Advanced Kubernetes

Gaining Mastery over your Deployments

Session Labs: Revision 1.2 - 12/16/20

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 adv-k8s-setup.pdf in http://github.com/brentlaster/conf/nfjs2020 for instructions and other things to be aware of.

Lab 1: Working with Kubernetes Probes

Purpose: In this lab, we'll learn how Kubernetes uses probes for determining the health of pods, how to set them up, and how to debug problems around them.

1. To run the examples in this set of labs, we need a Kubernetes instance. Start up a minikube instance on the machine by opening a terminal emulator window and entering the command below.

```
$ ./adv-k8s/extra/start-kube.sh
```

2. For this workshop, files that we need to use are contained in the directory **adv-k8s** in the home directory on the disk. Under that directory are subdirectories for each of the main topics that we will cover. For this section, we'll use files in the **roar-probes** directory. (roar is the name of the sample app we'll be working with.) Change into that directory. In the terminal emulator, enter

```
$ cd adv-k8s/roar-probes
```

3. In this directory, we have Helm charts to deploy the database and webapp parts of our application. You can use the "tree" command to see the overall structure if you are interested. Then create a namespace named "probes" to hold the deployment. Finally, deploy the app into the cluster with the "helm install" command. (In the last command, the first occurrence of "probes" indicates the namespace, the second is the name of the release, and the "." means using the files from this directory.)

```
$ tree
```

```
$ k create ns probes
```

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

```
$ helm install -n probes probes .
```

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

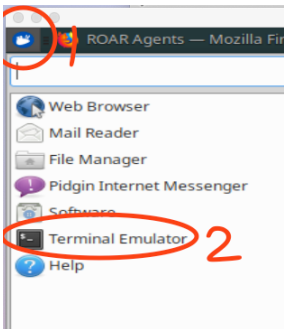
```
$ k get pods -n probes
```

5. You should see that while the web pod is running, the database pod is not ready at all. (If the web pod isn't "Running" yet, it may still need time to startup. You can run the command again after a minute or so to give it time to finish starting up.). Now, let's do a "describe" operation on the pod itself. Highlight and copy the pod name and paste it in place of the "<mysql-pod-name>" section in the command below.

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

6. 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.

7. You can see the YAML for this in the deployment template in the corresponding Helm chart. In a separate terminal window, take a look at that and find the section near the bottom with the **readinessProbe** spec.




```
$ cat ~/adv-k8s/roar-probes/charts/roar-db/templates/deployment.yaml
```

8. 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 simply calling the "version" command - which we should be able to do without a login.

9. You can choose to edit the deployment file with the "gedit" editor. Or you can use the "meld" tool to add the differences from a file that already has them. If using the meld tool, select the left arrow to add the changes from the second file into the deployment.yaml file. Then save the changes.

   Switch to ~/adv-k8s/roar-probes if not already there.

   ```
   $ cd ~/adv-k8s/roar-probes
   ```

   Either do:

