Beyond Kubernetes

An overview of the most popular Kubernetes applications that you should know about

Workshop Labs

Version 2.3 by Brent Laster

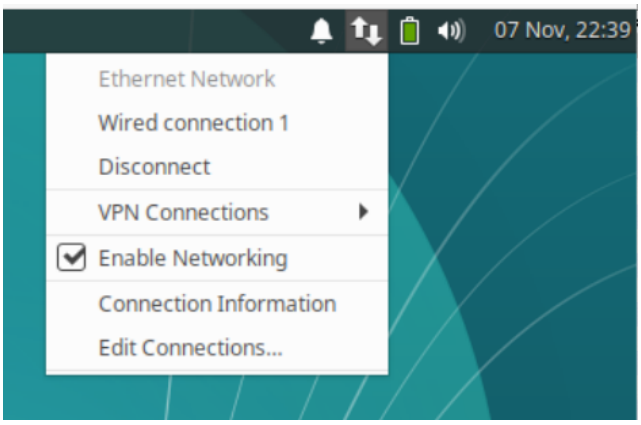
07/03/2022

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 *beyond-k8s.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!

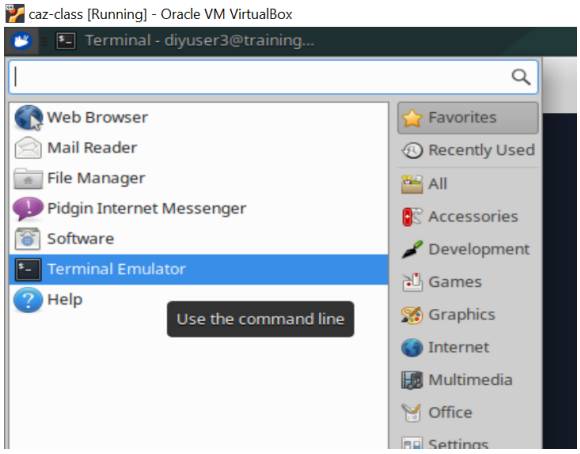
Throughout the labs, we will denote anything that is specific to the VirtualBox environment with a phrase like "**If running in the VM**".

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 beyond-k8s

$ git pull
```

If NOT running in the VM

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

$ cd beyond-k8s
```

3. **If not already done,** pre-pull images we will need for this workshop.

```
$ ./extra/image-prepull.sh

$ ./extra/image-prepull2.sh
```

4. **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
```

5. **If not already done, if running in the VM,** run the setup script to install argocd, tekton, and istio.

```
$ ./extra/setup.sh
```

6. 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
```

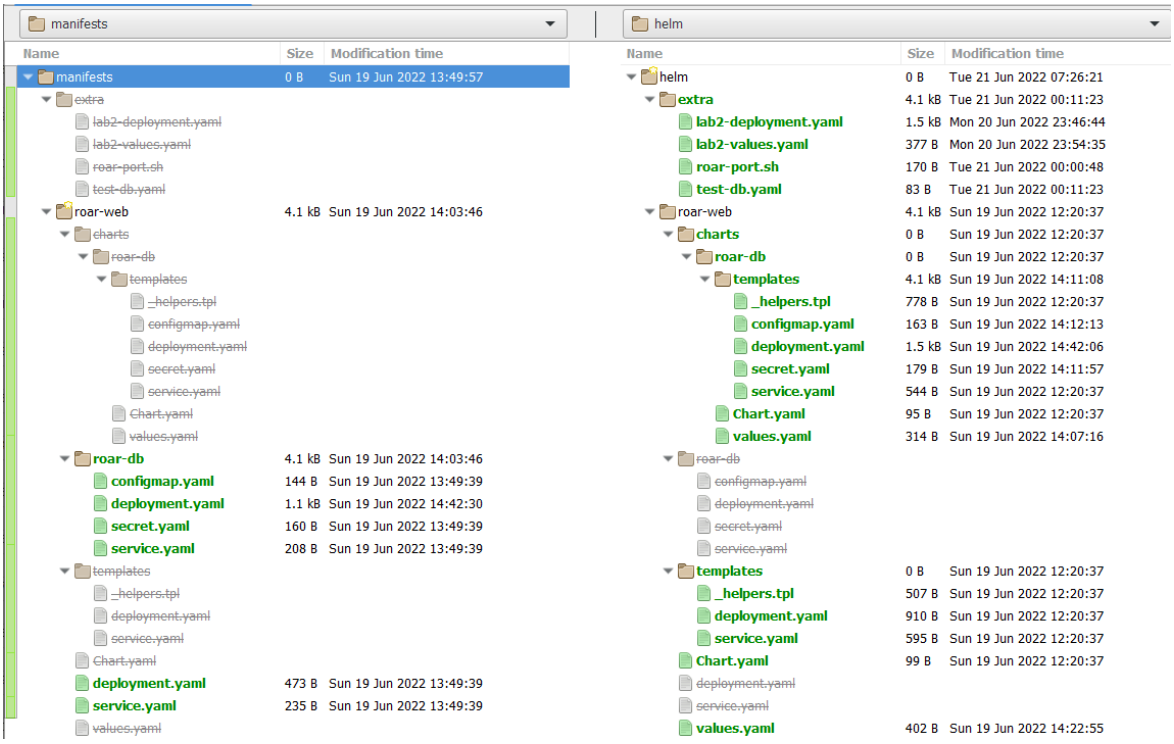
Lab 1 – Working with Helm

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

1. In our clone of the Git repository, we have different subdirectories for each of the key areas in the workshop. 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.

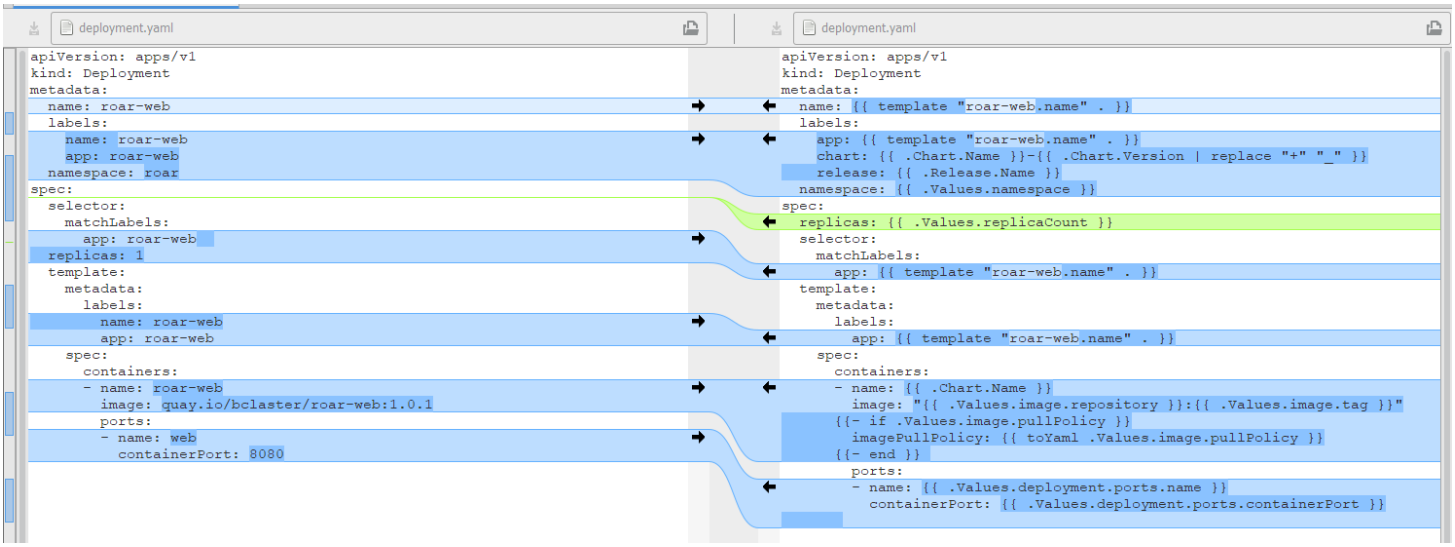
```
$ cd ~/beyond-k8s (if not already there)

$ meld manifests helm
```



2. 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
```



3. Go ahead and close meld (or any other diff tool you're using) when you're done looking at the differences. Now let's look at the differences in how these are deployed. First, create the namespace *roar* and then deploy the standard K8s yaml files in *manifests* with a Kubernetes apply command. You should see messages that indicate the items were created successfully.

   ```
   $ k create ns roar

   $ k apply -Rf manifests
   ```

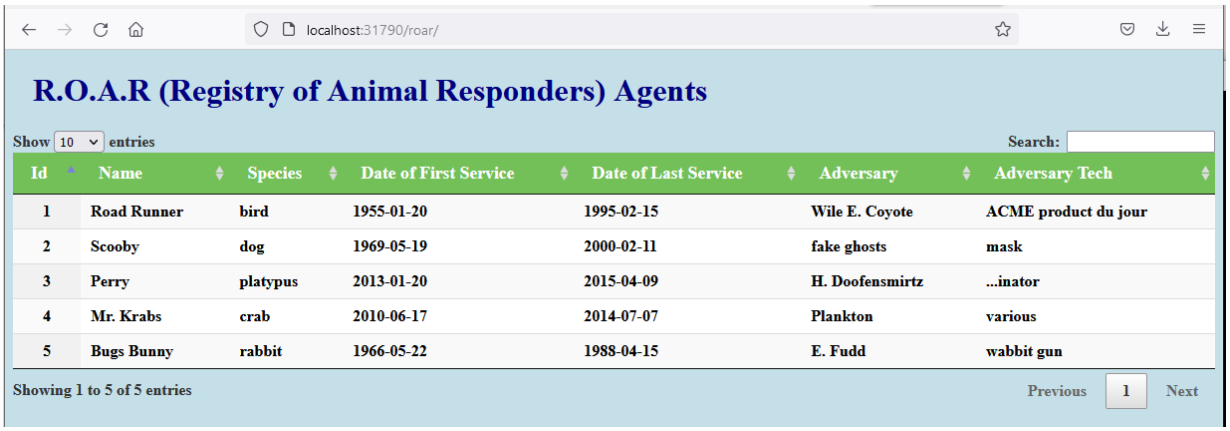
4. Let's verify the application is running as expected. The service for the web app is using a NodePort of 31790. If you are not running on the VM, you'll need to do a kubectl port-forward command for the service to be able to access the application on the running machine. (If you are running on the VM, then you can skip the first step.) Then, you can go to the URL in the second part below to see the application running. It should look like the figure shown.

If you are not running in the VM

```
$ k port-forward -n roar svc/roar-web 31790:8089
```

Whether you are running in the VM or not

<in browser> http://localhost:31790/roar



5. After you're done visiting the site, **if you're not running in the VM,** you can kill the port-forward command with a **ctrl-c** command.

6. 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

7. 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:

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

```
$ k get all
```

8. 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
```

