

## Kustomize Fundamentals

### Managing Kubernetes Manifests

#### Class Labs

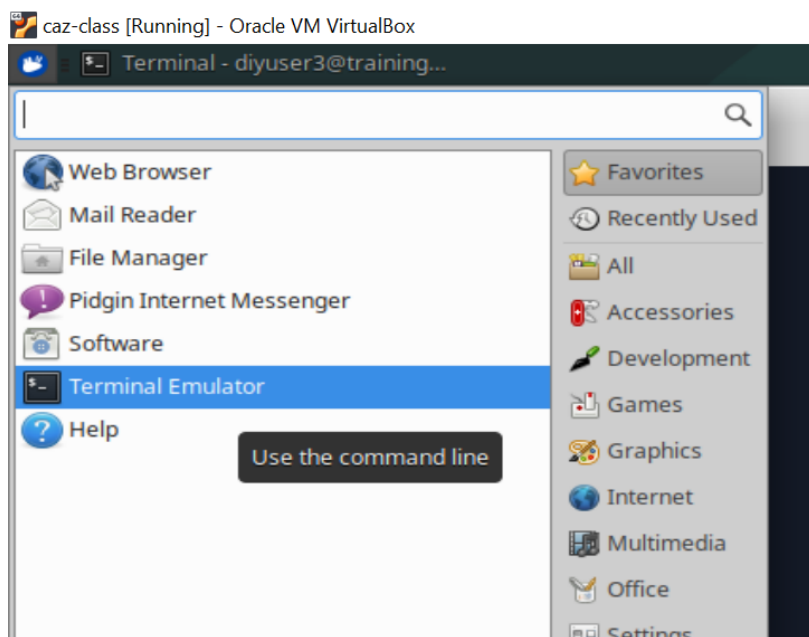
Version 1.2 by Brent Laster

11/09/2021

**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.

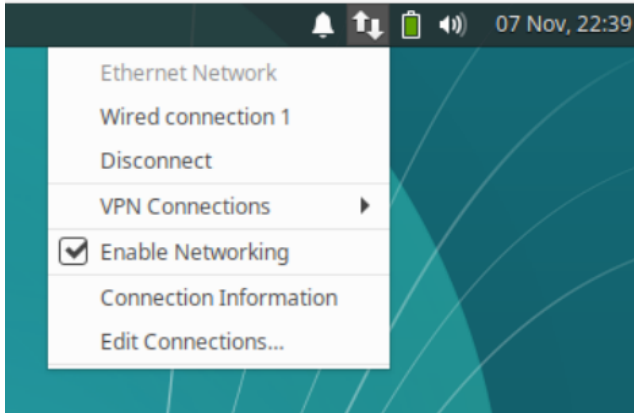


2. First, let's make sure we have the latest files for the class. For this course, we will be using a main directory *kz-fun* with subdirectories under it for the various labs. In the terminal window, `cd` into the main directory and update the files.

```
$ cd kz-fun
```

```
$ git pull
```

Note: If you see an error about "Could not resolve host: github.com", you may need to enable networking by selecting the up/down arrow icon at top right and selecting the option. See screenshot below.



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 *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 base
$ tree
```

2. Let's see what happens when we try to run "kustomize build" against these files. (On this system, I have "kustomize" aliased as "kz".) There will be an error.

```
$ kz build
```

3. 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". 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 -
```

4. 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. (On this machine, "kubectl" is aliased as just "k".)

```
$ k get all
```