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

   And change

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

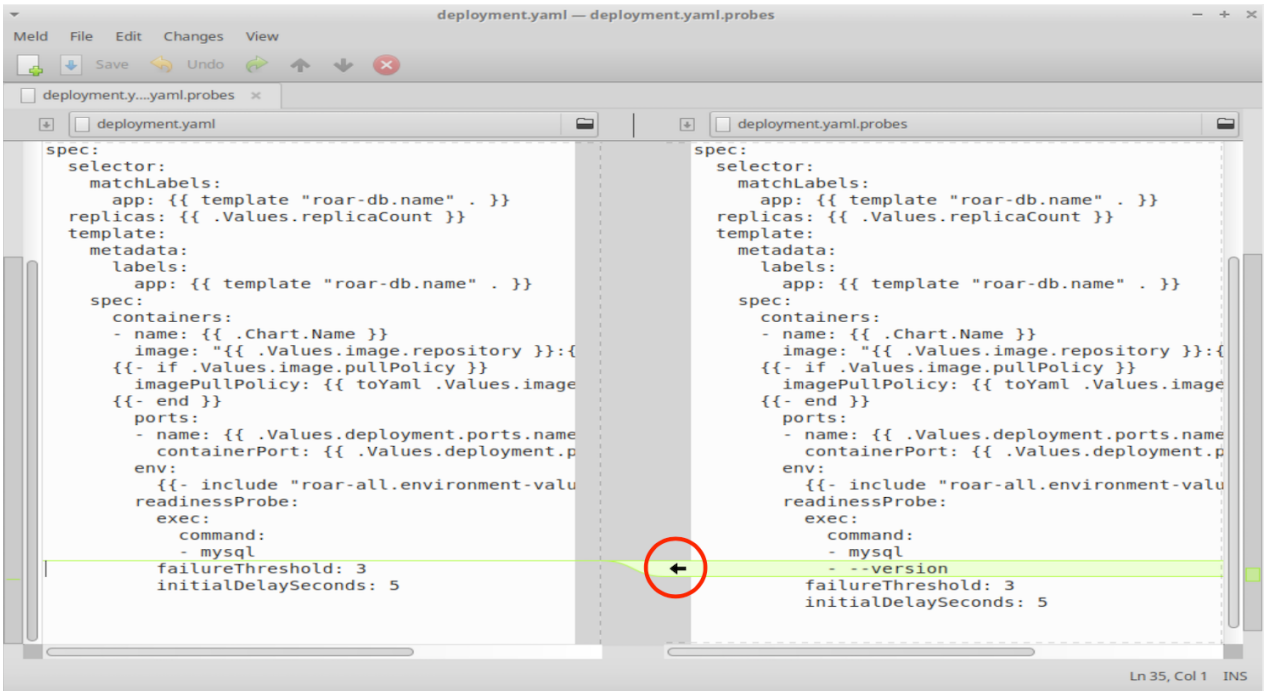
To add the line shown in bold

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

Or:

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

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



10. Upgrade the helm installation. After a minute or so, you can verify that you have a working mysql pod. (You may have to wait a moment and then check again.)

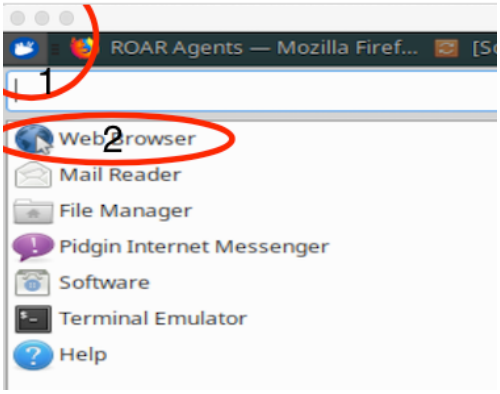
```
$ helm upgrade -n probes probes .
```

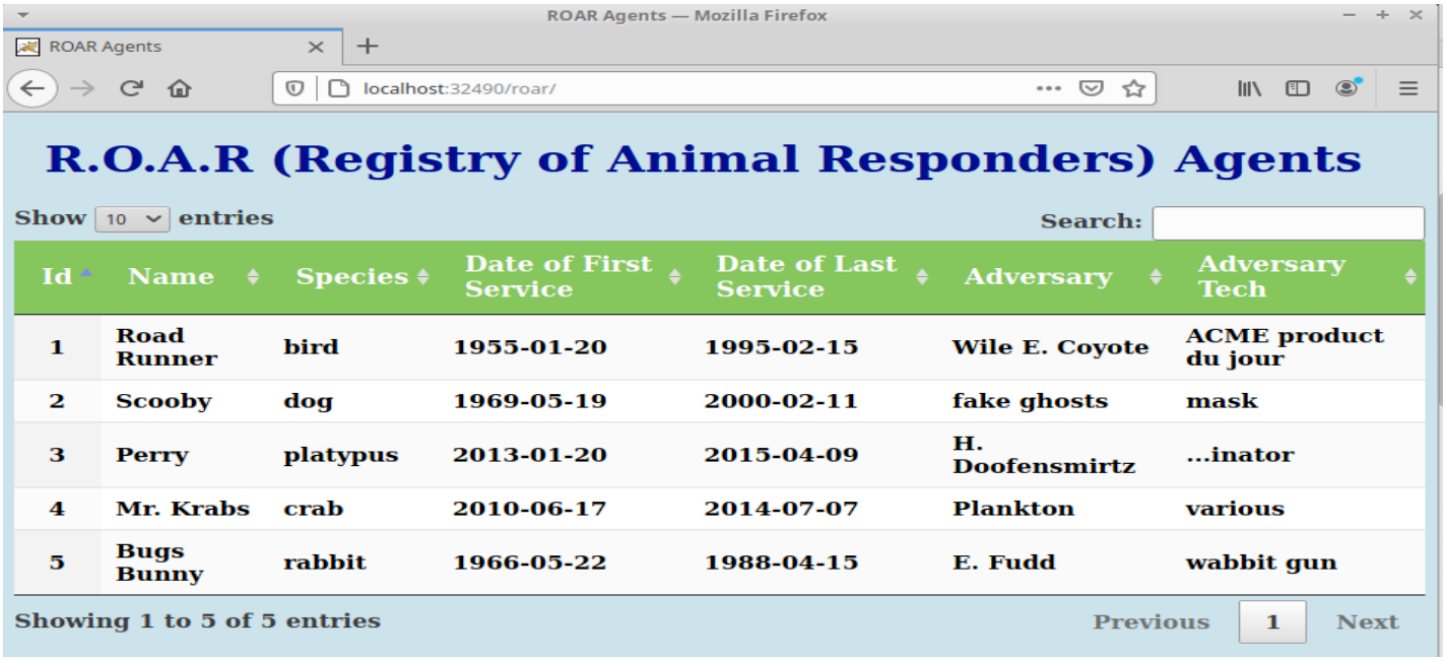
11. At this point, you can get the service's nodeport and then open up a browser on localhost to the URL below to see the application running.

```
$ k get svc -n probes
```