9. 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
```

10. After a minute or two, the application should be running in the cluster in the roar-helm namespace. If you want, you can repeat step 4 from above with the nodePort of the web service in this namespace to see the Helm-deployed app in action.

END OF LAB

Lab 2: 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 ~/beyond-k8s/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

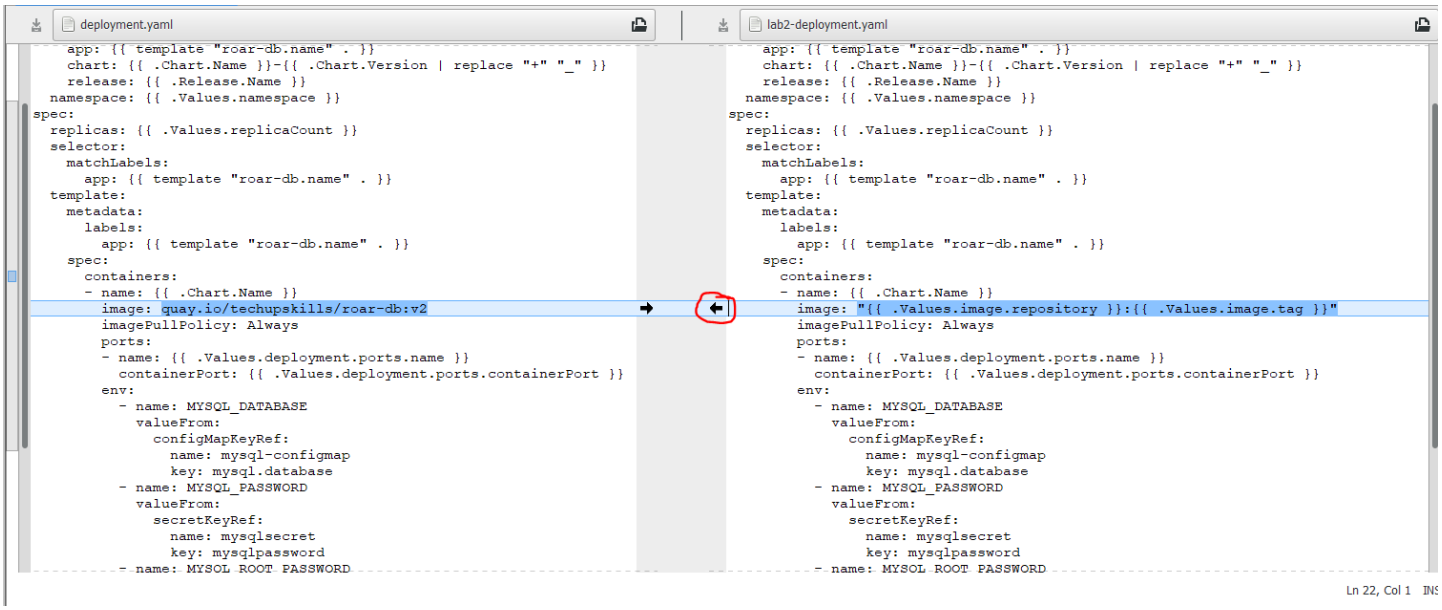
```
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

   Save your changes and exit the editor.

   Or:

```
$ meld charts/roar-db/templates/deployment.yaml ../extra/lab2-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.



3. 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:

   $ gedit charts/roar-db/values.yaml

   And add to the top of the file:

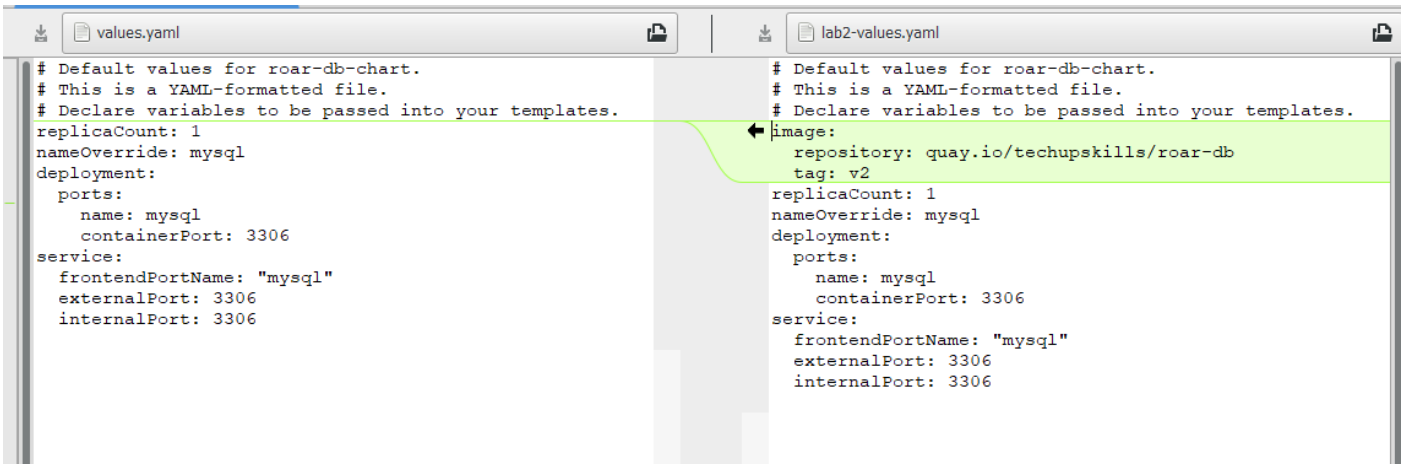
   ```
   image:
     repository: quay.io/techupskills/roar-db
     tag: v2
   ```

   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/lab2-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 a *kubectl port-forward <web app pod name> <nodeport>:8080 -n <namespace>*.

   To make things easier, there is a script that will do this for you. You can run it via

```
$ ~/beyond-k8s/extra/roar-port.sh <namespace> <nodeport>
```

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* .

   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. (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 ~/beyond-k8s/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
```

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

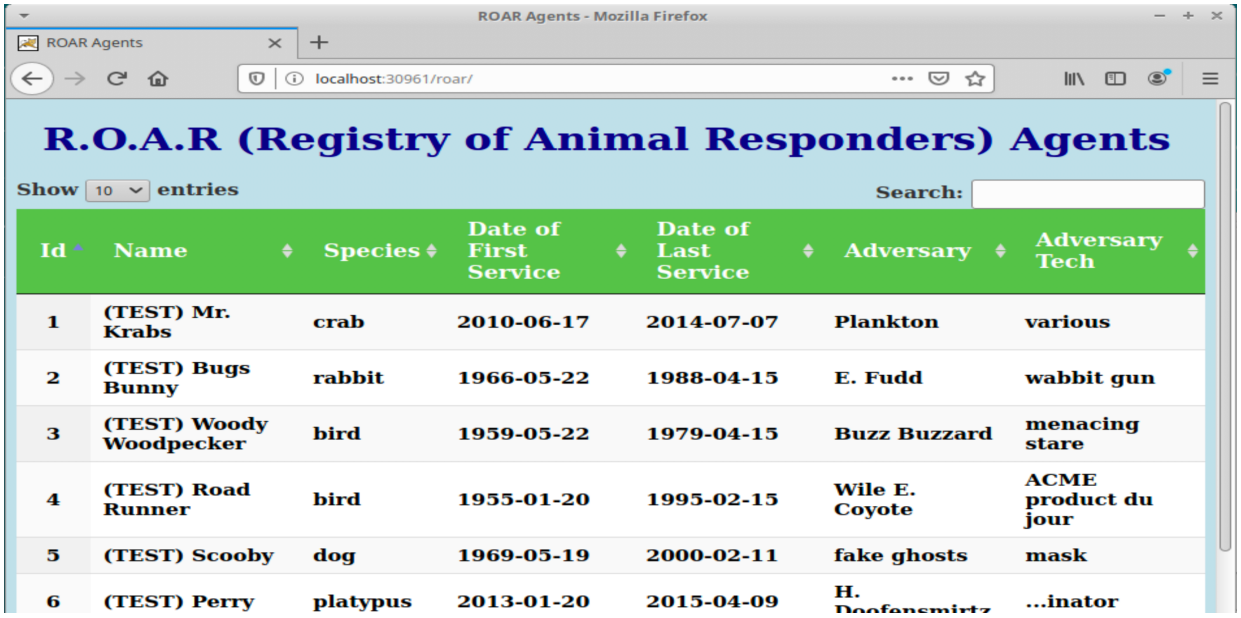
11. Get the nodeport for the new instance of the service.

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

12. **If you are not running in the VM**, use a new instance of the roar-port.sh command to pick up the new pod and do the port-forward.

```
$ ~/beyond-k8s/extra/roar-port.sh roar-helm <nodeport>
```

13. 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.)



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

```
$ Ctrl-C
```

END OF LAB

Lab 3 - 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.

```
$ cd ~/beyond-k8s/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 this system, 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 lab1 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. 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 used as the editor here. You may use a different one if you want.)

```
$ cat namespace.yaml

$ gedit 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.

```
$ gedit kustomization.yaml
```

The 2 lines are:

```
commonAnnotations:
  version: base
```

```
commonAnnotations:
  version: base

namespace: roar-kz

# List of resource files that kustomize reads, modifies
# and emits as a YAML string
resources:
- ./roar-db/deployment.yaml
- ./roar-db/service.yaml
- ./roar-db/secret.yaml
- ./roar-db/configmap.yaml
- ./roar-web/deployment.yaml
- ./roar-web/service.yaml
- namespace.yaml
```

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.

```
$ kz build | grep -a5 metadata

$ kz build | k 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.

```
$ k 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**, use a new instance of the roar-port.sh command to do the port-forward.

```
$ ~/beyond-k8s/extra/roar-port.sh roar-kz <nodeport>
```

12. **Open http://localhost:<nodeport>/roar in browser**

END OF LAB

Lab 4 - 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 ~/beyond-k8s/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
```

```
$ tree overlays (or ls -R overlays if you don't 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.)