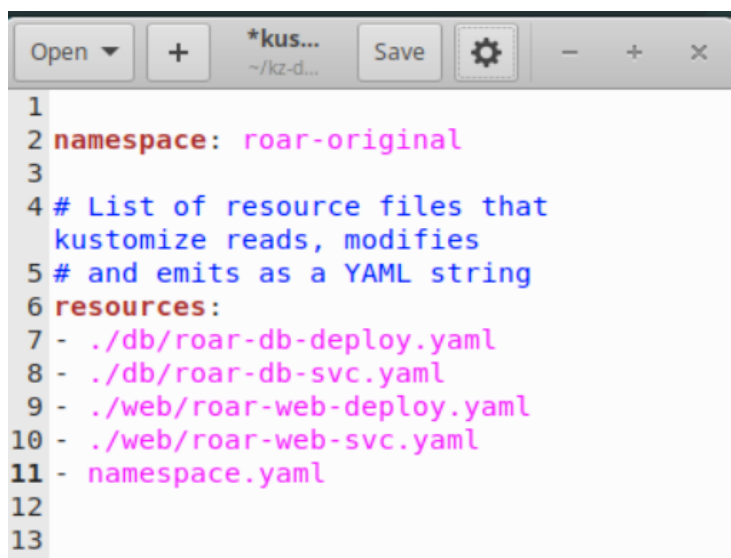
5. We have a *namespace.yaml* file in the directory. Take a look at it. It is setup to create a namespace. So how do we use it with Kustomize? Since it's another resource, we just need to include it in our list of resources. And then we also need to specify the namespace it creates "roar-original" in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

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.** (gedit is used as the editor here. You may use a different one if you want.)

```
$ cat namespace.yaml
```

```
$ gedit kustomization.yaml
```

(To turn on line numbers in gedit, click the "gear icon" in the upper right, then select "Preferences" from the menu and click "Display line numbers" in the pop-up box.)



```

1
2 namespace: roar-original
3
4 # List of resource files that
  customize reads, modifies
5 # and emits as a YAML string
6 resources:
7 - ./db/roar-db-deploy.yaml
8 - ./db/roar-db-svc.yaml
9 - ./web/roar-web-deploy.yaml
10 - ./web/roar-web-svc.yaml
11 - namespace.yaml
12
13

```

- 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.

```
$ kustomize build | grep -n3 original
```

- 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.

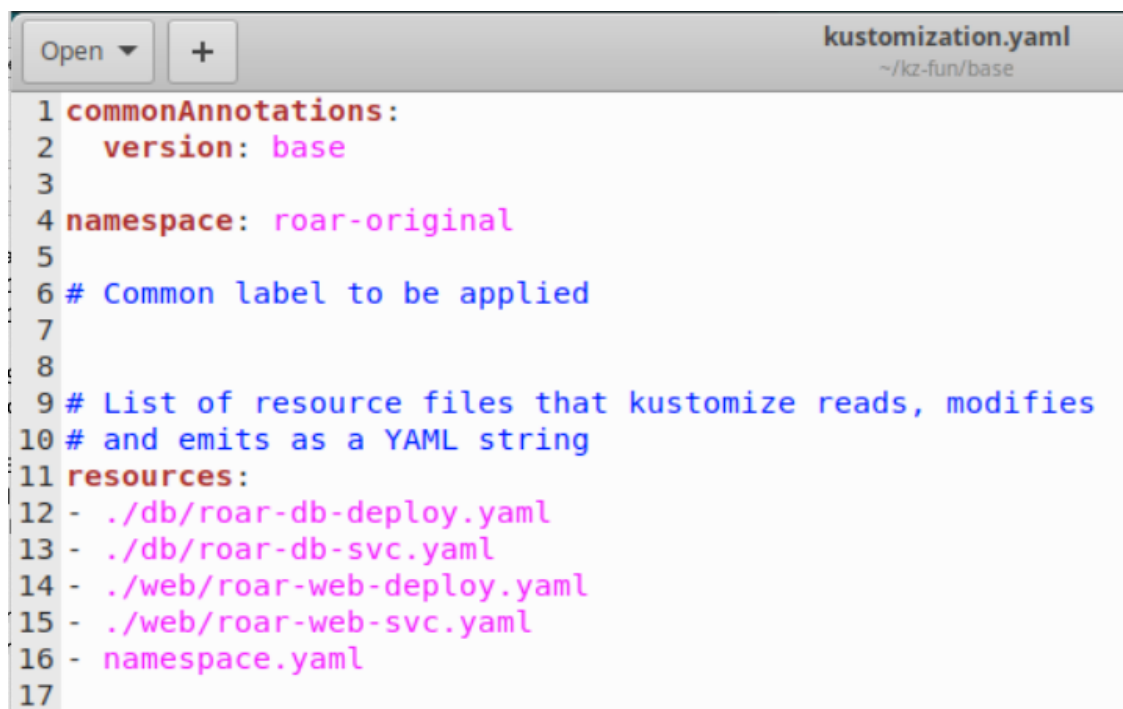
```
$ kustomize build | kubectl apply -f -
```

```
$ kubectl get ns
```

