# Kustomize Fundamentals

Managing Kubernetes Manifests

**Class Labs**

Version 1.0 by Brent Laster

11/01/2021

**Important Prereq:** These labs assume you have already followed the instructions in the separate setup document and have VirtualBox up and running on your system and have downloaded the *kz-fun.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!

**Startup - to do before first lab**

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



2. First, let's make sure we have the latest files for the class. For this course, we will be using a main directory *ts-k8s* with subdirectories under it for the various labs. In the terminal window, cd into the main directory and update the files.

$ cd kz-fun

$ git pull

3. Next, start up the paused Kubernetes (minikube) instance on this system using a script in the *extras* subdirectory. This will take several minutes to run.

$ extra/start-mini.sh

**Lab 1 - 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.**

1. Change to the ***lab1/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 lab1/base**

**$ tree**

1. Let's see what happens when we try to run "kustomize build" against these files.

**$ kz build**

(The kustomize command is aliased to just "kz" on this machine.)

1. 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.lab1". 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 ~/kz-fun/extra/kustomization.yaml kustomization.yaml**

**$ cat kustomization.yaml**

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

1. 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.

**$ k get all**

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

**$ cat namespace.yaml**

**$ gedit kustomization.yaml**

**Graphical user interface, text

Description automatically generated**

1. 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 original**

1. 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 -**

**$ kz get ns**

**$ kz get all -n roar-original**

1. Let's make one more change here. Let's apply a common label and 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**

**Graphical user interface, text

Description automatically generated**

1. Now you can run kustomize build and see the annotations and labels. Afterwards you can go ahead and apply the changes.

**$ kz build | grep -a5 metadata**

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

1. Now an instance of our application should be running in the roar-original namespace. 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-original | grep mysql**

**<find Nodeport>**

**open** [**http://localhost:<nodeport>/roar**](http://localhost:%3cnodeport%3e/roar) **in browser**

**Lab 2- 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-fun directory and create two the two directories.

**$ cd ~/kz-fun**

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

1. 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**

**﻿overlays**

**├── production**

**│   └── kustomization.yaml**

**└── staging**

**└── kustomization.yaml**

1. 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**

1. 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-original" 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**

1. Now you can do a kustomize build on each to verify it has the desired name in the output.

**$ kz build overlays/staging**

**$ kz build overlays/production**

1. 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.

**$ k create ns roar-staging**

**$ k create ns roar-production**

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

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

1. 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

Graphical user interface, text, application

Description automatically generated

1. Save your changes and then apply the variant.

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

1. **You can now find the nodeport for the service from roar-staging and refresh and see the test version of the data.**

**A picture containing graphical user interface

Description automatically generated**

**Lab 3 - Leveraging patches to update manifests**

**Purpose: In this lab, we’ll see how to leverage patches to update manifests.**

1. We'll turn our attention now to the "production" variant. Since this is intended to be production, we want to have it able to use external storage that is mounted rather than the volatile container storage it is currently using. There is already a file in the extra area that defines a PersistentVolume and PersistentVolumeClaim for us. Let's copy this file over to the "production" directory and then add it to our list of resources via editing the kustomization file and adding the highlighted line. Save your changes when done.

**$ cd ~/kz-fun/overlays/production**

**$ cp ../extra/storage.yaml .**

**$ gedit kustomization.yaml (add line "- storage.yaml" and save)**

**Graphical user interface, text, application, chat or text message

Description automatically generated**

1. So now this will define the PV and PVC objects we need. When the deployment is setup to use them via mounting the volume, a new directory for the database data will be created under /mnt/data on the system. You can verify that there is nothing there currently for this.

**$ ls /mnt**

**(should be empty)**

1. Now we need to patch the database deployment in our production overlay to include the appropriate volumes and volumeMounts values. To do this, we will include the patch information in our kustomization.yaml file. The definition for the patch is contained in a separate file already setup for you - "patch\_pv.yaml" in the "extra" folder. Copy that file over and take a look at it. Notice that it only includes the minimal information needed to identify the resource to patch (our deployment) and the data for the patch itself.

**$ cp ../extra/patch\_pv.yaml .**

**$ cat patch\_pv.yaml**

1. Now we'll add it into the kustomization file and tell Kustomize to update the resource via the use of a "patchesStrategicMerge" directive. Edit the kustomization.yaml file and add these lines and then save your changes when done. (The comment shown is optional.)