Look for the port > 3000 after the 8089: in the roar-web line. Plug that value in for <port> below.

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





END OF LAB

Lab 2: Working with Quotas

Purpose: In this lab, we'll explore some of the simple ways we can account for resource usage with pods and nodes, and setup and use quotas.

1. Before we launch any more deployments, let's set up some specific policies and classes of pods that work with those policies. First, we'll setup some priority classes. Take a look at the definition of the pod priorities and then apply the definition to create them.

```
$ cd ~/adv-k8s/extra
$ cat pod-priorities.yaml
$ k apply -f ./pod-priorities.yaml
```

2. Now, that the priority classes have been created, we'll create some resource quotas built around them. Since quotas are namespace-specific, we'll go ahead and create a new namespace to work in. Take a look at the definition of the quotas and then apply the definition to create them.

```
$ k create ns quotas
$ cat pod-quotas.yaml
$ k apply -f ./pod-quotas.yaml -n quotas
```

3. After setting up the quotas, you can see how things are currently setup and allocated.

```
$ k get priorityClasses
$ k describe quota -n quotas
```

4. In the roar-quota directory we have a version of our charts with requests, limits and priority classes assigned. You can take a look at those by looking at the end of the deployment.yaml templates. After that, go ahead and install the release.

```
$ cd ~/adv-k8s/roar-quotas
$ cat charts/roar-db/templates/deployment.yaml
$ cat charts/roar-web/templates/deployment.yaml
$ helm install -n quotas quota .
```

5. After a few moments, take a look at the state of the pods in the deployment. Notice that while the web pod is running, the database one does not exist. Let's figure out why. Since there is no pod to do a describe on, we'll look for a replicaset.

```
$ k get pods -n quotas
$ k get rs -n quotas
```

6. Notice the mysql replicaset. It has DESIRED=1, but CURRENT=0. Let's do a describe on it to see if we can find the problem.

```
$ k describe -n quotas rs <mysql rs name>
```

7. What does the error message say? The request for memory we asked for the pod exceeds the quota for the quota "pods-average". If you recall, the pods-average one has a memory limit of 5Gi. The pods-critical one has a higher memory limit of 10Gi. So let's change priority class for the mysql pod to be critical.

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

change the last line from

```
priorityClassName: average
```

to

```
priorityClassName: critical
```

being careful not to change the spaces at the start of the line.

