cloudpakforapplications-appmod

Introduction

App Modernization Workshop

In the workshop you will learn about foundational open source technologies and industry-wide accepted best practices for building modern, scalable, portable, and robust applications. You will learn migration strategies for moving legacy, monolithic WAS applications into Liberty containers running on Red Hat OpenShift, and some of the common pitfalls to watch out for.

Agenda Catagories

Build your own agenda from the following catagories. Browse the variants in the upper lefthand dropdown to view past agendas.

- Day 1
- Day 2

Also:

- Technology Used
- Presenters

Day 1

Topic	Description
Welcome, Introductions, Objectives and Setup	-
Lecture: Overview of Docker	
Lecture: Overview of Kubernetes	
Lab: Kubernetes 101	Series of Kubernetes Labs

BREAK	
Lecture: Helm	Helm Overview Lecture
Lab: Helm Overview	Series of Helm Labs
LUNCH	
Lecture: Operators Overview	Operators Overview
Lab: Kubernetes Extensions	Operator Labs
RECAP of Day 1	

Day 2

Topic	Description
RECAP of Day 1	
Lecture: S2I and Templates	Source to Image in OpenShift
Lab: S2I Open Liberty	Source to Image lab
BREAK	
Lectrue: CI/CD and Jenkins in OpenShift	
Lab: Jenkins Pipelines in OpenShift	
LUNCH	
Lecture: Overview of Tekton Pipelines	Tekton Overview Lecture

| | BREAK | | | Introduction: Managing Cloud-Native Applications across Cloud Environments | | | Introduction: IBM Garage | | | RECAP and Survey | |

Technology Used

- Containers
- Kubernetes
- Helm
- RedHat OpenShift
- IBM Kubernetes Service
- Jenkins
- Tekton
- MCM

Presenters

- Lee Zhang
- Remko de Knikker

Setup

Cloud Shell Setup Instructions

This section will guide you through the pre-requisites and setup of the environment used in this workshop labs. It is broken up into the following steps:

1. Sign up for IBM Cloud
2. Login to IBM Cloud
3. Open Cloud Shell
4. Connect to OpenShift Cluster

---

1. Sign up for IBM Cloud

You will need an IBM Cloud ID for the workshop. If you already have an IBM Cloud ID, proceed to the next section. To create an ID:

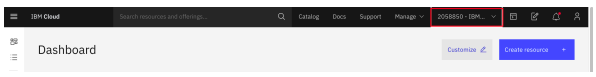
- Follow the steps outlined in NEWACCOUNT.

---

2. Login to IBM Cloud

To login to IBM Cloud,

1. Go to https://cloud.ibm.com in your browser and login.
2. Make sure that you are in the correct account#.



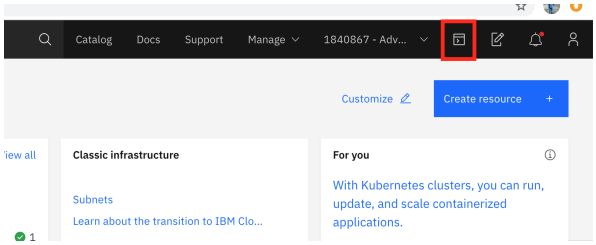
> Note: you may not have access to your OpenShift cluster if you are not in the right account#.

---

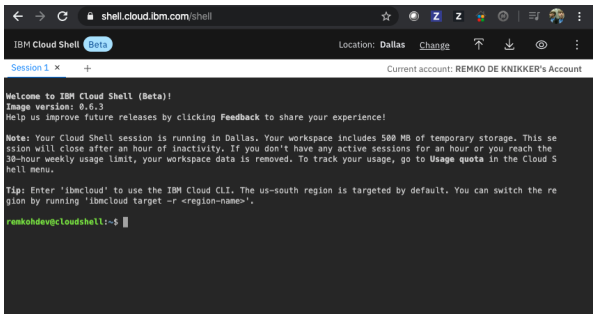
3. Open Cloud Shell

Most of the labs will run CLI commands. The IBM Cloud Shell is preconfigured with the full IBM Cloud CLI and tons of plug-ins and tools that you can use to manage apps, resources, and infrastructure.

1. From the IBM Cloud Home Page, select the terminal icon in the upper right hand menu.



2. It might take a few moments to create the instance and a new session which automatically logs you in through the IBM Cloud CLI.

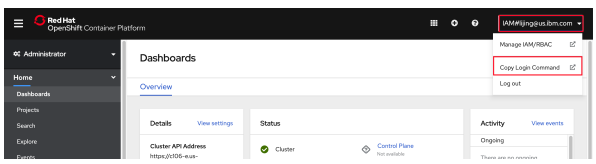


> *Note: Ensure the cloud shell is using the same account where your cluster is provisioned. Check that the account name shown in the top right of the screen, next to* `Current account` *is the correct one.*

---

4. Connect to OpenShift Cluster

1. In a new browser tab, go to https://cloud.ibm.com/kubernetes/clusters?platformType=openshift.
2. Select your cluster instance and open it.
3. Click `OpenShift web console` button on the top.
4. Click on your username in the upper right and select `Copy Login Command` option.



5. Click the `Display Token` link.

6. Copy the contents of the field `Log in with this token` to the clipboard. It provides a login command with a valid token for your username.
7. Go to the `Cloud Shell` tab.
8. Paste the `oc login command` in the IBM Cloud Shell terminal and run it.
9. After login to your cluster, set an environment variable for your cluster.

```
export CLUSTER_NAME=<your_cluster_name>
```

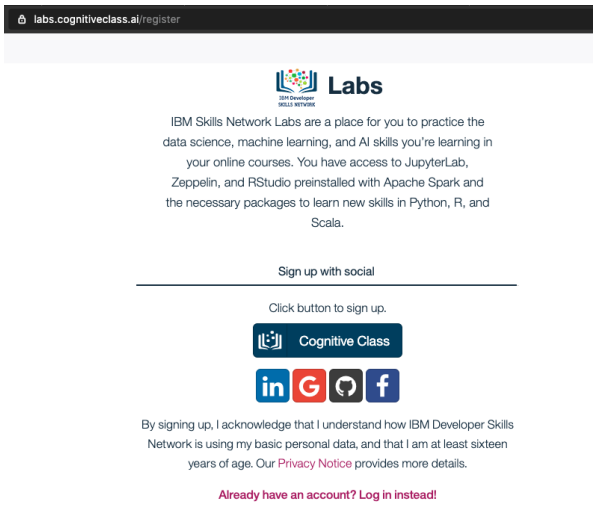
10. Verify you connect to the right cluster.

```
kubectl get pod
```

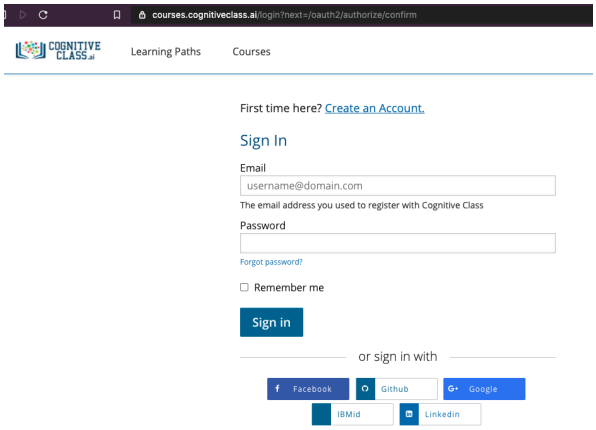
Access IBM Skills Network

1. Access IBM Skills Network

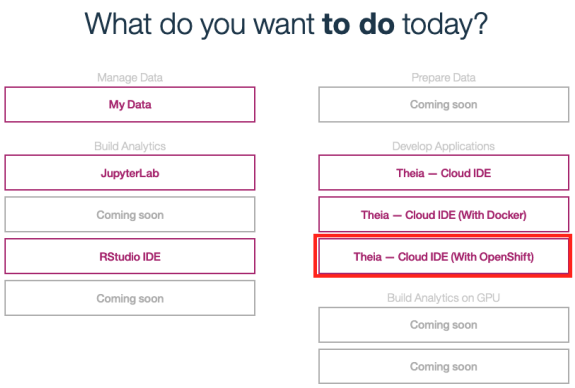
1. If you have registered your account, you can access the lab environment at https://labs.cognitiveclass.ai/ and go directly to step 6.
2. Navigate to https://labs.cognitiveclass.ai/register,



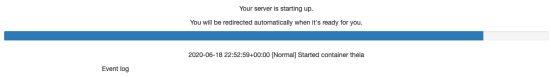
3. Create a new account with a Social login (LinkedIn, Google, Github or Facebook), or click the `Cognitive Class` button,



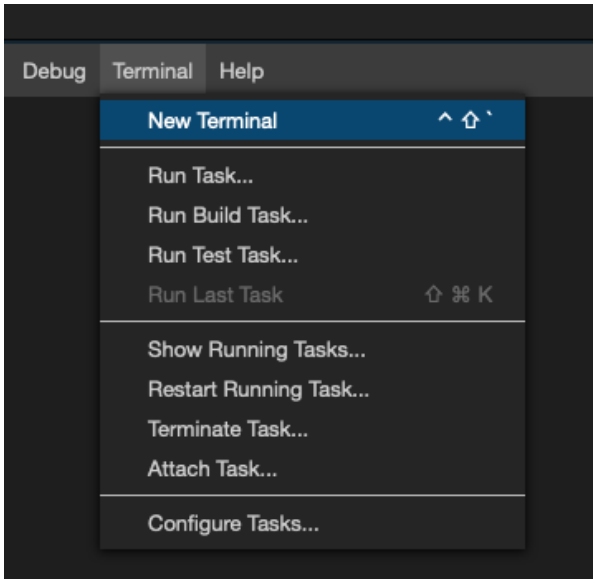
4. Click `Create an Account`,
5. Fill in your Email, Full Name, Public Username and password, click on the check boxes next to the `Privacy Notice` and `Terms of Service` to accept them. Then click on `Create Account`.
6. You will then be taken to a page with a list of sandbox environments. Click on the option for **Theia - Cloud IDE (With OpenShift)**



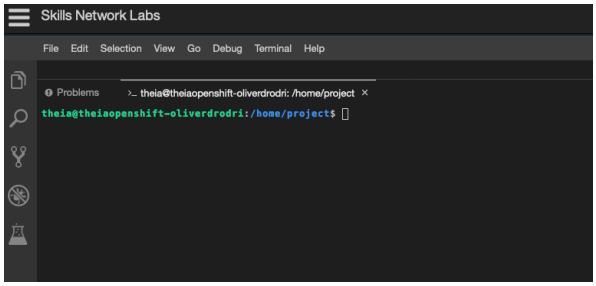
7. Wait a few minutes while your environment is created.



