/deployment.yaml
```

manifests/roar-web/deployment.yaml	helm/roar-web/templates/deployment.yaml
apiVersion: apps/v1	apiVersion: apps/v1
kind: Deployment	kind: Deployment
metadata:	metadata:
name: roar-web	name: {{ template "roar-web.name" . }}
labels:	labels:
name: roar-web	app: {{ template "roar-web.name" . }}
app: roar-web	chart: {{ .Chart.Name }}-{{ .Chart.Version replace "+" "_" }}
namespace: roar	release: {{ .Release.Name }}
spec:	spec:
selector:	replicas: {{ .Values.replicaCount }}
matchLabels:	selector:
app: roar-web	matchLabels:
replicas: 1	app: {{ template "roar-web.name" . }}
template:	template:
metadata:	metadata:
labels:	labels:
name: roar-web	app: {{ template "roar-web.name" . }}
app: roar-web	
spec:	spec:
containers:	containers:
- name: roar-web	- name: {{ .Chart.Name }}
image: quay.io/bclaster/roar-web:1.0.1	image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
ports:	imagePullPolicy: {{ toYaml .Values.image.pullPolicy }}
- name: web	ports:
containerPort: 8080	- name: {{ .Values.deployment.ports.name }}
	containerPort: {{ .Values.deployment.ports.containerPort }}

4. Go ahead and close meld (or any other diff tool you're using) when you're done looking at the differences. We've already seen how to deploy the standard Kubernetes manifests and how to look at the app running in the browser. Now let's install the helm release to see how those are deployed.
5. Now let's install the helm release.

```
$ helm install roar-helm helm/roar-web
```

You should see output like the following:

```
NAME: roar-helm
LAST DEPLOYED: Mon Jun 20 16:22:19 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

6. Take a look at the Helm releases we have running in the cluster now and the resources it added to the default namespace.

```
$ helm list -A
```

You should see output like the following:

<i>NAME</i>	<i>NAMESPACE</i>	<i>REVISION</i>	<i>UPDATED</i>
<i>STATUS</i>	<i>CHART</i>	<i>APP VERSION</i>	
<i>roar-helm</i>	<i>default</i>	<i>1</i>	<i>2022-06-20 16:22:19.1237099</i>
<i>-0400 EDT</i>	<i>deployed</i>	<i>roar-web-0.1.0</i>	

```
$ k get all
```

7. We really don't want this running in the default namespace. We'd rather have it running in a specific one for the application. Let's get rid of the resources in K8s tied to this release and verify that they're gone.

```
$ helm uninstall roar-helm
```

```
$ helm list -A
```

```
$ k get all
```

8. Now we can create a new namespace just for the helm version and deploy it into there. Note the addition of the “-n roar-helm” argument to direct it to that namespace.

```
$ k create ns roar-helm
```

```
$ helm install -n roar-helm roar-helm helm/roar-web
```

```
$ helm list -A
```

```
$ k get all -n roar-helm
```

9. After a minute or two, the application should be running in the cluster in the roar-helm namespace. If you want, you can look at the app running on your system. This service is setup as a type NodePort, so we need to first find the NodePort value. It will be after 8089: in the output of the following command and will have a value in the 30000's.

```
$ k get svc -n roar-helm
```

10. **If you are not running on the VM**, you will need to do a port-forward command first to expose the port, like the following:

```
$ k port-forward -n roar-helm svc/roar-web <nodeport-from-step-9>:8089
```

11. Now you can open up a browser session to the url below and see the running application from the roar-helm namespace.

<http://localhost:<nodeport-value-from-step-9>/roar>

END OF LAB

Lab 6: Templating with Helm

Purpose: In this lab, you'll get to see how we can change hard-coded values into templates, override values, and upgrade releases through Helm.

1. Take a look at the deployment template in the `roar-helm` directory and notice what the `"image"` value is set to.

```
$ cd ~/k8s-dev/helm/roar-web
```

```
$ grep image charts/roar-db/templates/deployment.yaml
```

Notice that the value for `image` is hardcoded to `"quay.io/techupskills/roar-db:v2"`.

2. We are going to change this to use the Helm templating facility. This means we'll change this value in the `deployment.yaml` file to have "placeholders". And we will put the default values we want to have in the `values.yaml` file. You can choose to edit the deployment file with the `"gedit"` editor if running in the VM or use another editor if not running in the VM. Or you can use the `"meld"` tool if running in the VM (or a different merge tool if not running in the VM) to add the differences from a file that already has them. If using the `meld` tool, select the right arrow to add the changes from the second file into the `deployment.yaml` file. Then save the changes.