**﻿patchesStrategicMerge:**

**- patch\_pv.yaml**

**$ gedit kustomization.yaml**

**Graphical user interface, text, application

Description automatically generated**

1. With your changes saved, create the new namespace, then build and deploy the generated manifests. After a few moments, you should be able to look at the /mnt area again and see that it is now populated with data.

$ k create ns roar-production

$ kz build | k apply -f -

$ ls /mnt

**Lab 4 - More advanced patching and generators**

**Purpose: In this lab, we’ll see how to use more advanced patching functionality in Kustomize and also how to work with generators to update objects together.**

1. In the manifest for our database service, we use environment variables to set several values that would be better set via configMaps or secrets. You can see this by looking at a subset of lines after "env" in the database deployment in the base.

**$ ﻿grep env -A8 ~/kz-dev/base/db/roar-db-deploy.yaml**

1. Let's see how we might could change one of these using Kustomize. Let's change the MYSQL\_PASSWORD to be picked up from a secret instead of from an environment variable. To do this we'll need to use a more extensive "JSON6902" type of patch. It would look like this:

﻿patchesJson6902:

- patch: |-

- op: replace

path: /spec/template/spec/containers/0/env/1

value:

name: MYSQL\_PASSWORD

valueFrom:

secretKeyRef:

name: dbp

key: dbp.info

target:

group: apps

kind: Deployment

name: mysql

version: v1

1. Since this code is a little complex to type in, the patch spec is already done for you in a file named "jsonpatch.yaml" in ~kz-fun/extra. We need to get this added into our kustomization.yaml file. We'll add it to staging to try it out. You can either get the text from the jsonpatch.yaml file and copy and paste it into kustomization.yaml in an editor or you can use the shortcut to append it below. If you use the shortcut, make sure to use two greater-than signs ">>" to append! Afterwards you can do a build to make sure you see the patch in the rendered output.

**$ cd ~/kz-fun/overlays/staging**

**$ cat ~/kz-fun/extra/jsonpatch.yaml >> kustomization.yaml**

**$ ﻿kz build | grep MYSQL\_DATABASE -A10**

1. If we applied this, we'd still be missing the secret. Let's add one using the "kustomize edit" command. We'll set up the secret to be based off of a file. So we'll create that data file first (in the overlays/staging directory).

**$ echo admin > dbp.info**

**$ ﻿kz edit add secret dbp --from-file=dbp.info**

1. Take a look at the changes. Then, build and deploy your changes. Notice in the output of the first command that the secret has had a unique identifier appended to it both in the definition and where it is used by the deployment. This is part of the generator functionality.

**$ ﻿kz build | grep -inA5 secret**

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

1. Find the nodeport where the app is running.

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

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

1. Open up a browser and go to [http://localhost:<NodePort>/roar/](http://localhost:%3cNodePort%3e/roar/)

You'll see that while our web app is up, there's no data being displayed.

Graphical user interface

Description automatically generated

1. The reason for this is that the password had an eol character added when we created it. Let's fix that using the -n option on the echo to update the password. Then we will rebuild the staging piece again. Take a look at the "secret" sections again and notice that there is a different unique identifier appended this time.

**$ echo -n admin > dbp.info**

**$ ﻿kz build | grep -inA5 secret**

1. Now you can rebuild and apply the updated manifest. Notice that when you do that, because the secret name changed and the database deployment was pointing to the updated secret name, the database deployment automatically got updated. This would not have happened without the "generator" functionality.

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

…

﻿service/mysql unchanged

service/roar-web unchanged

**deployment.apps/mysql configured**

deployment.apps/roar-web unchanged

1. Refresh the app in the web browser a couple of times and you should now see that since we got the correct password, our data shows up.

**Graphical user interface, table

Description automatically generated with medium confidence**

**Bonus: Lab 5 - Using Components**

**Purpose: In this lab, we’ll see how to use new Kustomize components to make functionality easily selectable between overlays.**

1. While it is nice to be able to use overlays to produce variants, if we want to selectively add functionality at a smaller scope, creating variants for each possible combination is not an ideal solution. A different approach is to compartmentalize the functionality into a Kustomize "component". To start with for doing this, we'll need to create a components directory at the same level as "base" and "overlays". To illustrate the functionality, we'll package up the external storage option as a component. For this, we'll need to create a new directory under "components".