8. You will be taken to a blank editor page once your environment is ready.
9. What we really need is access to the terminal. Click on the `Terminal` tab near the top of the page and select **New Terminal**

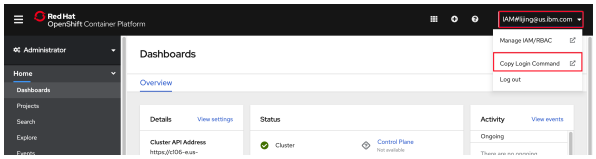


10. You can then click and drag the top of the terminal section upwards to make the terminal section bigger.



2. Connect to OpenShift Cluster

1. In a new browser tab, go to https://cloud.ibm.com/kubernetes/clusters?platformType=openshift.
2. Select your cluster instance and open it.
3. Click `OpenShift web console` button on the top.
4. Click on your username in the upper right and select `Copy Login Command` option.



5. Click the `Display Token` link.
6. Copy the contents of the field `Log in with this token` to the clipboard. It provides a login command with a valid token for your username.
7. Go to the `Cloud Shell` tab.
8. Paste the `oc login command` in the IBM Cloud Shell terminal and run it.
9. After login to your cluster, set an environment variable for your cluster.

```
export CLUSTER_NAME=<your_cluster_name>
```

10. Verify you connect to the right cluster.

```
kubectl get pod
```

---

3. Install s2i CLI tool

To install s2i CLI tool,

1. Download tar file.

```
curl -s https://api.github.com/repos/openshift/source-to-image/releas
  | grep browser_download_url \
  | grep linux-amd64 \
  | cut -d '"' -f 4 \
  | wget -qi -
```

2. Unzip tar file

```
tar xvf source-to-image*.gz
```

3. Make s2i CLI accessiblee.

```
sudo mv s2i /usr/local/bin
```

4. verify

```
s2i version
```

With that done, you can start the lab.

Day 1

Lab: Kubernetes 101

Lab overview

Lab 0: Setup - if you do not have access to a kubernetes environment, follow the suggestions here to get one. If you are in an IBM workshop and have already signed in to your kubernetes cluster, skip this lab and start with Lab 1.

Lab 1: This lab walks through creating and deploying a simple "guestbook" app written in Go as a net/http Server and accessing it.

Lab 2: Builds on lab 1 to expand to a more resilient setup which can survive having containers fail and recover. Lab 2 will also walk through basic services you need to get

started with Kubernetes and the IBM Cloud Container Service

Lab 3: Introduces configuration files used to describe deployments and services and their use to deploy an application.

Excercise - 1

Learn how to deploy an application to a Kubernetes cluster hosted within the IBM Container Service.

---

0. Prerequisites

Make sure you satisfy the prerequisites as outlined in Lab 0

---

1. Deploy the guestbook application

In this part of the lab we will deploy an application called `guestbook` that has already been built and uploaded to DockerHub under the name `ibmcom/guestbook:v1`.

1. Start by running `guestbook`:

```
kubectl create deployment guestbook --image=ibmcom/guestbook:v1
```

This action will take a bit of time. To check the status of the running application, you can use `$ kubectl get pods`.
You should see output similar to the following:

```
kubectl get pods
```

Eventually, the status should show up as `Running`.

```
1  $ kubectl get pods
```

```
NAME                          READY     STATUS              RESTARTS
guestbook-59bd679fdc-bxdg7    1/1       Running             0
```

The end result of the run command is not just the pod containing our application containers, but a Deployment resource that manages the lifecycle of those pods.

2. Once the status reads `Running`, we need to expose that deployment as a service so we can access it through the IP of the worker nodes. The `guestbook` application listens on port 3000. Run:

```
kubectl expose deployment guestbook --type="NodePort" --port=3000
```

3. To find the port used on that worker node, examine your new service:

```
$ kubectl get service guestbook
NAME        TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          A
guestbook   NodePort   10.10.10.253   <none>        3000:31208/TCP   1
```

We can see that our `<nodeport>` is `31208`. We can see in the output the port mapping from 3000 inside the pod exposed to the cluster on port 31208. This port in the 31000 range is automatically chosen, and could be different for you.

4. `guestbook` is now running on your cluster, and exposed to the internet. We need to find out where it is accessible. The worker nodes running in the container service get external IP addresses. Get the workers for your cluster and note one (any one) of the public IPs listed on the `<public-IP>` line. Replace `$CLUSTER_NAME` with your cluster name unless you have this environment variable set.

```
$ ibmcloud ks workers --cluster $CLUSTER_NAME
OK
ID                                                 Public IP        Pr
kube-hou02-pa1e3ee39f549640aebea69a444f51fe55-w1   173.193.99.136   10
```

We can see that our `<public-IP>` is `173.193.99.136`.

5. Now that you have both the address and the port, you can now access the application in the web browser at `<public-IP>:<nodeport>`. In the example case this is `173.193.99.136:31208`.

Congratulations, you've now deployed an application to Kubernetes!

When you're all done, continue to the next lab of this course.

Excercise - 2

In this lab, you'll learn how to update the number of instances a deployment has and how to safely roll out an update of your application on Kubernetes.

For this lab, you need a running deployment of the `guestbook` application from the previous lab. If you deleted it, recreate it using:

```
kubectl create deployment guestbook --image=ibmcom/guestbook:v1
```

---

1. Scale apps with replicas

A *replica* is a copy of a pod that contains a running service. By having multiple replicas of a pod, you can ensure your deployment has the available resources to handle increasing load on your application.

1. `kubectl` provides a `scale` subcommand to change the size of an existing deployment. Let's increase our capacity from a single running instance of `guestbook` up to 10 instances:

   ```
   kubectl scale --replicas=10 deployment guestbook
   ```

   Kubernetes will now try to make reality match the desired state of 10 replicas by starting 9 new pods with the same configuration as the first.
2. To see your changes being rolled out, you can run:

   ```
   kubectl rollout status deployment guestbook
   ```

The rollout might occur so quickly that the following messages might *not* display:

```
$ kubectl rollout status deployment guestbook
Waiting for rollout to finish: 1 of 10 updated replicas are available.
Waiting for rollout to finish: 2 of 10 updated replicas are available.
Waiting for rollout to finish: 3 of 10 updated replicas are available.
Waiting for rollout to finish: 4 of 10 updated replicas are available.
Waiting for rollout to finish: 5 of 10 updated replicas are available.
Waiting for rollout to finish: 6 of 10 updated replicas are available.
Waiting for rollout to finish: 7 of 10 updated replicas are available.
Waiting for rollout to finish: 8 of 10 updated replicas are available.
Waiting for rollout to finish: 9 of 10 updated replicas are available.
deployment "guestbook" successfully rolled out
```

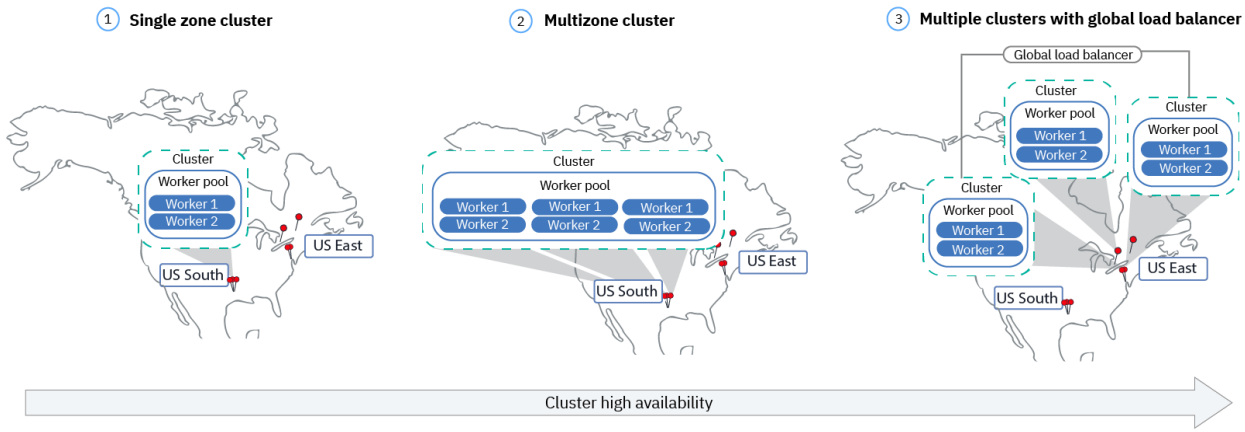
3. Once the rollout has finished, ensure your pods are running by using:

```
kubectl get pods
```

You should see output listing 10 replicas of your deployment:

```
$ kubectl get pods
NAME                         READY     STATUS    RESTARTS   AGE
guestbook-562211614-1tqm7    1/1       Running   0          1d
guestbook-562211614-1zqn4    1/1       Running   0          2m
guestbook-562211614-5htdz    1/1       Running   0          2m
guestbook-562211614-6h04h    1/1       Running   0          2m
guestbook-562211614-ds9hb    1/1       Running   0          2m
guestbook-562211614-nb5qp    1/1       Running   0          2m
guestbook-562211614-vtfp2    1/1       Running   0          2m
guestbook-562211614-vz5qw    1/1       Running   0          2m
guestbook-562211614-zksw3    1/1       Running   0          2m
guestbook-562211614-zsp0j    1/1       Running   0          2m
```

Tip: Another way to improve availability is to add clusters and regions to your deployment, as shown in the following diagram:



HA with more clusters and regions

---

2. Update and roll back apps

Kubernetes allows you to do rolling upgrade of your application to a new container image. This allows you to easily update the running image and also allows you to easily undo a rollout if a problem is discovered during or after deployment.

In the previous lab, we used an image with a `v1` tag. For our upgrade we'll use the image with the `v2` tag.

To update and roll back:

1. Using `kubectl`, you can now update your deployment to use the `v2` image. `kubectl` allows you to change details about existing resources with the `set` subcommand. We can use it to change the image being used.

   ```
   kubectl set image deployment/guestbook guestbook=ibmcom/guestbook:v2
   ```

   Note that a pod could have multiple containers, each with its own name. Each image can be changed individually or all at once by referring to the name. In the case of our `guestbook` Deployment, the container name is also `guestbook`. Multiple containers can be updated at the same time. (More information.)

