Troubleshooting Kubernetes

Learning to identify, understand, and fix the most common issues in the cluster

Class Labs

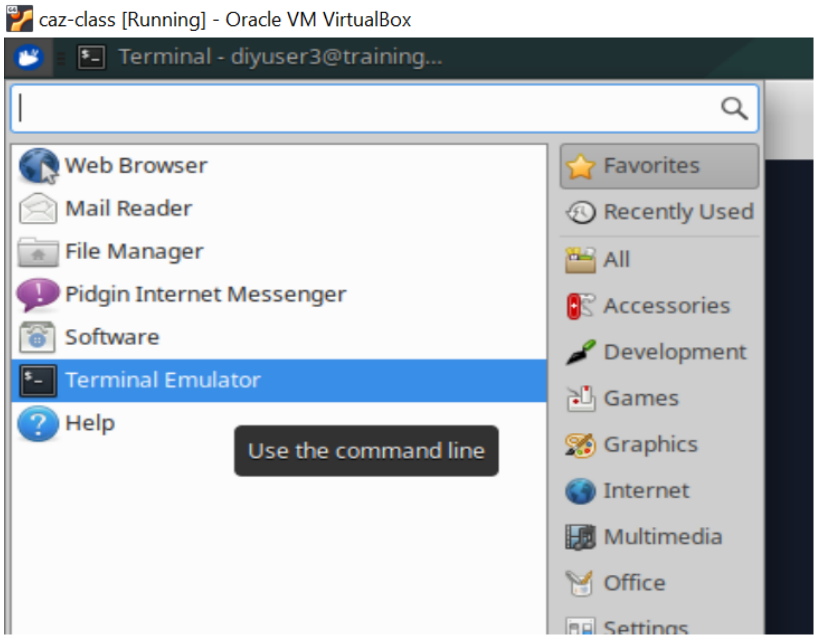
Version 2.1 by Brent Laster

6/16/2021

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 *ts-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!

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 ts-k8s-code

$ 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 - Pod Startup

Purpose: In this lab, we'll look at ways to identify and remediate issues with system resources when trying to get pods scheduled on nodes.

1. Change to the ***roar-ts1*** subdirectory (roar is the name of the sample app we'll be working with.) In this directory, we have Helm charts to deploy the database and webapp parts of our application. Use the "tree" command to view the structure of the directory tree and note that it has charts and pieces for both a webapp and a database piece. Each of these will have their own pod, deployment, and service.

```
$ cd roar-ts1
$ tree
```

2. Create a namespace named "ts" to hold the deployment and set it as the default.

```
$ k create ns ts

(The kubectl command is aliased to just "k" on
this machine.)
```

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

3. Finally, deploy the app into the cluster with the "helm install" command. (In the last command, the first occurrence of "ts" indicates the namespace, the second is the name of the release, and the "." means using the files from this directory.)

```
$ helm install -n ts ts .
```

4. Now let's see how things are progressing. Take a look at the overall status of the pods.

```
$ k get pods
```

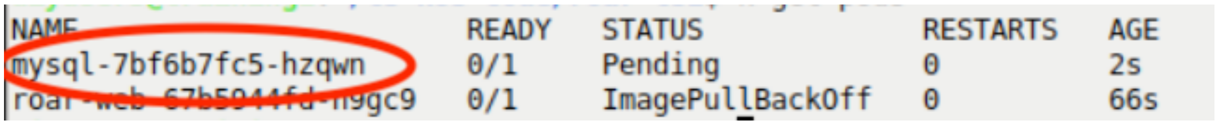
5. Notice that we only have one pod - one for roar-web (and it currently is in PENDING). Because this app has two parts, a webapp and a database piece, we should also have a pod for the mysql database piece. Since we don't have a pod to investigate, let's start one level up with the replicasets. Take a look at the replicasets. What do you notice about the one for the mysql piece?

```
$ k get rs
```

6. Notice that the line for the database replicaset (starting with "mysql-") has 0's in the DESIRED, CURRENT, and READY columns. We would assume it would be 1 for all of these. Take a look at the deployments and you'll see the same thing. While we could (and eventually should) go back and fix this in the chart, for the sake of time, we can simply try to scale this replicaset up to 1 for now. Then check to see that the pod shows up.

```
$ k get deploy

$ k scale deploy/mysql --replicas=1

$ k get pods
```