Either do:

```
$ gedit charts/roar-db/templates/deployment.yaml
```

And change

```
image: quay.io/techupskills/roar-db:v2
```

To

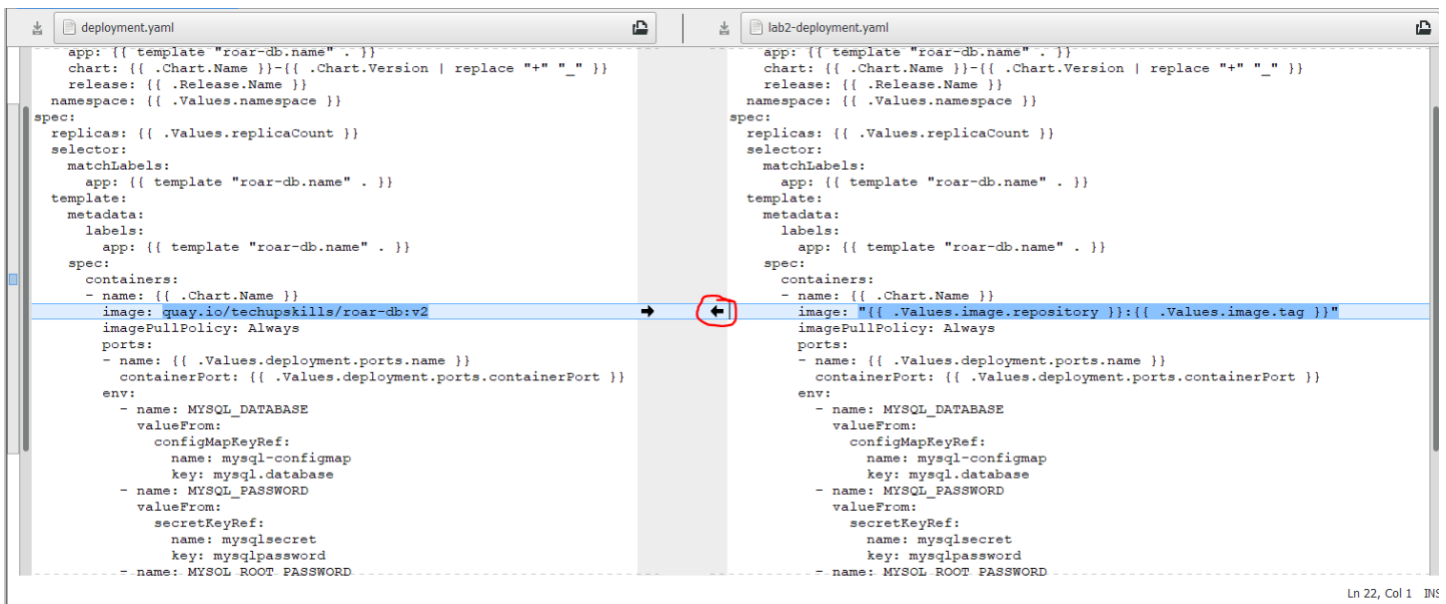
```
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

Save your changes and exit the editor.

Or:

```
$ meld charts/roar-db/templates/deployment.yaml ../extra/lab6-deployment.yaml
```

Then click on the arrow circled in red in the figure. This will update the template file with the change. Then Save your changes and exit `meld`.



- Now that we've updated the deployment template, we need to add default values. We'll use the same approach as in the previous step to add defaults for the *image.repository* and *image.tag* values in the chart's *values.yaml* file.

Either do:

```
$ <editor> charts/roar-db/values.yaml
```

And add to the top of the file:

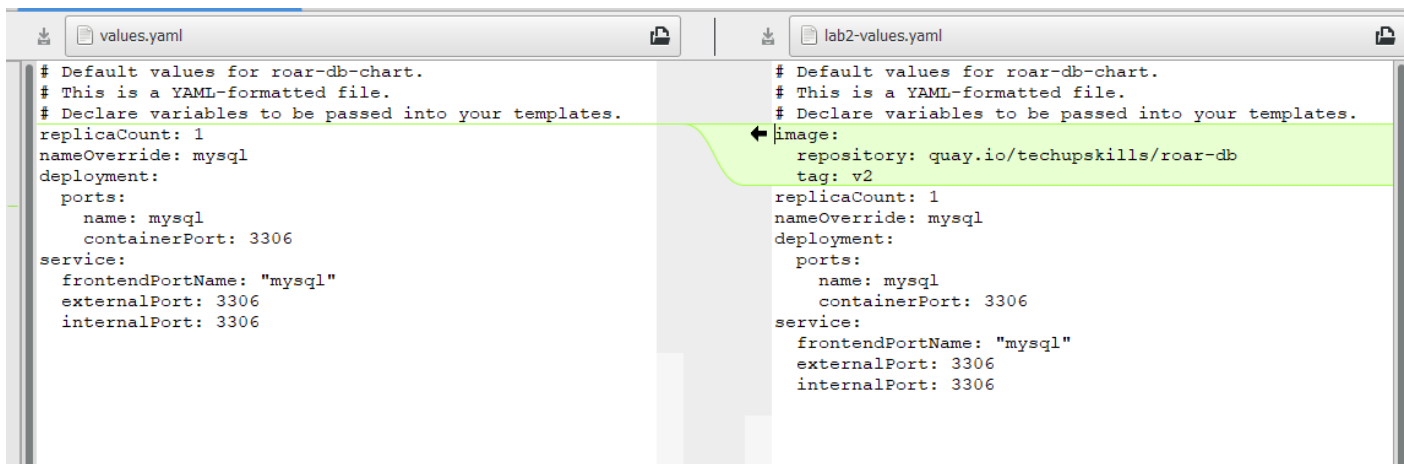
```
image:
  repository: quay.io/techupskills/roar-db
  tag: v2
```

Note that the first line should be all the way to the left and the remaining two lines are indented 2 spaces. Save your changes.

Or:

```
$ meld charts/roar-db/values.yaml ../extra/lab6-values.yaml
```

Then click on the arrow circled in red in the figure. This will update the values file with the change. Then Save your changes and exit meld.



4. Update the existing release.

```
$ helm upgrade -n roar-helm roar-helm .
```

5. Find the nodeport where the app is running and open it up in a browser.

