Helm Fundamentals

Class Labs: Revision 2.3 - 2/1/21

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 helm-fun-setup.pdf in http://github.com/brentlaster/safaridocs for instructions and other things to be aware of. You should also have a GitHub account.

Lab 1: Repos and Charts

Purpose: In this lab, we'll start working with Helm by adding a repo and pulling down a Helm chart, installing it, and then updating the release.

NOTE: Commands here start after the "$" and are intended to be run in a terminal session on the VM.

1. First, let's verify what version of Helm we have installed (should be 3.0+)

```
$ helm version
```

2. Now let's add a couple of public chart repositories

```
$ helm repo add stable https://charts.helm.sh/stable
$ helm repo add cm https://chartmuseum.github.io/charts
```

3. Verify that the charts were installed by listing the repos Helm knows about.

```
$ helm repo list
```

4. You should see the stable and cm repo listed. Now let's see what charts are available for download from these.

```
$ helm search repo stable   (What do you notice about all the charts here?)
$ helm search repo cm
```

5. There are a lot of charts available. We're going to setup **chartmuseum** - a repository for Helm charts - so we'll have our own local repository. First, let's find the available ChartMuseum versions.

```
$ helm search repo chartmuseum
```

The output should show you the currently available versions of the chart across repos.

6. Let's get some general information about the most recent version of the chart.

```
$ helm show chart cm/chartmuseum
```

7. Verify that no releases are currently installed on the cluster.

```
$ helm list
```

8. Now let's install chartmuseum locally. Recall that we need to give it a name when we install.

```
$ helm install local-chartmuseum cm/chartmuseum --version 2.15.0
```

9. Verify that the chart installation created a release in the cluster. What revision is it?

```
$ helm list
```

10. Notice the output you got when you did the helm install in step 8. This is telling us that in order to access this in the browser, we need to forward the port from within the cluster out. To save you typing/copying all of this, there is already a simple script that does this in the home directory named ./cm-port.sh. Run the script.

```
$ ./cm-port.sh
```

11. Verify that chartmuseum is up and running and accessible by opening up a browser from the upper left menu (see screenshot) and selecting web browser. Then **go to localhost:8080** and ensure you see the chartmuseum welcome message.

Lab 2: Changing values

Purpose: In this lab, you'll get to see how we can change values and upgrade releases through Helm, as well as learn some more Helm commands.

1. While we have ChartMuseum running, it would be nice not to have to also run the port forwarding script. The service that runs in the Kubernetes cluster for ChartMuseum is defaulted to be of type "ClusterIP" - mainly intended for traffic internal to the cluster. To see that, open a second terminal window and run the command to get the service info for the namespace where ChartMuseum is running. (The Kubernetes command line application, kubectl, is aliased to "k" on this machine.)

   ```
   $ k get svc
   ```

   Notice the output indicating the type of service is ClusterIP.

2. To be able to change this, we'll want to change the service type for our chartmuseum instance to be NodePort instead of ClusterIP. That will let us have a node in the range 30000-32767 that can be used for access. Let's first see what values we have that can be changed in our chartmuseum chart

   ```
   $ helm show values cm/chartmuseum
   ```

3. Lots of information there, but we'd like to change only the chart type and assign a node port. Let's look for the information around the text "ClusterIP"

   ```
   $ helm show values cm/chartmuseum | grep -n10  ClusterIP
   ```

4. Notice under the output, in the "service" section, we have the "type" set to ClusterIP and an empty setting for "nodePort".

