Building a Kubernetes Operator

Extending Kubernetes to Fit your Applications

Class Labs: Revision 1.5 - 4/18/21

Brent Laster, Lead Instructor - Tech Skills Transformations LLC

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 k8s-op-setup.pdf in http://github.com/skillsdocs/k8s-op for instructions and other things to be aware of. You should also have a GitHub account.

Lab 1: Scaffolding an operator with the Operator SDK

Purpose: In this lab, we'll explore the development environment for an operator and use the sdk to generate the initial code for our operator.

1. To start, we'll get the minikube environment up and running. Open a terminal window and enter the command below.

```
$ sudo minikube start --vm-driver=none --addons=registry --kubernetes-version=v1.19.0


        This will take a few minutes to run.
```

2. Now we'll verify that we have the two main tools installed that we need to work on our operator and note their versions: Go and operator-sdk. Open a terminal window and enter the following commands. (On this system, I have aliased "operator-sdk" to "ok".)

```
        $ go version

        $ ok
```

3. Our operator will be targeted for the "roar" application - a small Java-based webapp with a mysql backend. Create a directory to work in and then create a scaffold project for it with the operator-sdk init function. Note that you will need to supply your GitHub userid in the last step here.

```
        $ mkdir ops

        $ cd ops
```

```
$ ok init --domain=roarapp.com --repo=github.com/<your github id>/roar-op
```

4. **This will take a few minutes to run.** Notice at the end of the init step, the operator-sdk gives us a clue about what to do next - creating a new resource. The resource in this case equates to a new Custom Resource Definition (CRD) API and an associated controller. We'll set up the skeleton with a group equal to "roarapp" and an API version of "v1alpha1". Answer "y" to both prompts.

```
$ ok create api --group=roarapp --version=v1alpha1 --kind=RoarApp
```

5. This will have created a starting RoarApp resource API for us in api/v1alpha1/roarapp_types.go and a starting controller at controllers/roarapp_controller.go. Take a quick look at those files to see what is in each.

```
$ cat api/v1alpha1/roarapp_types.go
$ cat controllers/roarapp_controller.go
```

Lab 2: Defining the API

Purpose: In this lab, we'll modify the Go type definitions to set up the values we expect to be filled in for the desired state "spec" and the observed state "status".

1. In our last lab, we got a generated types file. We need to edit that file to add the appropriate types to use for these states. You edit the file and copy in/type the code or we could do something simpler to avoid confusion over where changes go and syntax errors, typos, etc.

   In this case, to keep it simple, and avoid those kind of issues you can use the visual merge tool **meld** and merge in the changes by clicking on the arrows that are pointing to the left from the right pane (circled in image). After you make the edits, save your changes, and close meld.

   (Note that you may need to stretch the meld window to see everything.)

```
$ meld api/v1alpha1/roarapp_types.go ~/ref/api/v1alpha1/roarapp_types.go
```

Click on arrows in red circles below. The files will then be identical. Save the changes and close Meld.
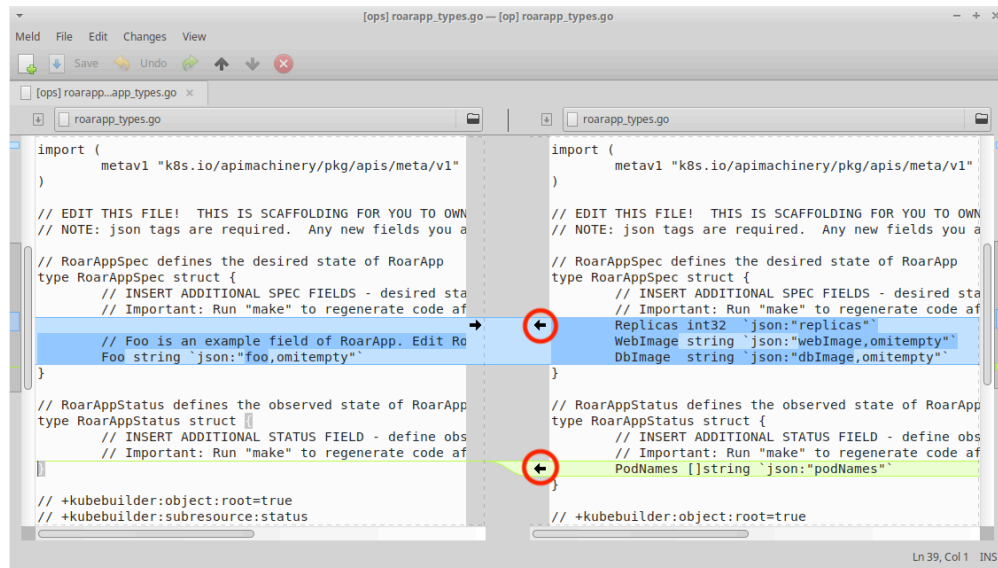
2. Whenever the types file (roarapp_types.go) is changed, you need to run the make command again to update the generated code for the resources. (What this does is update the zz_generated.deepcopy.go file to make sure we consistently implement an interface (runtime.Object) that is required for all K8S Kind types to implement.)

```
$ make generate
```