Save your changes and exit the editor. (Ignore the Gtk-WARNING message)

8. Upgrade the Helm release to get your changes deployed and then look at the pods again.

```
$ helm upgrade -n quotas quota .
$ k get pods -n quotas
```

9. Notice that while the mysql pod shows up in the list, its status is "Pending". Let's figure out why that is by doing a describe on it.

```
$ k describe -n quotas pod/<mysql pod name>
```

10. The error message indicates that there are no nodes available with enough memory to schedule this pod. Note that this does not reference any quotas we've setup. Let's get the list of nodes (there's only 1 here) and check how much memory is available on our node.

```
$ k get nodes
$ k describe node training1 | grep memory
```

11. Translating the 9991364Ki to Gi is roughly 10Gi. Our mysql pod is asking for 10Gi, but the web pod is already claiming 1Gi and other processes running on the node in other namespaces will be using several Gi. Let's drop the request down to 6Gi and see if that fixes things.

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

change the two lines near the bottom from

```
memory: "10Gi"
```

to

```
memory: "5Gi"
```

Save your changes and exit the editor.

12. Do a helm upgrade and add the "--recreate-pods" option to force the pods to be recreated. After a moment if you check, you should see the pods running now. Finally, you can check the quotas again to see what is being used.

```
$ helm upgrade -n quotas quota --recreate-pods .
(ignore the deprecated warning)
$ k get pods -n quotas
$ k describe quota -n quotas
```

END OF LAB

Lab 3: Selecting Nodes

Purpose: In this lab, we'll explore some of the ways we can tell Kubernetes which node to schedule pods on

1. The files for this lab are in the roar-affin subdirectory. Change to that, create a namespace, and do a Helm install of our release.

```
$ cd ~/adv-k8s/roar-affin
$ k create ns affin
$ helm install -n affin affin .
```

2. Take a look at the status of the pods in the namespace. You'll notice that they are not ready. Let's figure out why. Start with the mysql one and do a describe on it.

```
$ k get pods -n affin
$ k describe -n affin pod <mysql pod name>
```

3. In the output of the describe command, in the Events section, you can see that it failed to be scheduled because there were "0/1 nodes are available: 1 node(s) didn't match node selector". And further up, you can see that it is looking for a Node-Selector of "type=mini".

4. This means the pod definition expected at least one node to have a label of "type=mini". Take a look at what labels are on our single node now.

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

5. Since we don't have the desired label on the node, we'll add it and then verify it's there.

```
$ k label node training1 type=mini
$ k get nodes --show-labels | grep type
```

6. At this point, if you look again at the pods in the namespace you should see that the mysql pod is now running. Also, if you do a describe on it, you'll see an entry in the Events: section where it was scheduled.

```
$ k get pods -n affin
```

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

7. Now, let's look at the web pod. If you do a describe on it, you'll see similar messages about problems scheduling. But the node-selector entry will not list one. This is because we are using the node affinity functionality here. You can see the affinity definition by running the second command below.

```
$ k describe pod -n affin <web pod name>
$ k get -n affin pod <web pod name> -o yaml | grep affinity -A10
```

8. In the output from the grep, you can see that the nodeAffinity setting is "*requiredDuringSchedulingIgnoredDuringExecution*" and it would match up with a label of "*system=minikube*" or "*system=single*". But let's assume that we don't really need a node like that, it's only a preference. If that's the case we can change the pod spec to use "*preferredDuringSchedulingIgnoredDuringExecution*". To do that you can either edit the deployment.yaml file directly

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

and change

```
requiredDuringSchedulingIgnoredDuringExecution:
  nodeSelectorTerms:
  - matchExpressions:
```

