GitOps with ArgoCD

Continuous Delivery in Kubernetes

Class Labs: Revision 1.12 - 08/08/21

Brent Laster

Important Note: Prior to starting with this document, you should have the ova file for the class (the virtual machine already loaded into Virtual Box and have ensured that it is startable. See the setup doc **gitops-setup.pdf** in http://github.com/skilldocs/gitops for instructions and other things to be aware of. You should also have a GitHub account.

Note: *It is suggested to go ahead and do step 1 of lab 1 after you start the VM as it can take a few minutes to complete.*

Lab 1: Getting Started with Argo CD

Purpose: *In this lab, we'll start working with Argo CD by creating an app that represents our manifests and deploying that to the cluster.*

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

```
$ sudo minikube start --vm-driver=none --addons=registry --kubernetes-version=v1.19.0
```

2. Once your cluster starts up, you will have argocd running in its own argocd namespace. You can see the various pieces that are running there by looking at it in Kubernetes.

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

3. Note if you don't have an argocd namespace, try running these commands and check again.

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'

k get svc -n argocd | grep argocd-server
```

4. Next, we need to login to our argocd instance. Our instance has been patched to run as a LoadBalancer service (instead of ClusterIP) using port 32000. Login to the running instance. The userid and password

will be provided in the class. You can just answer y to the question that comes after the warning that pops up.

```
$ argocd login localhost:32000 (or port > 30000 from step 3)
```

5. Next we need to point argocd to the cluster we want to work with. Since we only have one cluster on this machine, it would default to that one automatically. But we'll go ahead and add it so we can see how its done.

```
$ argocd cluster add minikube
```

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
```

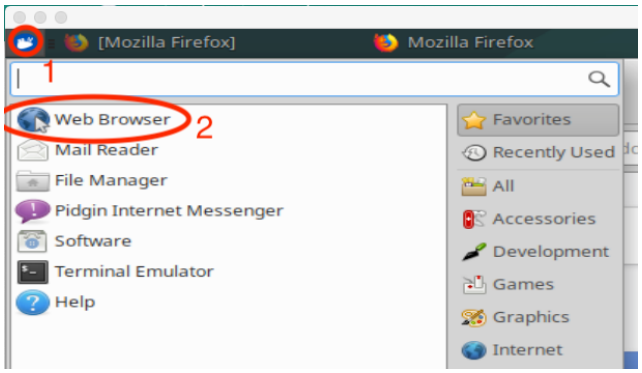
6. Let's set a couple of environment variables for convenience. The first one, ARGOCD_OPTS, tells the CLI which namespace ArgoCD is running in. You could skip this, but then you would have to supply that argument on every argocd command. The second variable tells ArgoCD where the API URL is for the target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

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

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

7. 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:32000/ (Click on Advanced and Accept…). Then login with the same Username and Password used on the command line.

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



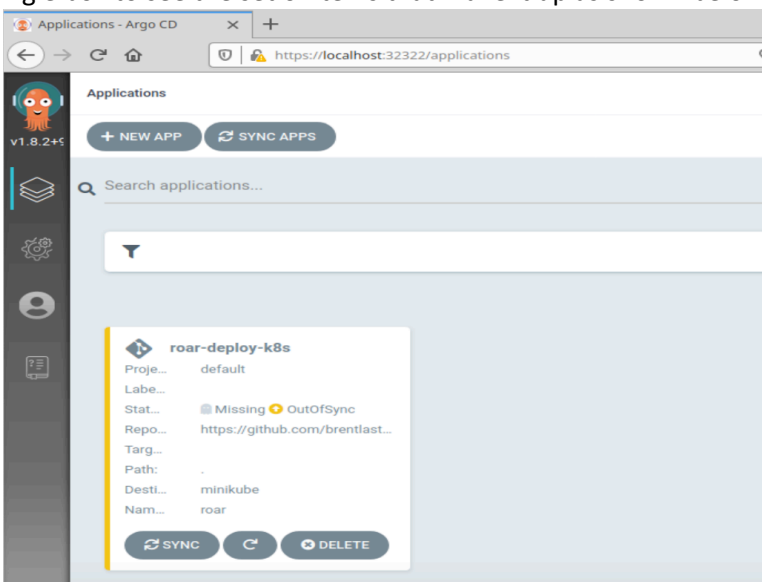
8. Now, we can create an "app" in Argo CD - essentially a way to manage the collection of Kubernetes manifests for services, deployments, etc.

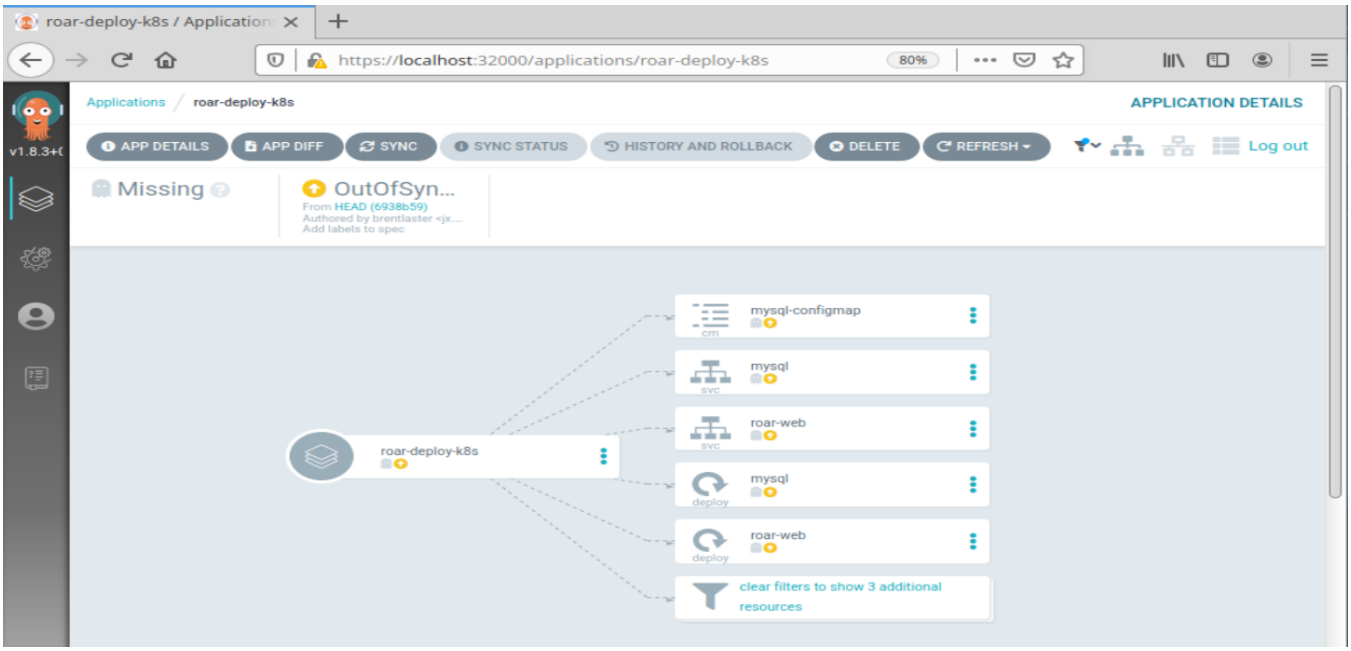
We'll use an existing set of manifests in GitHub for the ROAR webapp and database pieces. We'll also supply the destination cluster and namespace.

Back in the terminal, enter the command below. (Note: Make sure you've done this in a terminal where the preceding environment variables are set (from step 5).). *If you encounter any errors running this, make sure you have all 3 lines in the command and there are spaces as shown in the command string.*

```
$ argocd app create roar-deploy-k8s --repo
https://github.com/brentlaster/roar-deploy-k8s --path . --dest-server
$CLUSTER_IP --dest-namespace roar
```

9. You should get a message like "application 'roar-deploy-k8s' created". And if you go back to the browser, you'll be able to see a representation for the app and the set of objects making it up. Click on the large, single box to see the set of items that make it up as shown below.

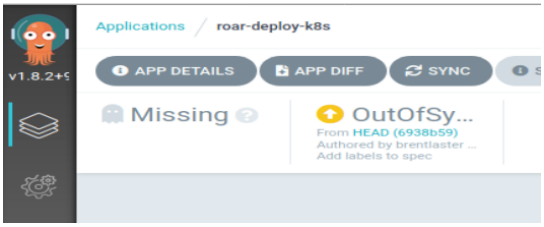


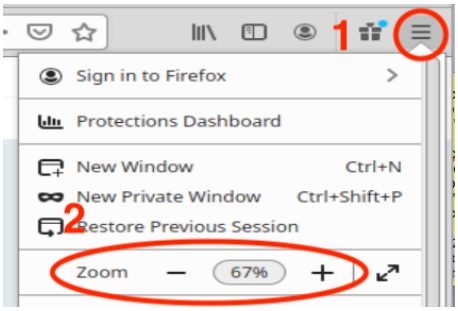


10. Back in the terminal, run the command below to see the current state of the app you just created.

```
$ argocd app list
```

The output should show you a number of fields with info about the app. Notice that the STATUS is currently "OutOfSync" and HEALTH is currently "Missing". If you look in the browser you'll see these same indicators near the top of the screen. (You may need to zoom out in the browser - see right screenshot below.)





11. The reason for the OutOfSync and Missing statuses is that while we've told Argo CD about the app, we haven't yet deployed it to the target cluster and namespace. Let's take a more detailed look at what will be deployed with our app. From the command line:

```
$ argocd app get roar-deploy-k8s
```

Notice the set of KIND objects in the list and other information.

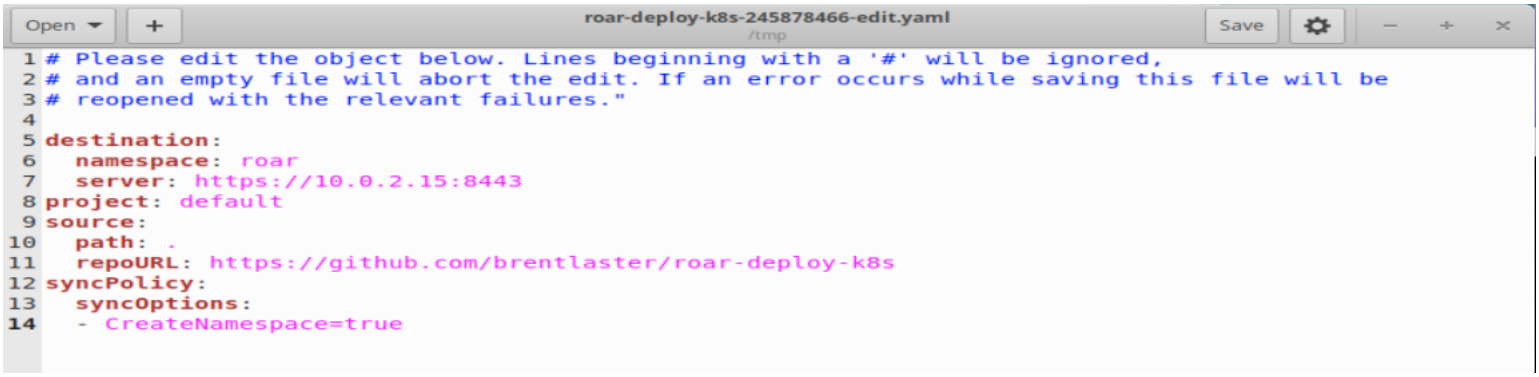
12. Before we sync our app, let's make one edit to it to automatically create the namespace for us.