```
$ k get svc -n roar-helm
```

Look for the NodePort setting in the service output (should be a number > 30000 after "8089:")

6. **If you are not running in the VM**, you may need to do a port-forward to be able to access the app on the local machine's browser. That is, you would do

```
$ k port-forward -n roar-helm service/roar-web
<nodeport-value>:8089
```

7. Open up a browser and go to <http://localhost:<NodePort>/roar/>

You should see the same webapp and data as before.

(If not running in the VM, you can kill the roar-port.sh run now with ctrl-c.)

8. Let's suppose we want to overwrite the image used here to be one that is for a test database. The image for the test database is on the quay.io hub at [quay.io/bclaster/roar-db-test:v4](https://quay.io/repository/bclaster/roar-db-test:v4).

We could use a long command line string to set it and use the template command to show the proposed changes between the rendered files. In the roar-web

subdirectory, run the commands below to see the difference. (You should be in the ~/k8s-dev/helm/roar-web. Note the "." In the commands.)

```
$ helm template . --debug | grep image
```

```
$ helm template . --debug
--set roar-db.image.repository=quay.io/bclaster/roar-db-test
--set roar-db.image.tag=v4 | grep image
```

9. Now, in another terminal window , start a watch of the pods in your deployed helm release. This is so that you can see the changes that will happen when we upgrade.

```
$ k get pods -n roar-helm --watch
```

10. Finally, let's do an upgrade using the new values file. In a **separate terminal window** from the one where you did step 9, execute the following commands:

```
$ cd ~/k8s-dev/helm/roar-web
```

```
$ helm upgrade -n roar-helm roar-helm .
--set roar-db.image.repository=quay.io/bclaster/roar-db-test
--set roar-db.image.tag=v4 --recreate-pods
```

Ignore the warning. Watch the changes happening to the pods in the terminal window with the watch running.

11. Repeat steps 5 and 6 to get the nodeport and do the port-forward if you need to. Then go back to your browser and refresh it. You should see a version of the (TEST) data in use now. (Depending on how quickly you refresh, you may need to refresh more than once.)

ROAR Agents - Mozilla Firefox

localhost:30961/roar/

R.O.A.R (Registry of Animal Responders) Agents

Show 10 entries Search:

Id	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
1	(TEST) Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
2	(TEST) Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun
3	(TEST) Woody Woodpecker	bird	1959-05-22	1979-04-15	Buzz Buzzard	menacing stare
4	(TEST) Road Runner	bird	1955-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
5	(TEST) Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
6	(TEST) Perry	platypus	2013-01-20	2015-04-09	H. Doofenmirtz	...inator

12. Go ahead and stop the watch from running in the window via Ctrl-C.

\$ Ctrl-C

END OF LAB

Lab 7 - Run a basic Kustomize example

Purpose: In this lab, we'll see how to make a set of manifests usable with Kustomize and how to use Kustomize to add additional changes without modifying the original files. (For these labs, we have "alias kz=kustomize" if you have kustomize installed. You may also use "kubectl kustomize" in place of "kz build" and "kubectl apply -k" in place of running Kustomize and pipeline to apply.)

1. Change to the **base** subdirectory. In this directory, we have deployment and service manifests for a simple webapp that uses a MySQL database and a file to create a namespace. You can see the files by running the tree command or recursive directory command (or tree if you have it installed).

\$ cd ~/k8s-dev/kz/base

\$ tree (or ls -R if you don't have tree installed)

2. Let's see what happens when we try to run "kustomize build" against these files. (On the VM, I have "kustomize" aliased as "kz".) There will be an error.

```
$ kz build (or kubectl kustomize)
```

3. Notice the error message about there not being a kustomization file. Let's add one. There's a basic one in the "extra" directory named "kustomization.yaml". Copy it over into the directory renaming it without the extension. Take a look at the contents to see what it does and then run the build command again, passing it to kubectl apply.

```
$ cp ../extra/kustomization.yaml kustomization.yaml
```

```
$ cat kustomization.yaml
```

```
$ kz build | k apply -f - (or kubectl apply -k .)
```

4. So which namespace did this get deployed to? It went to the "default" one which you can see by looking at what's in there. (On the VM, "kubectl" is aliased as just "k".)

```
$ k get all
```

5. We have a *namespace.yaml* file in the directory. Take a look at it. It is setup to create a namespace. So how do we use it with Kustomize? Since it's another resource, we just need to include it in our list of resources. And then we also need to specify the namespace it creates ("roar-kz") in the kustomization file.

Edit the kustomization.yaml file, and **add the namespace line** at the top (line 2) and **add namespace.yaml** at the end of the list of resources (line 11). **Save your changes and exit the editor when done.** (gedit is installed on the VM. You may use a different editor in place of <edit> if you want.)

```
$ cat namespace.yaml
```

```
$ <edit> kustomization.yaml
```

```

1
2 namespace: roar-kz
3
4 # List of resource files that kustomize reads, modifies
5 # and emits as a YAML string
6 resources:
7 - ./roar-db/deployment.yaml
8 - ./roar-db/service.yaml
9 - ./roar-db/secret.yaml
10 - ./roar-db/configmap.yaml
11 - ./roar-web/deployment.yaml
12 - ./roar-web/service.yaml
13 - namespace.yaml

```

6. Now that we've added the namespace resource, let's try the kustomize build command again to see if our namespace "roar-original" shows up where expected. You should see the manifest to create the namespace now included at the top of the output and the various resources having the namespace added.