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

- Let's make one more change here. Let's apply 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
```



```
kustomization.yaml
~/kz-fun/base

1 commonAnnotations:
2   version: base
3
4 namespace: roar-original
5
6 # Common label to be applied
7
8
9 # List of resource files that kustomize reads, modifies
10 # and emits as a YAML string
11 resources:
12 - ./db/roar-db-deploy.yaml
13 - ./db/roar-db-svc.yaml
14 - ./web/roar-web-deploy.yaml
15 - ./web/roar-web-svc.yaml
16 - namespace.yaml
17
```

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

```
$ kustomize build | grep -a5 metadata
```

```
$ kustomize build | kubectl apply -f -
```

10. 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.

```
$ kubectl get svc -n roar-original | grep web
```

<find Nodeport - second to last column - value after 8089 - value in the 30000's>

open <http://localhost:<nodeport>/roar> in browser

**END OF LAB**

## 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 the two directories.

```
$ cd ~/kz-fun
```

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

2. 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
```

3. 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
```

4. 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
```

5. 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
```

6. 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.

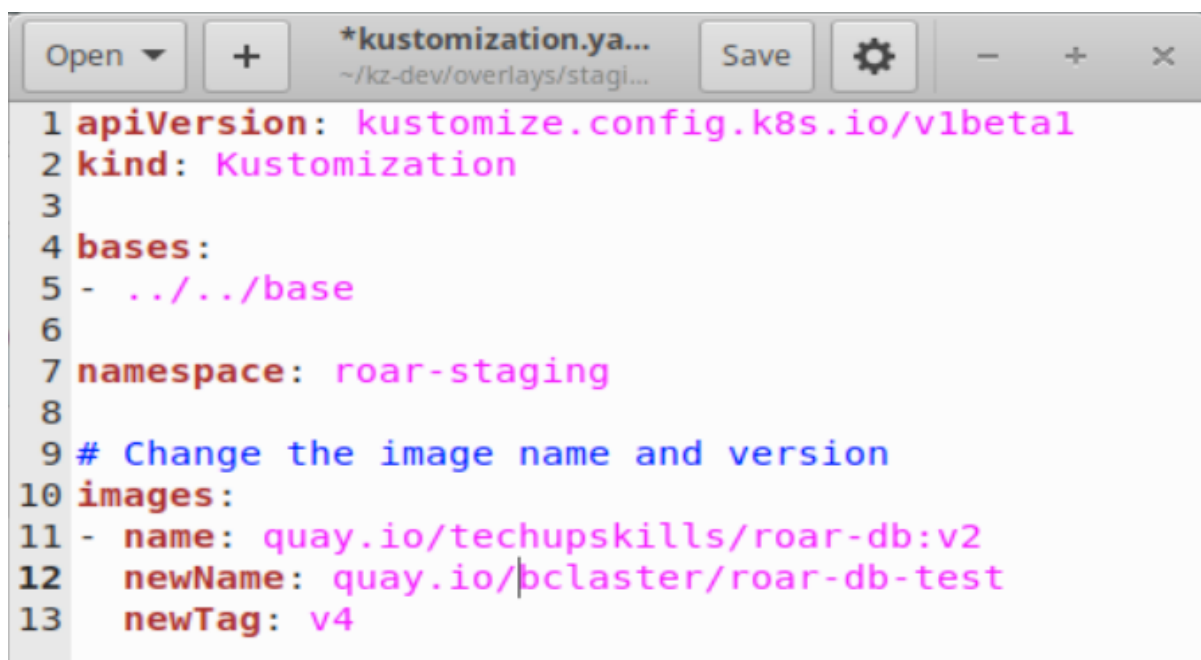
```
$ k create ns roar-staging
$ k create ns roar-production