Whether we use the command line or the UI to create the app, what we really get is a yaml file that is the “manifest” for the app. We can edit that to make changes. Edit the app and make the changes in the editor to add the yaml to automatically create the namespace.

```
$ export EDITOR=gedit
$ argocd app edit roar-deploy-k8s
```

Make the last 3 lines in the file equal to the set below - pay attention to the spacing and speillng - see the screenshot as a reference.

```
syncPolicy:
  syncOptions:
  - CreateNamespace=true
```

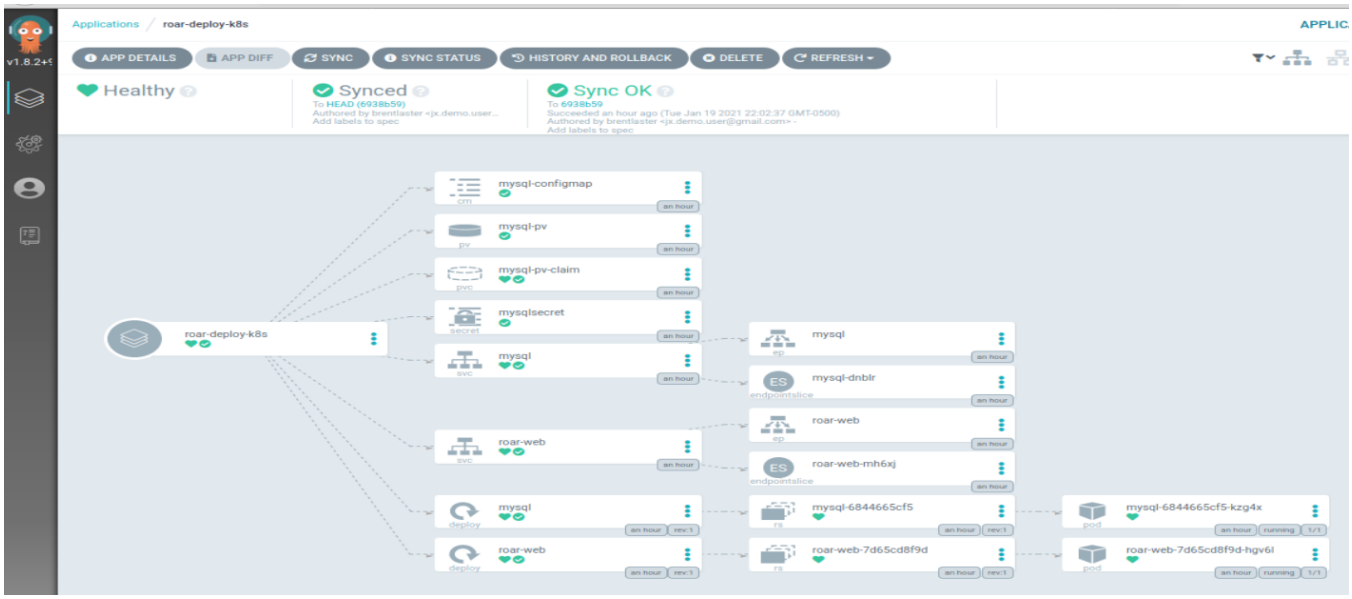


When you are done, save your changes and exit the editor. (You can ignore the Gtk warning message from the editor.)

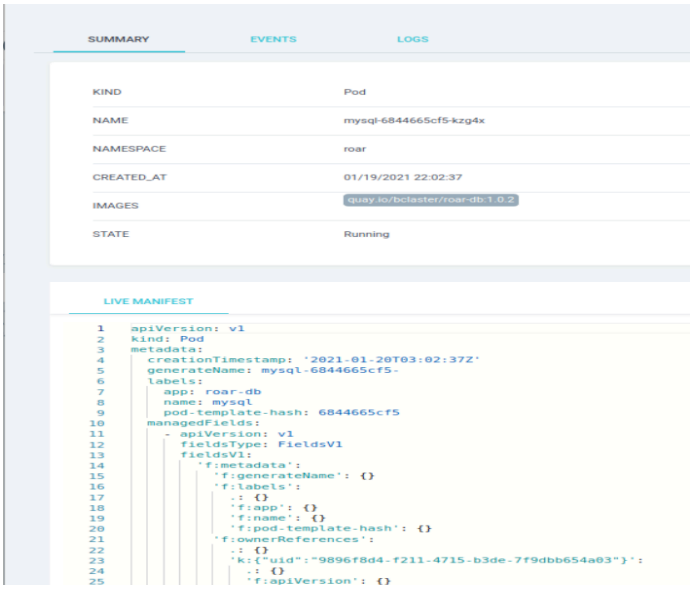
13. While ArgoCD can do automatic syncs to the target cluster and namespace, it defaults to manual syncs. So let's go ahead and tell it to sync now from the command line. Enter the command below.

```
$ argocd app sync roar-deploy-k8s
```

14. You’ll see updated status for the various kinds scroll by. And if you look back in the browser, you'll see that the sync has completed and things are starting up. (You may need to click on the “clear filters” items to see all of these.)



Optional: You can click on either of the pod objects on the far right (with the cube icons) to see information about the object as its running in the cluster. The SUMMARY tab shows the live manifest of the object (spec + status). The EVENTS tab shows the Kubernetes events that have occurred, and the LOGS tab shows the logs from the application in the pod.



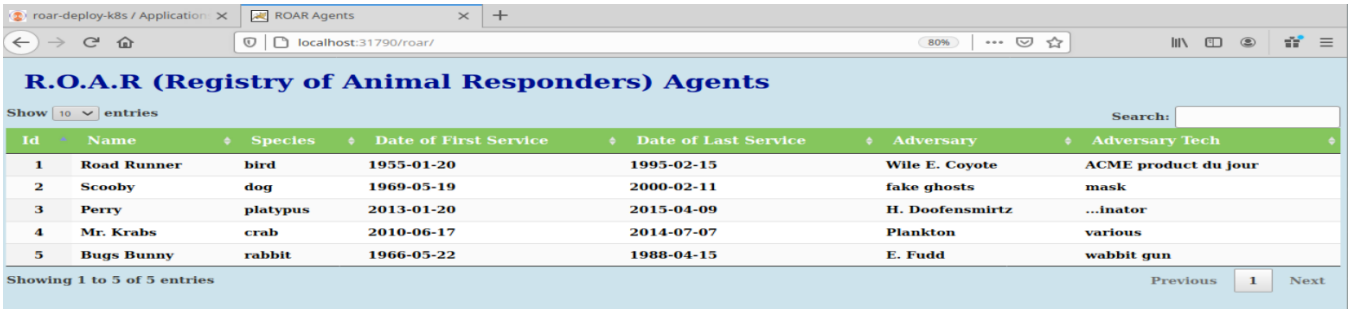
Optional: You can also see our app running in the cluster. From a command line, find the node port for the application and then open the URL shown below in a browser.

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

Look for the port number that is between “8089:” and “/TCP”. Plug that into the URL below.

http://localhost:<port number>/roar/

You should see the app running similar to this screenshot.

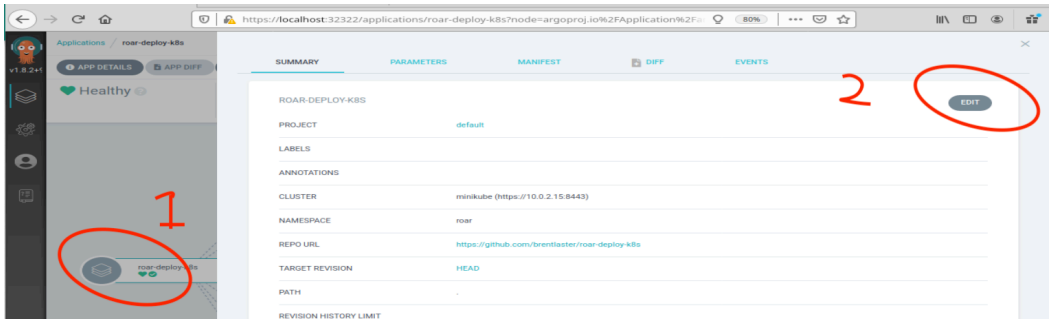


Lab 2: Dealing with Failed Deployments and Rollback

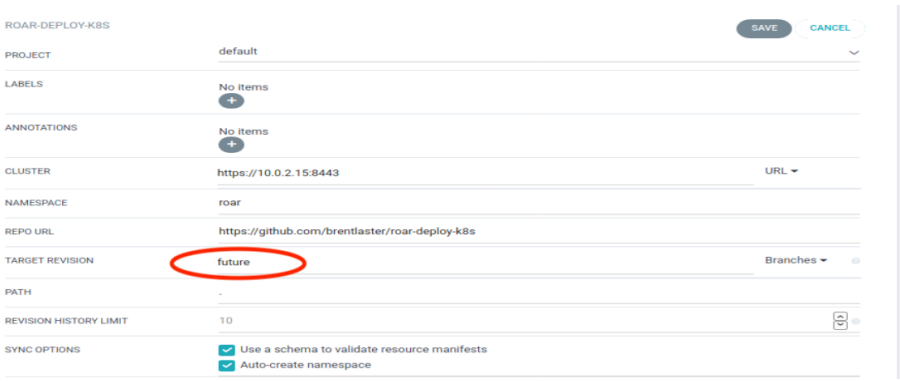
Purpose: *In this lab, we'll see what a failed deployment looks like and how to roll it back.*

1. Let's see what happens when a deployment fails. First, we'll edit the app to change the Target Revision field.

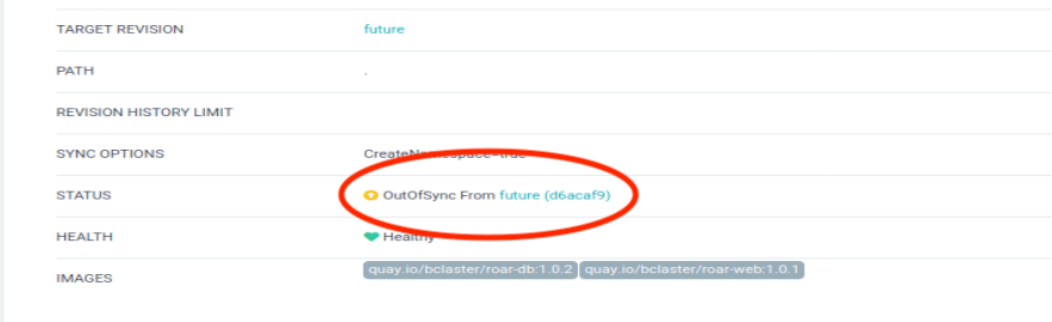
   In the browser tab with ArgoCD, click on the main app icon on the far left, then click the EDIT button in the upper right.



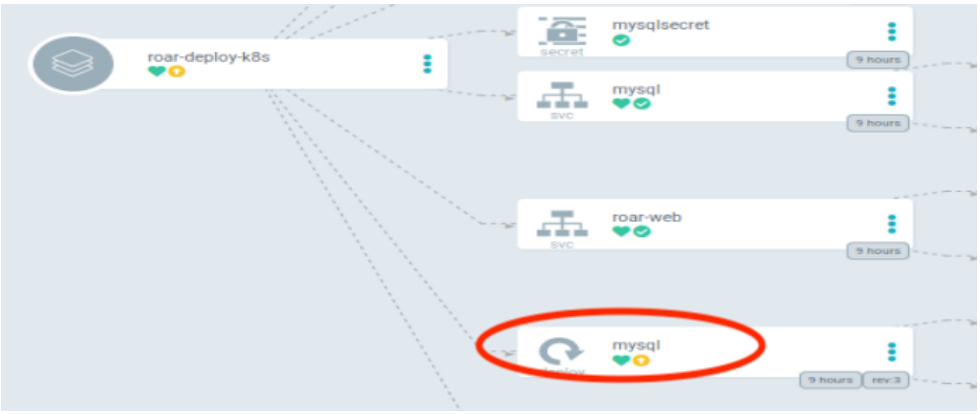
2. On the summary screen, edit the TARGET REVISION to change it from empty or "HEAD" to "future". Then click the SAVE button in the upper right corner to save the changes.