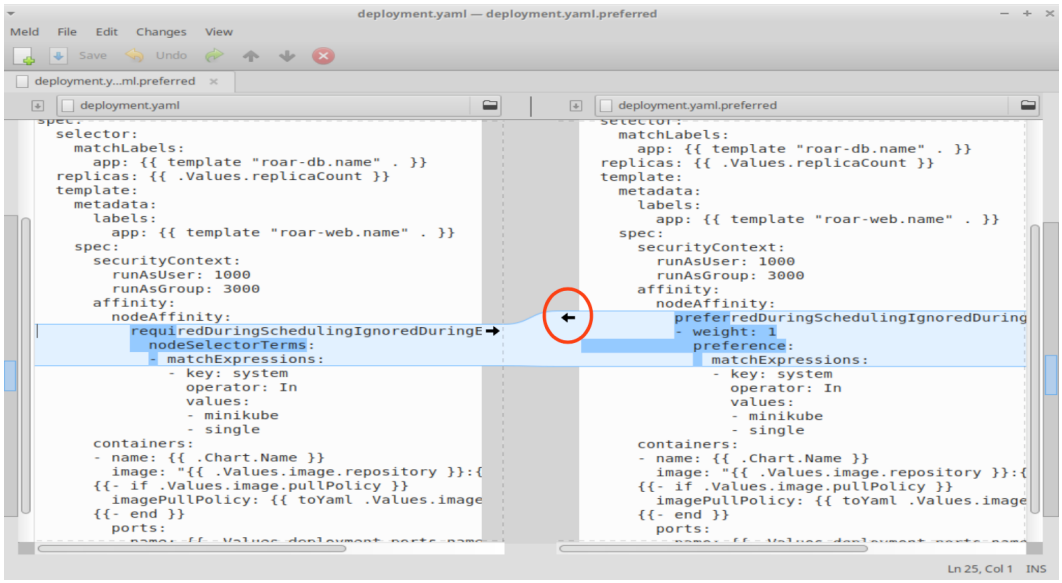
to

```
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 1
  preference:
    matchExpressions:
```

Or, you can use the meld tool to merge in the changes from a file in the extra area. Just issue the command below, click on the right arrow (the one circled in the figure) and save the files and exit meld.

(This assumes you are in the adv-k8s/roar-affin directory.)

```
$ meld charts/roar-web/templates/deployment.yaml ../extra/deployment.yaml.preferred
```



9. Now, upgrade the deployment with the recreate-pods option to see the changes take effect.

```
$ helm upgrade -n affin affin --recreate-pods .
```

10. After a few moments, you should be able to get the list of pods and see that the web one is running now too. You can do a describe if you want and see that it has been assigned to the training1 node since it was no longer a requirement to match those labels.

```
$ k get pods -n affin
```

END OF LAB

Lab 4: Working with Taints and Tolerations

Purpose: In this lab, we'll explore some of the uses of taints and tolerations in Kubernetes

1. The files for this lab are in the roar-taint subdirectory. Change to that, create a namespace, and do a Helm install of our release.

```
$ cd ~/adv-k8s/roar-taint
$ k create ns taint
$ helm install -n taint taint .
```

2. At this point, all pods should be running because there are no taints on the node. (You can do a get on the pods to verify if you want.). Let's add a taint on the node that implies that pods must be part of the roar app to be scheduled.

```
$ k taint nodes training1 roar=app:NoSchedule
```

3. Now, let's delete the release and install again. Then take a look at the pods.

```
$ helm delete -n taint taint
$ helm install -n taint taint .
$ k get pods -n taint
```

4. The web pod has failed to be scheduled. Do a describe to see why.

```
$ k describe -n taint pod <web pod name>
```

5. Notice that it says "*1 node(s) had taints that the pod didn't tolerate.*" So our database pod must have had a toleration for it since it was running. Take a look at the two tolerations in the database pod (at the end of the deployment.yaml file).

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

6. Notice the toleration for "*roar*" and "*Exists*". This says that the pod can run on the node even if the taint we created in step 2 above is there - regardless of the value. We need to add this to our web pod spec so it can run there as well.