7. Although we can now see the mysql (database) pod, note that it is in PENDING state. Let's get the name of the mysql pod and do a describe on it to see more information. (You can simply highlight, copy, and paste the name from the list into the command line.)



```
NAME                        READY   STATUS             RESTARTS   AGE
mysql-7bf6b7fc5-hzqwn       0/1     Pending            0          2s
roar-web-67b5944fd-n9gc9    0/1     ImagePullBackOff   0          66s
```

```
$ k describe pod <mysql pod name>
```

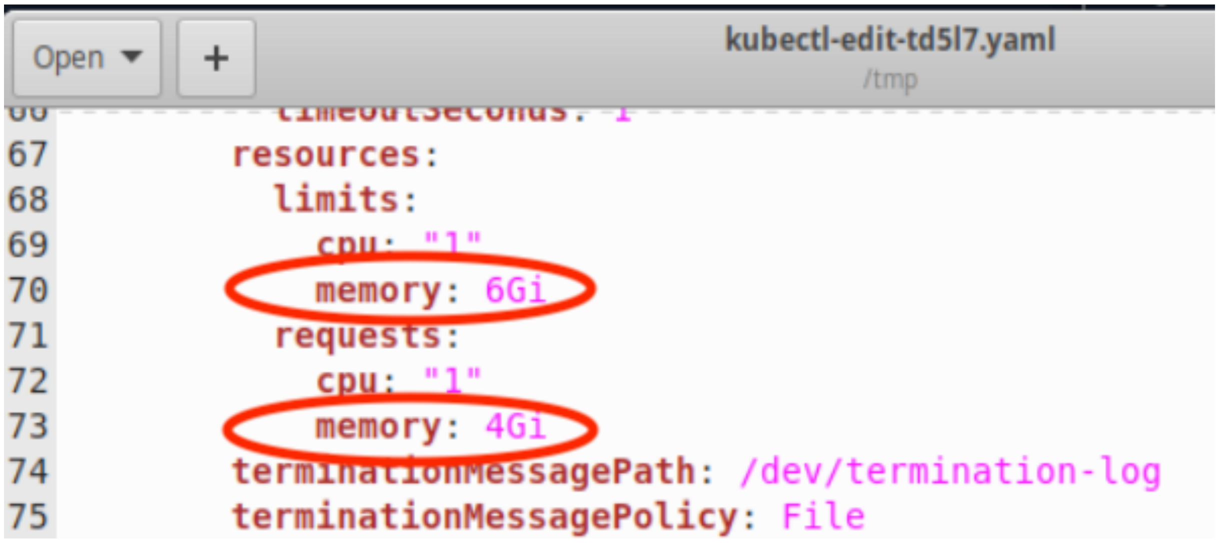
8. Notice near the bottom there's an Event that says *"0/1 nodes are available: Insufficient memory."* This means there is not enough memory on the node to schedule the pod. Run the commands below to see how much memory the pod is requesting and how much is available on the single node we have.

```
$ k get pod <mysql pod name> -o yaml | grep limits -A6

$ k describe node training1 | grep memory
```

9. Translating the 9992068Ki to Gi is roughly 10Gi. Our mysql pod is asking for 10Gi, but other processes running on the node in other namespaces could be using several Gi. Let's drop the limit and request values down to 6 and 4 respectively and see if that fixes things. We'll do this with the kubectl edit command to edit the deployment in place.

```
$ export EDITOR=gedit

$ k edit deploy/mysql
```

10. Change the two memory lines at line 70 and 73 as shown below. When you are done, save your changes and exit the editor.



```
67        resources:
68          limits:
69            cpu: "1"
70            memory: 6Gi
71          requests:
72            cpu: "1"
73            memory: 4Gi
74        terminationMessagePath: /dev/termination-log
75        terminationMessagePolicy: File
```

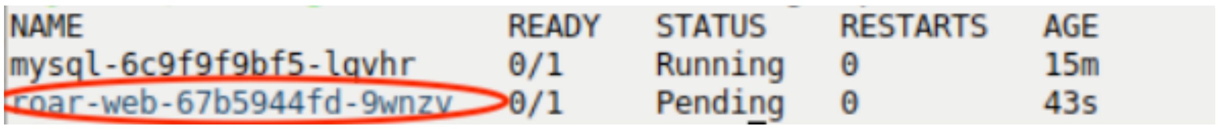
11. After a few moments, you should see a new mysql pod with a status of Running. At this point you can go ahead and delete any extra, old Replicasets that are out there and that will remove the extra old pod.

```
$ k get pods

(optional)$ k delete rs/<name of older mysql rs>
```

Lab 2- Understanding issues with node selection and extended Pod debugging

Purpose: In this lab, we'll look at ways to identify and remediate issues with getting scheduled on particular nodes and debug and fix why a pod won't start up.