5. Remind yourself of what release you currently have out there.

```
$ helm list
```

6. Go ahead and stop the port-forwarding that was running in the other window (via Ctrl-C).

```
$ Ctrl-C
```

7. We want to upgrade the service.type and service.nodePort values in our Helm release. How do we do that on the command line? Take a look at the first part of the help for helm upgrade.

```
$ helm upgrade --help | head
```

8. Notice the last line about being able to use the --set option to override values from the command line. We'll run an upgrade and try that - explicitly setting the service.type to NodePort and the node port itself to 31000. (You may want to copy and paste this one)

```
$ helm upgrade local-chartmuseum cm/chartmuseum --set
env.open.DISABLE_API=false --set service.type=NodePort --set
service.nodePort=31000
```

9. Take a look at the release you have out there now, its status and its history

```
$ helm list
$ helm status local-chartmuseum
$ helm history local-chartmuseum
```

10. Verify that the type of port has been changed for the running version.

```
$ k get svc
```

You should see it listed as a NodePort now with 31000 as the exposed port.

11. Now we'll add your chartmuseum repo to your list of repos for Helm and verify it's there.

```
$ helm repo add local http://localhost:31000
$ helm repo list
```

Lab 3: Creating a Helm Chart

Purpose: In this lab, we'll create a simple Helm chart and add it to our new repository

1. Let's use Helm to create a simple, default chart one that will spin up an nginx deployment.

```
$ helm create sample-chart
```

2. Let's see what Helm created in terms of the structure of files and directories.

```
$ tree sample-chart
```

3. Now take a look at some of the main files in the new chart.

```
$ cd sample-chart
$ cat Chart.yaml
$ cat values.yaml
$ cat templates/deployment.yaml
$ cat templates/service.yaml
```

4. Take a look at how Helm would render files in this chart.

```
$ helm template --debug . | head -n 50
```

5. Go ahead and install the chart.

```
$ helm install sample .
```

You will see some output like this:

```
NAME: sample
LAST DEPLOYED: <date/time>
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
```

```
1. Get the application URL by running these commands:

  export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/name=sample-chart,app.kubernetes.io/instance=sample"
-o jsonpath="{.items[0].metadata.name}")

  echo "Visit http://127.0.0.1:8080 to use your application"

  kubectl --namespace default port-forward $POD_NAME 8080:80
```

6. If you do a helm list command, you'll see that the chart was deployed in the "default" namespace (alongside our chart-museum one) and note that it is release version 1. You can also see the Kubernetes objects that were deployed in the default namespace for this.

```
$ helm list

$ k get all | grep sample
```

7. Take a look at the rendered templates that got deployed into the cluster.

```
$ helm get manifest sample | head -n 50
```

   This should look very similar to the output from the template command that was issued earlier.

8. You may have noticed earlier that this chart had a test built into it. Let's run the test now. Note the output and also note the pods that are there afterwards. Then take a look at the definition of the test afterward and see if you can understand how it all ties together.

```
$ helm test sample

$ k get pods

$ cat templates/tests/test-connection.yaml
```

9. We're done with this release now, so we can delete it.

```
$ helm delete sample
```

10. Take a look at the objects in the default namespace to see that the sample ones were removed. You may see a leftover test pod that did not get removed. If so, use the second command below to remove it.

```
$ k get all | grep sample

$ k delete pod/<sample-pod-name>
```

Lab 4: Charts and Dependencies

Purpose: In this lab, we'll deploy the chart for our sample webapp, and then see how to add another chart as a dependency for its database.

1. On your local machine is a directory with a set of projects that we'll be using in this class. There are subdirectories for different sections. Go to that directory. Update it with the latest version and switch to the quay.io branch. You can optionally look at any of the files you're interested in.

```
$ cd ~/helm-ws

$ git pull

$ git checkout quay.io

$ cat <files of interest>
```

2. Create a namespace for the database to run in and then deploy it via Helm

```
$ kubectl create ns  roar

$ cd roar-web

$ helm install -n roar roar .
```

Afterwards you should see a set of output like the following:

NAME: roar
LAST DEPLOYED: Wed Jun 3 22:23:59 2020
NAMESPACE: roar
STATUS: deployed
REVISION: 1
TEST SUITE: None

3. Take a look at the release that has been installed in the cluster.

```
$ helm list -n roar
```

You should see output like the following:

```
NAME  NAMESPACE  REVISION  UPDATED                                  STATUS    CHART     APP VERSION
roar  roar       1         2020-06-03 22:23:59.206949569 -0400 EDT  deployed  roar-web-0.1.0
```