```
$ kz build | grep -n3 roar-kz
```

7. Now we can go ahead and apply this again. Afterwards you can verify that the new namespace got created and that our application is running there.

```
$ kz build | k apply -f -
```

```
$ k get ns
```

```
$ k get all -n roar-kz
```

8. Let's make one more change here. Let's apply a common annotation to our manifests. Edit the kustomization file again and add the top 2 lines as shown in the screenshot. When you are done, save your changes and exit the editor.

```
$ <edit> kustomization.yaml
```

The 2 lines are:

```

commonAnnotations:
  version: base

```

```

1  commonAnnotations:
2    version: base
3
4  namespace: roar-kz
5
6  # List of resource files that kustomize reads, modifies
7  # and emits as a YAML string
8  resources:
9    - ./roar-db/deployment.yaml
10   - ./roar-db/service.yaml
11   - ./roar-db/secret.yaml
12   - ./roar-db/configmap.yaml
13   - ./roar-web/deployment.yaml
14   - ./roar-web/service.yaml
15   - namespace.yaml
16

```

9. Now you can run kustomize build and see the annotations. Afterwards you can go ahead and apply the changes. Look for the added annotation to all of the resources.

```
$ kustomize build | grep -a5 metadata
```

```
$ kustomize build | kubectl apply -f -
```

10. The instance of our application should be running in the roar-kz namespace. If you want to look at it, you can find the Nodeport where it is running and then open up the URL with that port in a browser to see the running application.

```
$ kubectl get svc -n roar-kz | grep web
```

```
<find Nodeport - second to last column - value
after 8089 - value in the 30000's>
```

11. If you are not running in the VM, do a port-forward from the service as before.
12. Open <http://localhost:<nodeport>/roar> in browser

END OF LAB

Lab 8 - Creating Variants

Purpose: In this lab, we'll see how to create production and stage variants of our simple application.

1. To illustrate how variants work, we'll first create a directory for the overlays that will create our staging and production variants. Change back to the *kz* directory and create the two directories.

```
$ cd ~/k8s-dev/kz
```

```
$ mkdir -p overlays/staging overlays/production
```

2. In order to pick up the necessary files to build the variants we'll need *kustomization.yaml* files in the directories pointing back to the appropriate resources. For simplicity, we'll just seed the directories with a *kustomization.yaml* file that points back to our standard bases. Execute the copy commands below to do this. After this, your directory tree should look as shown at the end of this step.

```
$ cp extra/kustomization.yaml.variant overlays/staging/kustomization.yaml
```

```
$ cp extra/kustomization.yaml.variant overlays/production/kustomization.yaml
```

```
$ ls -R overlays (or tree overlays if you have tree installed)
```

```
overlays
├── production
│   └── kustomization.yaml
└── staging
    └── kustomization.yaml
```

3. We now have an overlay file that we can use with Kustomize. Take a look at what's in it and then let's make sure we can build with it.

```
$ cat overlays/staging/kustomization.yaml
```

```
$ kz build overlays/staging
```

4. What namespace will this deploy to if we apply it as is? Look back up through the output from the previous step. Notice that if we applied it as is, it would go to the *roar-kz* namespace. Let's use separate namespaces for the staging overlay and the production overlay. To do that we'll just add the *namespace* transformer to the two new *kustomization.yaml* files. You can either edit the files and add the respective lines or just use the shortcut below.

```
$ echo namespace: roar-staging >> overlays/staging/kustomization.yaml
```

```
$ echo namespace: roar-production >> overlays/production/kustomization.yaml
```

5. Now you can do a *kustomize* build on each to verify it has the desired namespace in the output.

```
$ kz build overlays/staging | grep namespace
```

```
$ kz build overlays/production | grep namespace
```

6. Let's go ahead and apply these to get the variants of our application running. Since we didn't include a different namespace file to create the namespaces, we'll need to create those first. Then we can build and apply the variants. If you want afterwards, you can do the same thing we did at the end of lab 1 to find the nodeports and see the variants running. (You can ignore the warnings.)

```
$ k create ns roar-staging
```

```
$ k create ns roar-production
```

```
$ kz build overlays/staging | k apply -f -
```

```
$ kz build overlays/production | k apply -f -
```

7. Let's suppose that we want to make some more substantial changes in our variants. For example, we want to use test data in the version of our app running in the *roar-staging* namespace. The test data is contained in a different image at *quay.io/bclaster/roar-db-test:v4*. To make the change we'll use another transformer called "images". To use this, edit the *kustomization.yaml* file in the *overlays/staging* area and add the lines shown at the end of the file in the screenshot below (starting at line 10).

(There is also a "kustomization.yaml.test-image" file in the "extra" directory if you need a reference.)

```
$ <edit> overlays/staging/kustomization.yaml
```

```
images:
- name: quay.io/techupskills/roar-db:v2
  newName: quay.io/bclaster/roar-db-test
  newTag: v4
```

```
1
2 resources:
3 - ../../base
4
5 namespace: roar-staging
6
7 # Change the image name and version
8 images:
9 - name: quay.io/techupskills/roar-db:v2
10   newName: quay.io/bclaster/roar-db-test
11   newTag: v4|
```

8. Save your changes, close the editor, and then apply the variant.

```
$ kz build overlays/staging | k apply -f -
```

9. You can now find the nodeport for the service from roar-staging.