2. To check the status of the rollout, run:

```
kubectl rollout status deployment/guestbook
```

The rollout might occur so quickly that the following messages might *not* display:

```
$ kubectl rollout status deployment/guestbook
Waiting for rollout to finish: 2 out of 10 new replicas have been upda
Waiting for rollout to finish: 3 out of 10 new replicas have been upda
Waiting for rollout to finish: 3 out of 10 new replicas have been upda
Waiting for rollout to finish: 3 out of 10 new replicas have been upda
Waiting for rollout to finish: 4 out of 10 new replicas have been upda
Waiting for rollout to finish: 4 out of 10 new replicas have been upda
Waiting for rollout to finish: 4 out of 10 new replicas have been upda
Waiting for rollout to finish: 4 out of 10 new replicas have been upda
Waiting for rollout to finish: 4 out of 10 new replicas have been upda
Waiting for rollout to finish: 5 out of 10 new replicas have been upda
Waiting for rollout to finish: 5 out of 10 new replicas have been upda
Waiting for rollout to finish: 5 out of 10 new replicas have been upda
Waiting for rollout to finish: 6 out of 10 new replicas have been upda
Waiting for rollout to finish: 6 out of 10 new replicas have been upda
Waiting for rollout to finish: 6 out of 10 new replicas have been upda
Waiting for rollout to finish: 7 out of 10 new replicas have been upda
Waiting for rollout to finish: 7 out of 10 new replicas have been upda
Waiting for rollout to finish: 7 out of 10 new replicas have been upda
Waiting for rollout to finish: 7 out of 10 new replicas have been upda
Waiting for rollout to finish: 8 out of 10 new replicas have been upda
Waiting for rollout to finish: 8 out of 10 new replicas have been upda
Waiting for rollout to finish: 8 out of 10 new replicas have been upda
Waiting for rollout to finish: 8 out of 10 new replicas have been upda
Waiting for rollout to finish: 9 out of 10 new replicas have been upda
Waiting for rollout to finish: 9 out of 10 new replicas have been upda
Waiting for rollout to finish: 9 out of 10 new replicas have been upda
Waiting for rollout to finish: 1 old replicas are pending termination.
Waiting for rollout to finish: 1 old replicas are pending termination.
Waiting for rollout to finish: 1 old replicas are pending termination.
Waiting for rollout to finish: 9 of 10 updated replicas are available.
Waiting for rollout to finish: 9 of 10 updated replicas are available.
Waiting for rollout to finish: 9 of 10 updated replicas are available.
deployment "guestbook" successfully rolled out
```

3. Test the application as before, by accessing `<public-IP>:<nodeport>` in the browser to confirm your new code is active.
   Remember, to get the "nodeport" and "public-ip" use the following commands. Replace `$CLUSTER_NAME` with the name of your cluster if the environment variable is not set.:

   ```
   kubectl describe service guestbook
   ```

   and

   ```
   ibmcloud ks workers --cluster $CLUSTER_NAME
   ```

   To verify that you're running "v2" of guestbook, look at the title of the page, it should now be `Guestbook - v2`. If you are using a browser, make sure you force refresh (invalidating your cache).
4. If you want to undo your latest rollout, use:

   ```
   kubectl rollout undo deployment guestbook
   ```

   You can then use this command to see the status:

   ```
   kubectl rollout status deployment/guestbook
   ```

5. When doing a rollout, you see references to *old* replicas and *new* replicas. The *old* replicas are the original 10 pods deployed when we scaled the application. The *new* replicas come from the newly created pods with the different image. All of these pods are owned by the Deployment. The deployment manages these two sets of pods with a resource called a ReplicaSet. We can see the guestbook ReplicaSets with:

   ```
   ```

```
$ kubectl get replicasets -l app=guestbook
NAME                   DESIRED   CURRENT   READY     AGE
guestbook-5f5548d4f    10        10        10        21m
guestbook-768cc55c78   0         0         0         3h
```

Before we continue, let's delete the application so we can learn about a different way to achieve the same results:

To remove the deployment, use

```
kubectl delete deployment guestbook
```

To remove the service, use:

```
kubectl delete service guestbook
```

Congratulations! You deployed the second version of the app. Lab 2 is now complete. Continue to the next lab of this course.

Excercise - 3

Lab 3: Scale and update apps natively, building multi-tier applications.

In this lab you'll learn how to deploy the same guestbook application we deployed in the previous labs, however, instead of using the `kubectl` command line helper functions we'll be deploying the application using configuration files. The configuration file mechanism allows you to have more fine-grained control over all of resources being created within the Kubernetes cluster.

Before we work with the application we need to clone a github repo:

```
git clone https://github.com/IBM/guestbook.git
```

This repo contains multiple versions of the guestbook application as well as the configuration files we'll use to deploy the pieces of the application.

Change directory by running the command

```
cd guestbook/v1
```

You will find all the configurations files for this exercise in this directory.

1. Scale apps natively

Kubernetes can deploy an individual pod to run an application but when you need to scale it to handle a large number of requests a `Deployment` is the resource you want to use. A Deployment manages a collection of similar pods. When you ask for a specific number of replicas the Kubernetes Deployment Controller will attempt to maintain that number of replicas at all times.

Every Kubernetes object we create should provide two nested object fields that govern the object's configuration: the object `spec` and the object `status`. Object `spec` defines the desired state, and object `status` contains Kubernetes system provided information about the actual state of the resource. As described before, Kubernetes will attempt to reconcile your desired state with the actual state of the system.

For Object that we create we need to provide the `apiVersion` you are using to create the object, `kind` of the object we are creating and the `metadata` about the object such as a `name`, set of `labels` and optionally `namespace` that this object should belong.

Consider the following deployment configuration for guestbook application

guestbook-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook-v1
  labels:
    app: guestbook
    version: "1.0"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: guestbook
  template:
    metadata:
      labels:
        app: guestbook
        version: "1.0"
    spec:
      containers:
      - name: guestbook
        image: ibmcom/guestbook:v1
        ports:
        - name: http-server
          containerPort: 3000
```

The above configuration file create a deployment object named 'guestbook' with a pod containing a single container running the image `ibmcom/guestbook:v1`. Also the

configuration specifies replicas set to 3 and Kubernetes tries to make sure that at least three active pods are running at all times.

- Create guestbook deployment
  To create a Deployment using this configuration file we use the following command:

  ```
  kubectl create -f guestbook-deployment.yaml
  ```

- List the pod with label app=guestbook
  We can then list the pods it created by listing all pods that have a label of "app" with a value of "guestbook". This matches the labels defined above in the yaml file in the `spec.template.metadata.labels` section.

  ```
  kubectl get pods -l app=guestbook
  ```

When you change the number of replicas in the configuration, Kubernetes will try to add, or remove, pods from the system to match your request. To can make these modifications by using the following command:

```
kubectl edit deployment guestbook-v1
```

This will retrieve the latest configuration for the Deployment from the Kubernetes server and then load it into an editor for you. You'll notice that there are a lot more fields in this version than the original yaml file we used. This is because it contains all of the properties about the Deployment that Kubernetes knows about, not just the ones we chose to specify when we create it. Also notice that it now contains the `status` section mentioned previously.

To exit the `vi` editor, type `:q!`, of if you made changes that you want to see reflected, save them using `:wq`.

You can also edit the deployment file we used to create the Deployment to make changes. You should use the following command to make the change effective when you edit the deployment locally.

```
kubectl apply -f guestbook-deployment.yaml
```

This will ask Kubernetes to "diff" our yaml file with the current state of the Deployment and apply just those changes.

We can now define a Service object to expose the deployment to external clients.

guestbook-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: guestbook
  labels:
    app: guestbook
spec:
  ports:
  - port: 3000
    targetPort: http-server
  selector:
    app: guestbook
  type: LoadBalancer
```

The above configuration creates a Service resource named guestbook. A Service can be used to create a network path for incoming traffic to your running application. In this case, we are setting up a route from port 3000 on the cluster to the "http-server" port on our app, which is port 3000 per the Deployment container spec.

- Let us now create the guestbook service using the same type of command we used when we created the Deployment:

```
kubectl create -f guestbook-service.yaml
```

- Test guestbook app using a browser of your choice using the url `<your-cluster-ip>:<node-port>`
  Remember, to get the `nodeport` and `public-ip` use the following commands, replacing `CLUSTER_NAME` the name of your cluster if the environment variable is not already set.

  ```
  kubectl describe service guestbook
  ```

  and

  ```
  ibmcloud ks workers --cluster $CLUSTER_NAME
  ```

---

2. Connect to a back-end service.

If you look at the guestbook source code, under the `guestbook/v1/guestbook` directory, you'll notice that it is written to support a variety of data stores. By default it will keep the log of guestbook entries in memory. That's ok for testing purposes, but as you get into a more "real" environment where you scale your application that model will not work because based on which instance of the application the user is routed to they'll see very different results.

To solve this we need to have all instances of our app share the same data store - in this case we're going to use a redis database that we deploy to our cluster. This instance of redis will be defined in a similar manner to the guestbook.

redis-master-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
      role: master
  template:
    metadata:
      labels:
        app: redis
        role: master
    spec:
      containers:
      - name: redis-master
        image: redis:3.2.9
        ports:
        - name: redis-server
          containerPort: 6379
```

This yaml creates a redis database in a Deployment named 'redis-master'. It will create a single instance, with replicas set to 1, and the guestbook app instances will connect to it to persist data, as well as read the persisted data back. The image running in the container is 'redis:3.2.9' and exposes the standard redis port 6379.

- Create a redis Deployment, like we did for guestbook:

```
kubectl create -f redis-master-deployment.yaml
```

- Check to see that redis server pod is running:

```
$ kubectl get pods -lapp=redis,role=master
NAME                READY     STATUS    RESTARTS   AGE
redis-master-q9zg7  1/1       Running   0          2d
```

- Let us test the redis standalone. Replace the pod name `redis-master-q9zg7` with the name of your pod.

```
kubectl exec -it redis-master-q9zg7 redis-cli
```

The kubectl exec command will start a secondary process in the specified container. In this case we're asking for the "redis-cli" command to be executed in the container named "redis-master-q9zg7". When this process ends the "kubectl exec" command will also exit but the other processes in the container will not be impacted.
Once in the container we can use the "redis-cli" command to make sure the redis database is running properly, or to configure it if needed.

```
redis-cli> ping
PONG
redis-cli> exit
```

Now we need to expose the `redis-master` Deployment as a Service so that the guestbook application can connect to it through DNS lookup.

redis-master-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
spec:
  ports:
  - port: 6379
    targetPort: redis-server
  selector:
    app: redis
```

```
14      role: master
```

This creates a Service object named 'redis-master' and configures it to target port 6379 on the pods selected by the selectors "app=redis" and "role=master".