3. Now that we have the API defined (and info for spec/status and CRD validation markers), we can generate CRD manifests - also with a make target.

```
$ make manifests
```

4. This invokes controller-gen (again) to generate the CRD manifests. The resulting file will be at config/crd/bases/cache.example.com. Take a look at that file.

```
$ cat config/crd/bases/roarapp.roarapp.com_roarapps.yaml
```

Lab 3: Implementing the Controller

Purpose: In this lab, we'll work on implementing the custom controller for our RoarApp custom resource.

1. In our previous labs, we generated a number of scaffolded files. One of those was controller code for our custom resource. We need to update that code to have the specific packages and logic for our CR.

To help you understand the different areas involved and simplify typing, we'll do this in stages and merge in changes from a different reference set of code each time. **For each set of changes, we'll use meld as we did in the previous lab and then merge in more code. We will always just click the arrows in the right pane that are facing left to merge in the code. MAKE SURE TO SAVE/EXIT AFTER EACH MERGE.**

NOTES:

At each step there is a different _# in the name of the second file that is passed to meld.

Throughout the merges, you can ignore any line/changes that features your GitHub ID, such as the line that starts with roarappv1alpha1 "github.com/<your github userid>…

These meld commands assume you are in the "ops" directory when you run meld.

2. First, add in the other packages and core functionality we need from Kubernetes into the controller code.

```
$ meld controllers/roarapp_controller.go ~/ref/controllers/roarapp_controller_1.go
```

Then click on arrows in red circles below and save and exit.
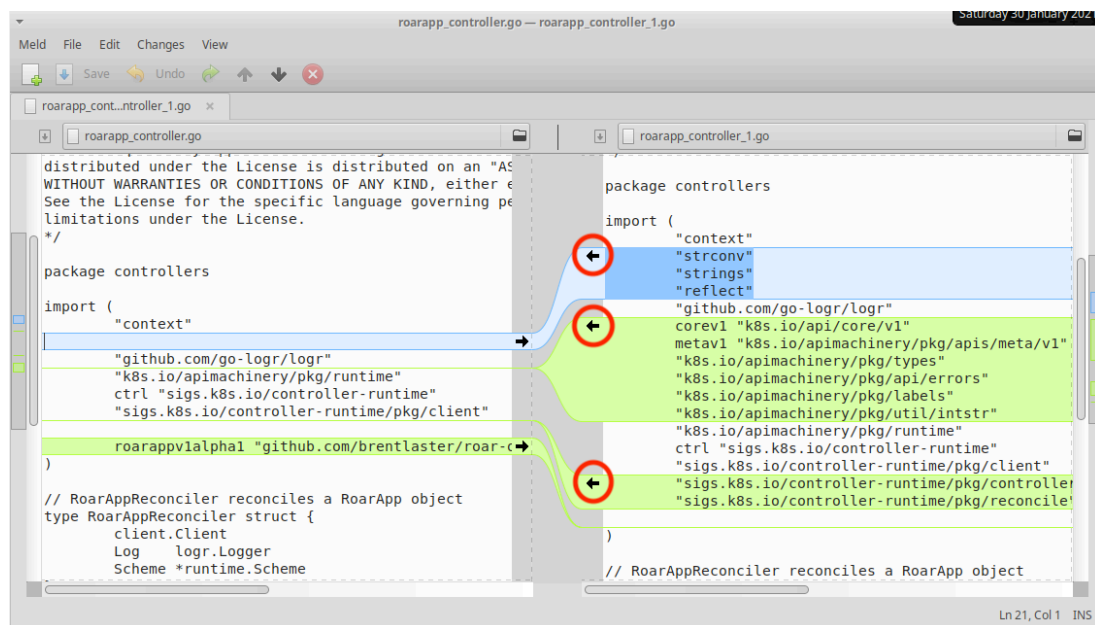
3. Next, we'll add in the code to reconcile the desired state of our CR against the observed state.

```
$ meld controllers/roarapp_controller.go ~/ref/controllers/roarapp_controller_2.go
```

Then click on arrows in red circles below and save and exit.