**$ mkdir -p /kz-fun/components/patch-storage**

1. Now we'll need to add a new kustomization.yaml file under there. The kustomization.yaml file will be of type Component. The kustomization yaml file is already created for you to move over to the components area with the command below. After you copy it, take a look at how it works.

**$ cd /kz-fun/components/patch-storage**

**$ cp ~/kz-fun/extra/storage-component.yaml kustomization.yaml**

**$ cat kustomization.yaml**

1. As you can tell from looking at the contents, the kustomization file for the component makes use of the two basic files we used earlier to do the patching to add the storage: storage.yaml to define the storage and PV and PVC, and patch-storage.yaml to add the volume and volume mounts into the database deployment. We need those files here for the component to use. Copy them over.

**$ cp ~/kz-fun/extra/storage.yaml .**

**$ cp ~/kz-fun/extra/patch-storage.yaml .**

1. When we added this to our production variant before, we added this spec into the kustomization.yaml file there. We're now going to add the functionality to the staging variant. To do this we'll just add a reference to the component into our kustomization.yaml file in the staging overlay. You can edit the file and add the lines below. Save your changes when done.

**$ cd ~/kz-fun/overlays/staging**

**$ gedit kustomization.yaml**

add these two lines:

﻿components:

* ../../components/patch-storage

**Graphical user interface, text, application

Description automatically generated**

1. Remove the existing data in the /mnt area, the namespace roar-staging, and the PV mysql-pv to get a clean environment. Then build and apply the staging overlay. After a few moments, you should be able to see a new "data" area under /mnt.

**$ sudo rm -rf /mnt/data; ls /mnt**

**$ k delete ns roar-staging**

**$ k delete pv mysql-pv**

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

**$ ls /mnt**

1. Let's do one more example and turn the database secret patch into a component. First create a new directory under the components area.

**$ mkdir $/kz-fun/components/db-secret**

1. Now we'll need to add a new kustomization.yaml file under there. The kustomization.yaml file will be of type Component. The kustomization yaml file is already created for you to move over to the components area with the command below. After you copy it, take a look at how it works.

**$ cd /kz-fun/components/db-secret**

**$ cp ~/kz-fun/extra/secret-component.yaml kustomization.yaml**

**$ cat kustomization.yaml**

1. As you can tell from looking at the contents, the kustomization file for the component makes use of the patch to change the deployment to use the secret and the password data file. We need the password data file here for the component to use. Copy it over.

**$ cp ~/kz-fun/overlays/staging/dbp.info .**

1. When we added this functionality to our staging variant before, we added the spec into the kustomization.yaml file there. We're now going to add the functionality to the production variant. To do this we'll just add a reference to the component into our kustomization.yaml file in the production overlay. You can edit the file and add the lines below at the bottom. Also, to prevent a conflict with the current staging storage, remove the line that includes the storage.yaml file in the resources and the patch for patch-pvc.yaml. When your file is done it should look as in the editor snapshot below.

**$ cd ~/kz-fun/overlays/productoin**

**$ gedit kustomization.yaml**

add these two lines at bottom:

﻿components:

* ../../components/db-secret

remove these lines:

* storage.yaml

patchesStrategicMerge:

* patch\_pv.yaml

**Graphical user interface, text, application

Description automatically generated**

1. Now you can build it, and see that the secret pieces have been added via the component. After that, you can build and apply, find the service nodeport as before, open in a browser and see the app running.

**$ kz build | grep -inA10 secret**

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

**END OF LABS**

**$ cp ~/kz-fun/extra/kustomization.yaml.patch-svc kustomization.yaml**

**$ cat kustomization.yaml**

1. Now, so that we get rid of the existing objects, delete the namespace and recreate it.

**$ k delete ns roar-staging**

**$ k create ns roar-staging**

1. Build and deploy from the test directory (make sure you are in the "overlays/test" directory).

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

1. Repeat steps 4 and 5 to refresh the running application. This time you should see the data showing up in the webapp.

**A picture containing graphical user interface

Description automatically generated**

to-do

install kustomize 3.8.9 on system - get from <https://github.com/kubernetes-sigs/kustomize/releases?q=3.8.9&expanded=true>