3. You'll see the STATUS field immediately change to "OutOfSync". Click the X in the upper left corner to close the parameters screen.

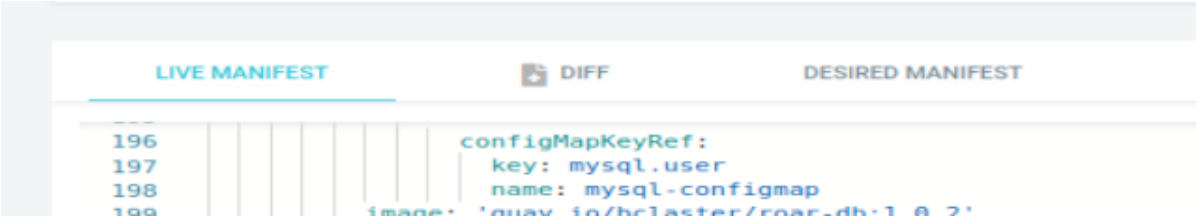


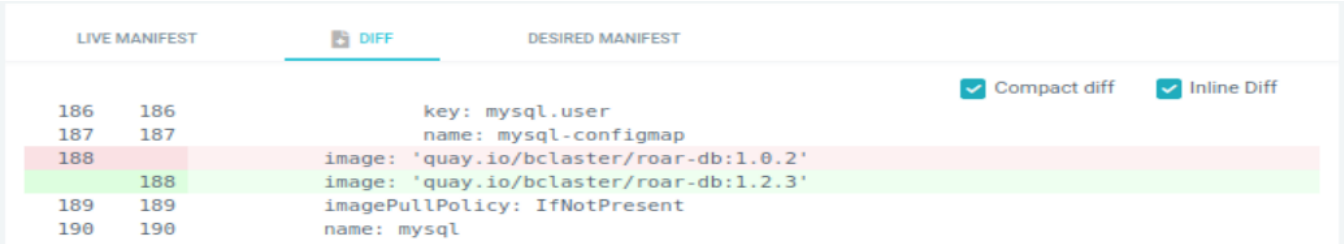
4. Now you can see which element is out of sync - the mysql deployment.



5. Click on that element and then on the **SUMMARY** screen, select the **DIFF** tab. In the DIFF tab on the upper right, click on both boxes for Compact diff and Inline Diff. You'll then be able to see the lines that are different.

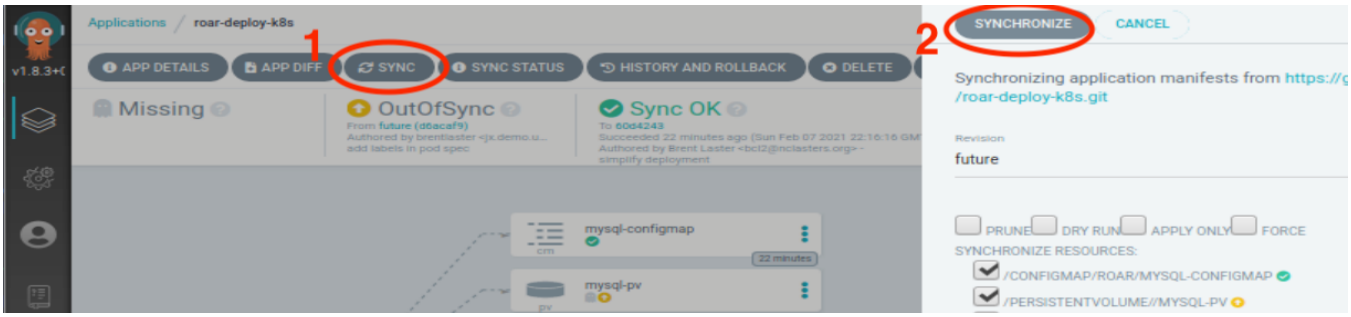
You can also see the manifest that is active in the cluster versus the one that is ready to be deployed by clicking on the LIVE MANIFEST vs DESIRED MANIFEST tabs, respectively.



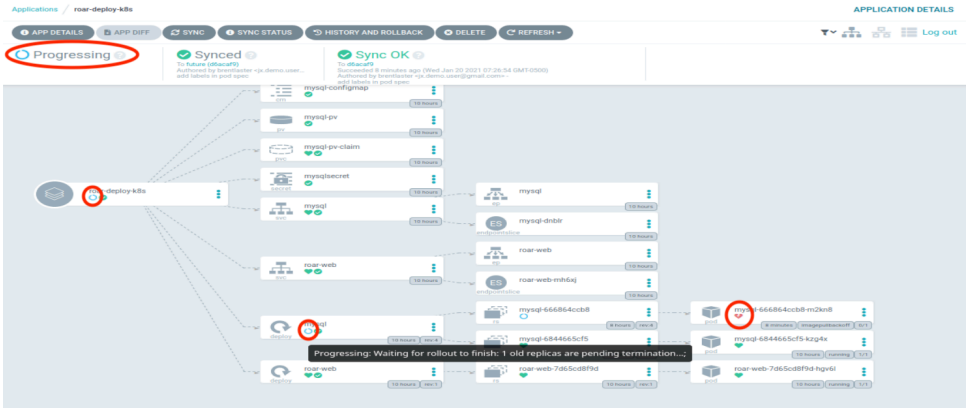


Close the screen by clicking on the "X" in the upper right corner when done.

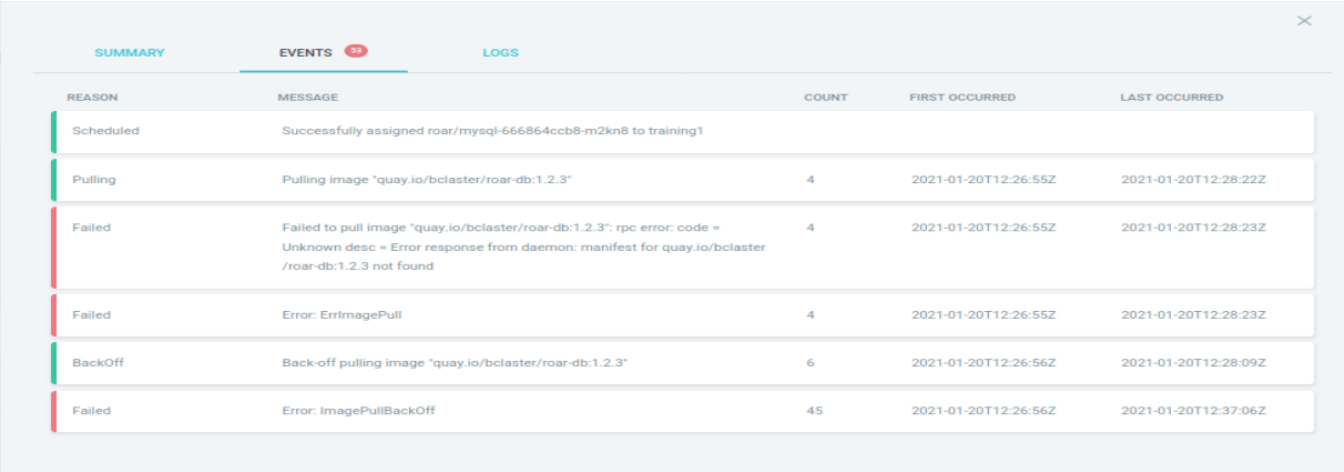
6. Manually synchronize the desired state to the cluster by clicking on the "**SYNC**" button on the top of the screen. When the popup appears, leave everything as it is and click on the "**SYNCHRONIZE**" button.



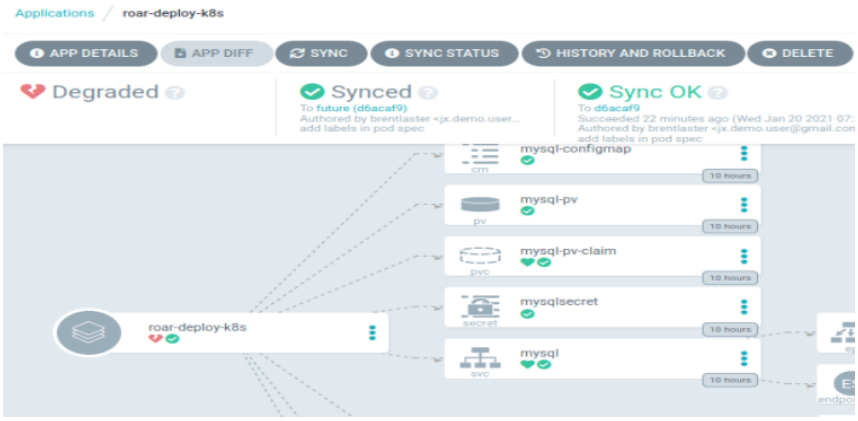
7. After the sync is attempted, you'll see that the app is in a "Progressing" state as it tries to reconcile the desired deployment with the live one. You'll also notice that there is a pod that is not ready yet, indicated by the broken red heart icon. (See circled sections in the second screenshot below.)



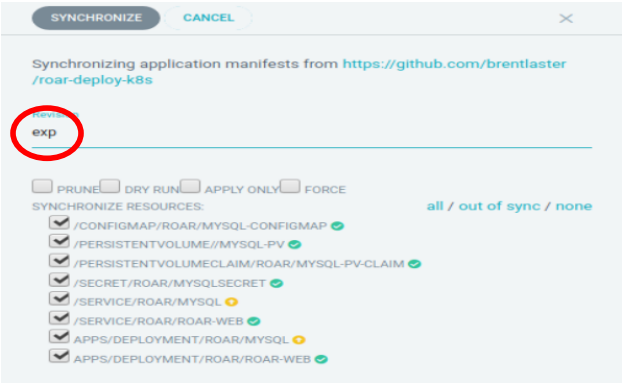
8. If you click on the un-synced pod element (the one with the broken red heart), you can see what is going on by looking at the EVENTS tab. After you are done, close that screen via the "X" in the upper right.



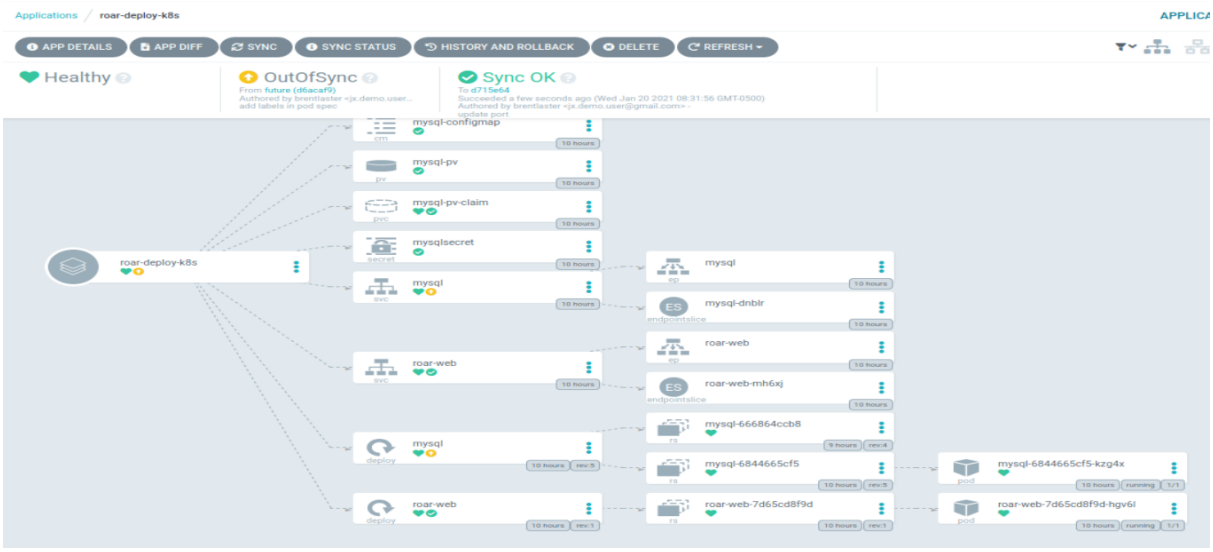
9. Eventually, the app status will switch to an overall status of “DEGRADED”. (You don’t need to wait for this.) Note that your original deployment is still synched and running though.