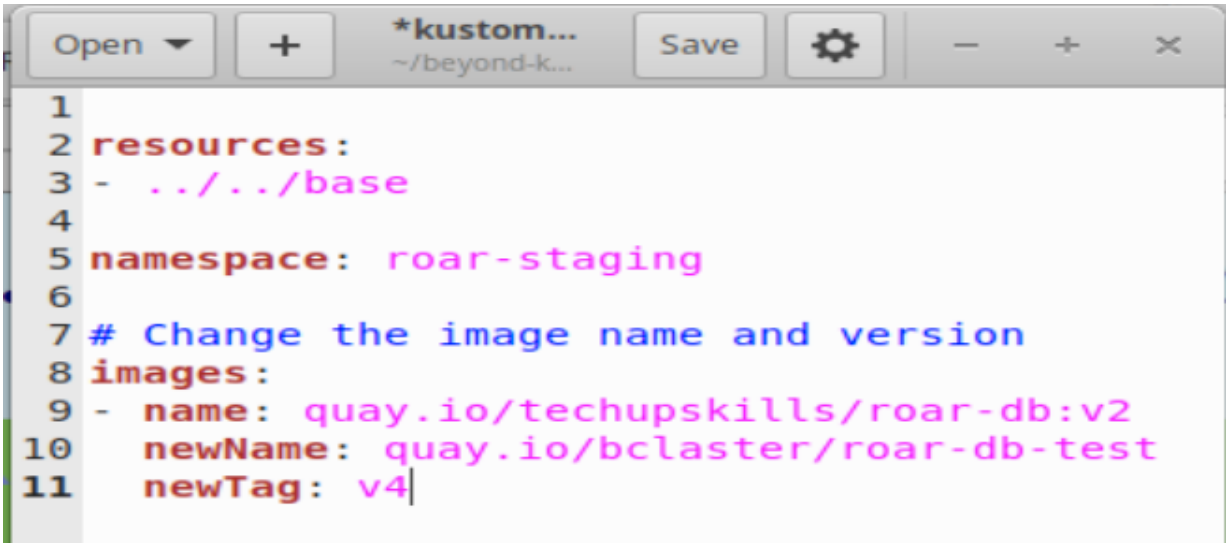
```
$ k create ns roar-staging
$ k create ns roar-production

$ 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.)

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

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



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

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

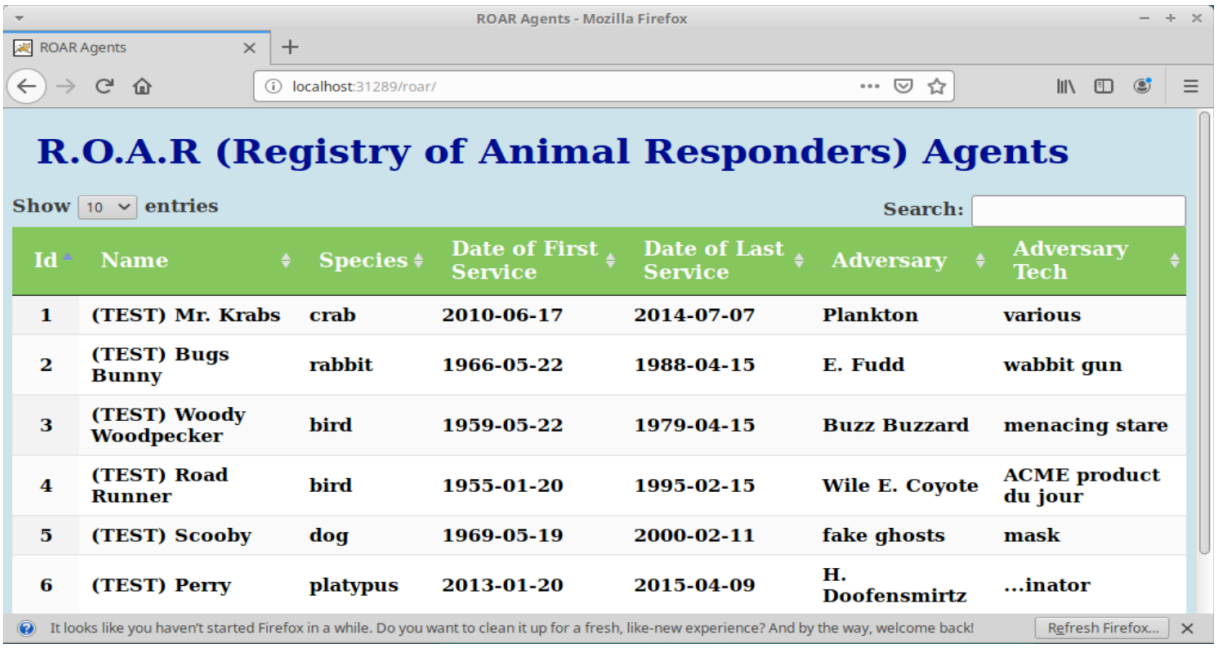
9. You can now find the nodeport for the service from roar-staging.

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

10. **If you are not running in the VM**, use the roar-port.sh command to pick up the new pod.

```
$ ~/beyond-k8s/extra/roar-port.sh roar-staging
<nodeport>
```

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



END OF LAB

Lab 5 – 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. 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 roar-istio
$ kubectl config current-context
$ kubectl config set-context <context from above> --
 namespace roar-istio
$ kubectl label namespace roar-istio 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 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 ~/beyond-k8s/istio

$ helm install roar-istio .
```

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 istio containers

```
$ kubectl logs <name of one of the pods> -c istio-init

$ kubectl logs <name of one of the pods> -c istio-proxy
```

6. 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
```

7. Let's send traffic to the pods and services with the "load-roar.sh" script. Running it gets the ingress host from the INGRESS_HOST environment variable, figures out the 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. To set this up, we first need to configure the INGRESS_HOST environment variable. This may need to be done in one of several different ways depending on how your cluster is setup.

If you are running in the VM, you can simply do

```
$ sudo minikube ip

$ export INGRESS_HOST=<ip returned from above>
```

If you are not running in the VM, you can try port-forwarding the ingress-service from the istio-system namespace, as in:

```
$ k get svc -n istio-system
   and get the nodeport associated with port 80 for
the istio-ingressgateway service

alternative command to get this:

$ k get svc -n istio-system | grep ingress | cut -d':' -f 3 | cut -d'/' -f1
```

```
    $ k port-forward -n istio-system svc/istio-
ingressgateway <nodeport value from above>:80

(example:

$ k get svc -n istio-system | grep ingress
istio-ingressgateway   LoadBalancer   10.96.236.76
<pending>
15021:31557/TCP,80:30674/TCP,443:32403/TCP,31400:31695/TCP,
15443:31059/TCP   12h

$ k port-forward -n istio-system svc/istio-ingressgateway
30674:80

)
```

```
    $ export INGRESS_HOST=127.0.0.1
```

If neither of these approaches work, consult https://istio.io/latest/docs/setup/getting-started/#determining-the-ingress-ip-and-ports to get more information on finding and setting the INGRESS_HOST value.

8. Now, you can execute the script to send traffic to the application and get the results back.

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

9. 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**.

10. Now, let's swap in another virtualservice spec that injects a delay of 5 seconds 50% 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
```

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

```
$ helm upgrade roar-istio .

$ ./load-roar.sh
```

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

12. Now, let's swap in another virtualservice spec that injects a 500 http error 50% 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
```

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

```
$ helm upgrade roar-istio .

$ ./load-roar.sh
```

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

14. Go ahead and reset the default namespace back to "default".

```
$ kubectl config set-context <context from earlier> --
    namespace default
```

END OF LAB

Lab 6: Working with Tekton Tasks

Purpose: In this lab, we'll see how to run and use a basic Tekton Task.

1. To run the examples in this set of labs, first change into the tekton directory for lab 6

```
$ cd ~/beyond-k8s/tekton/lab6
```

2. Before we do anything else, let's go ahead and open the Tekton dashboard. The dashboard service itself should have already been installed on your system via the setup script and should be running. So, all we need to do is a port-forward command and then open it up in a browser session. It is suggested to do this in a separate terminal session.

```
$ k -n tekton-pipelines port-forward svc/tekton-dashboard 9097:9097 &
```

Then, open a tab in the browser to: http://127.0.0.1:9097

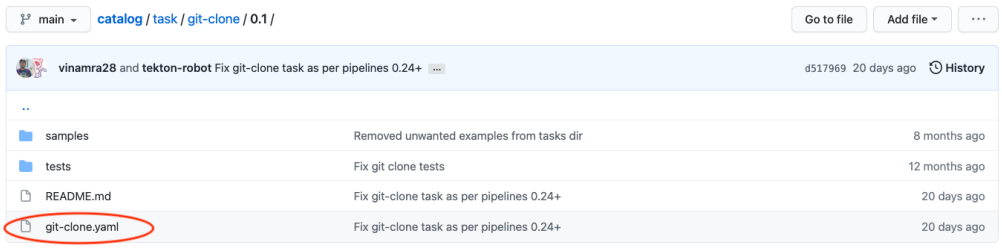
3. For this lab, we are going to see how to run a simple, predefined Task - git-clone. To start with, take a look at the actual Task as it is defined in the Tekton Catalog.

Go to the link for the main page of the task:

https://github.com/tektoncd/catalog/tree/main/task/git-clone/0.7

You can read about the task on there if you like. Note that is intended as a replacement for the former "Pipeline Resource" for a Git repository.

Click on the git-clone.yaml link in the list of files at the top to see the actual task definition.



(There's a lot of content in there, but if you skip past the possible parameters and environment settings, you can see most of the main functionality at the bottom.)

4. Create a new namespace for our tekton pieces.

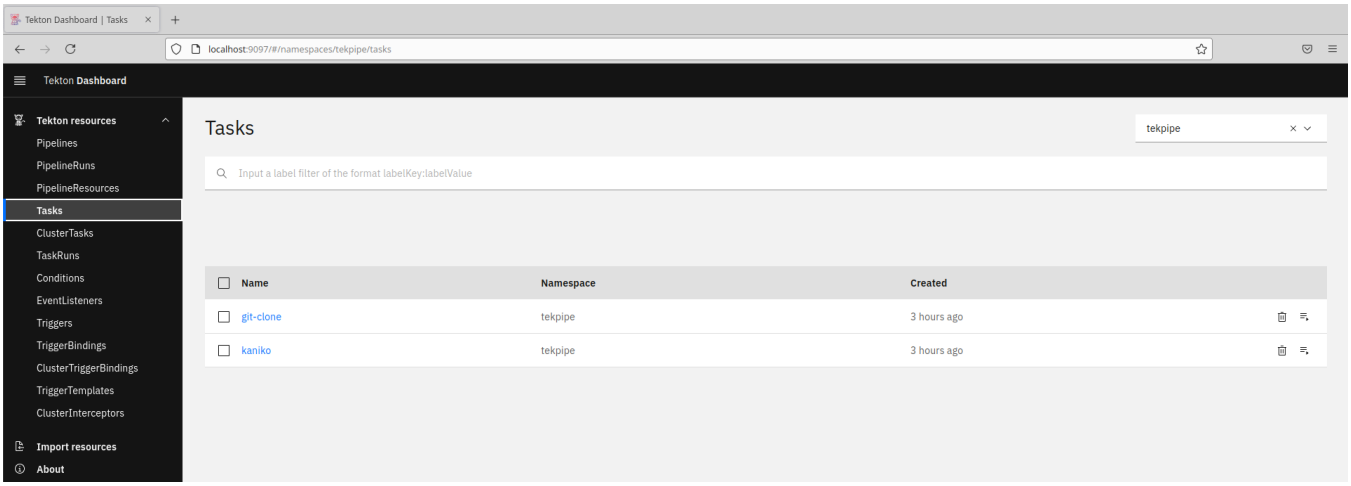
```
$ k create ns roar-tek
```

5. To run this task, we need it installed locally. We can do this easily via the "tkn hub" command. Execute the command below to install the git-clone Task in our tekpipe namespace. Then we'll go ahead and install the kaniko Task for building images that we'll need later as well.

```
$ tkn hub -n roar-tek install task git-clone
$ tkn hub -n roar-tek install task kaniko
```

6. You can now get a list of these from the command line. As well, you can look in the dashboard and see these items listed in the Tasks section (selected on the left-hand side). (Make sure to have the tekpipe namespace selected in the upper right.)

```
$ k get -n roar-tek tasks
```



7. In order to run this Task, we will need a persistent volume claim set up. There is already one defined in the tekton-pvc.yaml file in the current directory. Go ahead and apply it into the cluster.