in ~/kz-dev/overlays/staging - try build to see if it picks up and executes component patch-svc. if it does can use that to illues

that doesn't work , but see if you can make storage into a component

1. build and deploy
2. find nodeport, go to it and notice no data displayed. Explain that this is because service is now named mysql-staging. Do a build to show service name is "mysql-staging"
3. Not great way to fix this so we will try a patch on top of the overlay. Create new "test" directory at same level (sibling) to staging. In there copy over kustomization.yaml.patch-replace from extra to kustomization.yaml.
4. Look at what it does - notice resources that are pulled in and patch operation.
5. Now do a build and show that the service name has been patched.
6. Delete the staging namespace and recreate.
7. build from test area and deploy.
8. find nodeport and bring it up to show that it works.

To validate if this will fix the problem, let's edit the existing object. We'll set the EDITOR variable and then edit the deployment object. We'll also setup a watch command to watch the pod change.

(Do this one command in a separate terminal session:

**$ k get pods -w**)

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

**Change line 42 to use 1.0.1 instead of 1.10.1.**

**Text

Description automatically generated**

1. Save your changes to the deployment and close the editor. Eventually, you should see a new pod finished creating and running. The previous web pod will be terminated and removed. However, momentarily, the pod will change to a CrashLoopBackOff status. We'll figure that out in the next lab.

Leave the watch running in the other window for the next lab.

END OF LAB

**Lab 3 - Debugging failed and crashed containers within Pods**

**Purpose: In this lab, we’ll look at ways to troubleshoot failed containers within pods and how to spin up pods to debug them.**

1. We know from our last lab that the web pod is having some more serious issues. Let's start by doing a describe and a log. For this one, grab the name of the pod from the output of a "get pods" and use that instead of the label.

$ k get pods *(this is to get the name)*

Text

Description automatically generated

$ k describe pod <web pod name>

$ k logs <web pod name>

1. The describe doesn't tell us much that's meaningful. But the logs note several errors. Run the command below to find the first SEVERE error. It will look like the output below.

$ k logs <web pod name> | grep -m1 SEVERE -A2

**﻿***21-May-2021 22:02:20.234 SEVERE [main] org.apache.tomcat.util.digester.Digester.fatalError Parse fatal error at line [28] column [1]*

*org.xml.sax.SAXParseException; systemId: file:/usr/local/tomcat/webapps/roar/WEB-INF/web.xml; lineNumber: 28; columnNumber: 1; XML document structures must start and end within the same entity.*

1. If we want to debug further, we have a challenge because the container has crashed. Depending on the timing (between restarts) we might be able to exec into it and work from there. You can try this command, although you may get "container not found" messages unless the timing happens to be just right.

$ k exec -it <web pod name> -- bash

1. If you did get in, you can just "exit" out of that. Another approach we have for getting into a pod like this is using the kubectl debug command to copy it to another pod and start it with a different command. Try the example below for this. Once inside you can look at the problem file.

$ k debug <web pod name> -it --copy-to=roar-debug --container=roar-web -- sh

﻿$ cat /usr/local/tomcat/webapps/roar/WEB-INF/web.xml

(Note that when the restart cycle starts, your session will end.)

1. Since this is an XML error, we might want to run a debugging tool like an XML parser or linter to find out more. Unfortunately, we don't have any of those tools in this image. But I've created an image with the xmllint tool in it and we can use the *kubectl debug* command to create a debug container and attach to the pod. You can exit out of the previous connection if still in that and then run this command below.

﻿$ k debug <web pod name> -it --image=quay.io/techupskills/roar-debug:1.0.2 --share-processes --copy-to=roar-debug2 -- bash

1. Once in this session, we are in the debug pod, but have shared access to processes running on the original pod. We can see the processes with simple commands such as:

$ ps ax

1. In this pod, we have the xmllint tool. And we can access some things on the file system via the syntax of *"/proc/<process id>/root/<path>"*. Assuming you are at a place where you can do "ps ax" and see the "tomcat" process running (starts with /usr/local/openjdk-11/bin/java) then you can copy and paste this command to run xmllint against the problem file.