Now take a look at the resources that are installed in the cluster.

```
$ kubectl get all -n roar
```

4. Find the nodeport where the app is running and open it up in a browser.

```
$ kubectl get svc -n roar
```

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

In a browser, go to http://localhost:<NodePort>/roar/

You should see something like this:

5. Notice that we see our webapp, but there is no data in the table. This is because we don't have our database deployed and being pulled in. We have a database chart and resources on our local system. Let's see how to get it pulled in as a dependency. First, change into the directory for the database pieces. Again, you can look at any files of interest in there.

```
$ cd ~/helm-ws/roar-db

$ cat <files of interest>
```

6. We want to package this up so it can be used as a dependency for our web chart. Go ahead and run the command to package it up. You should be in the roar-db subdirectory still.

```
$ helm package .
```

For output, you should see something like this:

Successfully packaged chart and saved it to: /home/diyuser3/helm-ws/roar-db/roar-db-0.1.0.tgz

7. Upload the newly created package to our ChartMuseum instance.

```
$ curl --data-binary "@roar-db-0.1.0.tgz"  http://localhost:31000/api/charts
```

When done, you should see output like this:

{"saved":true}

8. Now that we have this package stored in our chart repository, we can add it as a dependency into our webapp's chart so it will have data to display. To do that we go back to the webapp's chart directory and create a requirements.yaml file.

```
$ cd ~/helm-ws/roar-web
$ gedit Chart.yaml
```

Add the **lines in bold** below (the dependencies section) - this would have been handled with a requirements.yaml in an earlier version of Helm.

apiVersion: v2
description: Helm chart for roar-web instance
name: roar-web
version: 0.1.0
dependencies:
- name: roar-db
version: 0.1.0
repository: http://localhost:31000

9. Save your changes to the Chart.yaml file, close the editor and update dependencies.

```
$ helm dep up
```

You should see output like this as Helm gets your new dependency:

Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "local" chart repository
...Successfully got an update from the "cm" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈Happy Helming!⎈
Saving 1 charts
Downloading roar-db from repo http://localhost:31000
Deleting outdated charts

10. If you look at the directory structure now, you'll have a new "charts" directory where the tgz file will be. With our new dependency in place, let's remind ourselves what is out there now and then go ahead and upgrade our webapp release.

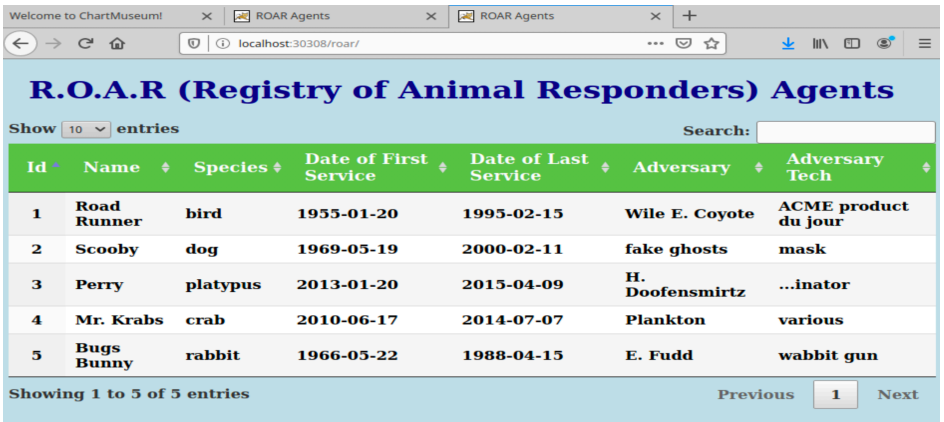
```
$ ls charts
$ helm list -n  roar
$ k get all -n  roar
$ helm  upgrade -n  roar roar  .  --recreate-pods
```

11. Take a look at what we have out there now for this release.

```
$ helm list -n  roar
$ k get all -n  roar
```

You should see the various database pieces in the cluster now.