$ kz build overlays/staging | k apply -f -
$ kz build overlays/production | k apply -f -
```

- 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
```

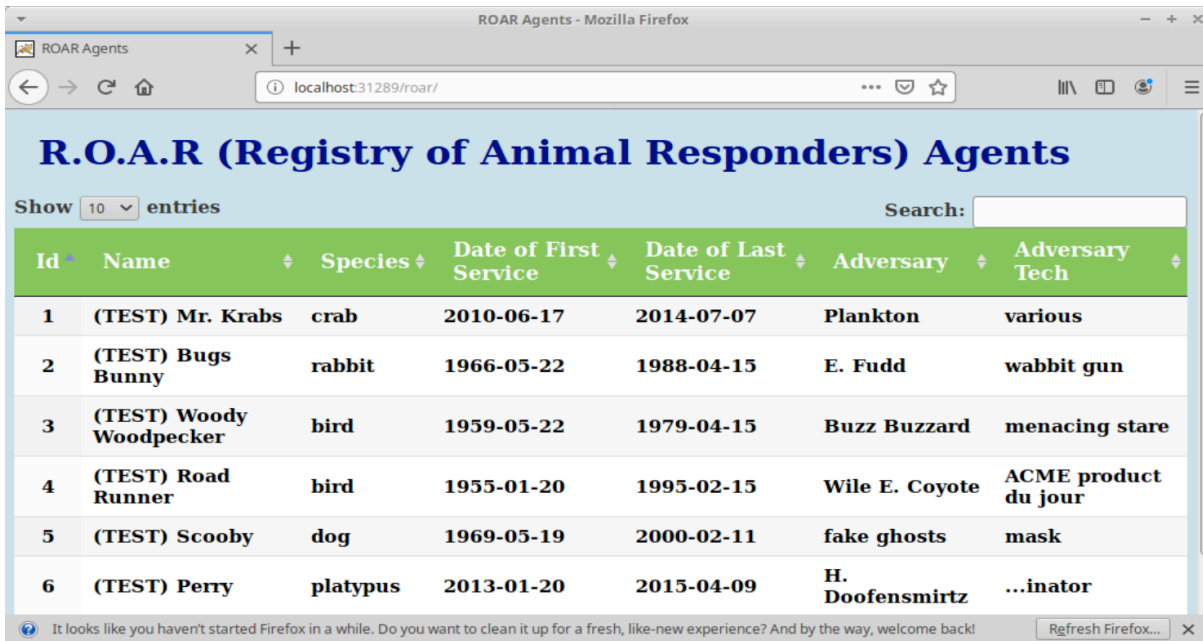


```
1 apiVersion: kustomize.config.k8s.io/v1beta1
2 kind: Kustomization
3
4 bases:
5 - ../../base
6
7 namespace: roar-staging
8
9 # Change the image name and version
10 images:
11 - name: quay.io/techupskills/roar-db:v2
12   newName: quay.io/bclaster/roar-db-test
13   newTag: v4
```

8. Save your changes and then apply the variant.

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

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



**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	(TEST) Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
2	(TEST) Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun
3	(TEST) Woody Woodpecker	bird	1959-05-22	1979-04-15	Buzz Buzzard	menacing stare
4	(TEST) Road Runner	bird	1955-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
5	(TEST) Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
6	(TEST) Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	...inator

It looks like you haven't started Firefox in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welcome back! Refresh Firefox...