1. Since we have our mysql pod in a Running state, let's switch focus to the web one. It's still in a Pending state, so let's do a describe to see if we can figure out what's wrong. Highlight and copy the NAME of the roar-web pod and then run the describe command below.



```
NAME                        READY   STATUS    RESTARTS   AGE
mysql-6c9f9f9bf5-lqvhr      0/1     Running   0          15m
roar-web-67b5944fd-9wnzv    0/1     Pending   0          43s
```

```
$ k describe pod <roar-web pod name>
```

2. At the bottom of the describe output, in the Events section, you should see an error similar to the following: "*Warning FailedScheduling 7m58s default-scheduler 0/1 nodes are available: 1 node(s) didn't match Pod's node affinity/selector.*" Node selection here is based on labels. Let's look at what label the pod wants and which ones our single node - training1 - has.

```
$ k describe pod <roar-web pod name> | grep Selector

$ k get nodes --show-labels
```

3. As you can see, there are a lot of labels on the node - added by the system. But there isn't one that matches the label the pod is expecting "type=mini". So, let's apply. one now. You can run a quick command to check afterwards.

```
$ k label node training1 type=mini

$ k get nodes --show-labels | grep type
```

4. Now, if you check the pod list a few times (*via k get pods*) you should eventually see that the web pod's status moves out of Pending. Unfortunately, it still has an error status of ErrImgPull or ImagePullBackOff. So, we need to solve this issue next.

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

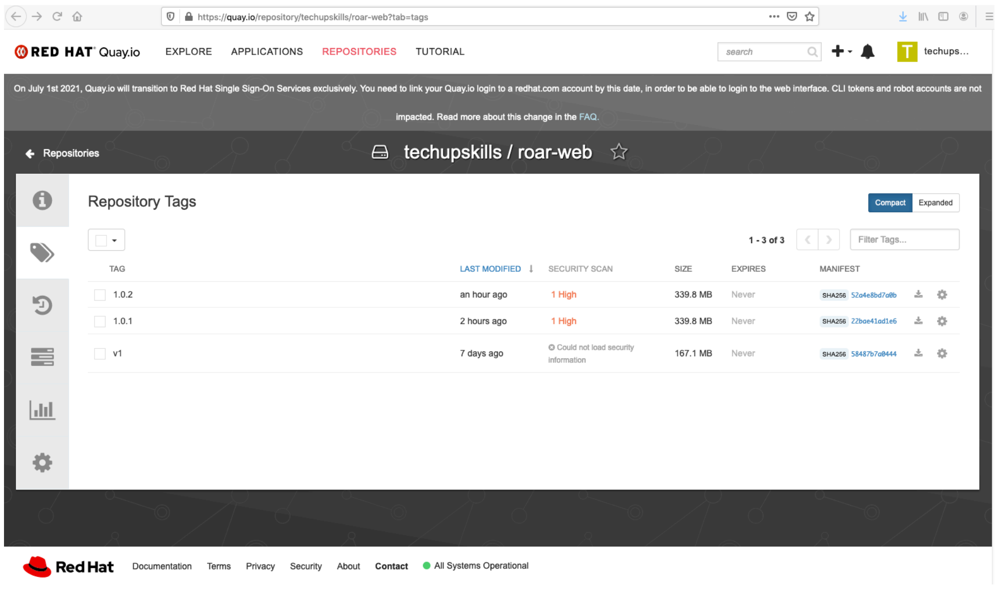
```
$ k logs <paste pod name here>
```

6. The output here confirms what is wrong – notice the part on "*trying and failing to pull image*". We need to get more detail though - such as the exact image name. We could use a describe command, but there's a shortcut using "get events" that we can do too.

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

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

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



On July 1st 2021, Quay.io will transition to Red Hat Single Sign-On Services exclusively. You need to link your Quay.io login to a redhat.com account by this date, in order to be able to login to the web interface. CLI tokens and robot accounts are not impacted. Read more about this change in the FAQ.