12. Finally, refresh the webapp in your browser and you should see data being displayed.



Lab 5: Templating

Purpose: In this lab, we'll see how to add templating to a manifest file and also another way to specify values and setup dependencies.

1. Let's suppose we want to create a helm deployment passing in a test database. We're going to change our dependency to be an actual copy of the chart so we can work with it more easily.

```
$ cd ~/helm-ws/roar-web (if not already there)
$ rm -rf charts/*
$ cp -R ../roar-db  charts/
```

2. Take a look at the deployment.yaml in the sub-chart. Notice that the image name is hardcoded and not templated.

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

```
   - name: {{ .Chart.Name }}
     image: quay.io/bclaster/roar-db-image:v1
     imagePullPolicy: Always
     ports:
```

3. Edit the file and change that line to a templated form:

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

Change line 19 from

```
image: quay.io/bclaster/roar-db-image:v1
```

to

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

Make sure that the indentation still lines up as before:

```
containers:
 - name: {{ .Chart.Name }}
   image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
   imagePullPolicy: Always
   ports:
```

Save the file and close the editor.

4. Now let's check our charts to make sure they're valid via the Helm lint function (you should still be in the roar-web directory).

```
$ helm lint --with-subcharts
```

What messages do you see? What does the error mean?

5. There's a "*nil pointer evaluation*" because we haven't yet defined anything in values.yaml for "Values.image.repository" or "Values.image.tag". Let's fix that now.

```
$ gedit charts/roar-db/values.yaml
```

Change the top from this

```
# Default values for roar-db-chart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
replicaCount: 1
nameOverride: mysql
deployment:
  ports:
    name: mysql
    containerPort: 3306
```

to this:

```
# Default values for roar-db-chart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
image:
  repository: quay.io/bclaster/roar-db-image
  tag: v1
replicaCount: 1
nameOverride: mysql
deployment:
  ports:
```

6. Save your changes and exit the editor. Then run the lint check again.

```
$ helm lint --with-subcharts
```

This time you should not see any errors (though you will still see INFO messages).

7. Create a new namespace and deploy this new version of the chart.

```
$ k create  ns  roar2
$ cd ~/helm-ws/roar-web (if not already there)
$ helm  install  roar2  -n  roar2  .
```

8. Find the nodeport and open up the webapp from the release in a browser.

```
$ k get svc -n roar2
```

Find the NodePort value (will be > 30000)

In the browser, go to http://localhost:<NodePort value>/roar/

Here you should see the same webapp and data as before.

9. Let's suppose we want to overwrite the image used here to be one that is for a test database. The image for the test database is on the quay.io hub at quay.io/bclaster/roar-db-test:v4 .
We could use a long command line string such as this to set it and use the template command to show the rendered files. In the roar-web subdirectory, run the commands below to see the difference.

```
$ helm template . --debug | grep image
```

```
$ helm template . --debug --set roar-
db.image.repository=quay.io/bclaster/roar-db-test   --set roar-
db.image.tag=v4  |  grep image
```

10. It's not always convenient to have to override things on the command line. But there are other ways to override values. Let's create a simple "extra" values file.

```
$ gedit  test-db.yaml
```

In the editor, add the following lines:

```
roar-db:
  image:
    repository: quay.io/bclaster/roar-db-test
    tag: v4
```

(For convenience, there is a test-db.yaml file in the "extra" area of the helm-ws area.)

Save your changes to the file and exit the editor.