10. Let’s try another branch. Click the “**SYNC**” button at the top. Then in the pop-up dialog (the one for **SYNCHRONIZE**) change the “Revision” field to be “exp”. This represents an experimental branch on our project. After making that change, click on the “SYNCHRONIZE” button.



11. After this, you’ll be able to see that a couple of items still show up as “Out Of Sync” (yellow circle with arrows). This is because the version in the app is “future” and the version synched to the cluster is from the “exp” branch.



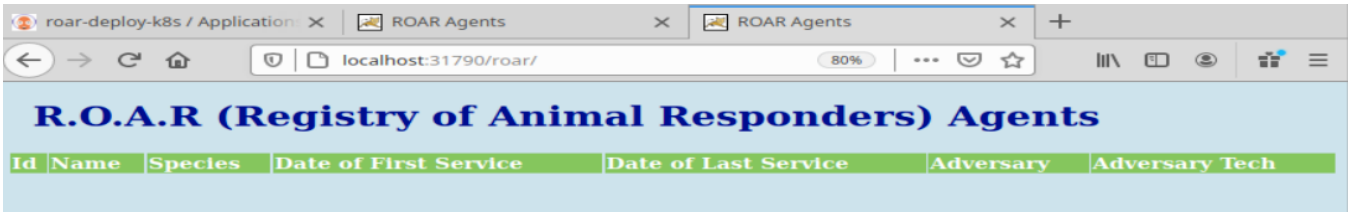
12. Now take a look at the app from "exp" running in the cluster. From a command line, find the node port for the application and then open the URL shown below in a browser.

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

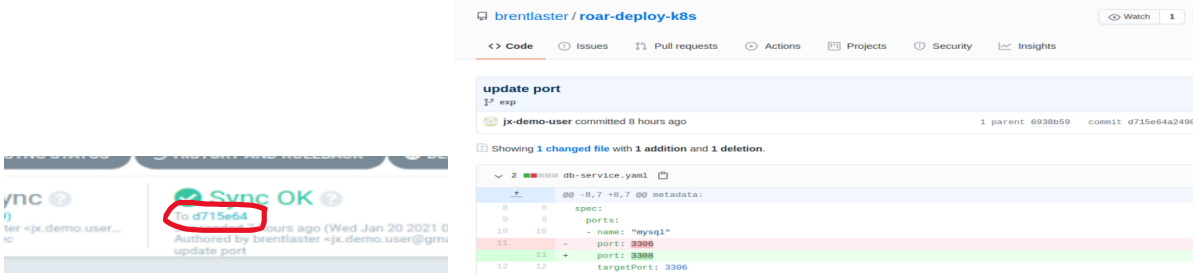
    Look for the port number that is between "8089:" and "/TCP". Plug that in to the URL below.

    http://localhost:<port number>/roar/

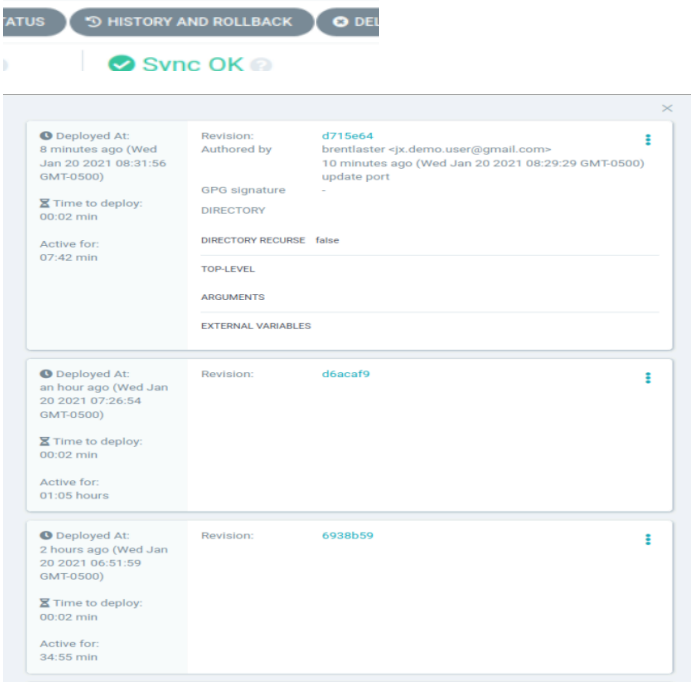
    You'll see something like below. Note that there is no data showing.



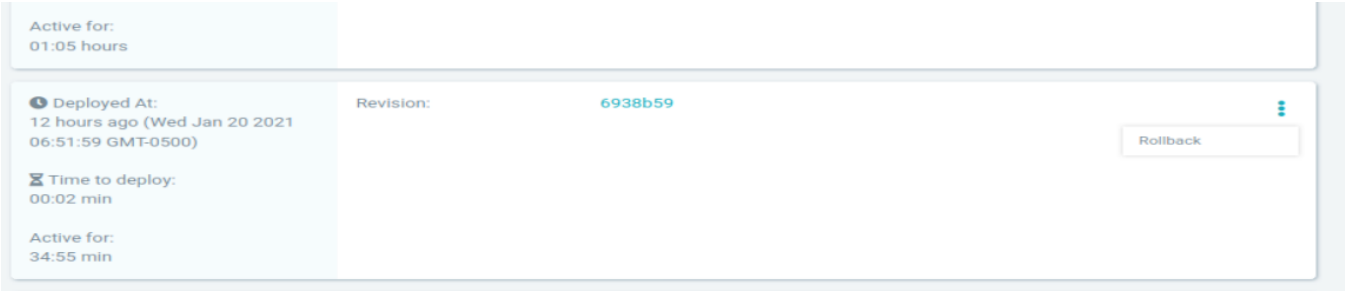
13. (**Optional)** If you want to see why this is broken, you can click on the revision link ("d715e64") at the top of the ArgoCD tab to go to the code in GitHub.



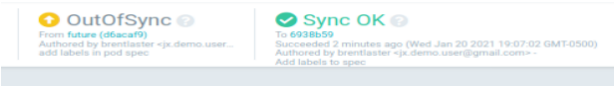
14. We want to rollback to a working state - meaning the original code deployed from main. To do that, click on the "HISTORY AND ROLLBACK" button at the top. You'll then see a list of deployments like the one below.



15. Find the entry for the original (first) revision, click on the 3 dots on the right and then click on the "ROLLBACK" pop-up button. Click on "OK" at the prompt.



16. The rollback syncs the cluster back to the desired version, but does not modify the app. So, while the app still reflects the "future" settings, the cluster is now synched to the desired revision. You can see this at the top of the screen.



17. Now, assuming the service port hasn't changed, if you refresh the screen where the roar web application is, you should see the version with the data in it - indicating that the correct version was applied back to the cluster.

Lab 3: Helm Projects and Automated Sync

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. 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
```

2. Now, we need to add local ssh information to argocd so the argocd pods can recognize the ssh requests for our local git repos.

   The middle command below will find and add ssh information to argocd. The top and bottom commands will show the entries before and after. (10.0.2.15 is the minikube ip address)

```
$ argocd cert list --cert-type ssh

$ ssh-keyscan 10.0.2.15 | argocd cert add-ssh --batch

$ argocd cert list --cert-type ssh
```

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

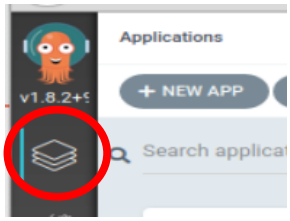
```
$ argocd repo add git@10.0.2.15:/git/repos/roar-k8s-helm.git --ssh-private-key-path ~/.ssh/id_rsa
```

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