- Create the service to access redis master:

```
kubectl create -f redis-master-service.yaml
```

- Restart guestbook so that it will find the redis service to use database:

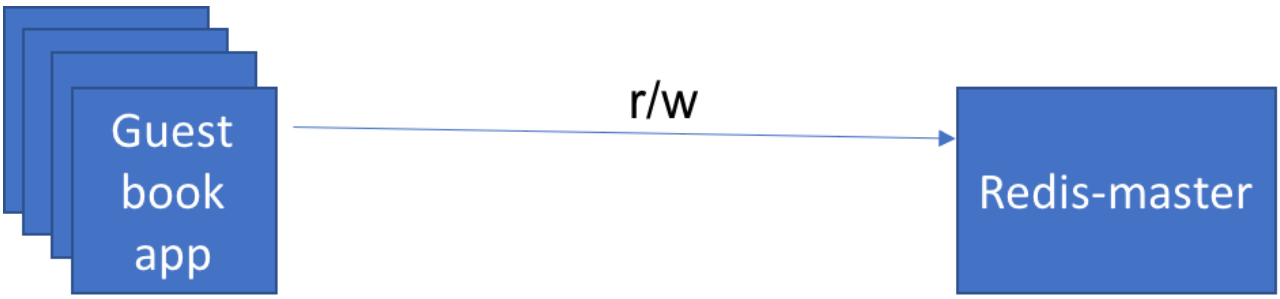
```
kubectl delete deploy guestbook-v1
```

```
kubectl create -f guestbook-deployment.yaml
```

- Test guestbook app using a browser of your choice using the url `<your-cluster-ip>:<node-port>` , or by simply refreshing the page if you already have the app open in another window.

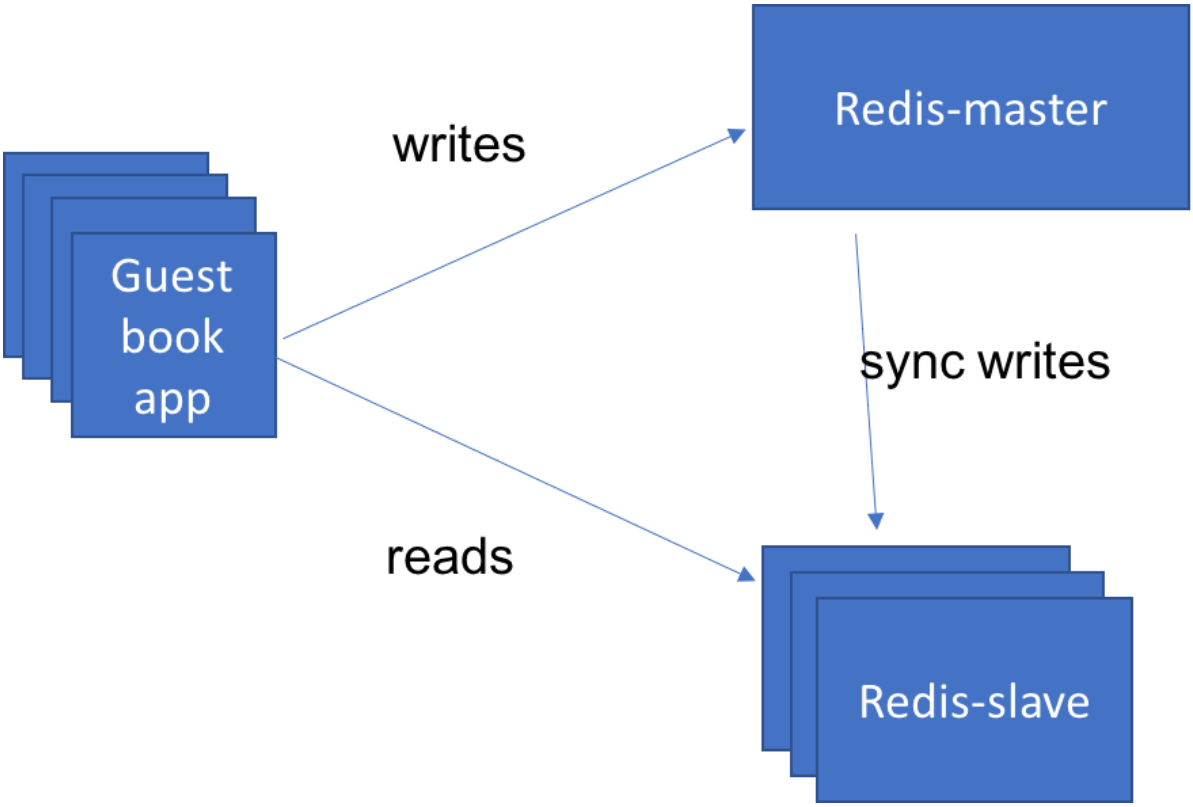
You can see now that if you open up multiple browsers and refresh the page to access the different copies of guestbook that they all have a consistent state. All instances write to the same backing persistent storage, and all instances read from that storage to display the guestbook entries that have been stored.

We have our simple 3-tier application running but we need to scale the application if traffic increases. Our main bottleneck is that we only have one database server to process each request coming though guestbook. One simple solution is to separate the reads and write such that they go to different databases that are replicated properly to achieve data consistency.



rw_to_master

Create a deployment named 'redis-slave' that can talk to redis database to manage data reads. In order to scale the database we use the pattern where we can scale the reads using redis slave deployment which can run several instances to read. Redis slave deployments is configured to run two replicas.



w_to_master-r_to_slave

redis-slave-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
    name: redis-slave
    labels:
      app: redis
      role: slave
spec:
    replicas: 2
    selector:
      matchLabels:
        app: redis
        role: slave
    template:
      metadata:
        labels:
          app: redis
          role: slave
      spec:
        containers:
        - name: redis-slave
          image: ibmcom/guestbook-redis-slave:v2
          ports:
          - name: redis-server
            containerPort: 6379
```

- Create the pod running redis slave deployment.

```
kubectl create -f redis-slave-deployment.yaml
```

    - Check if all the slave replicas are running

```
$ kubectl get pods -lapp=redis,role=slave
NAME                READY     STATUS    RESTARTS   AGE
redis-slave-kd7vx   1/1       Running   0          2d
redis-slave-wwcxw   1/1       Running   0          2d
```

- And then go into one of those pods and look at the database to see that everything looks right. Replace the pod name `redis-slave-kd7vx` with your own pod name. If you get the back `(empty list or set)` when you print the keys, go to the guestbook application and add an entry!

```
$ kubectl exec -it redis-slave-kd7vx  redis-cli
127.0.0.1:6379> keys *
1) "guestbook"
127.0.0.1:6379> lrange guestbook 0 10
1) "hello world"
2) "welcome to the Kube workshop"
127.0.0.1:6379> exit
```

Deploy redis slave service so we can access it by DNS name. Once redeployed, the application will send "read" operations to the `redis-slave` pods while "write" operations will go to the `redis-master` pods.

redis-slave-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
spec:
  ports:
  - port: 6379
    targetPort: redis-server
  selector:
    app: redis
    role: slave
```

- Create the service to access redis slaves.

```
kubectl create -f redis-slave-service.yaml
```

- Restart guestbook so that it will find the slave service to read from.

```
kubectl delete deploy guestbook-v1
```

```
kubectl create -f guestbook-deployment.yaml
```

- Test guestbook app using a browser of your choice using the url `<your-cluster-ip>:<node-port>` , or by simply refreshing the page if you have the app open in another window.

That's the end of the lab. Now let's clean-up our environment:

```
kubectl delete -f guestbook-deployment.yaml
kubectl delete -f guestbook-service.yaml
kubectl delete -f redis-slave-service.yaml
kubectl delete -f redis-slave-deployment.yaml
kubectl delete -f redis-master-service.yaml
kubectl delete -f redis-master-deployment.yaml
```

Lab: Helm 101

Helm is often described as the Kubernetes application package manager. So, what does Helm give you over using `kubectl` directly?

Objectives

These labs provide an insight on the advantages of using Helm over using Kubernetes directly through `kubectl`. In several of the labs there are two scenarios. The first scenario gives an example of how to perform the task using `kubectl`, the second scenario, using `helm`. When you complete all the labs, you'll:

- Understand the core concepts of Helm
- Understand the advantages of deployment using Helm over Kubernetes directly, looking at:
  - Application management
  - Updates
  - Configuration
  - Revision management
  - Repositories and chart sharing

Helm Status

Refer to Helm Status for more details.

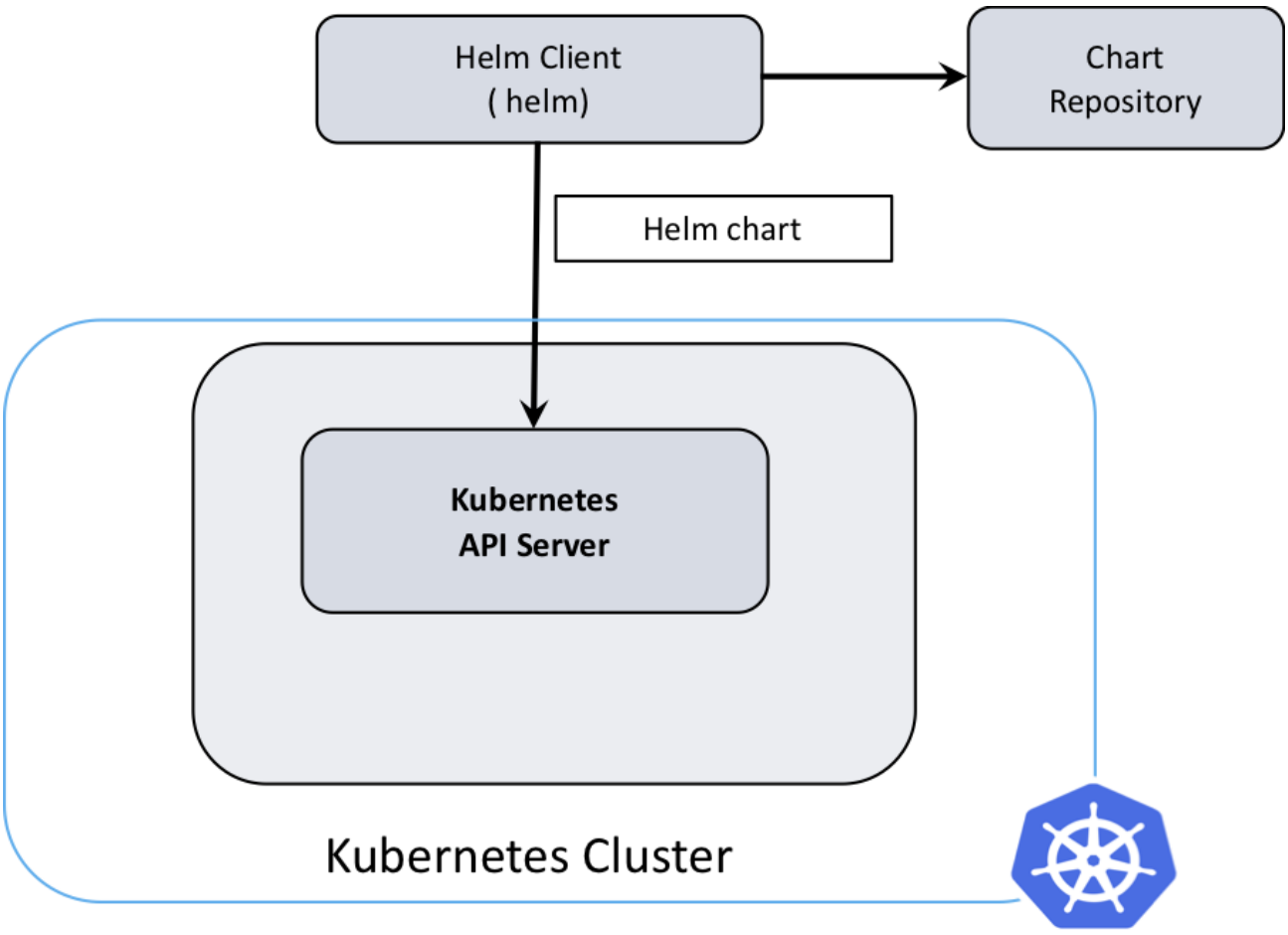
Prerequisites

- Have a running Kubernetes cluster. See the IBM Cloud Kubernetes Service or Kubernetes Getting Started Guide for details about creating a cluster.