11. Now, in one of your terminal windows, start a watch of the pods in your deployed helm release. This is so that you can see the changes that will happen when we upgrade. Save your changes to the file, and run the following command to pull in your custom definitions.

```
$ kubectl get pods -n  roar2  --watch
```

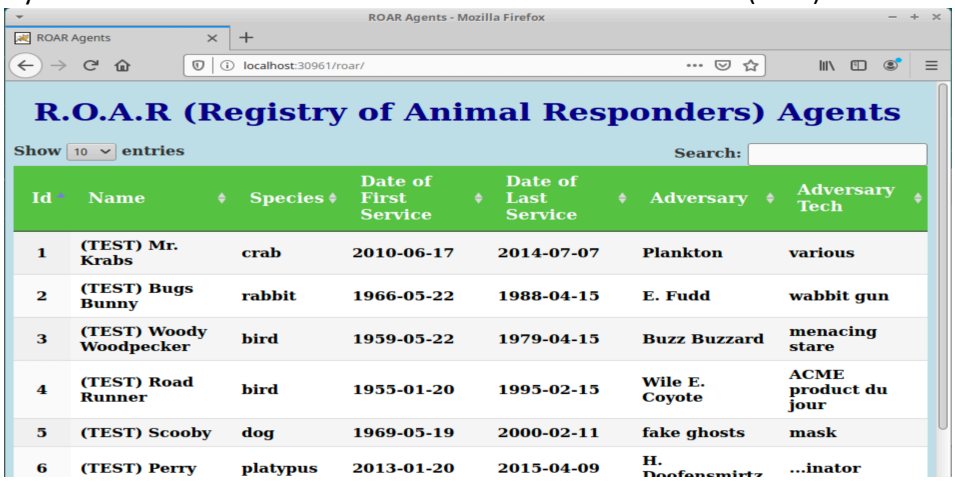
12. Finally, let's do an upgrade using the new values file. In a separate terminal window from the one where you did step 11, execute the following commands:

```
$ cd ~/helm-ws/roar-web

$ helm upgrade -n roar2 roar2 .  -f test-db.yaml --recreate-pods
```

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

13. Go back to your browser and refresh it. You should see a version of the (TEST) data in use now.



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

```
$ Ctrl-C
```

Lab 6: Using Functions and Pipelines

Purpose: In this lab, we'll see how to use functions and pipelines to expand what we can do in Helm charts.

1. In our last lab, we added a separate values file that we could use to swap out which database we used - the prod one or the test one. Now let's work on a way to do this based on a simple command line setting rather than having to have a separate file with multiple values. To start, let's make a copy of our working project that already functions. And then go to that directory.

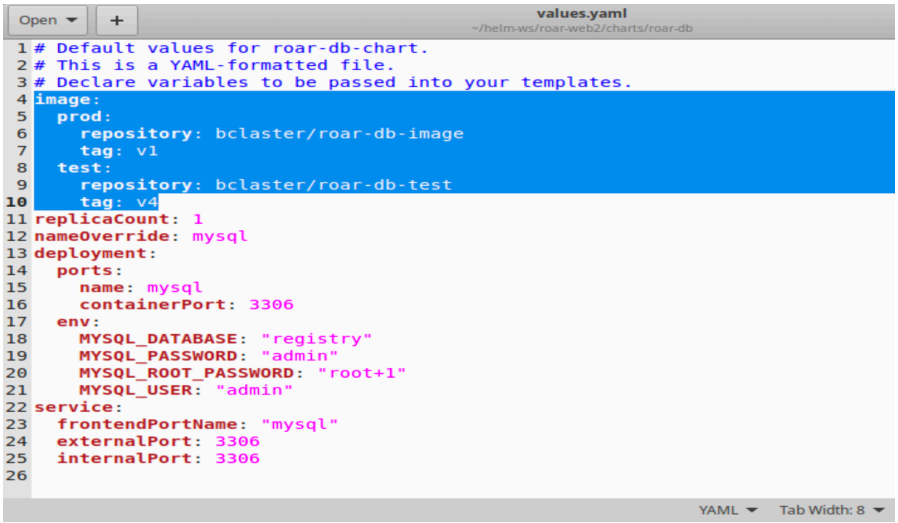
```
$ cd ~/helm-ws
$ cp -R roar-web roar-web2
$ cd roar-web2
```