4. Now, we'll add in code for a function that will create a new service to go along with our pods.

```
$ meld controllers/roarapp_controller.go ~/ref/controllers/roarapp_controller_3.go
```

Then click on the arrow in the red circle below and save and exit. (Note that you may need to scroll down in the right-hand pane to see the new function.

5. Finally, we'll add in code for the function that will create new pods of our application.

```
$ meld controllers/roarapp_controller.go ~/ref/controllers/roarapp_controller_4.go
```

Then click on the arrow in the red circle below and save and exit. (Note that you may need to scroll down in the right-hand pane to see the new function.)
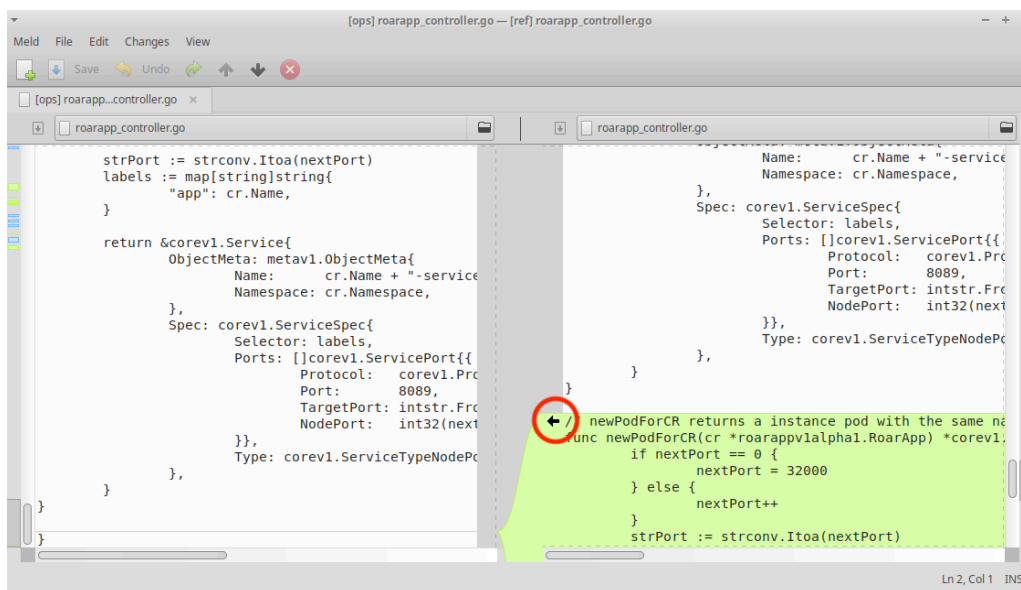
Lab 4: Specifying permissions and adding RBAC manifests

Purpose: In this lab, we'll add in the markers into the controller to allow it to interact with the resources it controls (manages).

1. The controller-gen utility already generated rules in roarapp_controller.go to manage our custom resource. Specifically
   a. For resources of group=roarapp.roarapp.com and resources=roarapps, the controller can get, watch, create, update, patch, and delete them.
   b. For resources of groups=roarapp.roarapp.com and resources=roarapps/status, the controller can do the usual status updates via get, update, and patch.

   These are expressed via markers in the controller code. Specifically:

```
// +kubebuilder:rbac:groups=roarapp.roarapp.com,resources=roarapps,verbs=get;list;watch;create;update;patch;delete

   // +kubebuilder:rbac:groups=roarapp.roarapp.com,resources=roarapps/status,verbs=get;update;patch
```

Now, because our controller needs to create and work with the general services and pods, we need to provide it with rbac access to those items - mainly get, list, watch, update, delete, and, for some, patch. We'll also include ones for deployments and jobs in case we want to/need to use those in the future. So we'll construct markers for those like the following:

// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;watch;create;update;delete

// +kubebuilder:rbac:groups=core,resources=services,verbs=get;list;watch;create;update;delete