$ ﻿xmllint /proc/$(ps ax | grep tomcat | awk '{print $1}' | head -n1)/root/usr/local/tomcat/webapps/roar/WEB-INF/web.xml

When this executes, you should see output similar to the following:

﻿/proc/6/root/usr/local/tomcat/webapps/roar/WEB-INF/web.xml:28: parser error : EndTag: '</' not found

^

(If you can't seem to catch it when the process is running, you can try deleting the mysql pod and the debug pod, letting K8s generate a new pod and then repeating steps 5-7)

You can just **exit** out of the debug pod when done.

1. Since we have a problem with the existing container, there will need to be a new image created to fix it. I've already created one at quay.io/techupskills/roar-web:1.0.2. We can use kubectl debug again here - to make a copy of our pod and replace the existing image reference with our new one. Use the command below to do that.

$ k debug <web pod name> --copy-to=web-test --set-image=roar-web=quay.io/techupskills/roar-web:1.0.2

1. After this, you can look at the logs to see that the new pod (web-test) has started successfully and isn't having the same issues. (It may take a couple of tries before you see it.)

$ k logs web-test

1. Now that we know that this image works, we can update the image in our existing deployment that is having the issues. We do this with the set image command. After this, you should see that there is a new image created that eventually will reach the Running state. You can also delete the test image, the debug pods, and any older replicasets (rs).

$ k set image deploy/roar-web roar-web=quay.io/techupskills/roar-web:1.0.2

$ k delete pod web-test

$ k delete pod roar-debug

$ k delete pod roar-debug2

$ k delete rs/<older rs names - leave newest/youngest one>

END OF LAB

**Lab 4 - Working with Probes**

**Purpose: In this lab, we’ll look at ways to debug issues when trying to use probes.**

1. Take a look at the pods in our namespace again. You should see that while the web pod is running, the database pod is not ready. Now, let’s do a “describe” operation on the mysql pod.

$ k get pods

$ k describe pod -l app=mysql

1. Note the error message near the bottom of the output mentioning the readiness probe failed. The readiness probe in this case is just an exec of a command to invoke mysql. The error implies that the call to “mysql” failed. But note that it doesn’t say it couldn’t find it. Rather, it wasn’t valid to call it that way since it tried to invoke it without a valid name and password to login.
2. You can see the YAML for this in the deployment template in the corresponding Helm chart. In the terminal window, take a look at that and find the section near the bottom with the **readinessProbe** spec.

$ cat charts/roar-db/templates/deployment.yaml

1. We actually don’t need to have a command login to verify readiness – we just need to know the mysql application responds. Let’s fix this by trying something simpler such as calling the “version” command - which we should be able to do without a login.

1. You can edit the deployment as we did in the first lab. Run the kubectl edit command. Add a line as shown below, paying careful attention to the spacing. (Remember to use spaces and not tabs.) When you are done, save your changes and exit the editor.

$ k edit deploy/mysql

And change (around line 58)

readinessProbe:

exec:

command:

- mysql

failureThreshold: 3

initialDelaySeconds: 5

To add the line shown in bold (remember to use spaces)

readinessProbe:

exec:

command:

- mysql

**- --version**

failureThreshold: 3

initialDelaySeconds: 5

Graphical user interface, text

Description automatically generated

1. Save your changes and exit the editor. At this point, you can get the service's nodeport and then open up a browser on localhost to the URL below to see the application running.

$ k get svc

Look for the port > 30000 after the 8089: in the roar-web line. Plug that value in for <port> below and open the URL up in a browser.

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

**Graphical user interface, text, application, email

Description automatically generated**

1. When the application comes up, you may notice something interesting about it - there is no data being displayed. Let's do the typical *logs* and *describe* commands to see if we can determine the problem. (Verify that you only have the 2 pods. If not, you may need to delete some remaining ReplicaSets.)

$ k logs -l app=mysql

$ k describe pod -l app=mysql

Graphical user interface, text

Description automatically generated

1. Since that didn't indicate any problem, let's verify that the database is actually accessible and has data in it. We'll do this with a *kubectl exec* command. You will need to copy the mysql pod name to use here.