- Have Helm installed and initialized with the Kubernetes cluster. See Installing Helm on IBM Cloud Kubernetes Service or the Helm Quickstart Guide for getting started with Helm.

Helm Overview

Helm is a tool that streamlines installation and management of Kubernetes applications. It uses a packaging format called "charts", which are a collection of files that describe Kubernetes resources. It can run anywhere (laptop, CI/CD, etc.) and is available for various operating systems, like OSX, Linux and Windows.



helm-architecture

Helm 3 pivoted from the Helm 2 client-server architecture to a client architecture. The client is still called `helm` and, there is an improved Go library which encapsulates the Helm logic so that it can be leveraged by different clients. The client is a CLI which users interact with

to perform different operations like install/upgrade/delete etc. The client interacts with the Kubernetes API server and the chart repository. It renders Helm template files into Kubernetes manifest files which it uses to perform operations on the Kubernetes cluster via the Kubernetes API. See the Helm Architecture for more details.

A chart is organized as a collection of files inside of a directory where the directory name is the name of the chart. It contains template YAML files which facilitates providing configuration values at runtime and eliminates the need of modifying YAML files. These templates provide programming logic as they are based on the Go template language, functions from the Sprig lib and other specialized functions.

The chart repository is a location where packaged charts can be stored and shared. This is akin to the image repository in Docker. Refer to The Chart Repository Guide for more details.

Helm Abstractions

Helm terms :

- Chart - It contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster. A chart is basically a package of pre-configured Kubernetes resources.
- Config - Contains configuration information that can be merged into a packaged chart to create a releasable object.
- helm - Helm client. It renders charts into manifest files. It interacts directly with the Kubernetes API server to install, upgrade, query, and remove Kubernetes resources.
- Release - An instance of a chart running in a Kubernetes cluster.
- Repository - Place where charts reside and can be shared with others.

To get started, head on over to Lab 1.

Setup Helm

For the Helm labs of this workshop we will be using Helm Version 3. If you are running the labs using the IBM Cloud Shell, we need to complete a couple of setup steps before we proceed. This section is broken up into the following steps:

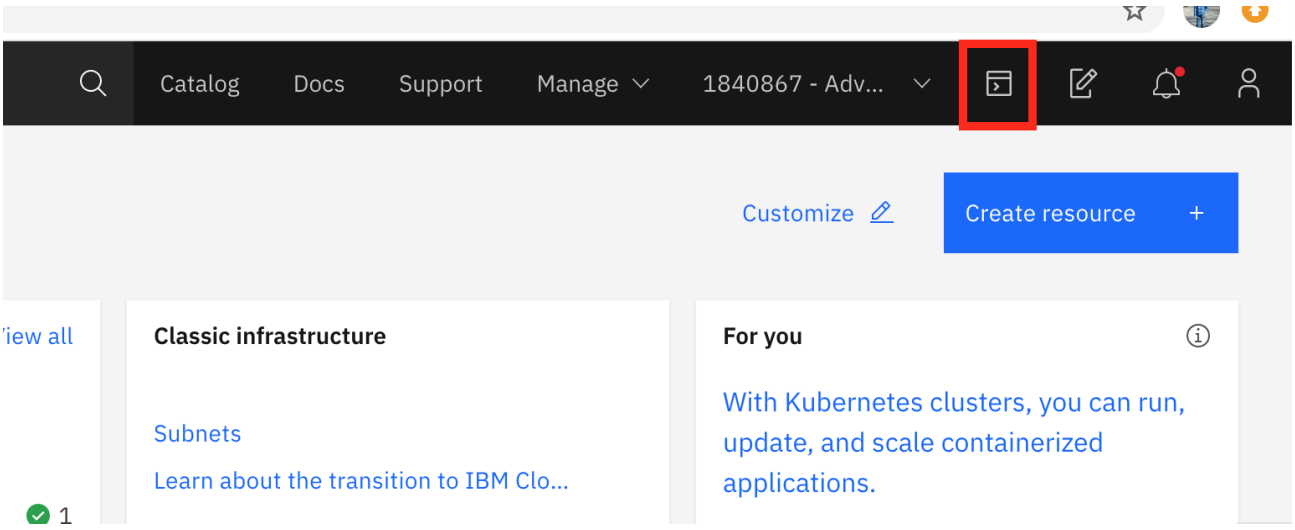
1. Access the Cloud Shell
2. Install Helm Version 3
3. Configure Kubectl

---

1. Access the Cloud Shell

If you do not already have it open from the workshop setup section, go ahead and open the Cloud shell.

1. From the IBM Cloud Home Page, select the terminal icon in the upper lefthand menu.



Terminal Button

> *Note: Ensure the cloud shell is using the same account where your cluster is provisioned. Ensure that the account name shown in the top right of the screen, next to* `Current account` *is the correct one.*

---

2. Install Helm Version 3

The version of Helm that comes pre-configured in the Cloud Shell needs to be updated:

1. Run the following commands to install Helm Version 3

```
wget https://get.helm.sh/helm-v3.2.0-linux-amd64.tar.gz
```

```
tar -zxvf helm-v3.2.0-linux-amd64.tar.gz
```

```
ls -al
```

```
echo 'export PATH=$HOME/linux-amd64:$PATH' > .bash_profile
```

```
source .bash_profile
```

```
helm version --short
```

The result is that you should have Helm Version 3 installed.

```
$ helm version --short
v3.2.0+ge11b7ce
```

3. Configure Kubectl

If you have not setup your kubectl to access your cluster, you can do so in the Cloud Shell.

1. Run the `ibmcloud ks clusters` command to verify the terminal and setup for access to the cluster

```
ibmcloud ks clusters
```

2. Configure the `kubectl` cli available within the terminal for access to your cluster. If you previously stored your cluster name to an environment variable, use that (ie. `$CLUSTER_NAME`), otherwise copy and paste your cluster name from the previous commands output to the `$CLUSTER_NAME` portion below.

```
ibmcloud ks cluster config --cluster $CLUSTER_NAME
```

3. Verify access to the Kubernetes API by getting the namespaces.

```
kubectl get namespace
```

4. You should see output similar to the following, if so, then your're ready to continue.

```
NAME              STATUS   AGE
default           Active   125m
ibm-cert-store    Active   121m
ibm-system        Active   124m
kube-node-lease   Active   125m
kube-public       Active   125m
kube-system       Active   125m
```

Lab 1 - Deploy Application

Let's investigate how Helm can help us focus on other things by letting a chart do the work for us. We'll first deploy an application to a Kubernetes cluster by using `kubectl` and then show how we can offload the work to a chart by deploying the same app with Helm.

The application is the Guestbook App, which is a sample multi-tier web application.

Scenario 1: Deploy the application using `kubectl`

In this part of the lab, we will deploy the application using the Kubernetes client `kubectl`. We will use Version 1 of the app for deploying here.

If you already have a copy of the guestbook application installed from the kube101 lab, skip this section and go the `helm` example in Scenario 2.

Clone the Guestbook App repo to get the files:

```
git clone https://github.com/IBM/guestbook.git
```

1. Use the configuration files in the cloned Git repository to deploy the containers and create services for them by using the following commands:

```
$ cd guestbook/v1

$ kubectl create -f redis-master-deployment.yaml
deployment.apps/redis-master created

$ kubectl create -f redis-master-service.yaml
service/redis-master created

$ kubectl create -f redis-slave-deployment.yaml
deployment.apps/redis-slave created

```

```
$ kubectl create -f redis-slave-service.yaml
service/redis-slave created

$ kubectl create -f guestbook-deployment.yaml
deployment.apps/guestbook-v1 created

$ kubectl create -f guestbook-service.yaml
service/guestbook created
```

Refer to the guestbook README for more details.

2. View the guestbook:

   You can now play with the guestbook that you just created by opening it in a browser (it might take a few moments for the guestbook to come up).

   - **Local Host:** If you are running Kubernetes locally, view the guestbook by navigating to `http://localhost:3000` in your browser.
   - **Remote Host:**
   - To view the guestbook on a remote host, locate the external IP and port of the load balancer in the **EXTERNAL-IP** and **PORTS** columns of the `$ kubectl get services` output.

```
$ kubectl get services
NAME           TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S
guestbook      LoadBalancer   172.21.252.107   50.23.5.136   3000:3
redis-master   ClusterIP      172.21.97.222    <none>        6379/T
redis-slave    ClusterIP      172.21.43.70     <none>        6379/T
.........
```

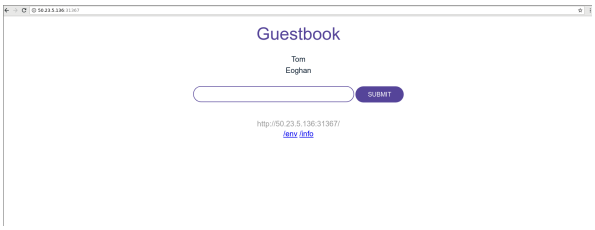
   In this scenario the URL is `http://50.23.5.136:31838`.

   Note: If no external IP is assigned, then you can get the external IP with the following command:

```
$ kubectl get nodes -o wide
NAME           STATUS    ROLES     AGE       VERSION        EXTERNA
10.47.122.98   Ready     <none>    1h        v1.10.11+IKS   173.193
```

In this scenario the URL is `http://173.193.92.112:31838`.