2. For switching between the prod and test database, we'll add two different versions of the image to choose from - one for prod and one for test. Edit and modify the *charts/roar-db/values.yaml* file and add the following values. (For reference, or if you're concerned about the typing, in the ~/helm-ws/extra folder, there is a values.yaml with the code already in it.). You can save and close the editor afterwards.

```
$ gedit charts/roar-db/values.yaml
```

Change the previous image definition to the following lines

```
image:
  prod:
    repository: quay.io/bclaster/roar-db-image
    tag: v1
  test:
    repository: quay.io/bclaster/roar-db-test
    tag: v4
```



3. Now, we'll add some processing to enable selecting one of these images based on passing a setting for "stage" of either "PROD" or "TEST". The design logic is

```
if  stage = PROD
      use   image.prod.repository :  image.prod.tag
else if  stage = TEST
      use   image.test.repository : image.test.tag
else
      display error message that we don't have a valid stage setting
```

Translating this into templating syntax, we use an if/(else if)/else flow. The "eq" function is used for comparison, passing the two things to compare after it. Also, there is a "required" function that will handle the default error checking. Putting it all together, it could look like this:

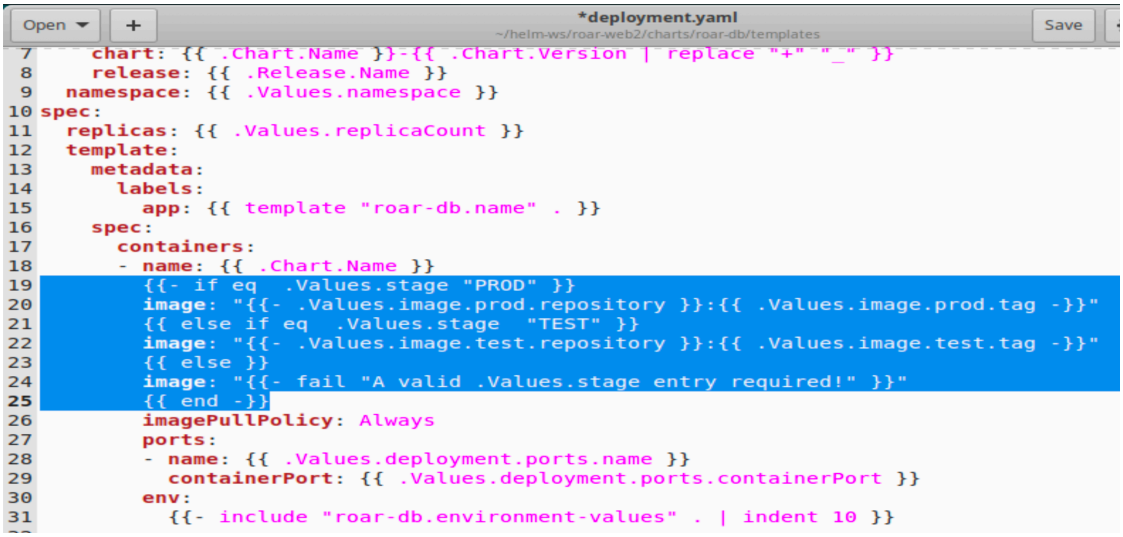
```
{{- if eq .Values.stage  "PROD" }}
  image: "{{- .Values.image.prod.repository }}:{{ .Values.image.prod.tag -}}"
{{ else if eq .Values.stage  "TEST" }}
  image: "{{- .Values.image.test.repository }}:{{ .Values.image.test.tag -}}"
{{ else }}
  image: "{{- fail "A valid .Values.stage entry required!"  }}"
{{ end -}}
```

4. Edit the *charts/roar-db/templates/deployment.yaml* file and make the changes shown below (add the if - end template pieces above into the containers section). Be sure you are in the helm-ws/roar-web2 directory and be sure to use spaces and not tabs.

(For convenience or if you have trouble with typing it in, there is a deployment.yaml file in the extras area.)