$ k exec -it <mysql pod name> -- bash

1. Now, you’ll be inside the db container. We can use one command to check that things look right here. (Type this at the /# prompt. Note no spaces between the options -u and -p and their arguments. You need only type the part in bold.) There are 2 separate steps. If everything looks good, then exit the container exec.

mysql@container-id:/$ **mysql -uadmin -padmin registry -e 'select \* from agents';**

mysql@container-id:/$ **exit**

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

1. Since the database looks ok, we need to look elsewhere for the problem. We'll next look at the service for the database in the next lab.

END OF LAB

**Lab 5 - Troubleshooting Services**

**Purpose: In this lab, we’ll troubleshoot how to determine the problem(s) when your service isn't accessible.**

1. When there is a potential problem with a service, one of the first things to check is whether the networking is working as expected for the cluster. An easy way to get some simple info is to use the cluster-info command. You can get basic info in its simple form or you can add the "dump" option to see more info.

**$ k cluster-info**

**$ k cluster-info dump**

1. This doesn't show any problem (that's easy to see at least). Let's try a different approach. We'll start up a basic pod (like busybox) in the cluster and run some basic commands through it. In that way, you are limiting the variables involved rather than using one of your custom pods to check this. Run the command below to start such a pod in the cluster. (Note the space between the last "--" and "sh".)

$ k run -it --rm debug --image=busybox:1.28 --restart=Never -n ts -- sh

1. With the pod running, let's check if the DNS is working. To do that, we'll simply use the nslookup command to see if can resolve Kubernetes.default (Kubernetes.default is a special service that provides a way for internal applications (in the cluster) to talk to the API server.) At the pod's prompt, run the command below.

**/ # nslookup kubernetes.default**

This should return output similar to this:

*Server: 10.96.0.10*

*Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local*

*Name: kubernetes.default*

*Address 1: 10.96.0.1 kubernetes.default.svc.cluster.local*

(If it were not working, you would see something like this:

*Server: 10.96.0.10*

*Address: 10.96.0.10*

*nslookup: can't resolve 'kubernetes.default')*

1. Now, let's check to see if we can see our database services We'll use a similar command. Run the command in bold below and you should see similar output.

**/ # nslookup mysql**

Server: 10.96.0.10

Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name: mysql

Address 1: 10.110.163.71 mysql.ts.svc.cluster.local

1. So far, so good. Next, we'll check to see if we can get to the services from within the cluster. To do this, we'll need the CLUSTER-IP address of our mysql service. Find this now with the following command (in a different terminal) and look for the item in the locations circled in red in the screenshot below.

$ k get svc mysql -o wide

****

1. Using the value from step 4, run the following command back in the busybox pod. Afterwards, you can exit this container. (That is the letter O, not the number 0 after the q.)

/ # wget -qO- <CLUSTER-IP for mysql>:3306

/ # exit

1. Notice the error message you got from that last operation (the "wget: can't connect to remote host…" output). This is a clue. To connect, a service needs endpoints. Let's check for those now. In a separate terminal:

$ k get ep

1. We have endpoints for the web service, but not for the database service. This is likely the problem. Endpoints are implemented via selectors between the service and the pods. Check the selector that's being used for the service. We'll use the jq tool here to easily parse the output.

$ k get svc mysql -o json | jq -j '.spec.selector'

The output should look like:

{

"name": "roar-db"

}

1. Now we should check to see what labels are on the database pod to see if they match.

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

You should see output something like this:

*mysql-78fd7d4744-wqn82 0/1 Running 0 26h app=mysql,pod-template-hash=78fd7d4744*

1. There is not a match for *name=roar-db.* But the pod does have an app=mysql label. So we could either add the label in the deployment that the service is looking for (it will spin up a new pod) or add the pod's label in the service. We'll do the former using another method - *kubectl patch*. Note: If you prefer not to use the patch, you can do a *k edit* on the deployment and modify the label that way. Here's the patch command - the syntax is important so be careful. There should be a space after "path": .

﻿$ k patch deploy mysql --type=json -p='[{"op": "add", "path": "/spec/template/metadata/labels/name", "value": "roar-db"}]'

1. If you still have 2 mysql pods, you can remedy that via scaling the deployment down to 0 and then back up to 1.

﻿$ k scale deploy/mysql --replicas=0

﻿$ k scale deploy/mysql --replicas=1

1. After the new pod gets done starting up, you should be able to refresh the browser and see data on the web page indicating everything is ok.

Table

Description automatically generated

END OF LAB