   You can do this either by editing the file directly

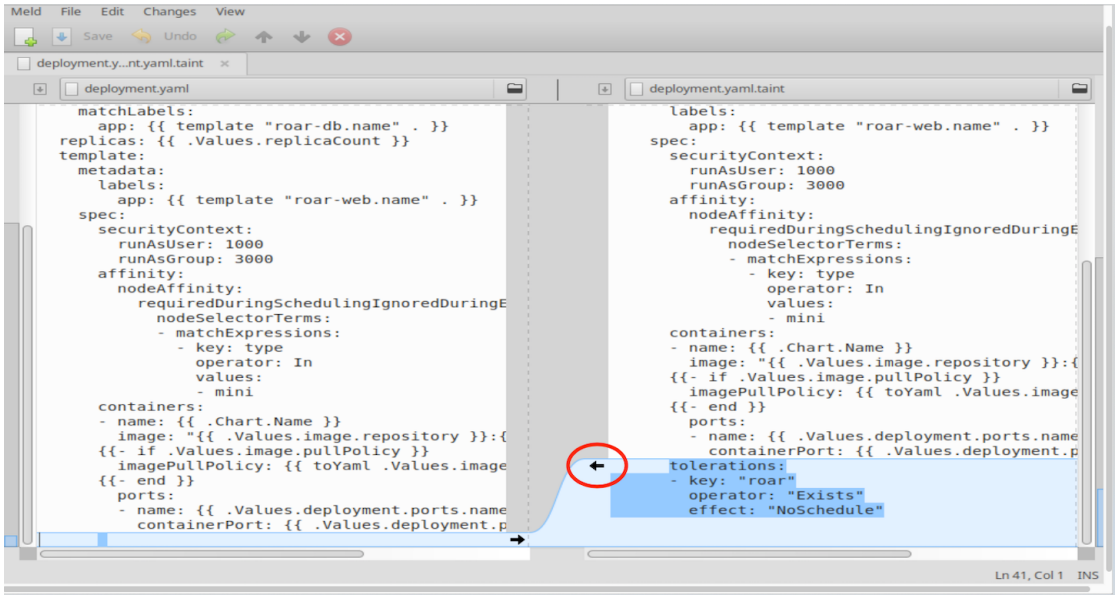
```
$ gedit charts/roar-web/templates/deployment.yaml
```

   and adding these lines (lining up with the same starting column as "containers:")

```
tolerations:
 - key: "roar"
   operator: "Exists"
   effect: "NoSchedule"
```

Or you can use the meld tool by running the following command, clicking on the right arrow (the one circled) and then saving the file and exiting meld.

```
$ meld charts/roar-web/templates/deployment.yaml ../extra/deployment.yaml.taint
```



7. Now with the toleration added for the web pod, do an upgrade to see if we can get the web pod scheduled now.

```
$ helm upgrade -n taint taint  .
$ k get pods -n taint
```

8. Now let's add one more taint for the other toleration that the mysql pod had. Afterwards, take a look at the state of the pods.

```
$ k taint nodes training1 use=database:NoExecute

$ k get pods -n taint
```

9. Why is the web pod not running? The database pod has a toleration for this taint. You can see that in the charts/roar-db/templates/deployment.yaml file near the bottom. You can also do a describe on the web pod again if you want to see that it didn't tolerate the new taint.

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

```
$ k describe -n taint pod <web pod name>
```

10. But the web pod doesn't have this toleration, so because of the "No Execute" policy, it gets kicked out. We could add a toleration to the web pod spec for this, but for simplicity, let's just remove the taint to get things running again.

```
$ k taint nodes training1 use:NoExecute-
$ k get pods -n taint
```

11. Go ahead and remove the other taint to prepare for future labs.

```
$ k taint nodes training1 roar:NoSchedule-
```

END OF LAB

Lab 5: Working with Security Contexts and RBAC

Purpose: In this lab, we'll learn more about what a pod security context is and why they are needed. We'll also see how to create a service account with RBAC.

1. The files for this lab are in the roar-context subdirectory. Change to that, create a namespace, and do a Helm install of our release.

```
$ cd ~/adv-k8s/roar-context
$ k create ns context
$ helm install -n context context .
```

2. If you look at the pods in the namespace, you'll see that the mysql pod is running, but the web pod is not. Do a describe on the web pod to see what the problem is.

```
$ k get pods -n context
$ k describe -n context pod <web pod name>
```

3. In the "Events" section, you'll see an error like "*Error: container has runAsNonRoot and image has non-numeric user (tomcat), cannot verify user is non-root*". This happens because we have a Pod Security Policy (discussed in next section) that specifies that pods can't run as the root user. We could go back and update the container to have a numeric non-root user. Or we can add a Security Context definition in our pod spec. We'll add the Security Context definition. You can use either of the following approaches.