```
$ k get svc -n roar-staging | grep web
```

10. If you are not running in the VM, do a port-forward from the service as before.

11. Refresh and see the test version of the data.

R.O.A.R (Registry of Animal Responders) Agents

Show 10 entries Search:

Id	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
1	(TEST) Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
2	(TEST) Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun
3	(TEST) Woody Woodpecker	bird	1959-05-22	1979-04-15	Buzz Buzzard	menacing stare
4	(TEST) Road Runner	bird	1955-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
5	(TEST) Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
6	(TEST) Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	...inator

It looks like you haven't started Firefox in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welcome back! Refresh Firefox...

END OF LAB

Lab 9 - 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 look at the built-in Kubernetes dashboard. You should already have this installed from the setup. You can use a simple port-forward to access. Do this in a secondary terminal session.

```
$ k port-forward -n kubernetes-dashboard svc/kubernetes-dashboard :443
```

2. Take the port you get back from the command above - the one after "127.0.0.1" and open up a browser to:

https://localhost:<port>

3. In the browser, you'll see a login screen. We'll use the token option to get in. You may already have the token from the setup. If not, in the k8s-dev/monitoring directory is a script to generate the token. Run the script and then copy the output.

```
$ ./get-token.sh
```


- At the login screen, select "Token" as the access method, and paste the token you got from the step above.

← → ↻ 🏠 ⚠ Not secure | https://localhost:58115/#/login

RE Data Portal Octantis integrations/promo... Golang basics - writ... 00_999_Fast_track_...

Kubernetes Dashboard

☒ Token
Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out section.

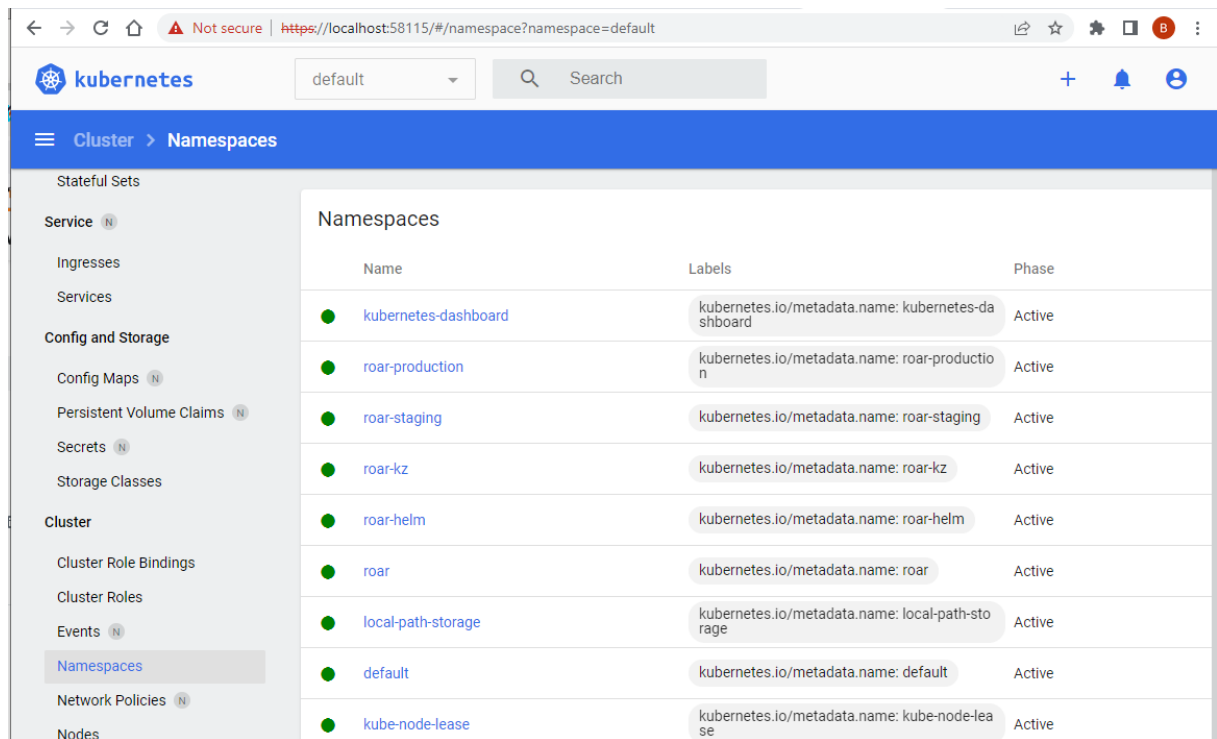
☐ Kubeconfig
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about [Clusters](#) section.

Enter token *

.....

Sign in

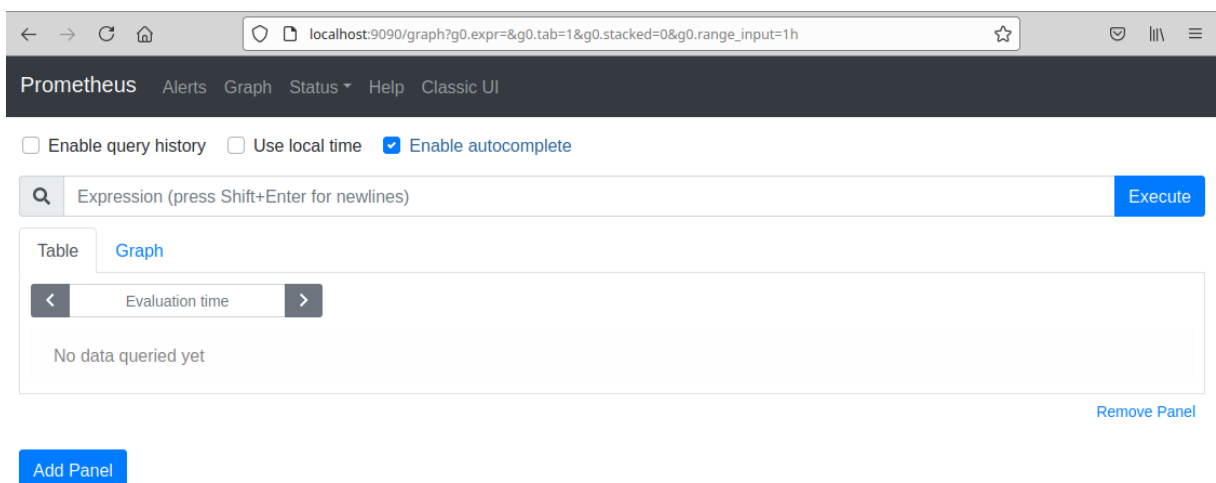
- The dashboard for our cluster will now show. You can choose K8s objects on the left and get a list of them, explore them, etc.



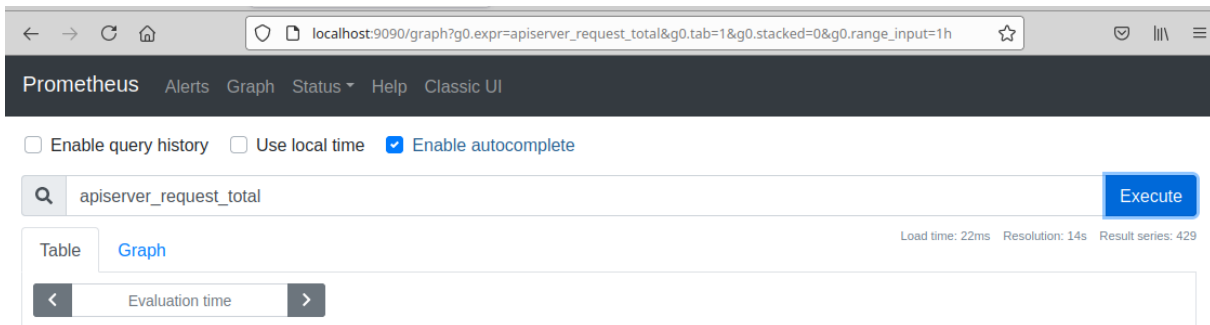
6. Now let's look at some metrics gathering with a tool called Prometheus. You should already have these installed from the setup. First, we will do a port-forward to access the Prometheus UI in our browser.