- Navigate to the output given (for example `http://50.23.5.136:31838`) in your browser. You should see the guestbook now displaying in your browser:



Scenario 2: Deploy the application using Helm

In this part of the lab, we will deploy the application by using Helm. We will set a release name of `guestbook-demo` to distinguish it from the previous deployment. The Helm chart is available here. Clone the Helm 101 repo to get the files:

```
git clone https://github.com/IBM/helm101
```

A chart is defined as a collection of files that describe a related set of Kubernetes resources. We probably then should take a look at the the files before we go and install the chart. The files for the `guestbook` chart are as follows:

```
.
├── Chart.yaml    \\ A YAML file containing information about the chart
├── LICENSE       \\ A plain text file containing the license for the char
├── README.md     \\ A README providing information about the chart usage,
├── templates     \\ A directory of templates that will generate valid Kub
│   ├── _helpers.tpl               \\ Template helpers/definitions that ar
│   ├── guestbook-deployment.yaml  \\ Guestbook app container resource
│   ├── guestbook-service.yaml     \\ Guestbook app service resource
│   ├── NOTES.txt                  \\ A plain text file containing short u
│   ├── redis-master-deployment.yaml  \\ Redis master container resource
│   ├── redis-master-service.yaml     \\ Redis master service resource
│   ├── redis-slave-deployment.yaml   \\ Redis slave container resource
│   └── redis-slave-service.yaml      \\ Redis slave service resource
└── values.yaml   \\ The default configuration values for the chart
```

Note: The template files shown above will be rendered into Kubernetes manifest files before being passed to the Kubernetes API server. Therefore, they map to the manifest files that we deployed when we used `kubectl` (minus the helper and notes files).

Let's go ahead and install the chart now. If the `helm-demo` namespace does not exist, you will need to create it using:

```
kubectl create namespace helm-demo
```

1. Install the app as a Helm chart:

   ```
   $ cd helm101/charts

   $ helm install guestbook-demo ./guestbook/ --namespace helm-demo
   NAME: guestbook-demo
   ...
   ```

   You should see output similar to the following:

   ```
   NAME: guestbook-demo
   LAST DEPLOYED: Mon Feb 24 18:08:02 2020
   NAMESPACE: helm-demo
   STATUS: deployed
   REVISION: 1
   TEST SUITE: None
   NOTES:
   1. Get the application URL by running these commands:
     NOTE: It may take a few minutes for the LoadBalancer IP to be availa
         You can watch the status of by running 'kubectl get svc -w guest
     export SERVICE_IP=$(kubectl get svc --namespace helm-demo guestbook-
     echo http://$SERVICE_IP:3000
   ```

   The chart install performs the Kubernetes deployments and service creations of the redis master and slaves, and the guestbook app, as one. This is because the chart is a

collection of files that describe a related set of Kubernetes resources and Helm manages the creation of these resources via the Kubernetes API.
Check the deployment:

```
kubectl get deployment guestbook-demo --namespace helm-demo
```

You should see output similar to the following:

```
$ kubectl get deployment guestbook-demo --namespace helm-dem
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
guestbook-demo   2/2     2            2           51m
```

To check the status of the running application pods, use:

```
kubectl get pods --namespace helm-demo
```

You should see output similar to the following:

```
$ kubectl get pods --namespace helm-demo
NAME                            READY     STATUS    RESTARTS   AGE
guestbook-demo-6c9cf8b9-jwbs9   1/1       Running   0          52m
guestbook-demo-6c9cf8b9-qk4fb   1/1       Running   0          52m
redis-master-5d8b66464f-j72jf   1/1       Running   0          52m
redis-slave-586b4c847c-2xt99    1/1       Running   0          52m
redis-slave-586b4c847c-q7rq5    1/1       Running   0          52m
```

To check the services, use:

```
kubectl get services --namespace helm-demo
```

```
$ kubectl get services --namespace helm-demo
NAME             TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)
guestbook-demo   LoadBalancer   172.21.43.244    <pending>     3000:31
redis-master     ClusterIP      172.21.12.43     <none>        6379/TC
redis-slave      ClusterIP      172.21.176.148   <none>        6379/TC
```

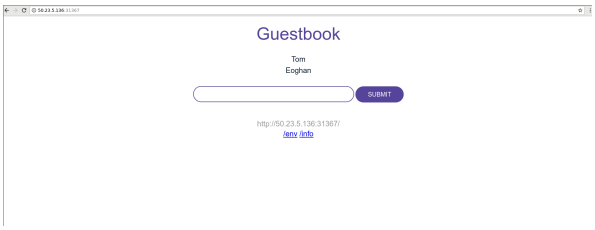
2. View the guestbook:
   You can now play with the guestbook that you just created by opening it in a browser (it might take a few moments for the guestbook to come up).
   - **Local Host:** If you are running Kubernetes locally, view the guestbook by navigating to `http://localhost:3000` in your browser.
   - **Remote Host:**
   - To view the guestbook on a remote host, locate the external IP and the port of the load balancer by following the "NOTES" section in the install output. The commands will be similar to the following:

```
$ export SERVICE_IP=$(kubectl get svc --namespace helm-demo guestbo
$ echo http://$SERVICE_IP
http://50.23.5.136
```

   Combine the service IP with the port of the service printed earlier. In this scenario the URL is `http://50.23.5.136:31367`.
   Note: If no external IP is assigned, then you can get the external IP with the following command:

```
$ kubectl get nodes -o wide
NAME           STATUS    ROLES     AGE       VERSION        EXTERNA
10.47.122.98   Ready     <none>    1h        v1.10.11+IKS   173.193
```

   In this scenario the URL is `http://173.193.92.112:31367`.
   - Navigate to the output given (for example `http://50.23.5.136:31367`) in your browser. You should see the guestbook now displaying in your browser:



Conclusion

Congratulations, you have now deployed an application by using two different methods to Kubernetes! From this lab, you can see that using Helm required less commands and less to think about (by giving it the chart path and not the individual files) versus using `kubectl`. Helm's application management provides the user with this simplicity.

Move on to the next lab, Lab2, to learn how to update our running app when the chart has been changed.

Lab 2 - Update Application

In Lab 1, we installed the guestbook sample app by using Helm and saw the benefits over using `kubectl`. You probably think that you're done and know enough to use Helm. But what about updates or improvements to the chart? How do you update your running app to pick up these changes?

In this lab, we're going to look at how to update our running app when the chart has been changed. To demonstrate this, we're going to make changes to the original `guestbook` chart by:

- Removing the Redis slaves and using just the in-memory DB
- Changing the type from `LoadBalancer` to `NodePort`.

It seems contrived but the goal of this lab is to show you how to update your apps with Kubernetes and Helm. So, how easy is it to do this? Let's take a look below.

Scenario 1: Update the application using `kubectl`

In this part of the lab we will update the previously deployed application Guestbook, using Kubernetes directly.

1. This is an **optional** step that is not technically required to update your running app. The reason for doing this step is "house keeping" - you want to have the correct files for the current configuration that you have deployed. This avoids making mistakes if you have future updates or even rollbacks. In this updated configuration, we remove the Redis slaves. To have the directory match the configuration, move/archive or simply remove the Redis slave files from the guestbook repo tree:

```
cd guestbook/v1
rm redis-slave-service.yaml
rm redis-slave-deployment.yaml
```

> Note: you can reclaim these files later with a `git checkout -- <filename>` command, if desired

2. Delete the Redis slave service and pods:

```
$ kubectl delete svc redis-slave --namespace default
service "redis-slave" deleted
$ kubectl delete deployment redis-slave --namespace default
deployment.extensions "redis-slave" deleted
```

3. Update the guestbook service from `LoadBalancer` to `NodePort` type:

```
sed -i.bak 's/LoadBalancer/NodePort/g' guestbook-service.yaml
```

> Note: you can reset the files later with a `git checkout -- <filename>` command, if desired

4. Delete the guestbook service:

```
kubectl delete svc guestbook --namespace default
```

5. Re-create the service with `NodePort` type:

```
kubectl create -f guestbook-service.yaml
```

6. Check the updates, using

```
kubectl get all --namespace default
```

```
$ kubectl get all --namespace default
NAME                                  READY     STATUS    RESTARTS   AGE
pod/guestbook-v1-7fc76dc46-9r4s7      1/1       Running   0          1h
pod/guestbook-v1-7fc76dc46-hspnk      1/1       Running   0          1h
pod/guestbook-v1-7fc76dc46-sxzkt      1/1       Running   0          1h
pod/redis-master-5d8b66464f-pvbl9     1/1       Running   0          1h

NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(
service/guestbook      NodePort    172.21.45.29    <none>        3000:
service/kubernetes     ClusterIP   172.21.0.1      <none>        443/T
service/redis-master   ClusterIP   172.21.232.61   <none>        6379/

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/guestbook-demo       3/3     3            3           1h
deployment.apps/redis-master         1/1     1            1           1h

NAME                                          DESIRED   CURRENT   READY
replicaset.apps/guestbook-v1-7fc76dc46        3         3         3
replicaset.apps/redis-master-5d8b66464f       1         1         1
```

Note: The service type has changed (to `NodePort`) and a new port has been allocated (`31989` in this output case) to the guestbook service. All `redis-slave` resources have been removed.

7. View the guestbook
   Get the public IP of one of your nodes:

```
kubectl get nodes -o wide
```

Navigate to the IP address plus the node port that printed earlier.

Scenario 2: Update the application using Helm

In this section, we'll update the previously deployed `guestbook-demo` application by using Helm.

Before we start, let's take a few minutes to see how Helm simplifies the process compared to using Kubernetes directly. Helm's use of a template language provides great flexibility and power to chart authors, which removes the complexity to the chart user. In the guestbook example, we'll use the following capabilities of templating:

- Values: An object that provides access to the values passed into the chart. An example of this is in `guestbook-service`, which contains the line `type: {{ .Values.service.type }}`. This line provides the capability to set the service type during an upgrade or install.
- Control structures: Also called “actions” in template parlance, control structures provide the template author with the ability to control the flow of a template’s generation. An example of this is in `redis-slave-service`, which contains the line `{{- if .Values.redis.slaveEnabled -}}`. This line allows us to enable/disable the REDIS master/slave during an upgrade or install.

The complete `redis-slave-service.yaml` file shown below, demonstrates how the file becomes redundant when the `slaveEnabled` flag is disabled and also how the port value is set. There are more examples of templating functionality in the other chart files.

```
{{- if .Values.redis.slaveEnabled -}}
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
spec:
  ports:
  - port: {{ .Values.redis.port }}
    targetPort: redis-server
  selector:
    app: redis
    role: slave
{{- end }}
```

Enough talking about the theory. Now let's give it a go!

1. First, lets check the app we deployed in Lab 1 with Helm. This can be done by checking the Helm releases:

```
helm list -n helm-demo
```

Note that we specify the namespace. If not specified, it uses the current namespace context. You should see output similar to the following:

```
$ helm list -n helm-demo
NAME           NAMESPACE REVISION  UPDATED
guestbook-demo helm-demo 1         2020-02-24 18:08:02.017401264 +0000
```

The `list` command provides the list of deployed charts (releases) giving information of chart version, namespace, number of updates (revisions) etc.