## END OF LAB

### 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)
```





2. 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)

3. 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
```

4. 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
```

```

1
2 resources:
3 - ../../base
4 - storage.yaml
5
6 namespace: roar-production
7
8 # add in persistent storage
9 patchesStrategicMerge:
10 - patch_pv.yaml
11

```

5. With your changes saved, 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.

```
$ 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-fun/base/db/roar-db-deploy.yaml
```

2. 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

```

3. 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
```

4. 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
```

5. 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 -
```

- 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:")

- Open up a browser and go to <http://localhost:<NodePort>/roar/>

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



- 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
```

- 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
```

10. 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.

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	(TEST) Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
2	(TEST) Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun
3	(TEST) Woody Woodpecker	bird	1959-05-22	1979-04-15	Buzz Buzzard	menacing stare
4	(TEST) Road Runner	bird	1955-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
5	(TEST) Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
6	(TEST) Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	...inator

Showing 1 to 6 of 6 entries Previous 1 Next

## 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
```

2. 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
```

3. 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_pv.yaml .
```

4. 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
```

```

30 target:
31   group: apps
32   kind: Deployment
33   name: mysql
34   version: v1
35 secretGenerator:
36 - files:
37   - dbp.info
38   name: dbp
39   type: Opaque
40
41 components:
42 - ../components/patch-storage
43

```

5. 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 pv mysql-pv
```

```
$ k delete ns roar-staging
```

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

```
$ ls /mnt
```

6. 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
```

7. 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
```

8. 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 .
```

9. 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/production
```

```
$ gedit kustomization.yaml
```

add these two lines at bottom:

components:

```
- ../../components/db-secret
```

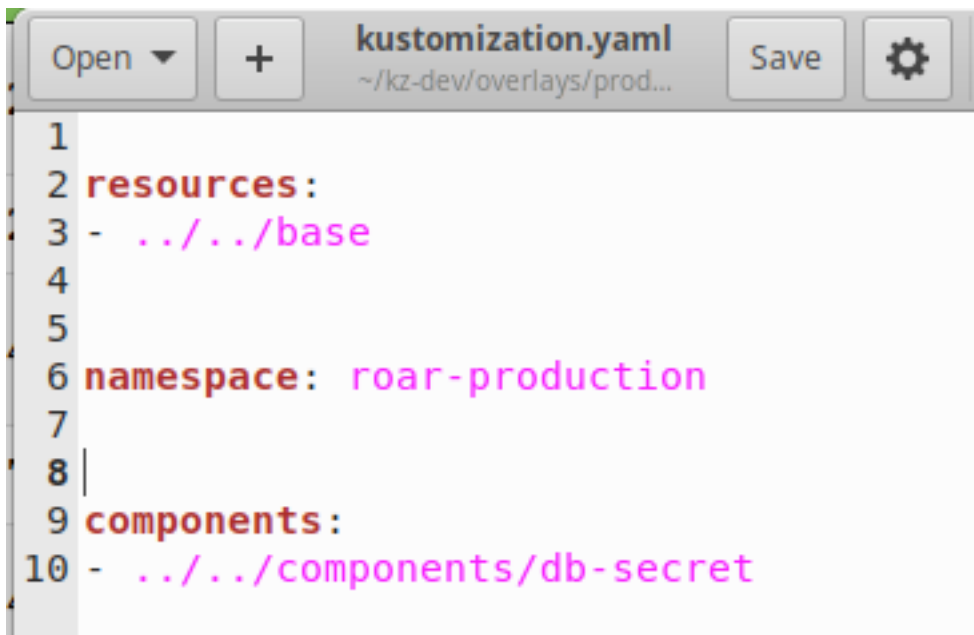
remove these lines:

```
- storage.yaml
```

patchesStrategicMerge:

```
- patch_pv.yaml
```





```
1
2 resources:
3 - ../../base
4
5
6 namespace: roar-production
7
8 |
9 components:
10 - ../../components/db-secret
```

10. 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