2. Now, we'll add these into our controller code. You can just run the meld command below, click on the arrow circled in red, save your updated file, and exit meld.

```
$ meld controllers/roarapp_controller.go ~/ref/controllers/roarapp_controller_5.go
```
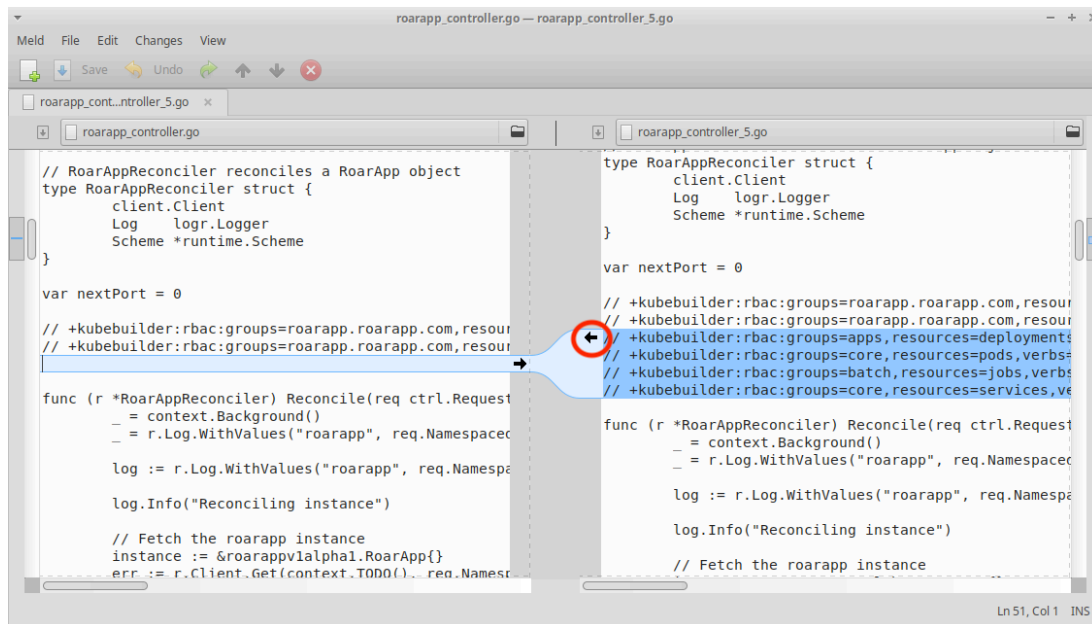

3. After we've added the necessary rbac markers, we need to run the make command to create the ClusterRole manifest.

```
$ make manifests
```

4. After running the step above, you can look at the ClusterRole manifest for the operator at config/rbac/role.yaml.

```
$ cat config/rbac/role.yaml
```

Lab 5: Building and Testing the Controller

Purpose: In this lab, we'll see how to build and test the controller/operator locally that we've created in the past labs.

1. First, we need to execute the install target to register our CRD with the Kubernetes apiserver.

   ```
   $ make install
   ```

2. If you look at the CRDs in the cluster now, you'll be able to see the one for the roarapp Kind. (On this system, I've aliased "k" to "kubectl".)

   ```
   $ k get crds
   ```

3. In a terminal session, in the ops directory, run the command below. You'll see logs from the controller starting up, but it won't do anything significant yet.

   ```
   $ make run ENABLE_WEBHOOKS=false
   ```

4. Open a second terminal window. In the second terminal window, take a look at an example RoarApp manifest that we have. This file defines the needed info for our app - the number of replicas and the images we will use for the database and web pieces.

   ```
   $ cat ref/roarapp.yaml
   ```

5. Notice that we have it set to 3 replicas. Now, in the second terminal window, go ahead and create a new namespace, and apply the manifest.

   ```
   $ k create ns op-test
   $ k apply -n op-test -f ref/roarapp.yaml
   ```

6. In the original terminal where your controller is running, you should be able to see that the operator has reconciled the desired state of 3 to the observed state. This is indicated by the messages in the logs - see the ones around "Currently available" and "Reconciling instance".