```
$ argocd proj add-source helmproj git@10.0.2.15:/git/repos/roar-k8s-helm.git

$ argocd proj add-destination helmproj https://10.0.2.15:8443 helm-demo
```

5. Now, let's create a new app definition in the UI for our project. 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 - check both boxes

SOURCE

Repository URL = git@10.0.2.15:/git/repos/roar-k8s-helm.git

Revision = HEAD

Path = helm

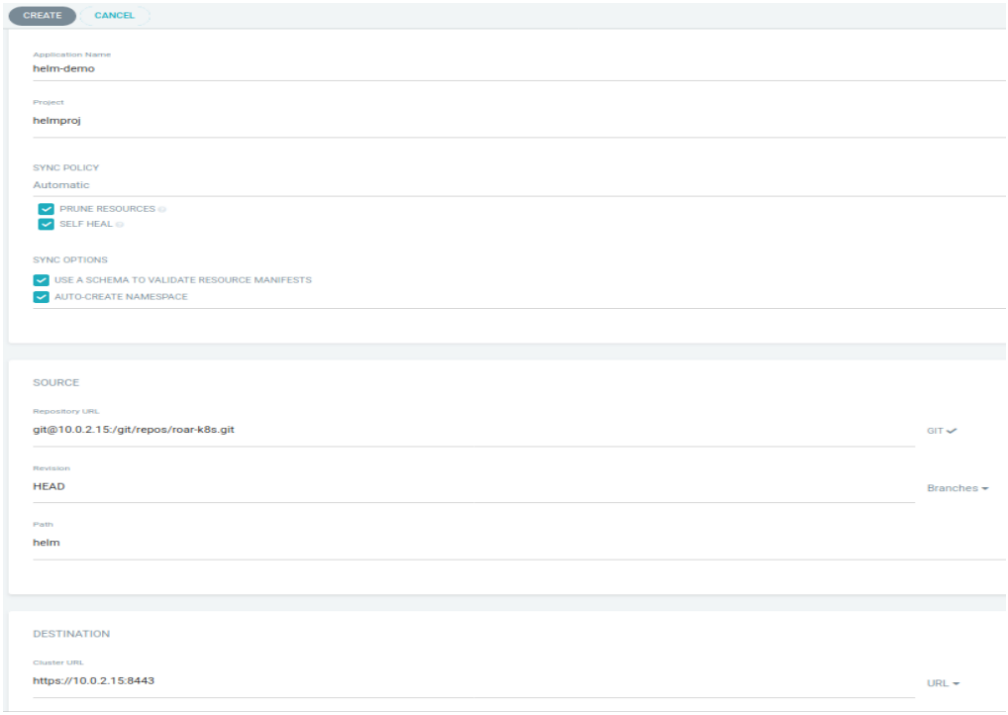
DESTINATION

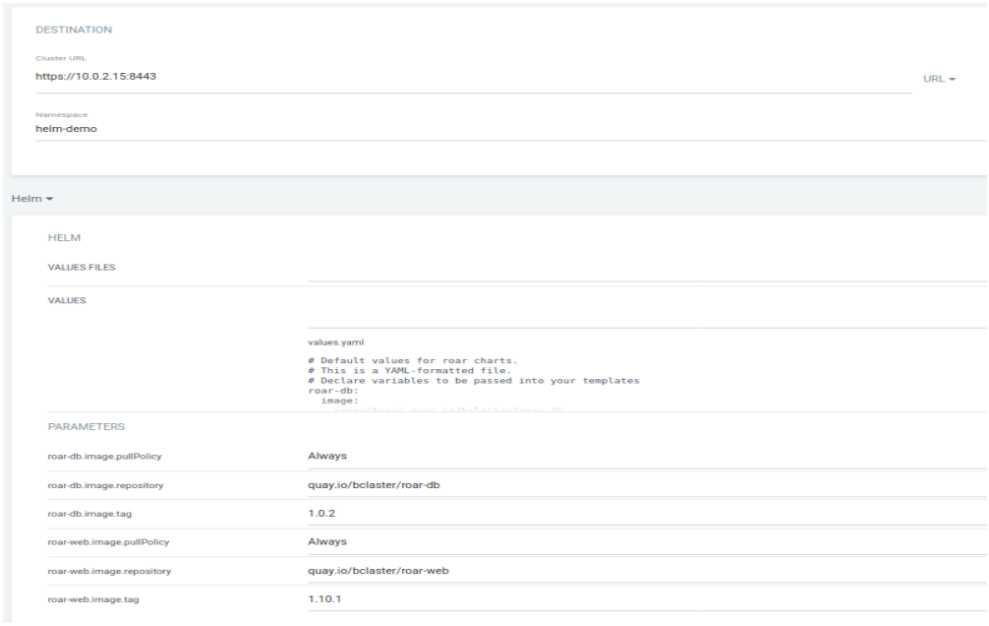
Cluster URL = https://10.0.2.15:8443

Namespace = helm-demo

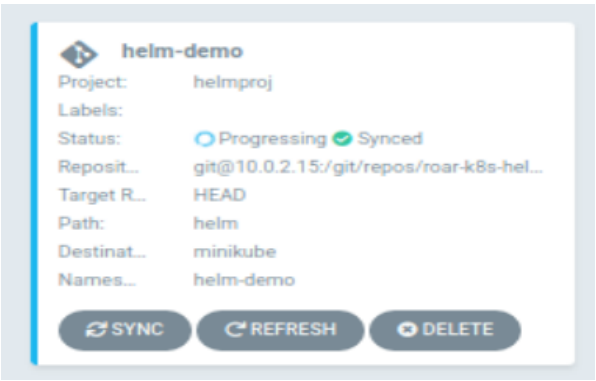
HELM

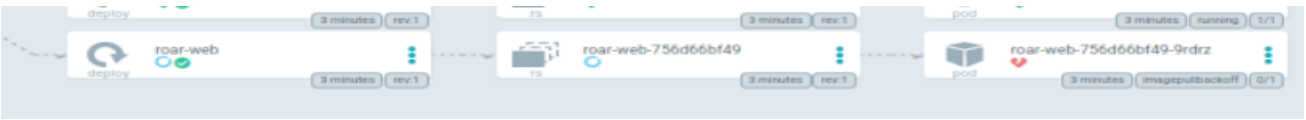
This section will have already found our root/global values.yaml file and populated the PARAMETERS section from it. You can leave it as-is.



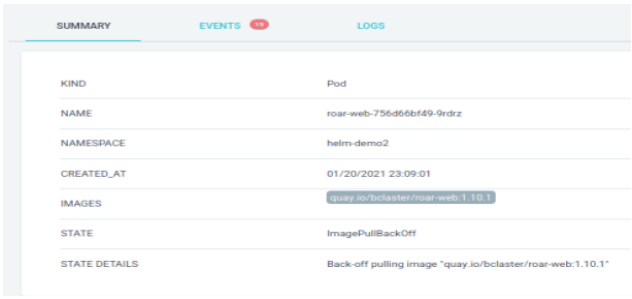


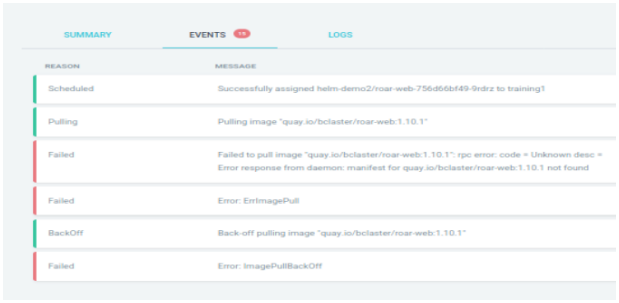
6. When you're done, click on the "**CREATE**" button at the top to save the app. Click on the new app you created. You should see the new app automatically going through the "Progressing" stage while it tries to get things starting up. However, there is a problem with the web pod as indicated by the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





7. Click on that item and look at the SUMMARY and/or EVENTS tabs to see what's wrong.





8. 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 git@10.0.2.15:/git/repos/roar-k8s-helm.git

   (answer yes if prompted for confirmation to connect)

   $ cd roar-k8s-helm/helm

   $ gedit values.yaml

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

```
1  # Default values for roar charts.
2  # This is a YAML-formatted file.
3  # Declare variables to be passed into your templates
4  roar-db:
5    image:
6      repository: quay.io/bclaster/roar-db
7      tag: 1.0.2
8      pullPolicy: Always
9  roar-web:
10   image:
11     repository: quay.io/bclaster/roar-web
12     tag: 1.0.1
13     pullPolicy: Always
14
15
```

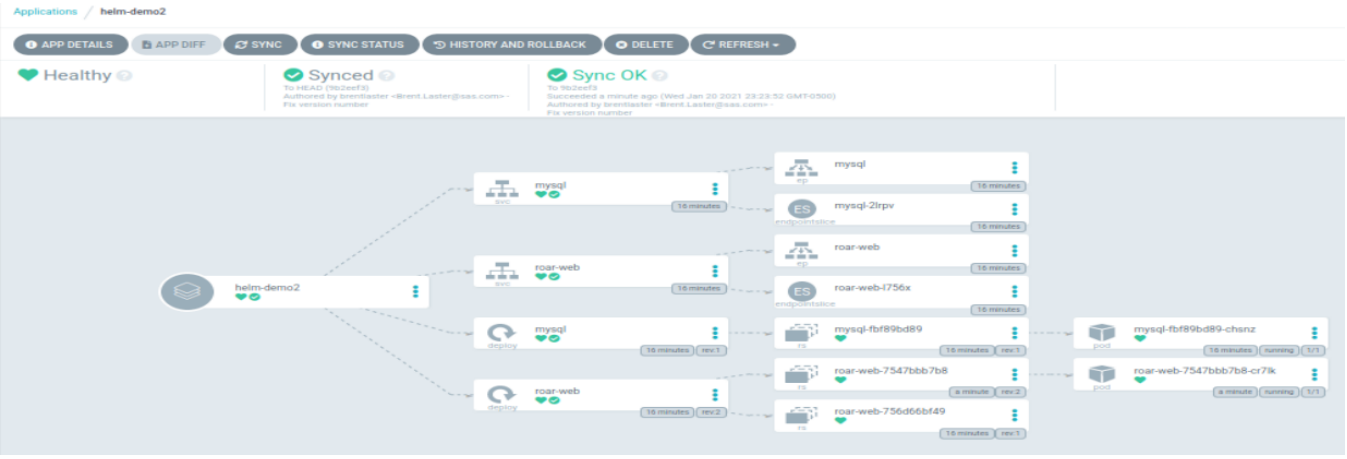
   $ git diff (This is just to verify this is the only change that has been made in your repo.)

   $ git commit -am "Fix version number"

   $ git push origin main

9. 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.)



10. **(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 4: Working with CI/CD pipelines, Jenkins, and ArgoCD

Purpose: *In this lab, we'll see how to use Jenkins to run a simple CI/CD pipeline that can be paired with ArgoCD to automatically deploy changes.*

1. First, let's take a look at a pipeline we have setup in Jenkins to work with. We're using a pipeline script stored in an external Jenkinsfile. To access it, we need to clone down the "roar-min" repository first.

   ```
   $ cd ~