Repository Tags

TAG	LAST MODIFIED	SECURITY SCAN	SIZE	EXPIRES	MANIFEST
1.0.2	an hour ago	1 High	339.8 MB	Never	SHA256 52a4e8bd7a0b
1.0.1	2 hours ago	1 High	339.8 MB	Never	SHA256 22bae41ad1e6
v1	7 days ago	Could not load security information	167.1 MB	Never	SHA256 58487b7a0444

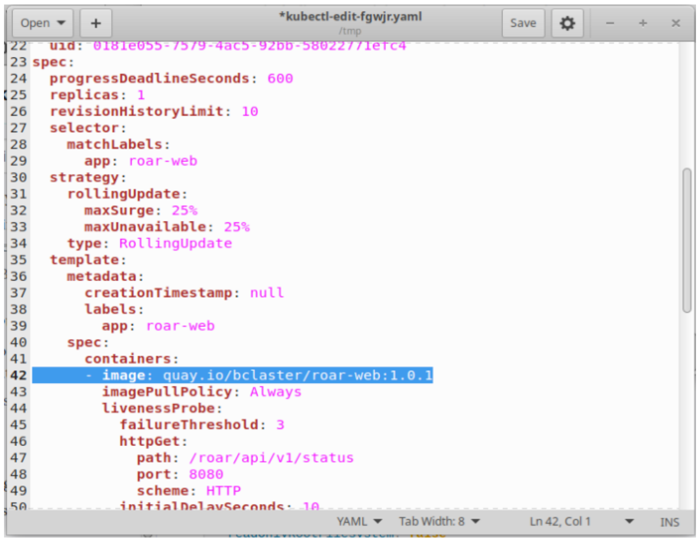
8. You can see that we don't have an image with the tag "1.10.1". Instead we have a "1.0.1". So there's probably a typo.

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)
```

```
$ export EDITOR=gedit

$ k edit deploy/roar-web
```

Change line 42 to use 1.0.1 instead of 1.10.1.



```
22   uid: 0181e055-7579-4ac5-92bb-58022771efc4
23 spec:
24   progressDeadlineSeconds: 600
25   replicas: 1
26   revisionHistoryLimit: 10
27   selector:
28     matchLabels:
29       app: roar-web
30   strategy:
31     rollingUpdate:
32       maxSurge: 25%
33       maxUnavailable: 25%
34     type: RollingUpdate
35   template:
36     metadata:
37       creationTimestamp: null
38       labels:
39         app: roar-web
40     spec:
41       containers:
42       - image: quay.io/bclaster/roar-web:1.0.1
43         imagePullPolicy: Always
44         livenessProbe:
45           failureThreshold: 3
46           httpGet:
47             path: /roar/api/v1/status
48             port: 8080
49             scheme: HTTP
50           initialDelaySeconds: 10
```

9. 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.

   You can also stop the watch in the other terminal window if you want.

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. Be sure to grab the name of the new pod from the last lab.

```
$ k describe pod <web pod name>

$ k logs <web pod name>
```

2. The describe doesn't tell us much that's meaningful. But the logs note several errors. Scroll back up through that output and eventually you'll find the first SEVERE error. It will look like the following and be right before the set of "at java.xml/com…" lines.

```
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.
```

3. 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
```

4. 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.)
```

5. 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. Do that with 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
```

6. 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
```

7. 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.
```

8. 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
```

9. 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.

```
$ k logs web-test
```

10. 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 the older mysql pod.

```
$ 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 <older mysql pod>
```

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. Highlight and copy the pod name and paste it in place of the "<mysql-pod-name>" section in the command below.

```
$ k get pods

$ k describe pod <mysql-pod-name>
```

2. 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.

3. 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
```

4. 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.

5. 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.