```
$ k apply -n roar-tek -f tekton-pvc.yaml
```

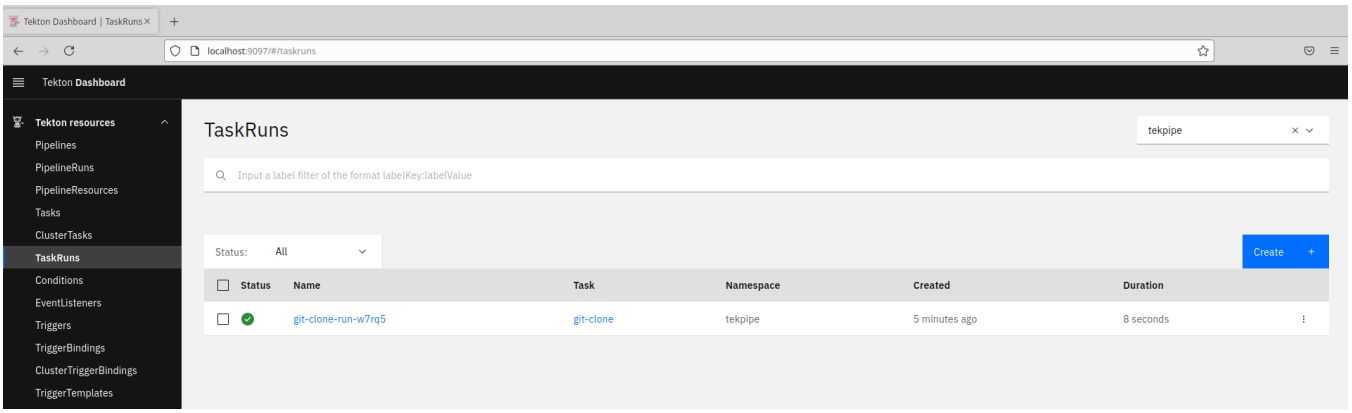
8. Now we can attempt to run our task via the **tkn** command line utility. The fairly long command is below. You can type this in if you want or you can simply run the shell script **tkn-git-clone.sh** that is in the directory. Execute this from a terminal session.

```
$ export TKN_EXE=<location of your tkn executable>

$ ./tkn-git-clone.sh
```

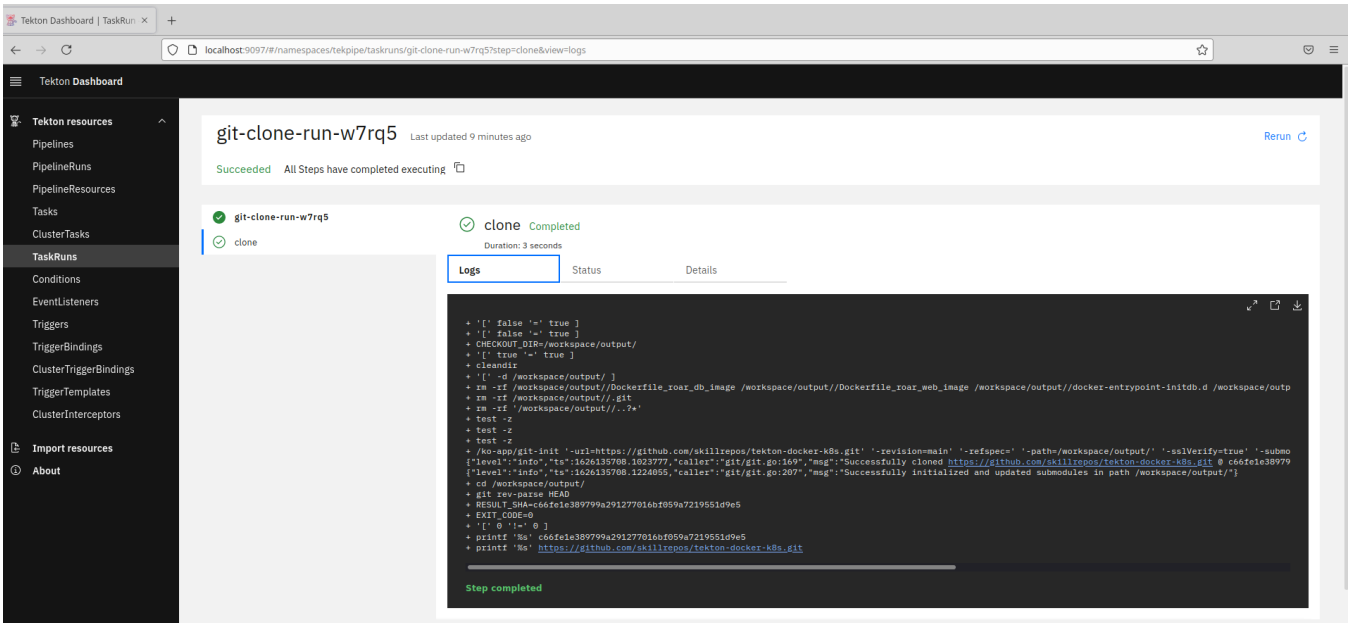
OR

```
$ tkn task start git-clone --namespace=roar-tek \
--param url=https://github.com/skillrepos/tekton-docker-k8s.git \
        --param revision=main \
        --param deleteExisting=true \
--workspace name=output,claimName=git-files-pvc \
        --use-param-defaults \
        --showlog
```

9. This will create a TaskRun object and run the pre-defined git-clone task. You can see it execute in the terminal window or you can go to the Dashboard and look at it there, under the TaskRuns section (on the left). Select the "tekpipe" namespace selected in the upper right box.



10. You can click on that TaskRun there to get more details, logs, and status. Click on the TaskRun and browse those tabs.



11. Now let's see how this same TaskRun can be done via a manifest file. In the same directory, take a look at the file **taskrun-git-clone.yaml**. Then you can use the "kubectl create" command (NOT apply) to create the TaskRun object and invoke it.

```
$ cat taskrun-git-clone.yaml

$ k create -n roar-tek -f taskrun-git-clone.yaml
```

12. From here, you can look at the new TaskRun and its details, logs, etc. in the dashboard just as you did the previous one.

Lab 7: Creating Tekton Pipelines

Purpose: In this lab, we'll see how to create a Tekton Pipeline to orchestrate our tasks and run it.

1. To work with the examples in this set of labs, first change into the directory for lab3.

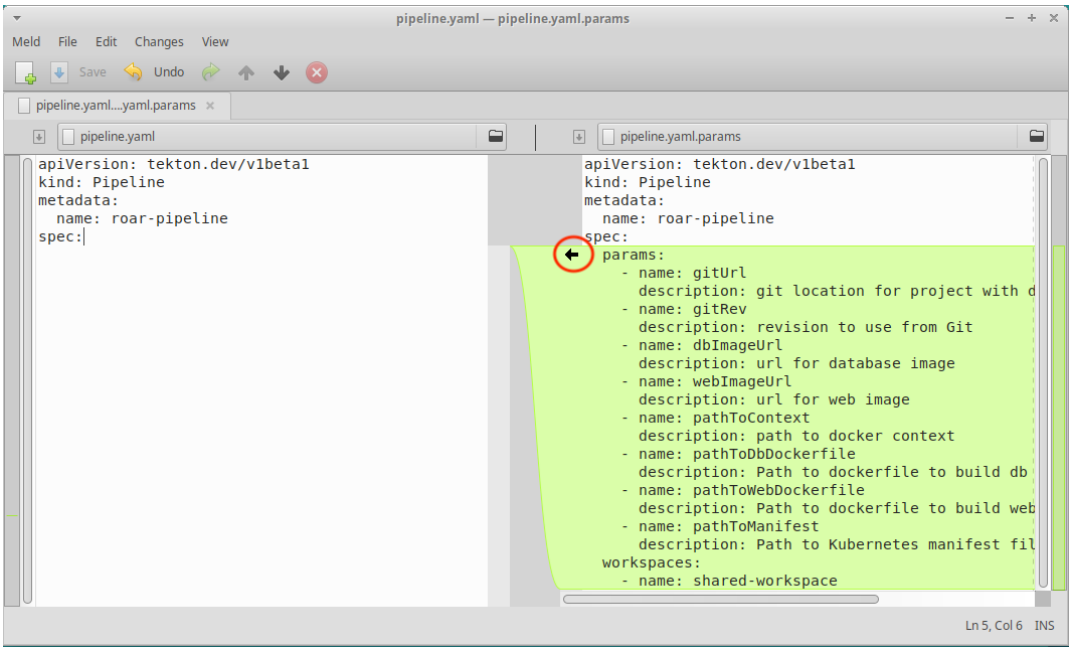
   ```
   $ cd ~/beyond-k8s/tekton/lab7
   ```

2. Because of the amount of code involved, we'll use the approach of constructing the pipeline spec via adding in a piece at a time via a diffing/merge tool. If you are running in the VM, you can use “meld”. Otherwise, you can use whatever diffing/merge tool you have installed. We'll start by adding in all of the parameters that the pipeline will need.

   These parameters include:
   a. The Git URL and revision (branch) where the content to build our images is stored.
   b. The locations for our updated database and webapp images after they are rebuilt.
   c. Paths to the Docker files for building the updated images.
   d. Path to the Kubernetes manifest to updated with the new images.

   ```
   $ meld pipeline.yaml pipeline.yaml.params
   ```

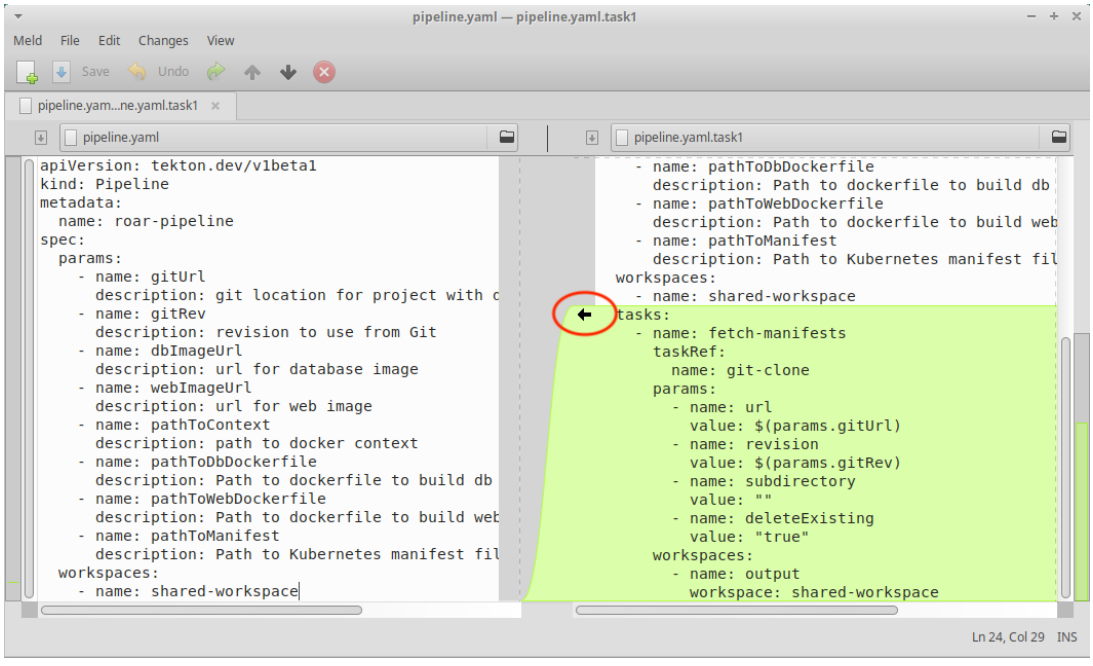
   Review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.



3. Next, we'll add in the call to the first task - fetch-manifests. Notice that the "taskRef" element tells us that this is really a call to the "git-clone" task that we saw in the first lab. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task1
```

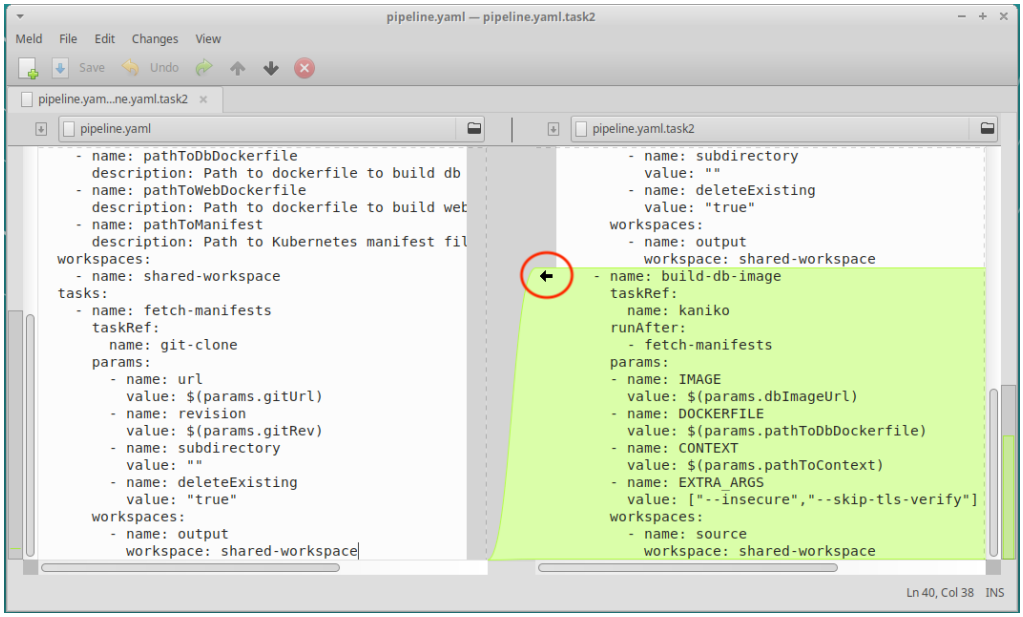
Review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.