2. We now know the release is there from step 1., so we can update the application:

```
$ cd helm101/charts

$ helm upgrade guestbook-demo ./guestbook --set redis.slaveEnabled=fa
Release "guestbook-demo" has been upgraded. Happy Helming!
...
```

A Helm upgrade takes an existing release and upgrades it according to the information you provide. You should see output similar to the following:

```
$ helm upgrade guestbook-demo ./guestbook --set redis.slaveEnabled=fa
Release "guestbook-demo" has been upgraded. Happy Helming!
NAME: guestbook-demo
LAST DEPLOYED: Tue Feb 25 14:23:27 2020
NAMESPACE: helm-demo
STATUS: deployed
REVISION: 2
TEST SUITE: None
NOTES:
```

```
1. Get the application URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace helm-demo -o jsonpath="{
  export NODE_IP=$(kubectl get nodes --namespace helm-demo -o jsonpat
  echo http://$NODE_IP:$NODE_PORT
```

The `upgrade` command upgrades the app to a specified version of a chart, removes the `redis-slave` resources, and updates the app `service.type` to `NodePort`. Check the updates, using:

```
kubectl get all --namespace helm-demo
```

```
$ kubectl get all --namespace helm-demo
NAME                                  READY   STATUS    RESTARTS   AG
pod/guestbook-demo-6c9cf8b9-dhqk9     1/1     Running   0          20
pod/guestbook-demo-6c9cf8b9-zddn2     1/1     Running   0          20
pod/redis-master-5d8b66464f-g7sh6     1/1     Running   0          20

NAME                     TYPE        CLUSTER-IP       EXTERNAL-IP   P
service/guestbook-demo   NodePort    172.21.43.244    <none>        3
service/redis-master     ClusterIP   172.21.12.43     <none>        6

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/guestbook-demo      2/2     2            2           20h
deployment.apps/redis-master        1/1     1            1           20h

NAME                                           DESIRED   CURRENT   READY
replicaset.apps/guestbook-demo-6c9cf8b9        2         2         2
replicaset.apps/redis-master-5d8b66464f        1         1         1
```

> Note: The service type has changed (to `NodePort`) and a new port has been allocated (`31202` in this output case) to the guestbook service. All `redis-slave` resources have been removed.

When you check the Helm release with `helm list -n helm-demo`, you will see the revision and date has been updated:

```
$ helm list -n helm-demo
NAME            NAMESPACE REVISION  UPDATED
guestbook-demo  helm-demo 2         2020-02-25 14:23:27.06732381 +000
```

3. View the guestbook
   Get the public IP of one of your nodes:

```
kubectl get nodes -o wide
```

   Navigate to the IP address plus the node port that printed earlier.

Conclusion

Congratulations, you have now updated the applications! Helm does not require any manual changing of resources and is therefore so much easier to upgrade! All configurations can be set on the fly on the command line or by using override files. This is made possible from when the logic was added to the template files, which enables or disables the capability, depending on the flag set.

Check out Lab 3 to get an insight into revision management.

Lab 3 - Revisions

Let's say you deployed different release versions of your application (i.e., you upgraded the running application). How do you keep track of the versions and how can you do a rollback?

Scenario 1: Revision management using Kubernetes

In this part of the lab, we should illustrate revision management of `guestbook` by using Kubernetes directly, but we can't. This is because Kubernetes does not provide any support for revision management. The onus is on you to manage your systems and any updates or changes you make. However, we can use Helm to conduct revision management.

Scenario 2: Revision management using Helm

In this part of the lab, we illustrate revision management on the deployed application `guestbook-demo` by using Helm.

With Helm, every time an install, upgrade, or rollback happens, the revision number is incremented by 1. The first revision number is always 1. Helm persists release metadata in Secrets (default) or ConfigMaps, stored in the Kubernetes cluster. Every time your release changes, it appends that to the existing data. This provides Helm with the capability to rollback to a previous release.

Let's see how this works in practice.

1. Check the number of deployments:

```
helm history guestbook-demo -n helm-demo
```

You should see output similar to the following because we did an upgrade in Lab 2 after the initial install in Lab 1:

```
$ helm history guestbook-demo -n helm-demo
REVISION    UPDATED                     STATUS      CHART           A
1           Mon Feb 24 18:08:02 2020    superseded  guestbook-0.2.0
2           Tue Feb 25 14:23:27 2020    deployed    guestbook-0.2.0
```

2. Roll back to the previous revision:
   In this rollback, Helm checks the changes that occured when upgrading from the revision 1 to revision 2. This information enables it to makes the calls to the Kubernetes API server, to update the deployed application as per the initial deployment - in other words with Redis slaves and using a load balancer.
   Rollback with this command:

```
helm rollback guestbook-demo 1 -n helm-demo
```

```
$ helm rollback guestbook-demo 1 -n helm-demo
Rollback was a success! Happy Helming!
```

Check the history again:

```
helm history guestbook-demo -n helm-demo
```

You should see output similar to the following:

```
$ helm history guestbook-demo -n helm-demo
REVISION    UPDATED                     STATUS      CHART           A
1           Mon Feb 24 18:08:02 2020    superseded  guestbook-0.2.0
2           Tue Feb 25 14:23:27 2020    superseded  guestbook-0.2.0
```

```
5   3           Tue Feb 25 14:53:45 2020    deployed    guestbook-0.2.0
```

Check the rollback, using:

```
kubectl get all --namespace helm-demo
```

```
1   $ kubectl get all --namespace helm-demo
2   NAME                                  READY   STATUS    RESTARTS   AG
3   pod/guestbook-demo-6c9cf8b9-dhqk9     1/1     Running   0          20
4   pod/guestbook-demo-6c9cf8b9-zddn      1/1     Running   0          20
5   pod/redis-master-5d8b66464f-g7sh6     1/1     Running   0          20
6   pod/redis-slave-586b4c847c-tkfj5      1/1     Running   0          5m
7   pod/redis-slave-586b4c847c-xxrdn      1/1     Running   0          5m
8
9   NAME                     TYPE           CLUSTER-IP       EXTERNAL-IP
10  service/guestbook-demo   LoadBalancer   172.21.43.244    <pending>
11  service/redis-master     ClusterIP      172.21.12.43     <none>
12  service/redis-slave      ClusterIP      172.21.232.16    <none>
13
14  NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
15  deployment.apps/guestbook-demo      2/2     2            2           20h
16  deployment.apps/redis-master        1/1     1            1           20h
17  deployment.apps/redis-slave         2/2     2            2           5m1
18
19  NAME                                           DESIRED   CURRENT   READY
20  replicaset.apps/guestbook-demo-26c9cf8b9       2         2         2
21  replicaset.apps/redis-master-5d8b66464f        1         1         1
22  replicaset.apps/redis-slave-586b4c847c         2         2         2
```

You can see from the output that the app service is the service type of `LoadBalancer` again and the Redis master/slave deployment has returned. This shows a complete rollback from the upgrade in Lab 2

Conclusion

From this lab, we can say that Helm does revision management well and Kubernetes does not have the capability built in! You might be wondering why we need `helm rollback` when you could just re-run the `helm upgrade` from a previous version. And that's a good question. Technically, you should end up with the same resources (with same parameters) deployed. However, the advantage of using `helm rollback` is that helm manages (ie. remembers) all of the variations/parameters of the previous `helm install\upgrade` for you. Doing the rollback via a `helm upgrade` requires you (and your entire team) to manually track how the command was previously executed. That's not only tedious but very error prone. It is much easier, safer and reliable to let Helm manage all of that for you and all you need to do it tell it which previous version to go back to, and it does the rest.

Lab 4 awaits.

Lab 4 - Helm Repositories

A key aspect of providing an application means sharing with others. Sharing can be direct counsumption (by users or in CI/CD pipelines) or as a dependency for other charts. If people can't find your app then they can't use it.

A means of sharing is a chart repository, which is a location where packaged charts can be stored and shared. As the chart repository only applies to Helm, we will just look at the usage and storage of Helm charts.

Using charts from a public repository

Helm charts can be available on a remote repository or in a local environment/repository. The remote repositories can be public like Bitnami Charts or IBM Helm Charts, or hosted repositories like on Google Cloud Storage or GitHub. Refer to Helm Chart Repository Guide for more details. You can learn more about the structure of a chart repository by examining the chart index file in this lab.

In this part of the lab, we show you how to install the `guestbook` chart from the Helm101 repo.

1. Check the repositories configured on your system:

   ```
   helm repo list
   ```

   The output should be similar to the following:

   ```
   $ helm repo list
   Error: no repositories to show
   ```

> Note: Chart repositories are not installed by default with Helm v3. It is expected that you add the repositories for the charts you want to use. The Helm Hub provides a centralized search for publicly available distributed charts. Using the hub you can identify the chart with its hosted repository and then add it to your local respoistory list. The Helm chart repository like Helm v2 is in "maintenance mode" and will be deprecated by November 13, 2020. See the project status for more details.

2. Add `helm101` repo:

```
helm repo add helm101 https://ibm.github.io/helm101/
```

Should generate an output as follows:

```
$ helm repo add helm101 https://ibm.github.io/helm101/
"helm101" has been added to your repositories
```

You can also search your repositories for charts by running the following command:

```
helm search repo helm101
```

```
$ helm search repo helm101
NAME              CHART VERSION  APP VERSION DESCRIPTION
helm101/guestbook 0.2.1                      A Helm chart to deploy Gu
```

3. Install the chart
   As mentioned we are going to install the `guestbook` chart from the Helm101 repo. As the repo is added to our local respoitory list we can reference the chart using the `repo name/chart name`, in other words `helm101/guestbook`. To see this in action, you will install the application to a new namespace called `repo-demo`.
   If the `repo-demo` namespace does not exist, create it using:

```
kubectl create namespace repo-demo
```

Now install the chart using this command:

```
helm install guestbook-demo helm101/guestbook --namespace repo-demo
```

The output should be similar to the following:

```
$helm install guestbook-demo helm101/guestbook --namespace repo-demo
NAME: guestbook-demo
LAST DEPLOYED: Tue Feb 25 15:40:17 2020
NAMESPACE: repo-demo
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
1. Get the application URL by running these commands:
  NOTE: It may take a few minutes for the LoadBalancer IP to be availa
        You can watch the status of by running 'kubectl get svc -w gue
  export SERVICE_IP=$(kubectl get svc --namespace repo-demo guestbook-
  echo http://$SERVICE_IP:3000
```

Check that release deployed as expected as follows:

```
$ helm list -n repo-demo
NAME           NAMESPACE   REVISION UPDATED
guestbook-demo repo-demo   1        2020-02-25 15:40:17.627745329 +000
```

Conclusion

This lab provided you with a brief introduction to the Helm repositories to show how charts can be installed. The ability to share your chart means ease of use to both you and your consumers.