7. In the other terminal window (where you did step 5) take a look at what the operator has created in the new "op-test" namespace. You should see 3 pods and 3 corresponding services.
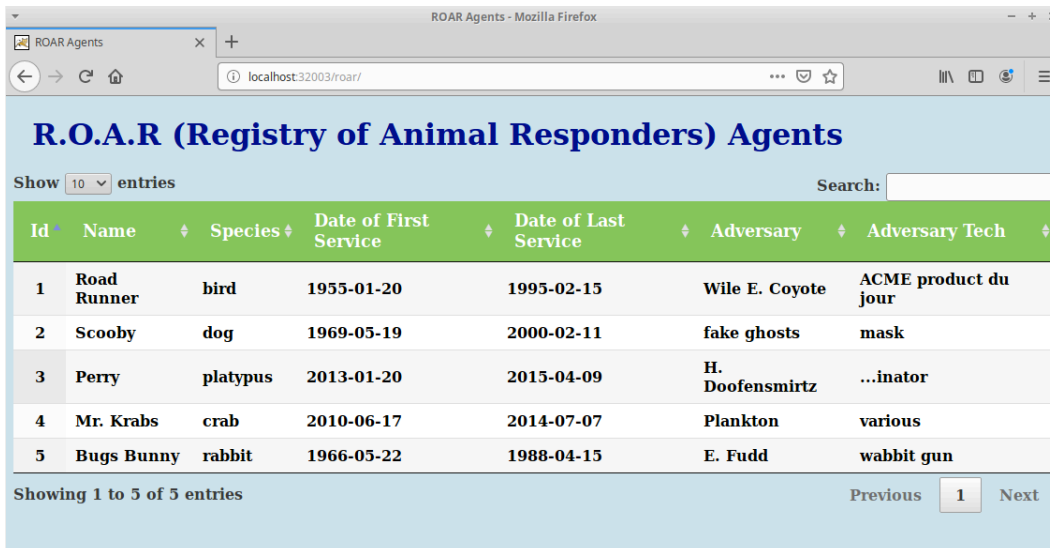
```
$ k get all -n op-test
```

8. The last part of the pod and service names are the port numbers as "node ports". **It will take a few minutes for the pods to get to the "Running" status.** Once the pods show a STATUS of "Running", open a browser and enter the URL below into it to see the actual app running there (substituting the port number from the name into <port>). (Note that this is "http" and not "https".)

http://localhost:<port>/roar

(You can open a browser by clicking on the mouse face up in the upper left corner of the VM screen and clicking on the "Web Browser" entry.)

You should see something like this:

9. At this point, you can go ahead and stop the test of the controller that's running in the one terminal via Ctrl-C. And you can also delete the op-test namespace.

```
Ctrl-C
$ k delete ns op-test
```

Lab 6: Creating the Operator Image and Deploying it into a cluster

Purpose: In this lab, we'll see how to build and run the operator that we've created in the past labs.

1. Now that we've verified that the controller works as expected running outside of the cluster, we will package it up as an image and install it in the cluster. First, take a look at the Dockerfile that was created for you.

    ```
    $ cat Dockerfile
    ```

2. Now, run the make target docker-build to package it up. We have a local registry running on the VM, so you can tag it with localhost:5000 to reference that registry. (This will take a few minutes to run.)

    ```
    $ make docker-build IMG=localhost:5000/roarapp-operator:v0.0.1
    ```

3. Now, push the image to the local registry.

    ```
    $ make docker-push IMG=localhost:5000/roarapp-operator:v0.0.1
    ```

4. You can now deploy the operator into the cluster. This make target will create an "ops-system" namespace to deploy this into.

    ```
    $ make deploy IMG=localhost:5000/roarapp-operator:v0.0.1
    ```

5. Take a look at what is in the ops-system namespace. You should see the various K8S objects that make up your operator.

    ```
    $ k get all -n ops-system
    ```

6. Now, apply the manifest for the new type into the ops-system namespace to see it create the RoarApp objects.

    ```
    $ k apply -n ops-system -f ~/ref/roarapp.yaml
    ```

7. Take a look at what is running in ops-system now.

```
$ k get all -n ops-system
```

8. In one of the terminal windows, start a watch on pods in the ops-system namespace.

```
$ k get pods -n ops-system -w
```

9. In a separate terminal window, edit the manifest in ref to have only 1 replica.

```
$ gedit ref/roarapp.yaml
```

Change "replicas: 3" to "replicas: 1"

Save and exit the editor.

10. In that same terminal, apply the changed manifest and observe what happens back in the window with the watch.

```
$ k apply -n ops-system -f ~/ref/roarapp.yaml
```

11. If you look at what's in the namespace now, you can see that the operator has scaled down to one instance of our app.

```
$ k get all -n ops-system
```

12. If you want, you can open up the url "localhost:<nodeport>/roar/" for any of the services and see that instance of the app running in a browser.