4. Now comes the call to the second task - build-db-image. Notice that the "taskRef" element tells us that this is really a call to the "kaniko" task. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task2
```

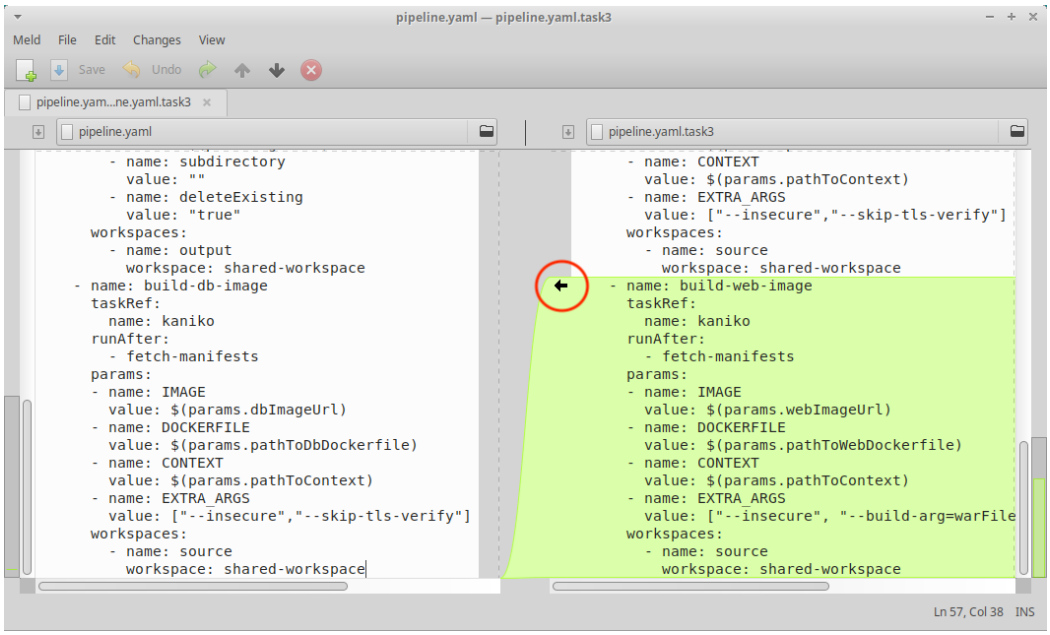
Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.



5. Add in the call to the third task - build-web-image. Notice that the "taskRef" element tells us that this is really a call to the "kaniko" task. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task3
```

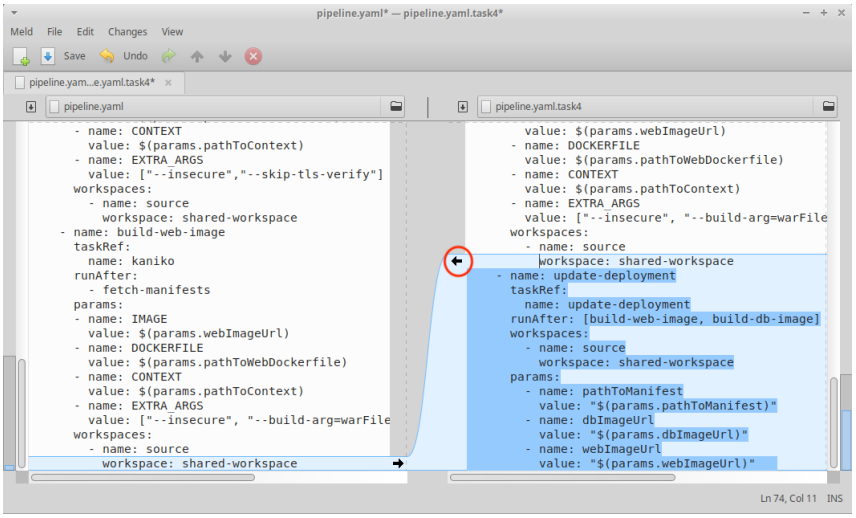
Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.



6. Add in the call to the fourth task - the one to update the deployment manifest with the updated image information. Notice that the "taskRef" element points to one of the tasks we created earlier. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task4
```

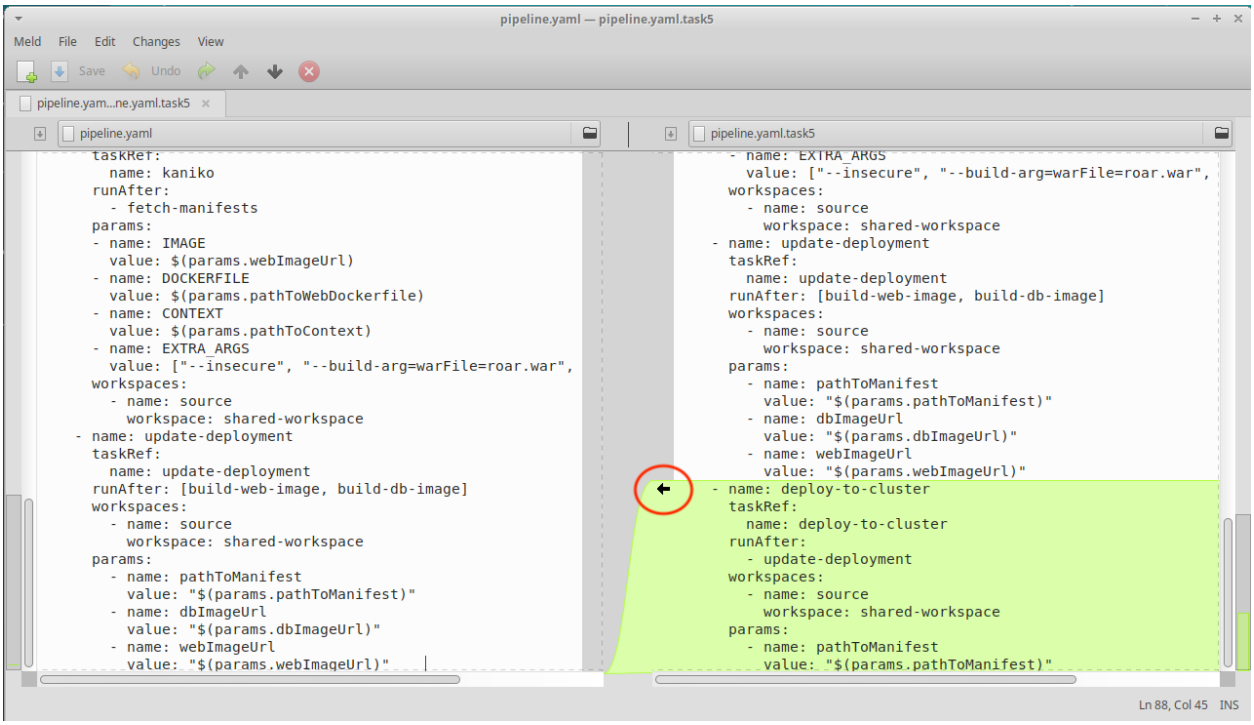
Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.



7. Finally, we'll add the call to the fifth task - the one to update the cluster based on our updated deployment manifest with the updated image information. Notice that the "taskRef" element points to one of the tasks we created earlier. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task5
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

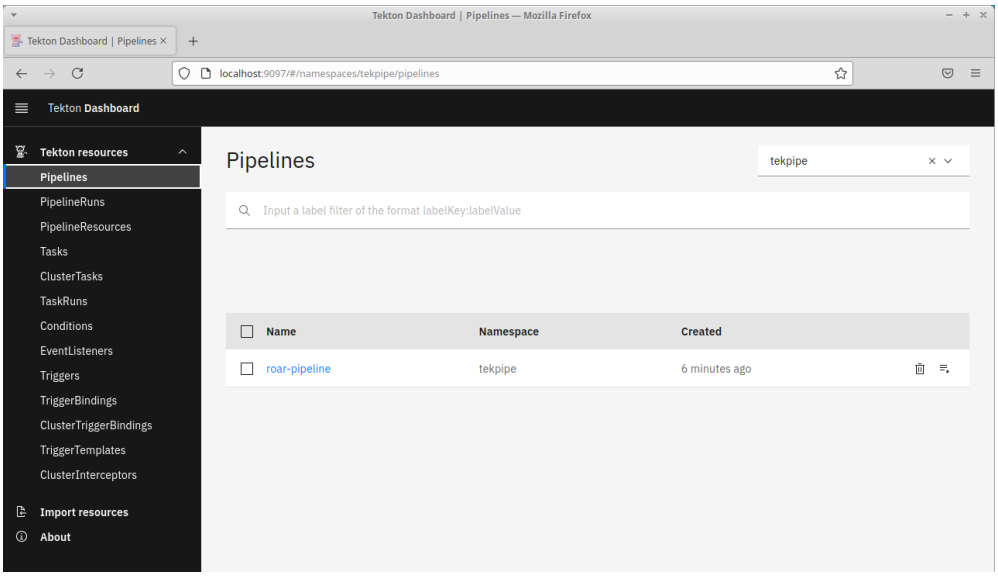


8. Our Pipeline spec is complete. But we need to install two other custom tasks that the pipeline will need. These are in the tasks.yaml file. You can look at them and then apply them so they will be available to the pipeline.

```
$ cat tasks.yaml

$ k apply -f tasks.yaml -n roar-tek
```

9. Go ahead and instantiate the pipeline in the cluster and then open up the Tekton dashboard to note the roar-pipeline object is present in the tekpipe namespace.