```
$ kubectl port-forward -n monitoring svc/monitoring-kube-prometheus-prometheus 9090:9090
```

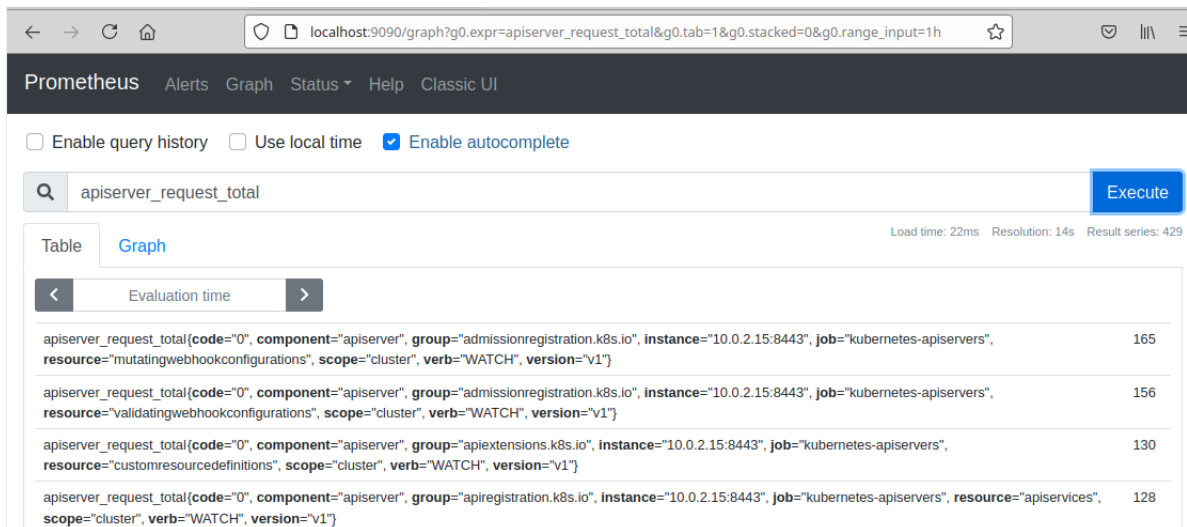
7. Now, in a new browser tab, go to <http://localhost:9090>. This may take a while, but eventually you should see a screen like the one below:



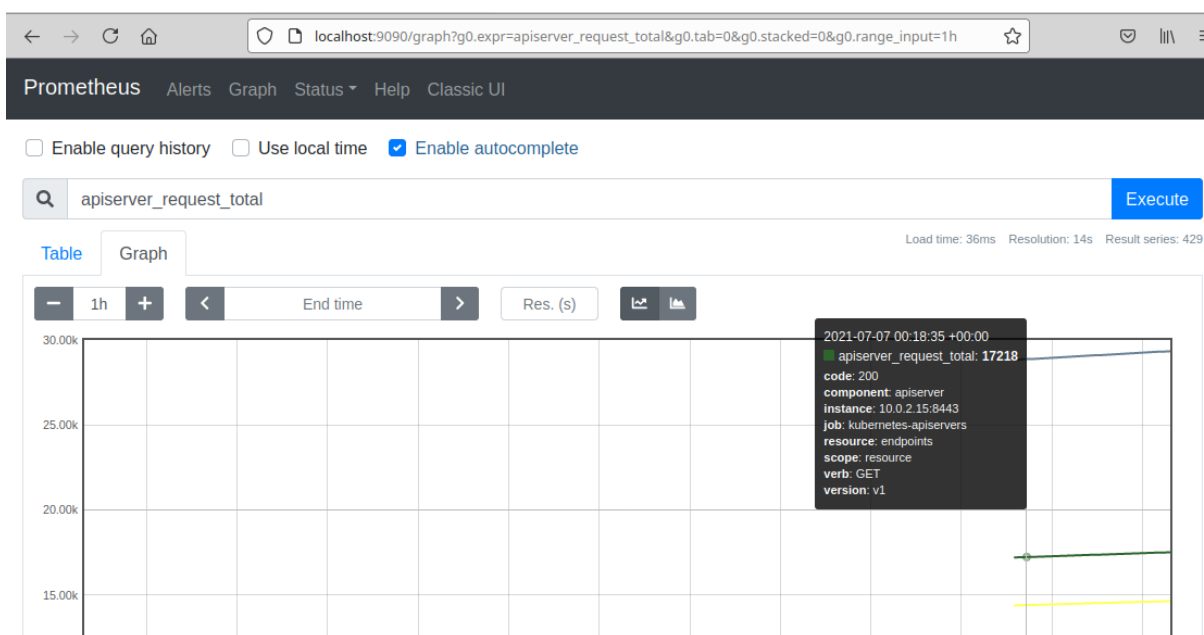
8. Prometheus comes with a set of built-in metrics. Just start typing in the “Expression” box. For example, let’s look at one called “**apiserver_request_total**”. Just start typing that in the Expression box. After you begin typing, you can select it in the list that pops up. After you have got it in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



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



10. 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. You can click "Execute" again to refresh.



11. You can also see the metrics being automatically exported for the node. **If not running in the VM**, do a port forward on the node-exporter service.