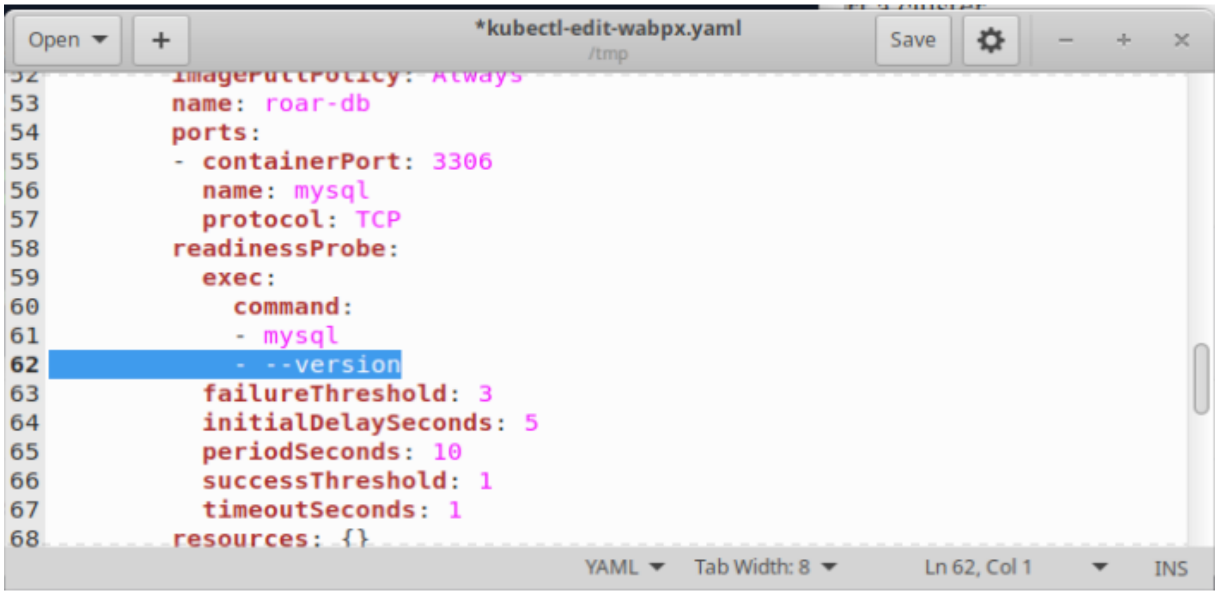
```
$ k edit deploy/mysql
```

And change (around line 58)

```
readinessProbe:
  exec:
    command:
    - mysql
  failureThreshold: 3
  initialDelaySeconds: 5
```

To add the line shown in bold

```
readinessProbe:
  exec:
    command:
    - mysql
    - --version
  failureThreshold: 3
  initialDelaySeconds:  5
```



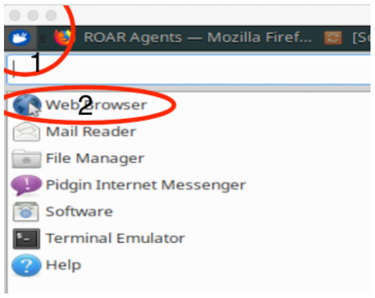
```
        imagePullPolicy: Always
        name: roar-db
        ports:
        - containerPort: 3306
          name: mysql
          protocol: TCP
        readinessProbe:
          exec:
            command:
            - mysql
            - --version
          failureThreshold: 3
          initialDelaySeconds: 5
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 1
        resources: {}
```

6. 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
```

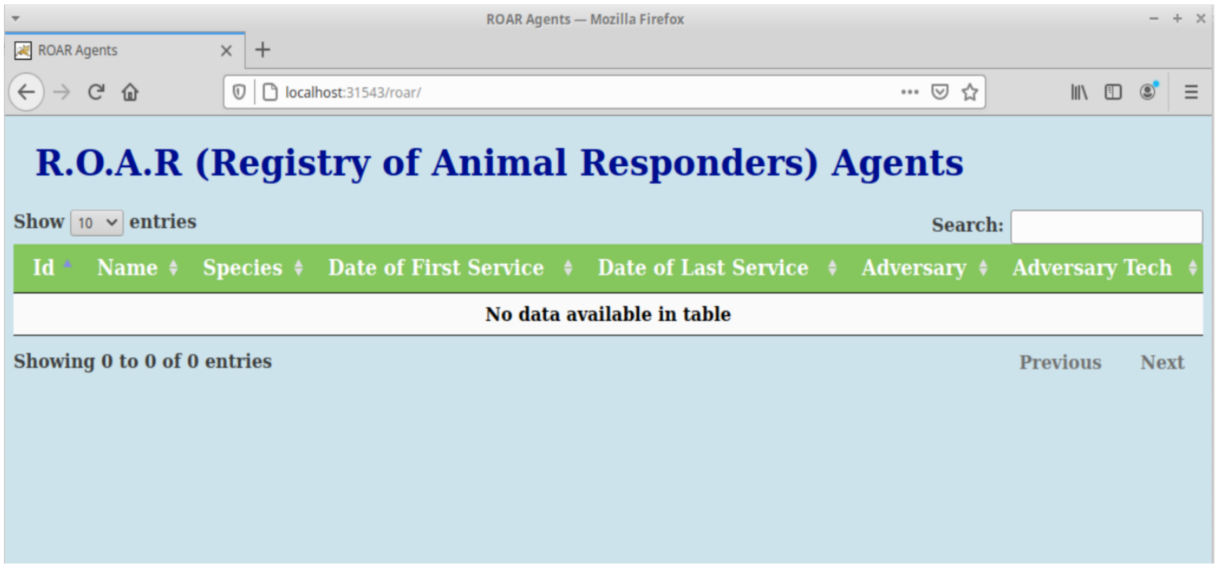
Look for the port > 30000 after the 8089: in the roar-web line. Plug that value in for <port> below and open the URL up in a browser.

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



7. 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.