```
$ k apply -f pipeline.yaml -n roar-tek
```



10. Now that we have a pipeline, we need some way to run it. To do this, we need a PipelineRun object - similar to the TaskRun object we used in Lab 1. The PipelineRun needs to ultimately render a Pipeline object with the necessary parameters to pass into the Pipeline. There is already a spec for a PipelineRun object in this directory. Take a look at it via the command below. Note that it has a PipelineRef to the Pipeline we just created. And it specifies parameter values needed for the Pipeline to run.

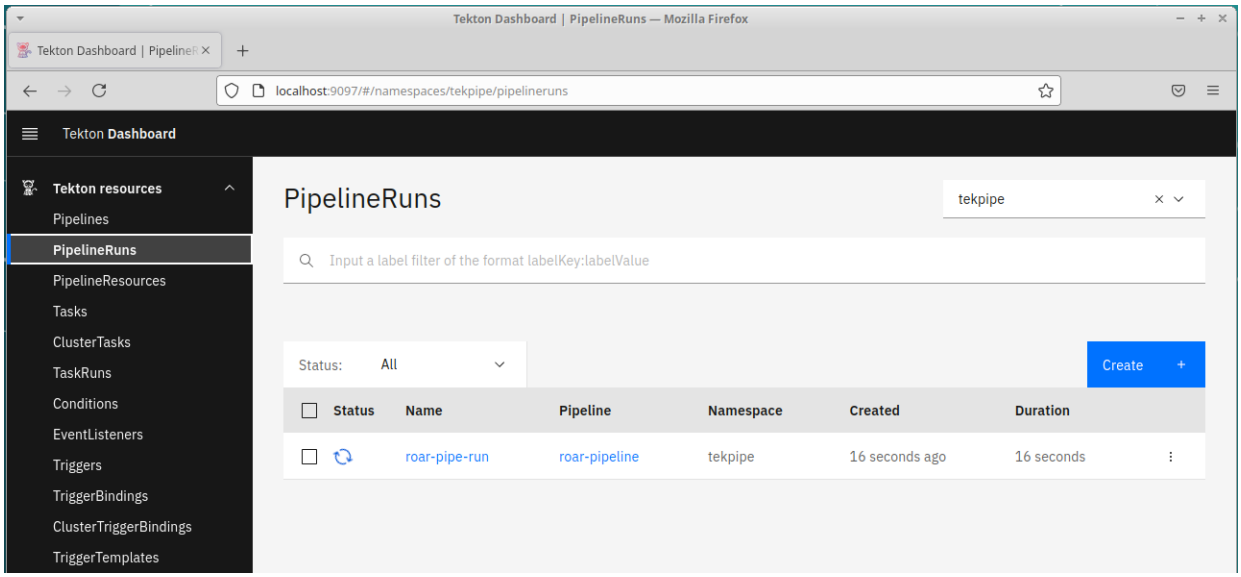
```
$ cat pipelinerun.yaml
```

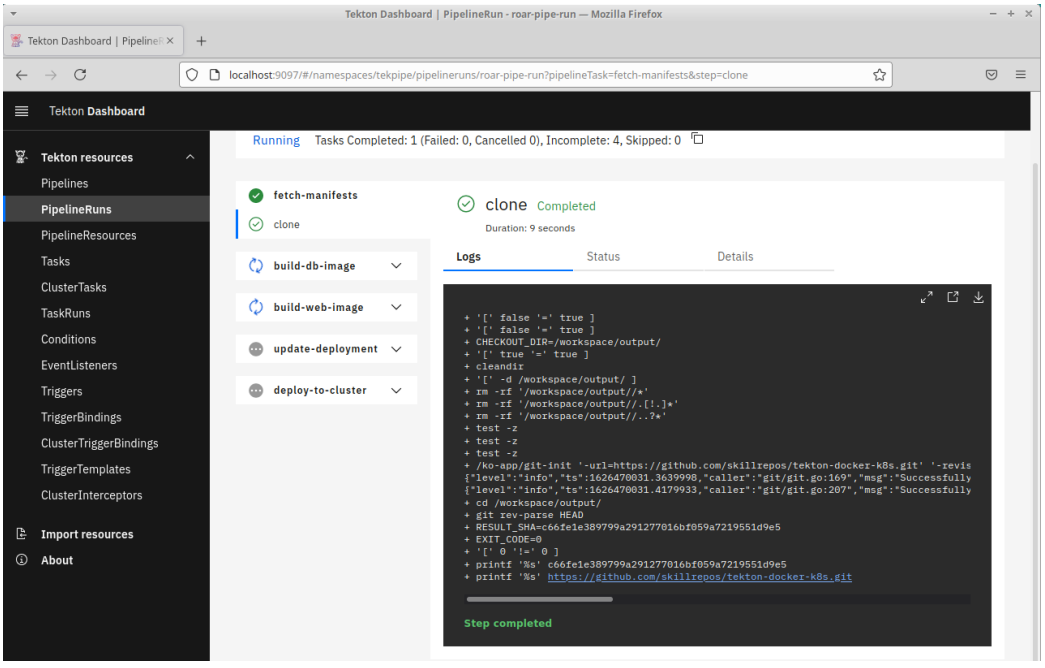
11. As well, we need a Service Account to run our Pipeline under as well as the corresponding Role and RoleBinding. In this directory, there is a file with the necessary specs in it - pipeline-rbac.yaml. Take a look at that file and then apply it to set up the needed rbac pieces for running the pipeline. Notice in the RBAC settings we are setting up the RoleBinding to be active in the target namespace for our app - roar.

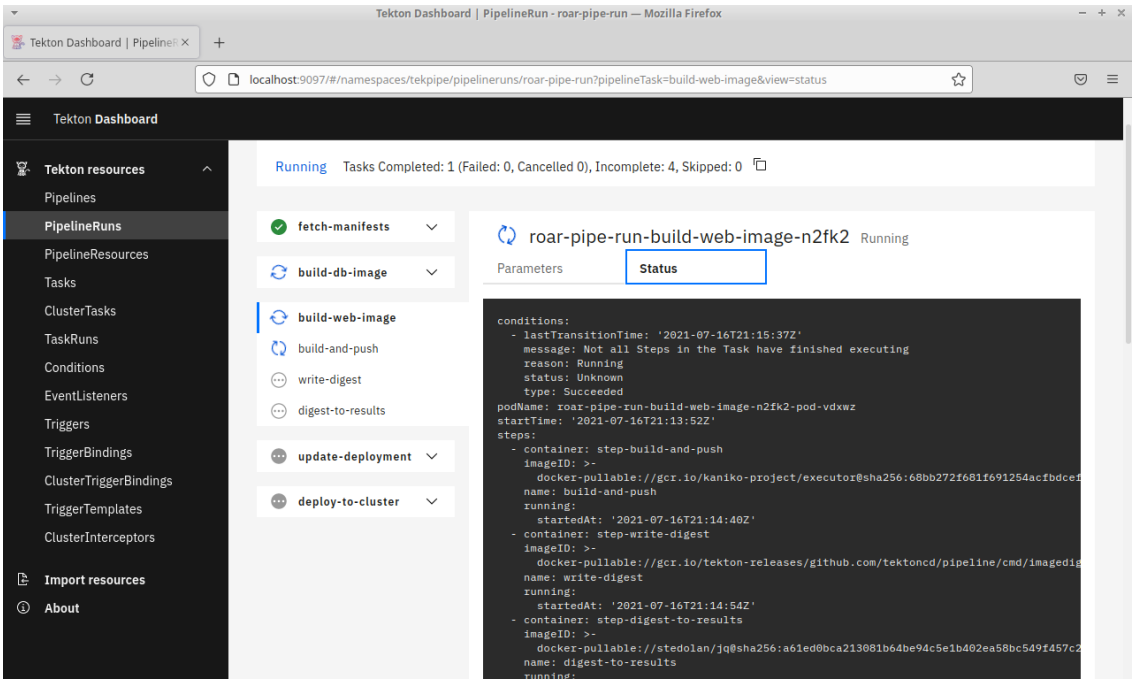
```
$ cat pipeline-rbac.yaml

$ k apply -f pipeline-rbac.yaml
```

12. Now apply the PipelineRun manifest. After you apply this , you can switch back to the dashboard and see the PipelineRun. You can also see the Logs and Status by clicking on those tabs. Note this will take quite a while to run.

```
$ k apply -n roar-tek -f pipelinerun.yaml
```







13. You can see the pods that are running the different tasks by looking in the namespace.

```
$ k get pods -n roar-tek
```

Lab 8: Getting Started with Argo CD

Note: For this lab, you will need to have a GitHub free account and a GitHub Personal Access Token (PAT). Follow the instructions at the link below to generate the token. Keep a copy of it somewhere so you can copy and paste it to use later in the lab

https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token

Purpose: *In this lab, we'll see how to add a Helm project to ArgoCD with automatic synching. And then we'll make a change to correct an issue and see the automatic sync in action.*

1. At this point, you should have argocd running in its own argocd namespace. You can see the various pieces that are running there by looking at it in Kubernetes. Find the port that corresponds to 443 for argocd-server.

```
$ k get all -n argocd
```

2. You will need to get the initial password for logging in which is stored in a secret. To get the initial password:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret -o
      jsonpath="{.data.password}" | base64 -d; echo
```

3. Now we need to make the argocd service accessible by port forwarding. Run the command below in another terminal.

```
$ kubectl port-forward -n argocd svc/argocd-server 31700:443
```

4. Next, we need to login to our argocd instance. Userid is “admin” and password is the one from step 2 above. (Answer “y” to any prompts about proceeding insecurely.)

```
$ argocd login localhost:31700
```

5. OPTIONAL: After you login , you can change the default password through the command line utility if you want. Note that you will need to login again with the new password afterwards.

```
$ argocd account update-password
```

6. Next we need to point argocd to the cluster we want to work with. If you only have one, it will default to that one. Otherwise, you'll need to add the cluster name you want.

```
$ kubectl config get-contexts

$ argocd cluster add (current one from list above)
```

You should see output like the following:

```
INFO[0000] ServiceAccount "argocd-manager" created in namespace
"kube-system"

INFO[0000] ClusterRole "argocd-manager-role" created
```

```
      INFO[0000] ClusterRoleBinding "argocd-manager-role-binding"
created

        Cluster 'https://10.0.2.15:8443' added
```

7. Let's set a couple of environment variables for convenience. The first one, ARGOCD_OPTS, tells the CLI which namespace ArgoCD is running in. You could skip this, but then you would have to supply that argument on every argocd command. The second variable tells ArgoCD where the API URL is for the target cluster.

```
$ export ARGOCD_OPTS='--port-forward-namespace argocd'
```

If you are running in the VM:

```
$ export CLUSTER_IP=https://$(sudo minikube ip):8443
```

If you are NOT running in the VM:

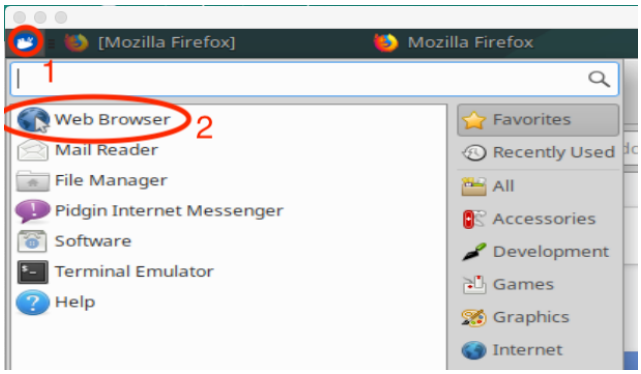
You'll need to get the cluster ip or you can try the setting below if appropriate.

```
$ export CLUSTER_IP=https://kubernetes.default.svc
```