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

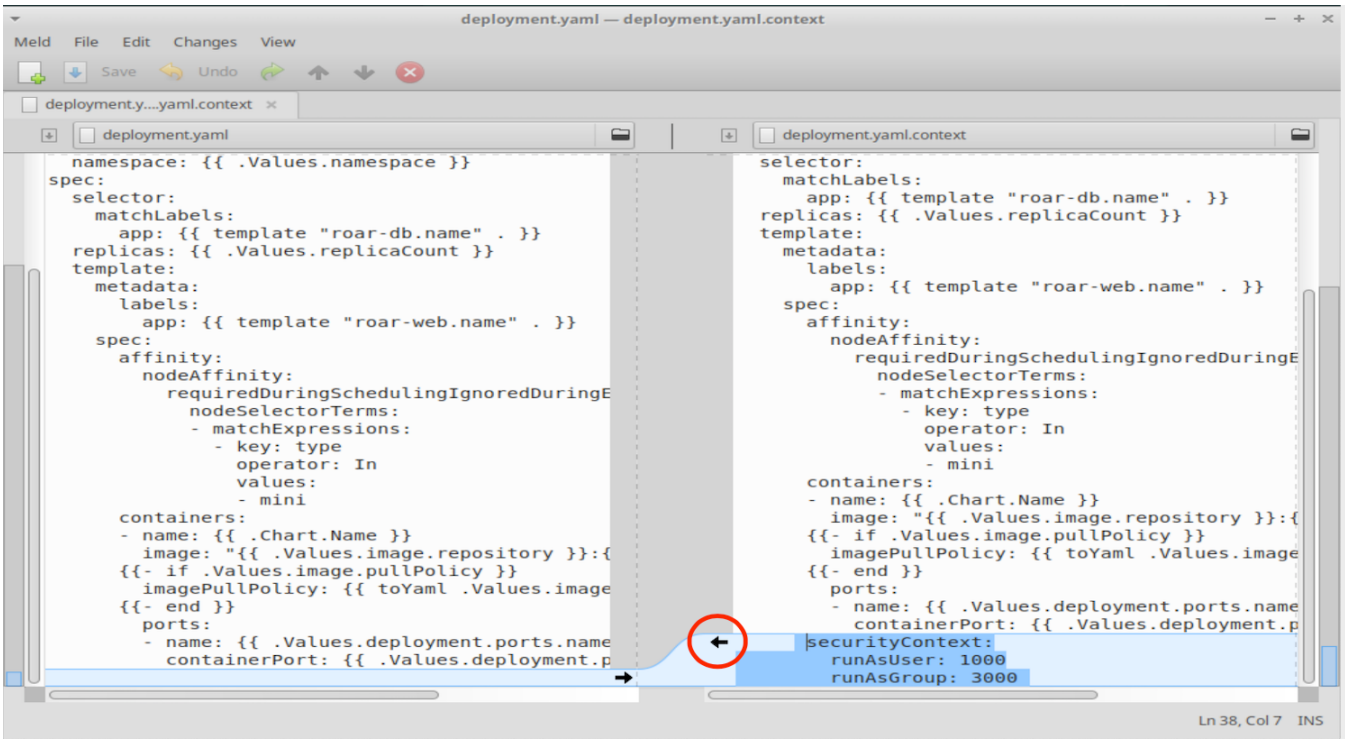
and add the following lines at the bottom (ensuring they line up with the "containers:" column)

```
securityContext:
  runAsUser: 1000
  runAsGroup: 999
```

OR

```
$ meld charts/roar-web/templates/deployment.yaml ../extra/deployment.yaml.context
```

In meld, click on the right arrow, save your changes and close the application.



4. Once you've made the changes to the pod spec, go ahead and redeploy the Helm chart. Both pods should now run as expected. (Note that you may need to refresh a time or two to see the Running state.)

```
$ helm upgrade -n context context --recreate-pods .
$ k get pods -n context
```

5. In addition to controlling access and authorization for pods with security contexts, we can use RBAC to control access and authorization in Kubernetes clusters and namespaces. For the next lab, we'll need a service account with limited access for some of the examples. Since this will be tied to a new namespace, go ahead and create the namespace and then create the service account.

```
$ k create ns policy
$ k create sa -n policy roar-account
```

6. To keep things simple, we'll bind this account to an existing, built-in role that's available in the cluster - *edit*. Let's take a look at what this role includes:

```
$ k describe clusterrole edit
```