```
$ kubectl port-forward -n monitoring svc/monitoring-prometheus-node-exporter 9100:9100
```

12. And then open up a browser to <http://localhost:9100/metrics>

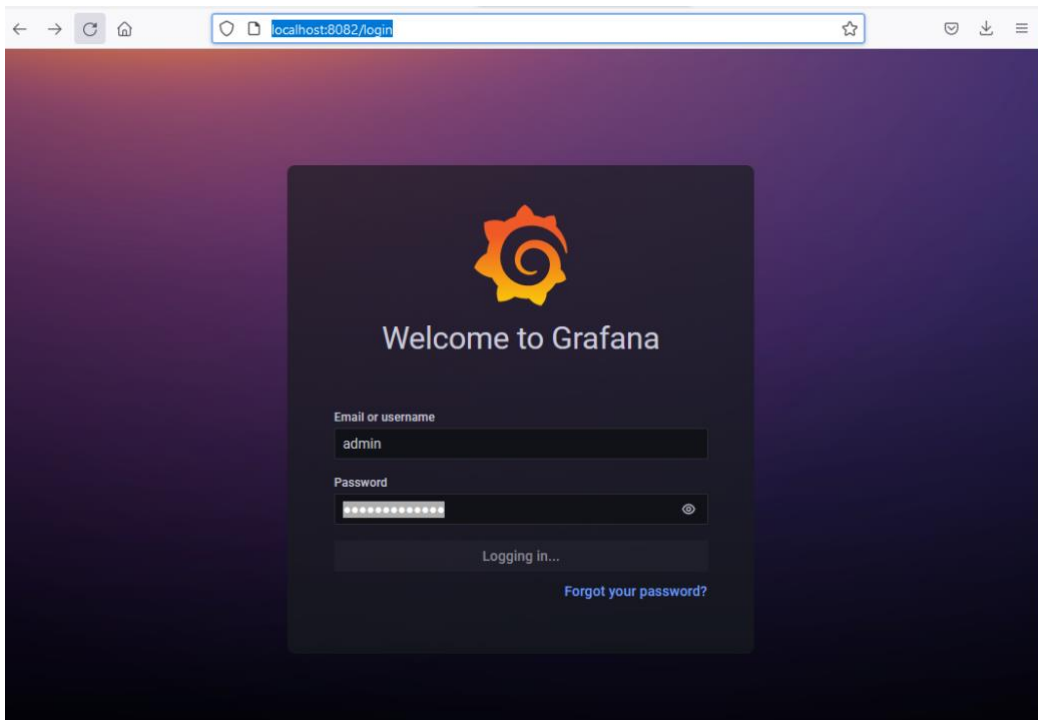
13. Finally, let's take a look at Grafana. First you need to get the default Grafana password. You should have that from the setup. But if not, you can do that with the following command to decode it from the secret.