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

  <add in lines>
```



5. Save your changes, exit the editor, and do a dry-run to make sure the image comes out as expected. To simplify seeing the change, we'll grep for image with a few lines of context (from the roar-web2 subdir).

```
$ helm install --set roar-db.stage=TEST --dry-run FOO . | grep image -n3
```

You should see output like the following:

```
65-     spec:
66-       containers:
67-       - name: roar-db
68:         image: "quay.io/bclaster/roar-db-test:v4"
69:         imagePullPolicy: Always
70-         ports:
71-         - name: mysql
72-           containerPort: 3306
--
```

6. While this appears to work, what happens if we pass a lower-case value?

```
$ helm install --set roar-db.stage=test   --dry-run FOO . | grep image -n3
```

This is not what we want. Let's make our arguments case-insensitive. To do this, we'll pipe the value we pass in through a pipeline and to another template function called "upper" to upper-case it first.

7. Edit the deployment file and make the changes shown below.

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

Change line 19 from

```
{{- if eq .Values.stage  "PROD" }}
```

to

```
{{- if eq  ( .Values.stage | upper )  "PROD" }}
```

and change line 21 from

```
{{ else if eq .Values.stage  "TEST" }}
```

to

```
{{ else if eq ( .Values.stage | upper )  "TEST" }}
```

```
14        labels:
15          app: {{ template "roar-db.name" . }}
16      spec:
17        containers:
18        - name: {{ .Chart.Name }}
19          {{- if eq ( .Values.stage | upper )  "PROD" }}
20          image: "{{- .Values.image.prod.repository }}:{{ .Values.image.prod.tag -}}"
21          {{ else if eq ( .Values.stage | upper )  "TEST" }}
22          image: "{{- .Values.image.test.repository }}:{{ .Values.image.test.tag -}}"
23          {{ else }}
24          image: "{{- fail "A valid .Values.stage entry required!" }}"
25          {{ end -}}
```

8. Save your changes and try running the command with the lower-case setting again. This time it should work.

```
$ helm install --set roar-db.stage=test --dry-run FOO . | grep image -n3
```

9. What happens if we don't pass in a setting for stage? Try it and see.

```
$ helm install --dry-run FOO . | grep image -n3
```

We get an error because we don't have any value set and one of our functions fails because it doesn't have the correct type to compare.

10. We'd rather have a default that works. Let's set one up in the *values.yaml* of the parent project that will be passed in to the child project. In the *helm-ws/roar-web2* directory, edit the *values.yaml* file and add the settings below.

```
$ gedit ~/helm-ws/roar-web2/values.yaml

<add these lines>

roar-db:
    stage: PROD
```

Open + *values.yaml ~/helm-ws/roar-web2

```
# Default values for roar-web chart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
replicaCount: 1
image:
  repository: bclaster/roar-web-image
  tag: v1
  pullPolicy: Always
deployment:
  ports:
    name: web
    containerPort: 8080
service:
  portType: NodePort
  frontendPortName: frontend
  externalPort: 8080
  internalPort: 8089
roar-db:
  stage: PROD
```

11. Now try running the same command as you did in step 8.

```
$ helm install --dry-run FOO . | grep image -n3
```

Notice that this time it worked. Also notice that the value was passed down from the parent project.

12. Finally let's try deploying a running test instance of our application and a running instance of the prod version of our application.

```
$ k create ns roar-prod
$ helm install --set roar-db.stage=PROD roar-prod -n roar-prod .
$ k create ns roar-test
$ helm install --set roar-db.stage=TEST roar-test -n roar-test .
```

13. Now, you can get the NodePort for both versions. and open them in a browser to see the results.

```
$ k get svc -n roar-prod | grep web
$ k get svc -n roar-test | grep web
```

14. As a reminder, the port numbers that are > 30000 are the ones you want. You can open each of them in a browser at http://localhost:<nodeport>/roar/ and view the different instances.

THE END – THANKS!