7. To do the actual binding, we'll create a *rolebinding* object. This will create a *rolebinding* named roar-editor that allows our roar-account to have the *cluster edit* permissions.

```
$ k create rolebinding -n policy roar-editor --clusterrole=edit
--serviceaccount=policy:roar-account
```

8. Next, we'll create a new *role* named "*roar-policies*" in the "*policy*" namespace that allows for the "use" verb to be done against a *pod security policy* that we'll create in the next lab.

```
$ k create role -n policy policy:roar-policies   --verb=use
--resource=podsecuritypolicies.extensions
--resource-name=roar-policies
```

9. Finally, we'll create some aliases to allow us to run with and without the service account.

```
$ alias k-admin='k -n policy'

$ alias k-editor='k --as=system:serviceaccount:policy:roar-account -n policy'
```

END OF LAB

Lab 6: Working with Pod Security Policies

Purpose: In this lab, we'll learn more about what a pod security policy is and how they are used.

1. The files for this lab are in the roar-psp subdirectory and do a Helm install of our release. (We created a namespace to use in the last lab.)

```
$ cd ~/adv-k8s/roar-psp
$ helm install -n policy policy .
```

2. Take a look at the pods that are running in the namespace. You'll notice that only the web pod is running. This chart does not include a deployment for the database one. We've broken that one out as a separate manifest so we can try some things out with pod security policies. You can see the definition in extra/roar-db-pod.yaml.

```
$ k-admin get pods
$ cat ../extra/roar-db-pod.yaml
```

3. Take a look at what pod security policies we have in place currently. We're going to actually delete these to see what it's like when we don't have any.

```
$ k get psp

$ k delete psp restricted

$ k delete psp privileged
```

4. Let's try to deploy the pod yaml to create the missing pod. We can try it with both the admin account and the one tied to our particular service account. In both cases, you'll see an error like " *Error from server (Forbidden): error when creating "roar-db-pod.yaml": pods "mysql" is forbidden: no providers available to validate pod request*" because there are no psps currently defined.

```
$ k-admin apply -f ../extra/roar-db-pod.yaml

$ k-editor apply -f ../extra/roar-db-pod.yaml
```

5. Now, we'll create our custom psp. Take a look at the definition of our custom one (named roar-policies) and create it. (Note the hostPort settings in the definition.) Then you can do a describe to see more details.

```
$ cat ../extra/psp.yaml

$ k-admin apply -f ../extra/psp.yaml

$ k describe psp roar-policies
```

6. Try to create the pod now that we have our new psp in place. You will see error messages but note what is different between them.

```
$ k-admin apply -f ../extra/roar-db-pod.yaml

$ k-editor apply -f ../extra/roar-db-pod.yaml
```

7. Why did one look like it could access our policy and one couldn't? Use the "auth can-i" command to see if they can actually use the policy?

```
$ k-admin auth can-i use podsecuritypolicy/roar-policies

$ k-editor auth can-i use podsecuritypolicy/roar-policies
```

8. At this point, we could create a *rolebinding* to bind the *serviceaccount* used by the k-editor alias to the *role* that has "use" access to our pod security policy. However, we can also just create a *rolebinding* for the default service account of the pod. The second option is more secure. We'll do both here.

```
$ k-admin create rolebinding roar-account:policy:roar-policies
--role=policy:roar-policies --serviceaccount=policy:roar-account
```

```
$ k-admin create rolebinding default:policy:roar-policies
--role=policy:roar-policies --serviceaccount=policy:default
```

9. Now, we can run the command again for k-editor and see that it is referencing our policy (note the host port error).

```
$ k-editor apply -f ../extra/roar-db-pod.yaml
```

10. While we can now access the psp, we still have the problem with host ports. To correct that, we can go back and edit the psp.yaml file in extra and then apply it again.

```
$ gedit ../extra/psp.yaml
```

Edit the line for max hostPort to be 3306. Save the changes and exit the editor.

```
$ k apply -f ../extra/psp.yaml
```

11. If we run the command for the pod again, it should succeed.

```
$ k-editor apply -f ../extra/roar-db-pod.yaml
```

12. Finally, if we check the yaml for the pod, we can see the associated psp.

```
$ k-editor get pod mysql -o yaml | grep psp
```

END OF LAB