```
$ kubectl get secret --namespace monitoring monitoring-grafana -o jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

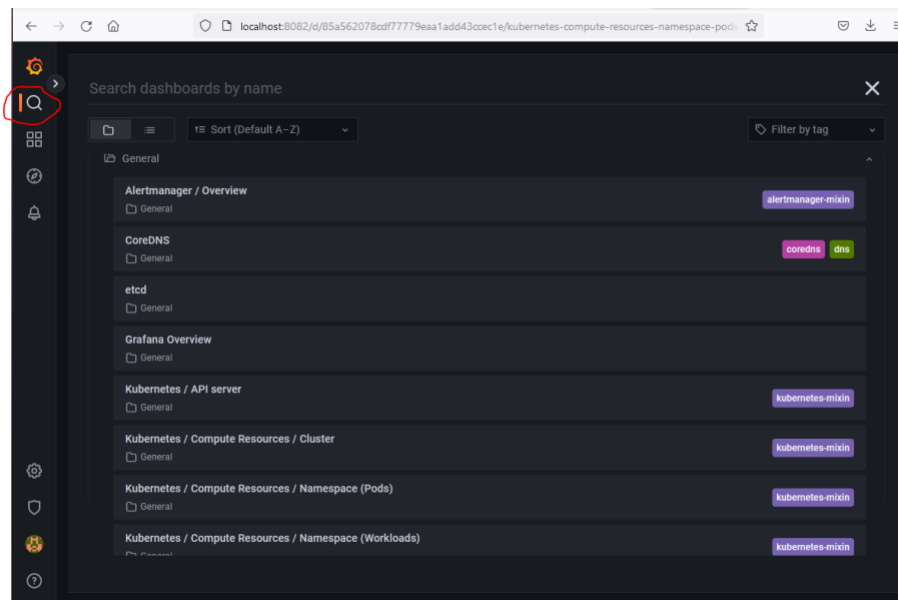
14. Then you can do a port forward for the "monitoring-grafana" service. You can pick a port or let it pick one. In this case, I've used 8082.

```
$ k port-forward -n monitoring svc/monitoring-grafana 8082:80
```

15. Open a browser to localhost:8082 (or whatever port you used). Login with username "admin" and the password from step 16.

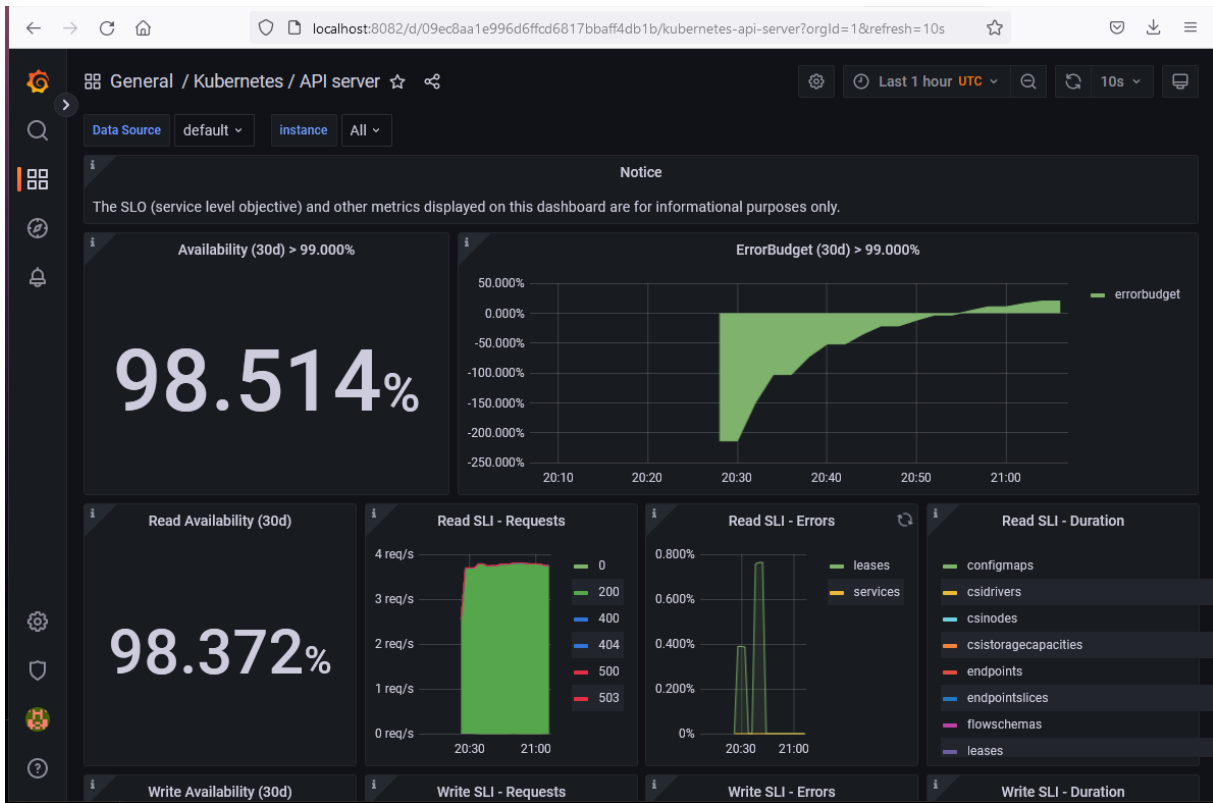


16. Click on the magnifying glass for "search" (left red circle in figure below). This will provide you with a list of built-in graphs you can click on as demos and explore.



17. Click on one of the links to view one of the demo graphs (such as the "Kubernetes / API server" one) shown in the figure below). You can then

explore others by discarding/saving this one and going back to the list and selecting others.



END OF LAB