   $ git clone git@localhost:/git/repos/roar-min

   $ gedit roar-min/Jenkinsfile
   ```

2. You can scroll down through the code to see what the pipeline does. It is divided into sections called "stages" for the main operations. The lines that start with "sh" are making calls out to the shell (OS) to run commands.
   Basically, it
   1. pulls code for our ROAR app
   2. builds the deliverables and packages them up
   3. then pulls them in to Docker images
   4. those Docker images are pushed into our local Docker repository running on the minikube instance
   5. Finally, the deploy stages use Kustomize to update the Kubernetes yaml files (manifests) for the project and push them into Git. That is where ArgoCD will kick in.

3. As you look through this, you can see that we have a "**checkout scm**" command to pull down the source code in the project, then a stage to build the binaries of the project with the build tool gradle. Then we

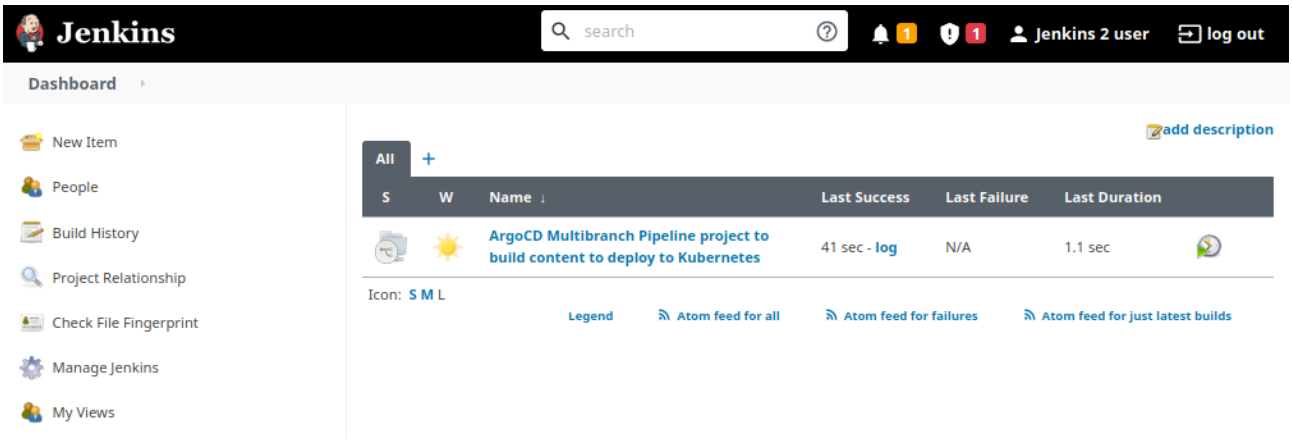
have stages to create updated Docker images - one for the "**stage**" level and one for the "**prod**" level. These Docker images get tagged with the version numbers of "**0.0.<Build Number>**" and "**1.0.<Build Number>**" respectively.

4. Finally, we have stages to use a tool called Kustomize to update the manifests with those version numbers (the tags of the Docker images) and push those updates to a Git repository.

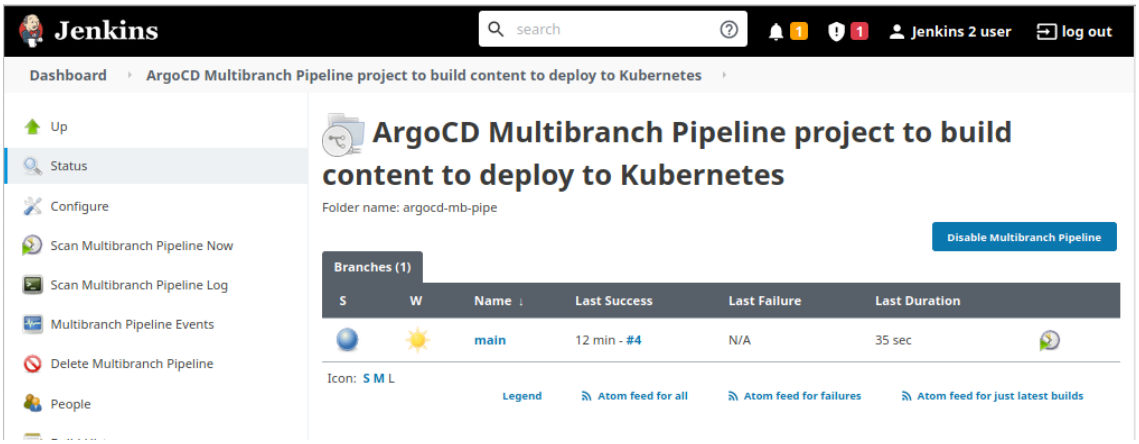
   This is the repository (roar-min-deploy) that our ArgoCD instance is watching and will sync from to update our cluster when changes are made.

5. Now, let's look at the Jenkins project that references this. Open up Jenkins at localhost:8080, and sign in with the userid and password provided in class.

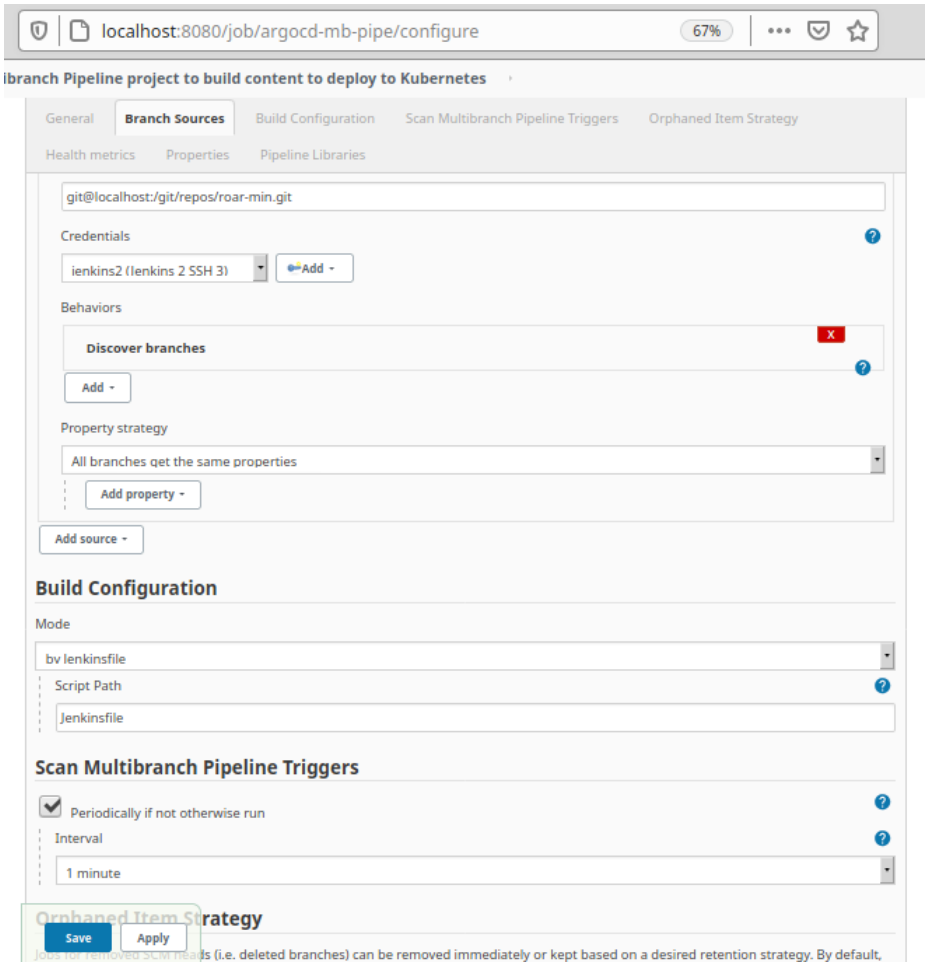
6. On the Jenkins dashboard (home screen) you'll see there is one project with the name starting with "ArgoCD". This is a multibranch pipeline project which means it will monitor for any changes in a branch in the repository and if that branch has a Jenkinsfile with the pipeline in it, it will execute that pipeline in Jenkins.



7. This multibranch project automatically creates a "sub-job" for every branch it finds in the repository that has a Jenkinsfile in it. In this case, the only branch we have is "main" so if you click on the name of the top-level project, you'll see the project for "main".



8. If you want to see how that project was configured, you can select the "**Configure**" menu entry on the left side and look at the details. Probably the most interesting part is the "**Branch Sources**" which points to the deploy repository to watch and the scan interval (lower part of the page) with the interval set to 1 minute.



9. Let's setup a new project and related pieces in ArgoCD to react to updates to the deployment files in Git.

   First, we'll create the project and add the repo and cluster we're interested in as the source (-s) and destination (-d) respectively. (The "*" means any namespace.)

   ```
   $ argocd proj create jenkins-proj -d https://10.0.2.15:8443,* -s
   git@10.0.2.15:/git/repos/roar-min-deploy.git
   ```

   Note that the repo is only for the deployment manifests as that's all ArgoCD is interested in - not the actual project source code or Docker files.

10. Next, we'll add the deployment repo so we can use it in our app.

    ```
    $ cd ~
    ```

```
$ argocd repo add git@10.0.2.15:/git/repos/roar-min-deploy.git --
ssh-private-key-path ~/.ssh/id_rsa
```

11. Now we'll create an app with a single command-line invocation:

```
$ argocd app create jenkins-stage --project jenkins-proj --repo
git@10.0.2.15:/git/repos/roar-min-deploy.git --dest-server
https://10.0.2.15:8443 --path overlays/stage --sync-policy auto
```

12. Let's look at what the app will be applying. Clone down a copy of the roar-min-deploy repository.

```
$ git clone git@localhost:/git/repos/roar-min-deploy
```

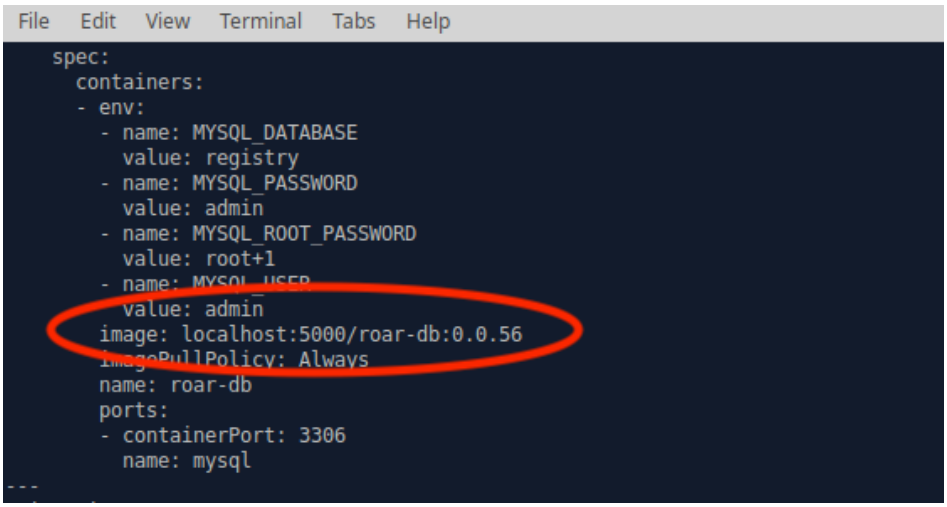
13. Change into the cloned directory area for the staged version and run kustomize to build the manifests.