```
$ k logs <mysql pod name>

$ k describe pod <mysql pod name>
```



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

Show 10 entries Search:

Id	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
No data available in table						

Showing 0 to 0 of 0 entries Previous Next

8. 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.

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

9. 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.)

10. 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 do some general operation like that is to 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
```

2. 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')
```

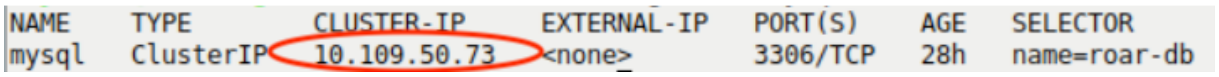
3. Now, let's check to see if we can see our database services We'll use a similar command. Run the command in bold below and you should see similar output.

```
/ # 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
```

4. 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
```



```
NAME    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   SELECTOR
mysql   ClusterIP   10.109.50.73   <none>        3306/TCP   28h   name=roar-db
```

5. 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
```

6. 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
```

7. 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"
        }
```

8. 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
```

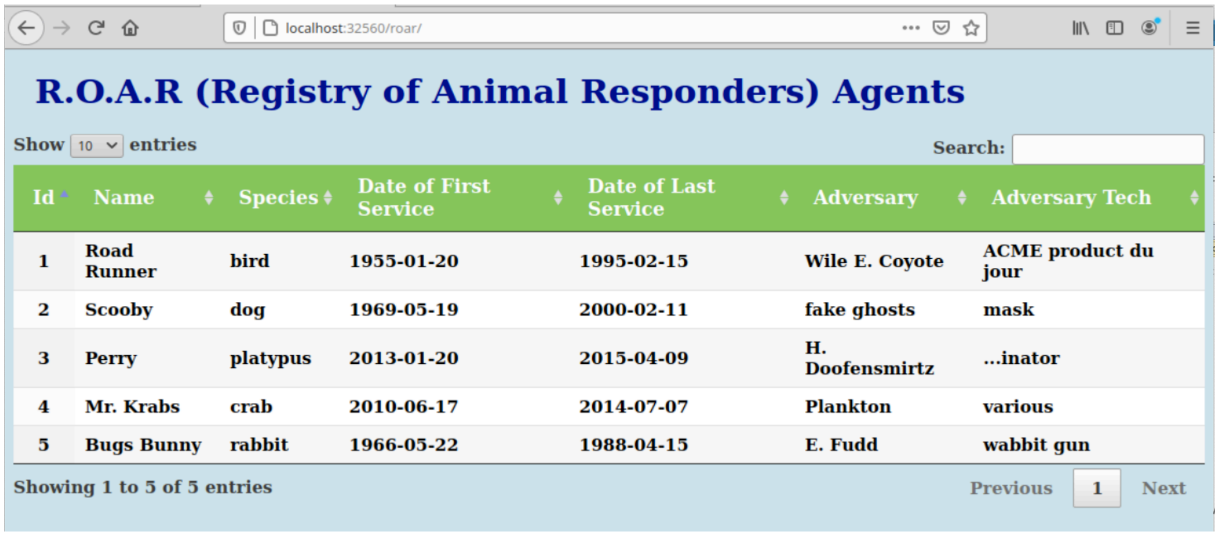
9. There is not a match for *name=roar-db.* But the pod does have an app=mysql label. So we could either add the label in the deployment that the service is looking for (it will spin up a new pod) or add the pod's label in the service. We'll do the former using another method - *kubectl patch*. Note: 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"}]'
```

10. 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
```

11. 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 working.



localhost:32560/roar/

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

Show 10 entries Search:

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

Showing 1 to 5 of 5 entries Previous 1 Next

<u>END OF LAB</u>