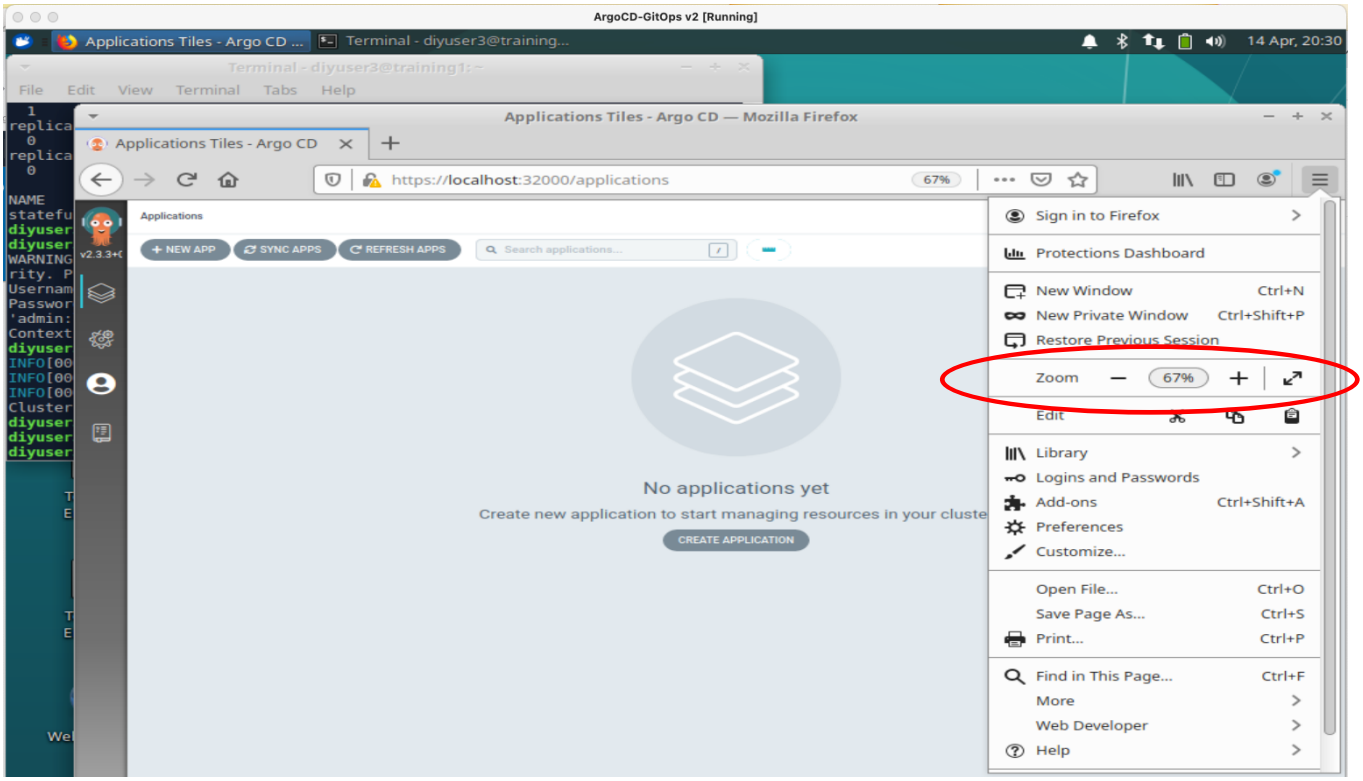
8. Before we go further, let's open up the browser interface to Argo CD. Do that by opening up a browser and then going to the url https://localhost:31700/ (Click on Advanced and Accept…). Then login with the same Username and Password used on the command line.

If running in the VM

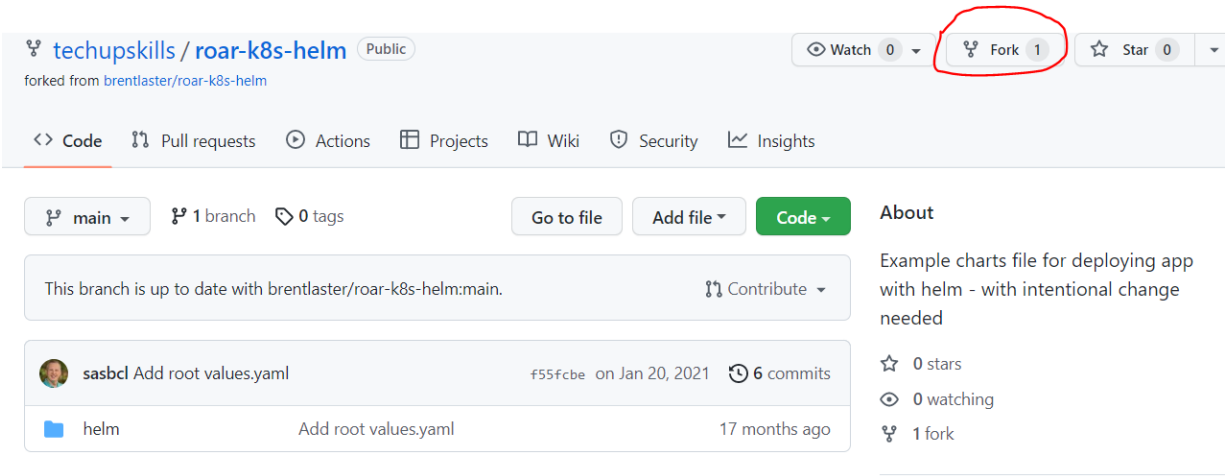
(The screenshots below show how to get a browser window (there's also a shortcut on the desktop).



9. You can scale out the screen by clicking on the settings and then set the Zoom value back to a smaller version.



10. Now, let's create a fork of a GitHub repository that you can use for this lab. Open up GitHub.com, log in, and go to https://github.com/techupskills/roar-k8s-helm. This is a helm repo for our sample app. Click on the "Fork" button in the upper right to make a copy in your namespace.



11. First, let's create a new project to work with instead of the default one. In a terminal, create a new project with the command below.

```
$ argocd proj create helmproj
```

12. Now we'll add our forked Git repo as a repo for our project. After running the command below, you should see a message that indicates the repository was added.

```
$ argocd repo add https://github.com/<your-github-userid>/roar-k8s-helm.git
```

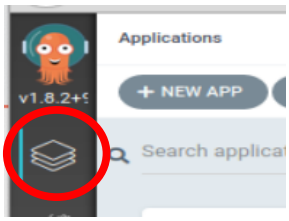
13. Also add it to our project as a source AND add the cluster as a destination to the project with the namespace at the end.

```
$ argocd proj add-source helmproj https://github.com/<your-github-
            userid>/roar-k8s-helm.git

$ argocd proj add-destination helmproj $CLUSTER_IP roar-argo
```

14. Create a new namespace for our project

```
$ k create ns roar-argo
```

15. Now, let's create a new app definition in the UI for our project. Go back to the browser tab with the UI. Click on the "stack of squares" on the left side, then select the "**+ NEW APP**" button. Set the values as follows: (Notice that along the way, ArgoCD recognizes there are Helm charts in the repo and automatically adds the "Helm" section at the bottom.)