```
$ cd roar-min-deploy/overlays/stage
```

```
$ kustomize build
```

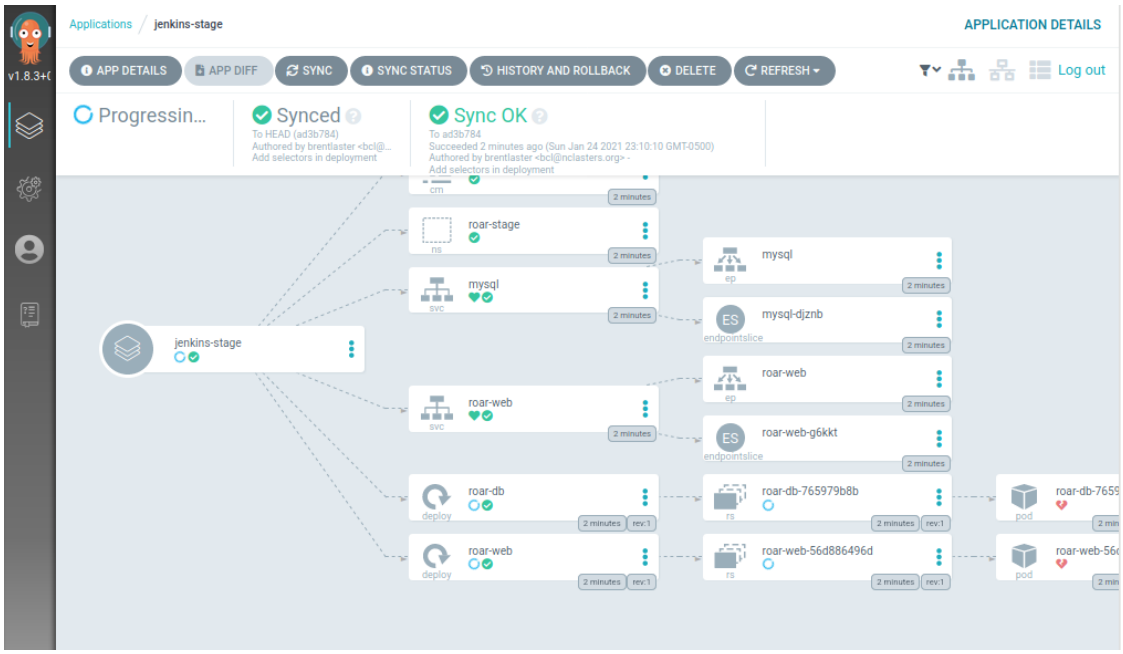
14. The output is the manifests that will be fed into Kubernetes, constructed by applying the overlay pieces and patch on top of the base resources.

    If you scroll and look at the image definitions in the deployment parts, you'll see that it's trying to pull images from our local registry (localhost:5000).



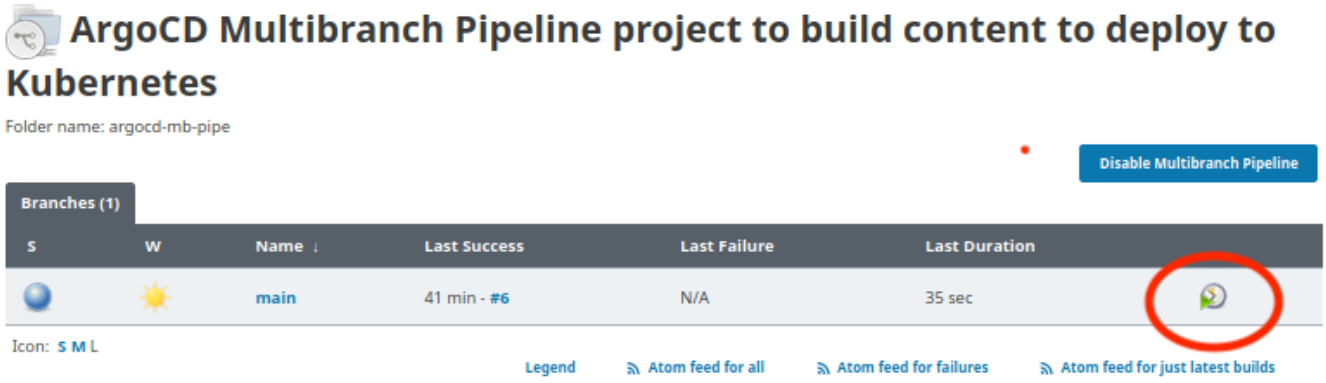
    But we haven't pushed any images to that repo yet. So we would expect our pods trying to pull those images to fail.

    Switch back to ArgoCD, click on the **jenkins-stage** app and notice that that is the case here - the pods are failing to start (as indicated by the red broken heart in the pod blocks on the far right).



15. Switch to the browser tab with Jenkins.

    To see if our pipeline works, start the Jenkins pipeline running by getting out of the Configuration screen for the project and going back to the main project page (**http://localhost:8080/job/argocd-mb-pipe/**). Click on the small button at the end of the status row to schedule a build run.

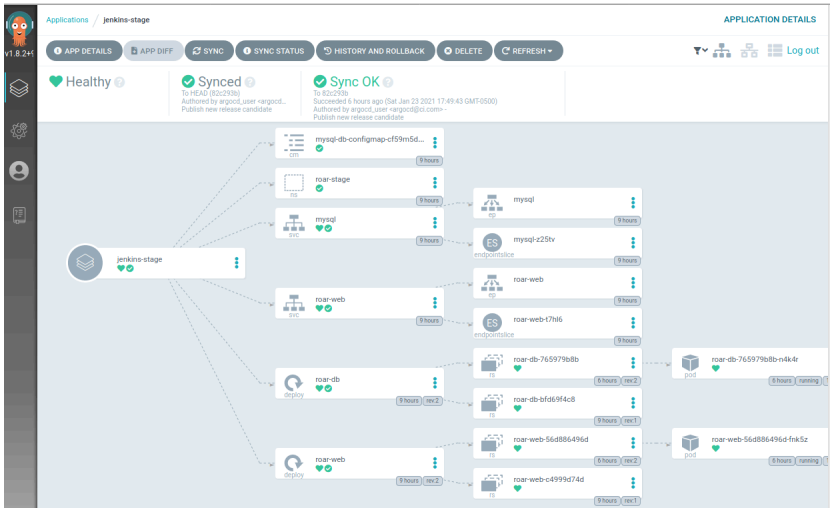


16. This will execute the pipeline and eventually cause our ArgoCD instance to sync the changes to the cluster. You'll be able to see activity happening in the lower left of the Jenkins screen. You can leave this running while we go on to the next section.

Lab 5: Deploying to Production

Purpose: In this lab, we'll, create an app to deploy our prod version.

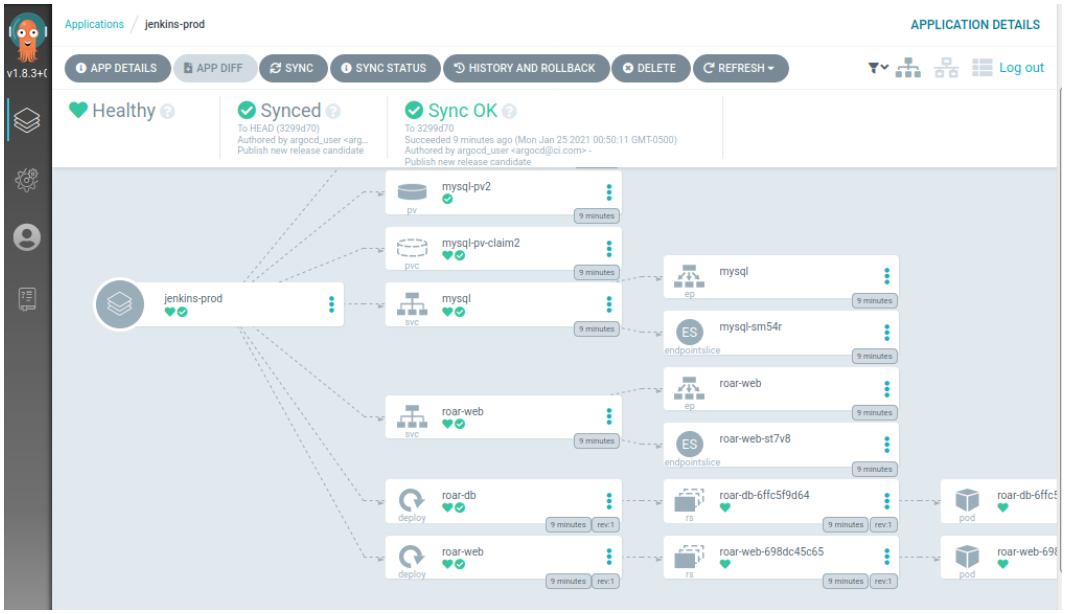
1. Look at the jenkins-stage app we did in the last lab. At this point, if everything built and got deployed ok, you should see everything in sync and healthy (all green).



2. This was all for the stage level. Now, let's add another app for the RC (production) level. Execute the command below:

```
$ argocd app create jenkins-prod --project jenkins-proj --repo
git@10.0.2.15:/git/repos/roar-min-deploy.git --dest-server
https://10.0.2.15:8443 --path overlays/prod --sync-policy auto
```

3. Since we setup the automatic sync, this should start up and sync everything and get it running.



4. Now if you want to verify that the stage and prod are running the expected versions of the images we generated, you can use the commands below.

```
$ k get pods -n roar-stage
$ k describe -n roar-stage pod <a pod-name from step above> | grep Image:
```

You should see image numbers starting with “0.0”

```
$ k get pods -n roar-prod
$ k describe -n roar-prod pod <a pod-name from step above> | grep Image:
```

You should see image numbers starting with “1.0”

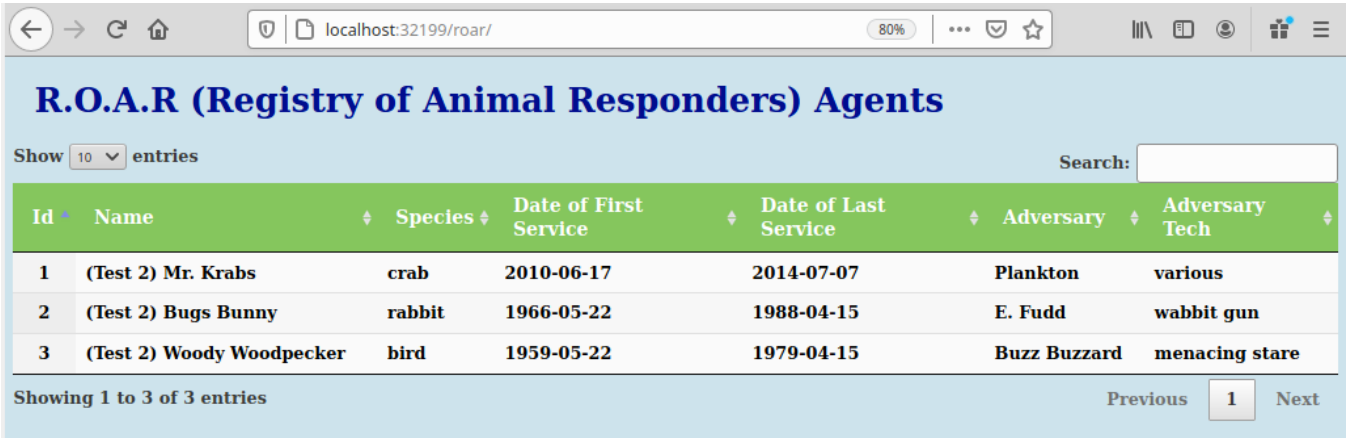
5. You can also see that we’ve deployed two different versions of our application. Take a look first at the stage version.

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

Look for the 5 digit port # that starts with “3” after the “8089” in the PORT(S) column.

Open up the link in the browser as http://localhost:<port number from above>/roar/

You should see something like this:



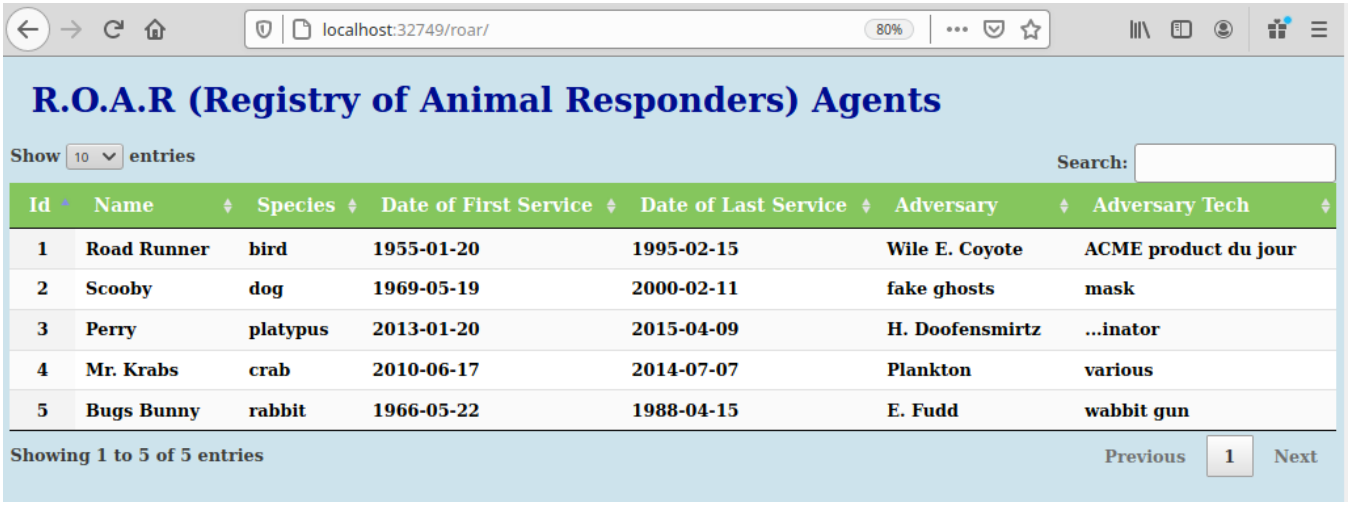
6. Now let’s look at the prod version of our application. Similar procedure for the other namespace.

```
$ k get svc -n roar-prod roar-web
```

Look for the 5 digit port # that starts with “3” after the “8089” in the PORT(S) column.

Open up the link in the browser as http://localhost:<port number from above>/roar/

You should see something like this:



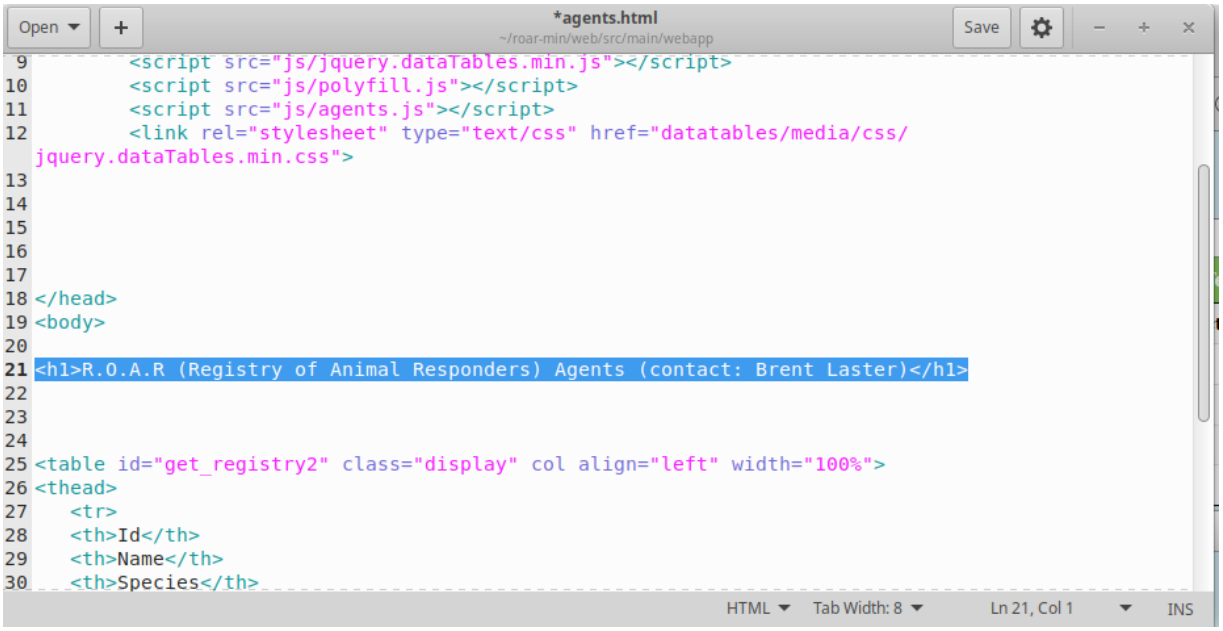
Lab 6: Seeing the entire workflow.

Purpose: In this lab, we’ll make a change to our source code and see the whole pipeline in practice through having our changed deployed into Kubernetes with ArgoCD.

1. We have a clone of the repository for our app already. Change into that directory and edit the file web/src/main/webapp/agents.html.

```
$ cd ~/roar-min
$ gedit web/src/main/webapp/agents.html
```

2. Change line 21 (that starts with “<h1>” and add/modify the text between the “<h1>” and “</h1>” tags in some way. For my example, I put in the “(contact: Brent Laster)” text.

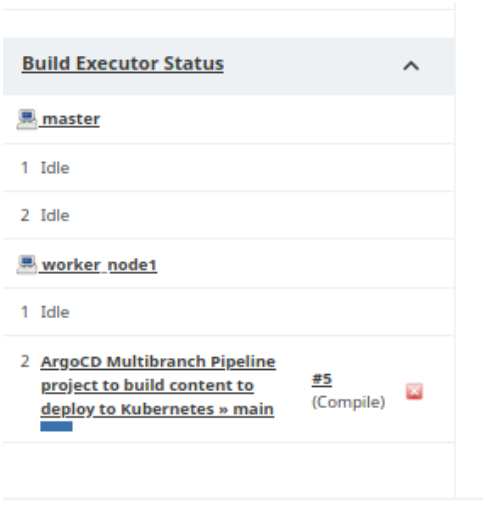


3. Save your changes and close the editor. Back on the command line, commit and push your changes into the git repository.

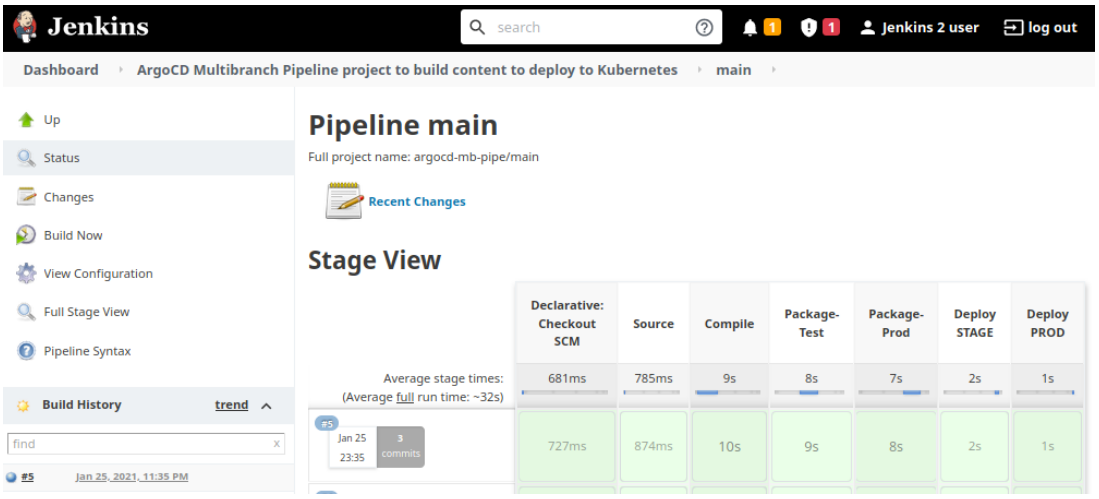
```
$ git commit -am "update page text"

$ git push origin main:main
```

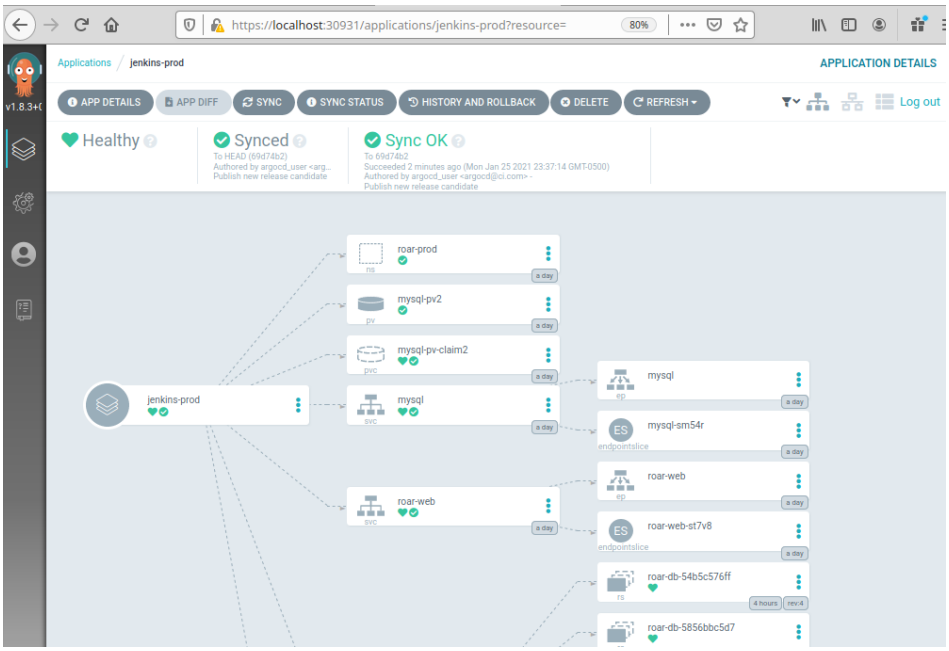
4. At this point, switch back to the Jenkins instance. (You may need to log in again.) After a minute or so, you should see the change be detected and a new build for the multibranch pipeline be automatically kicked off.



5. After a few moments, the build pipeline should complete successfully and if you look at the output, ou can see the green success indication in the various stages that it ran.



6. Switch back to the ArgoCD instance in the browser. (You may need to login again.) After a few minutes, our prod app should automatically sync things up as indicated in the GUI interface.

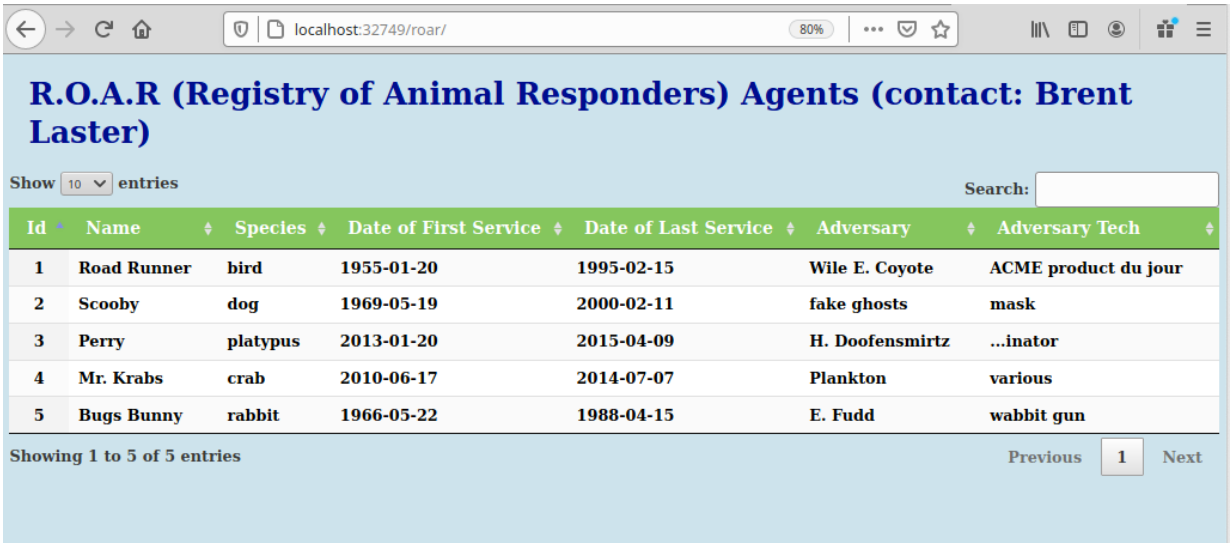


7. At this point, you can get the service ports for the staging and production instances so you can view the updates. (They may be the same as before.)

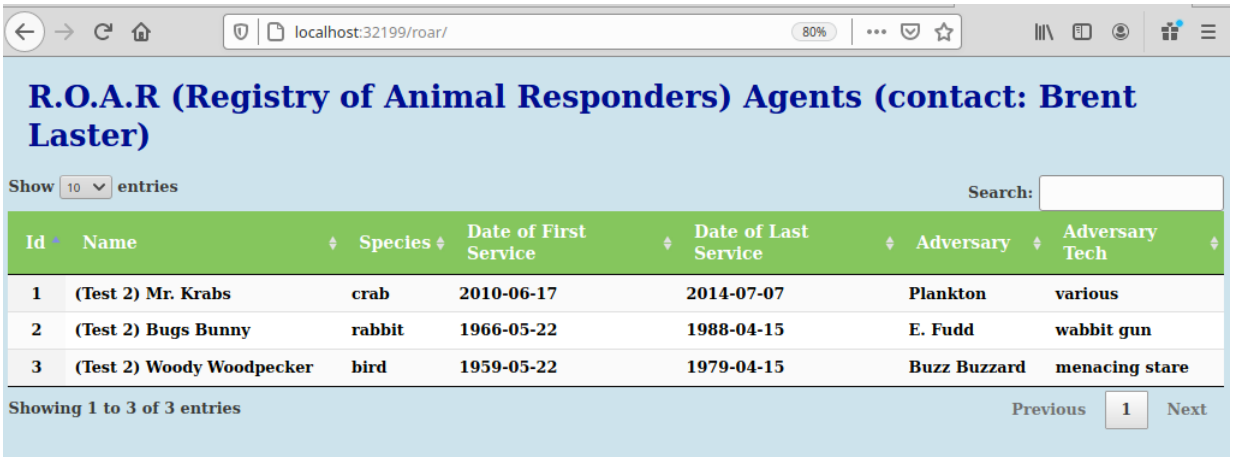
```
$ k get svc -n roar-stage  roar-web

$ k get svc -n roar-prod roar-web
```

8. refresh the screen with your app running and you should be able to see your change in prod as it has been deployed into the cluster automatically. (Note: You may need to clear the cache on your browser to see the update.)



9. You can also go back and manually sync the staging area and see your change in that version as well.



Note: If you need to clear the cache, you can go to Settings, then enter “cache” in the search box, select the “Clear Data” button, then check “Cached Web Content” and select the “Clear” button.