GENERAL

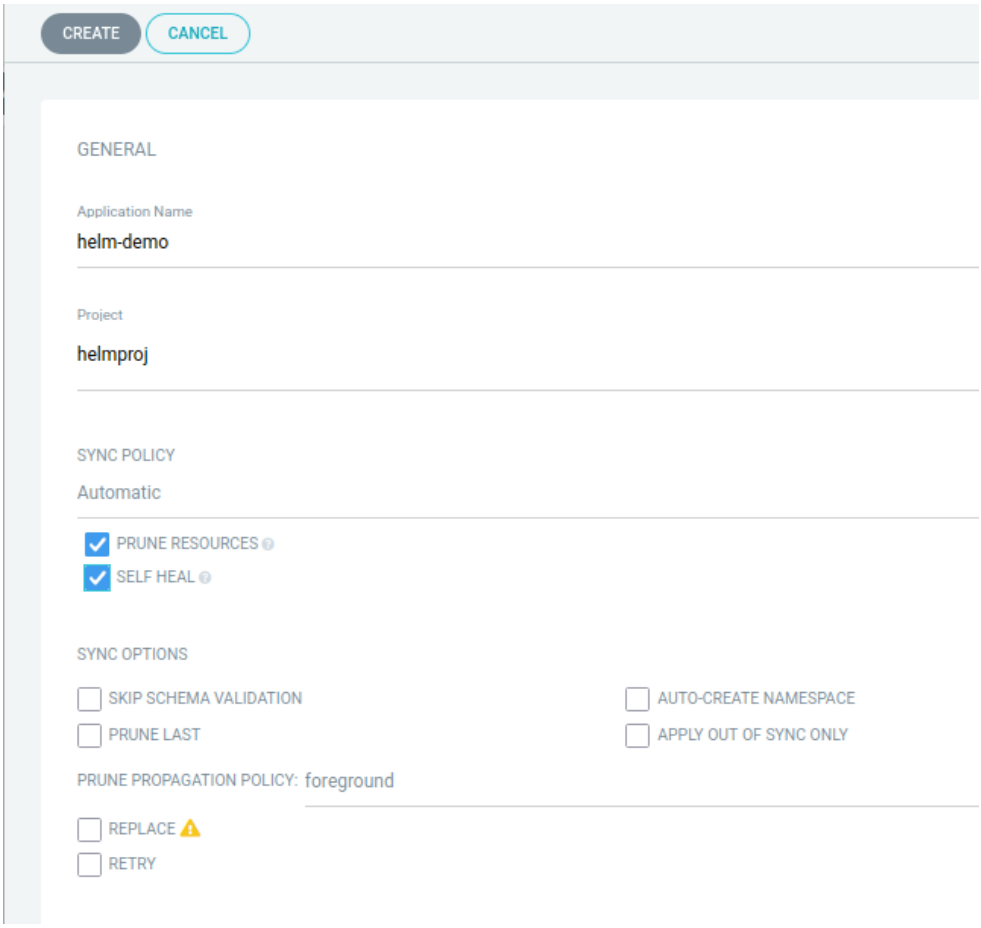
Application Name = helm-demo

Project = helmproj

SYNC POLICY = Automatic and check both boxes

SYNC OPTIONS - skip

PRUNE PROPAGATION POLICY - skip

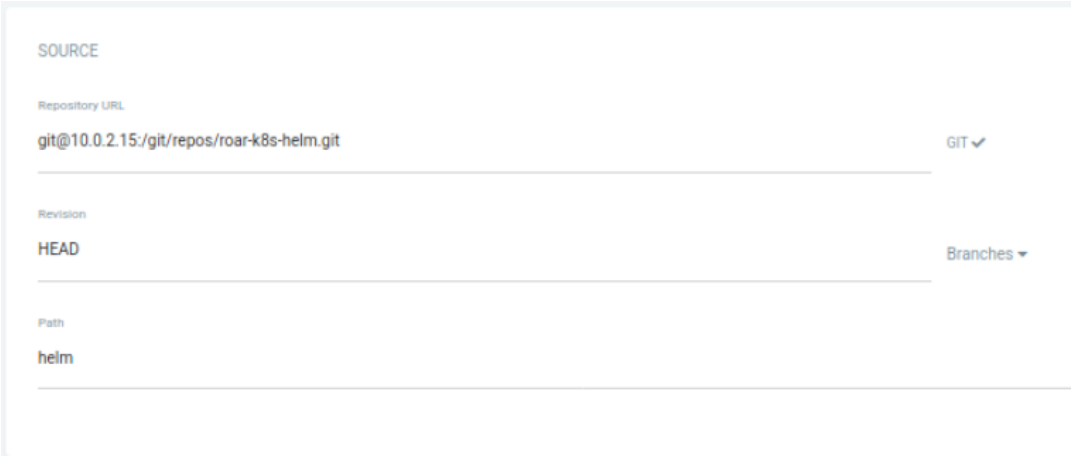


SOURCE

Repository URL = https://github.com/<your-github-userid>/roar-k8s-helm

Revision = HEAD

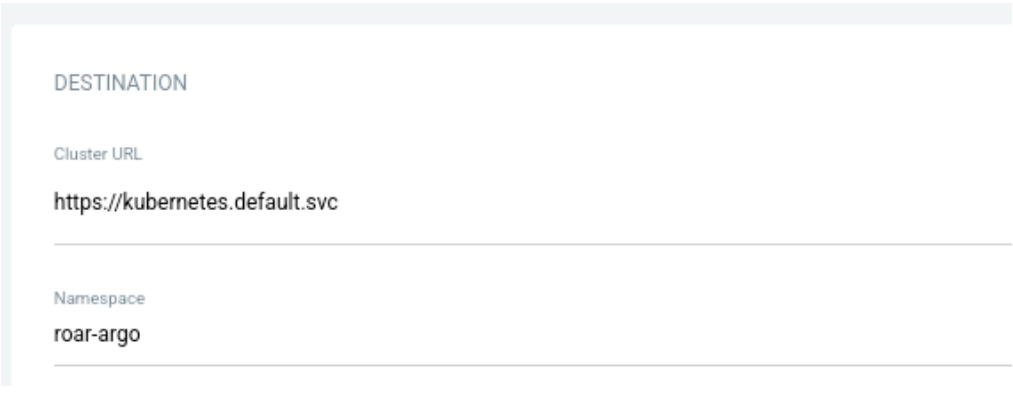
Path = helm



DESTINATION

Cluster URL = <same value used for CLUSTER_IP>

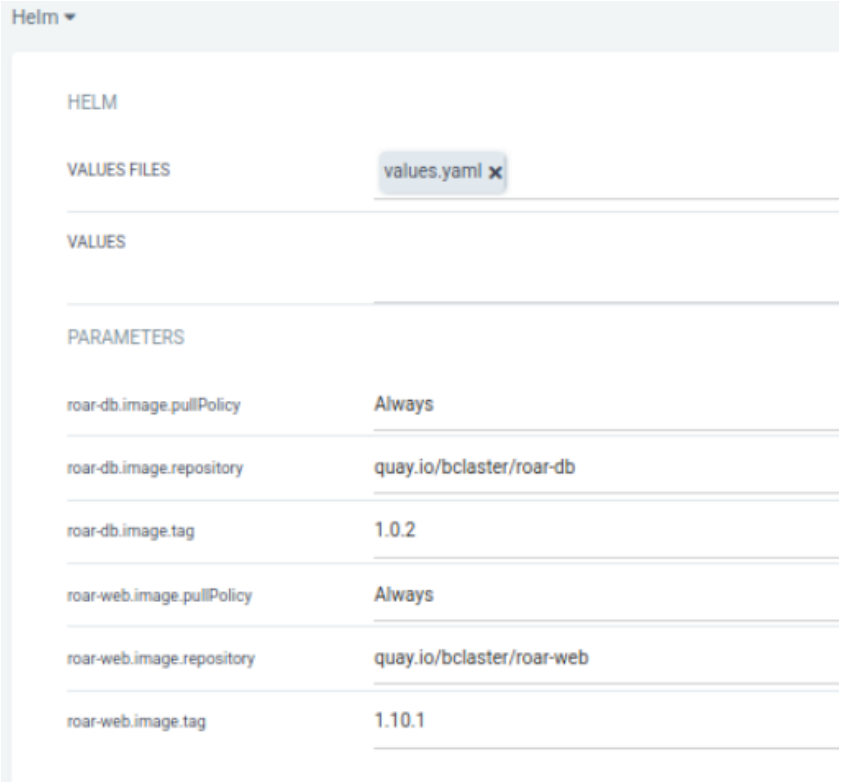
Namespace = roar-argo



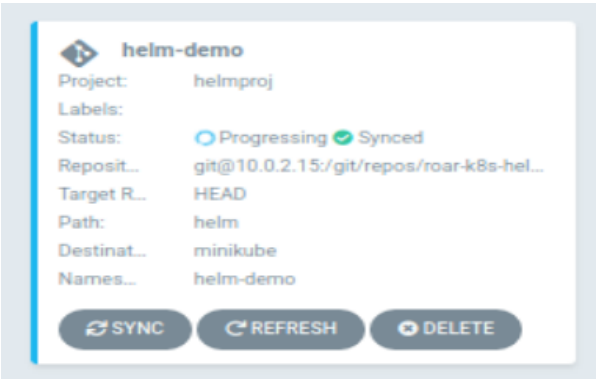
HELM

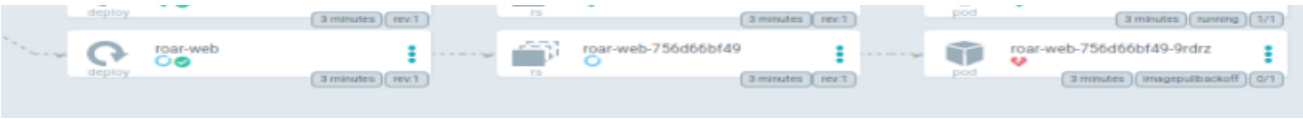
VALUES FILES = values.yaml

You can leave the rest of it as-is.

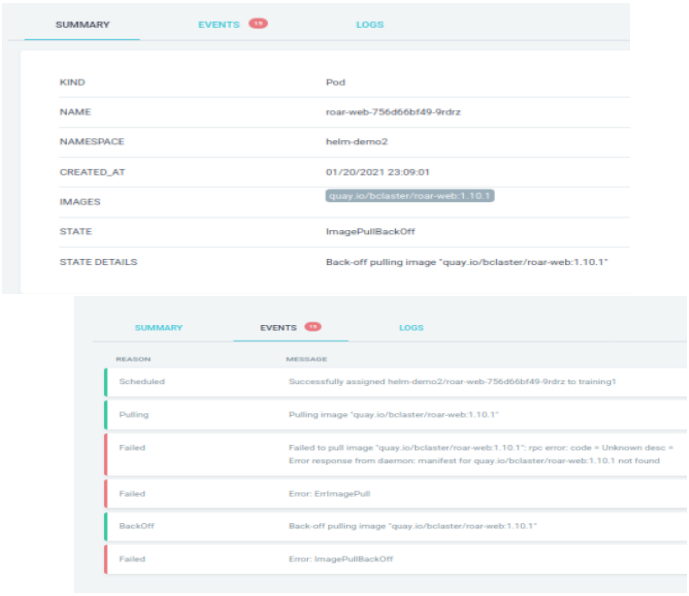


16. When you're done, click on the "**CREATE**" button at the top to save the app. Click on the new app you created. You should see the new app automatically going through the "Progressing" stage while it tries to get things starting up. However, there is a problem with the web pod as indicated by the red broken heart. (Click on the app to see that.)





17. Click on that item (the one with the red heart) and look at the SUMMARY and/or EVENTS tabs to see what's wrong.



18. The deployment can't get the requested disk image. The reason is that there is a typo here in the images semantic version tag: Instead of "1.10.1", it should be "1.0.1". Let's fix that with the following sequence. (All commands should be run in a terminal)

$ git clone https://github.com/<your-github-userid>/roar-k8s-helm

(Answer yes if prompted for confirmation to connect)

$ cd roar-k8s-helm/helm

$ <edit> values.yaml

Change the tag on line 12 from "1.10.1" to "1.0.1", save your changes and exit the editor.

```
# Default values for roar charts.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates
roar-db:
  image:
    repository: quay.io/bclaster/roar-db
    tag: 1.0.2
    pullPolicy: Always
roar-web:
  image:
    repository: quay.io/bclaster/roar-web
    tag: 1.0.1
    pullPolicy: Always
```

$ git diff (This is just to verify this is the only change that has been made in your repo.)

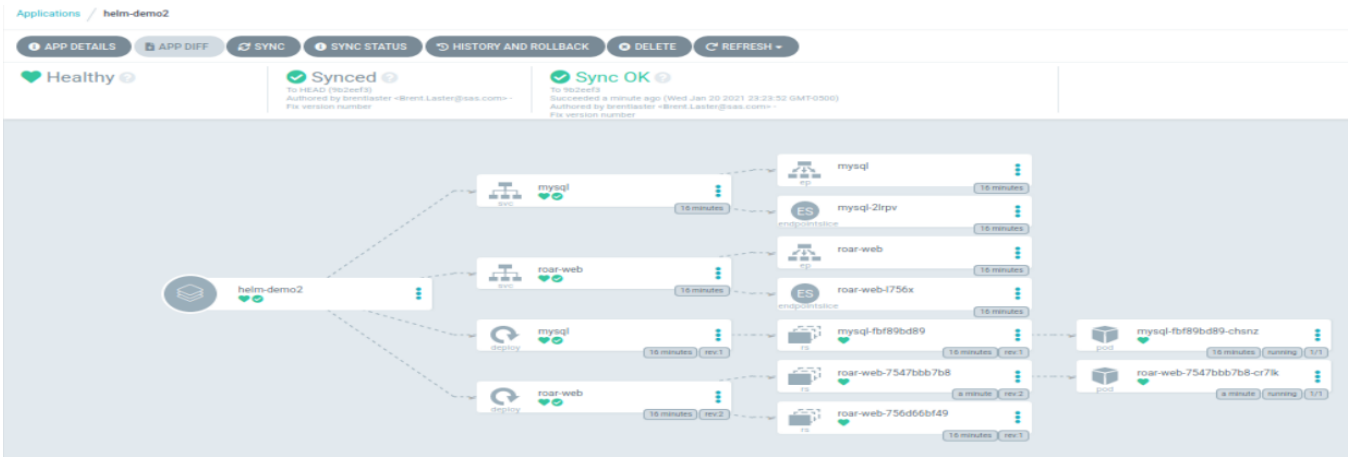
$ git commit -am "Fix version number"

$ git push

At this point, you'll be prompted for a sign-in/Private Access Token or password. Wherever it asks for a token or a password, you can just copy and paste in the token you generated in GitHub prior to this lab. An example dialog that may come up is shown below.

19. After this, you can go back to the main app page. After about 3 minutes or so (be patient), the automated sync setting will kick in and the stage will change to "Progressing" and eventually should all go green.

    (Since the auto sync interval in ArgoCD is so long, you may just want to let this run while we go through the next section.)



20. **(Optional)** If you want, you can find the service node port and use it to look at the running app as we've done before.

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 ~/beyond-k8s/op
```

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

```
$ k create ns roar-op
```

3. First, we want to deploy our CRD into the cluster. 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/roarapp_crd.yaml
```

4. Go ahead and deploy the CRD and verify that it is in there. (You can ignore the deprecation warning.)

```
$ k create -f crds/roarapp_crd.yaml

$ k get crd
```

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

```
$ k 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.

```
$ k create -n roar-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.

```
$ k get pods -n roar-op
```

9. 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
```

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/op-new/blob/main/controllers/roarapp_controller.go if you're interested. The Reconcile function starts around line 56. NOTE: This is not intended to represent coding best practices – just a quick and simple (and contrived) example.

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

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

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

```
$ k get pods -n op
```

13. 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.

```
$ k describe -n op pod <example roarapp pod name> | grep IP

http://<IP address>:8080/roar/
```

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

```
$ k get RoarApp -n op

$ k describe -n op RoarApp
```

15. 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=<editor of your choice>

$ 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