Lab: Kubernetes Extensions

- Access the client terminal
- Login to ibmcloud
- Connect to your cluster
- Create a Custom Resource
- Operators
  - Ready Made Operators
- Create a Custom Resource and Operator using the Operator SDK
  - Install sdk-operator
  - Create the Operator
  - Cleanup
- Application CRD

Access the web-terminal

Go to the URL for your web-terminal, given to you by the instructor team.

Login

```
export CLUSTER_NAME=<your cluster name>
ibmcloud login
oc config current-context
```

```
ibmcloud login
API endpoint: https://cloud.ibm.com

Email> a.newell1@remkoh.dev

Password>
Authenticating...
OK
```

```
Select an account:
1. A Newell's Account (ed69e4e9e2d74660b6cc45d0e47cd8b7)
2. Advowork (e2b54d0c3bbe4180b1ee63a0e2a7aba4) <-> 1840867
Enter a number> 2
Targeted account Advowork (e2b54d0c3bbe4180b1ee63a0e2a7aba4) <-> 1840867


Select a region (or press enter to skip):
1. au-syd
2. in-che
3. jp-osa
4. jp-tok
5. kr-seo
6. eu-de
7. eu-gb
8. us-south
9. us-south-test
10. us-east
Enter a number> 10
Targeted region us-east
```

Connect to your cluster by downloading the cluster config,

```
ibmcloud ks cluster config --cluster $CLUSTER_NAME
```

Check you can access the cluster,

```
oc config current-context
oc get nodes
```

Create a Custom Resource (CR)

https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/

Custom Resource Definitions (CRD) were added in Kubernetes v1.7 in June 2017. A CRD defines Custom Resources (CR). A CR is an extension of the Kubernetes API that allows you to store your own API Objects and lets the API Server handle the lifecycle of a CR. On their own, CRs simply let you store and retrieve structured data.

For instance, our Guestbook application consists of an object `Guestbook` with attributes `GuestbookTitle` and `GuestbookSubtitle`, and a Guestbook handles objectes of type `GuestbookMessage` with attributes `Message`, `Sender`.

You have to ask yourself if it makes sense if your objects are added as a Custom Resource to Kubernetes or not. If your API is a Declarative API you can consider adding a CR.

- Your API has a small number of small objects (resources).
- The objects define configuration of applications or infrastructure.
- The objects are updated relatively infrequently.
- Users often need to read and write the objects.
- main operations on the objects are CRUD (create, read, update and delete).
- Transactions between objects are not required.

It doesn't immediately make sense to store messages by Guestbook users in Kubernetes, but it might make sense to store meta-data about a Guestbook deployment, for instance the title and subtitle of a Guestbook deployment, assigned resources or replicas.

Another benefit of adding a Custom Resource is to view your types in the Kubernetes Dashboard.

If you want to deploy a Guestbook instance as a Kubernetes API object and let the Kubernetes API Server handle the lifecycle events of the Guestbook deployment, you can create a Custom Resource Definition (CRD) for the Guestbook object as follows. That way you can deploy multiple Guestbooks with different titles and let each be managed by Kubernetes.

```
cat <<EOF >>guestbook-crd.yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: guestbooks.apps.ibm.com
spec:
```

```
  group: apps.ibm.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                guestbookTitle:
                  type: string
                guestbookSubtitle:
                  type: string
  scope: Namespaced
  names:
    plural: guestbooks
    singular: guestbook
    kind: Guestbook
    shortNames:
    - gb
EOF
```

- You can see that the `apiVersion` is part of the `apiextensions.k8s.io/v1` API Group in Kubernetes, which is the API that enables extensions, and the `kind` is set to `CustomResourceDefinition`.
- The `served` flag can disable and enable a version.
- Only 1 version can be flagged as the storage version.
- The `spec.names.kind` is used by your resource manifests and should be CamelCased.

Create the Custom Resource for the Guestbook witht he command,

```
oc create -f guestbook-crd.yaml
```

When run in the terminal,

```
$ kubectl create -f guestbook-crd.yaml
customresourcedefinition.apiextensions.k8s.io/guestbooks.apps.ibm.com crea
```

You have now added a CR to the Kubernetes API, but you have not yet created a deployment of type Guestbook yet.

Create a resource specification of type Guestbook named `my-guestbook`,

```
cat <<EOF >>my-guestbook.yaml
apiVersion: "apps.ibm.com/v1"
kind: Guestbook
metadata:
  name: my-guestbook
spec:
  guestbookTitle: "The Chemical Wedding of Remko"
  guestbookSubtitle: "First Day of Many"
EOF
```

And to create the `my-guestbook` resource, run the command

```
oc create -f my-guestbook.yaml
```

When run in the terminal,

```
$ oc create -f my-guestbook.yaml
guestbook.apps.ibm.com/my-guestbook created
```

If you list all Kubernetes resources, only the default Kubernetes service is listed. To list your Custom Resources, add the extended type to your command.

```
$ oc get all
```

```
NAME    TYPE    CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
service/kubernetes    ClusterIP    172.21.0.1    <none>    443/TCP    5d14

$ oc get guestbook
NAME    AGE
my-guestbook    8m32s
```

To read the details for the `my-guestbook` of type `Guestbook`, describe the instance,

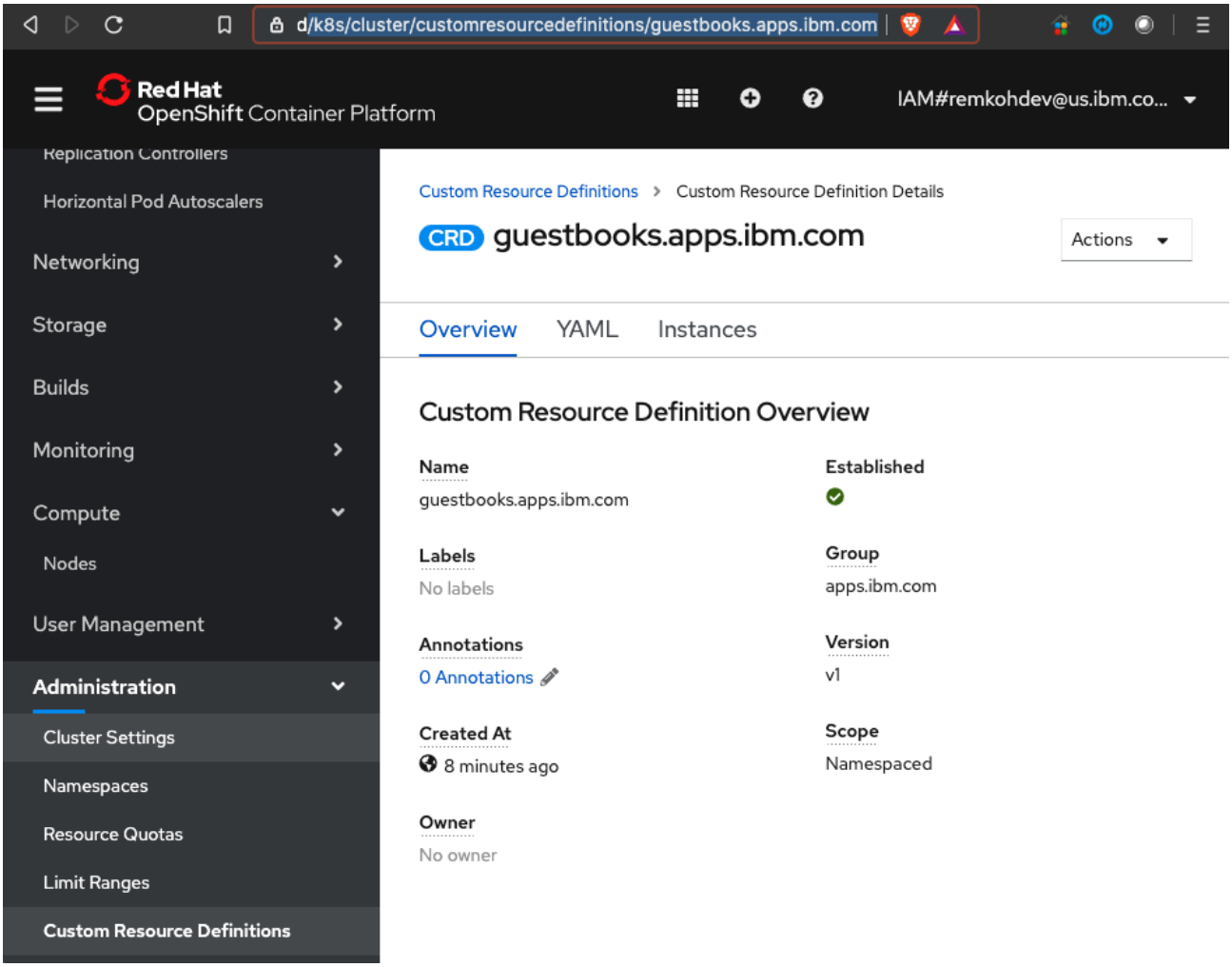
```
$ oc describe guestbook my-guestbook

Name:         my-guestbook
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  apps.ibm.com/v1
Kind:         Guestbook
Metadata:
  Creation Timestamp:  2020-06-30T20:31:36Z
  Generation:          1
  Resource Version:    1081471
  Self Link:           /apis/apps.ibm.com/v1/namespaces/default/guestbooks
  UID:                 dcbdcafc-999d-4051-9244-0315093357e7
Spec:
  Guestbook Subtitle:  First Day of Many
  Guestbook Title:     The Chemical Wedding of Remko
Events:                <none>
```

Or retrieve the resource information by specifying the type,

```
$ oc get Guestbook -o yaml
apiVersion: v1
items:
- apiVersion: apps.ibm.com/v1
  kind: Guestbook
  metadata:
    creationTimestamp: "2020-07-02T04:41:57Z"
    generation: 1
    name: my-guestbook
    namespace: default
    resourceVersion: "1903244"
    selfLink: /apis/apps.ibm.com/v1/namespaces/default/guestbooks/my-guest
```

```
    uid: 3f774899-3070-4e00-b74c-a6a14654faeb
  spec:
    guestbookSubtitle: First Day of Many
    guestbookTitle: The Chemical Wedding of Remko
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

In the Kubernetes Dashboard web console, you can browse to Custom Resource Definitions and find the Guestbook CRD.



Administration > Custom Resource Definitions

You have now created a new type or Custom Resource (CR) and created an instance of your new type. But just having a new type and a new instance of the type, does not add as much control over the instances yet, we can basically only create and delete a static type with

some descriptive meta-data. With a custom controller or `Operator` you can over-write the methods that are triggered at certain lifecycle events.

Cleanup

```
oc delete guestbook my-guestbook
oc delete customresourcedefinition guestbooks.apps.ibm.com
```

Operators

https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

Operators are clients of the Kubernetes API that act as controllers for a Custom Resource.

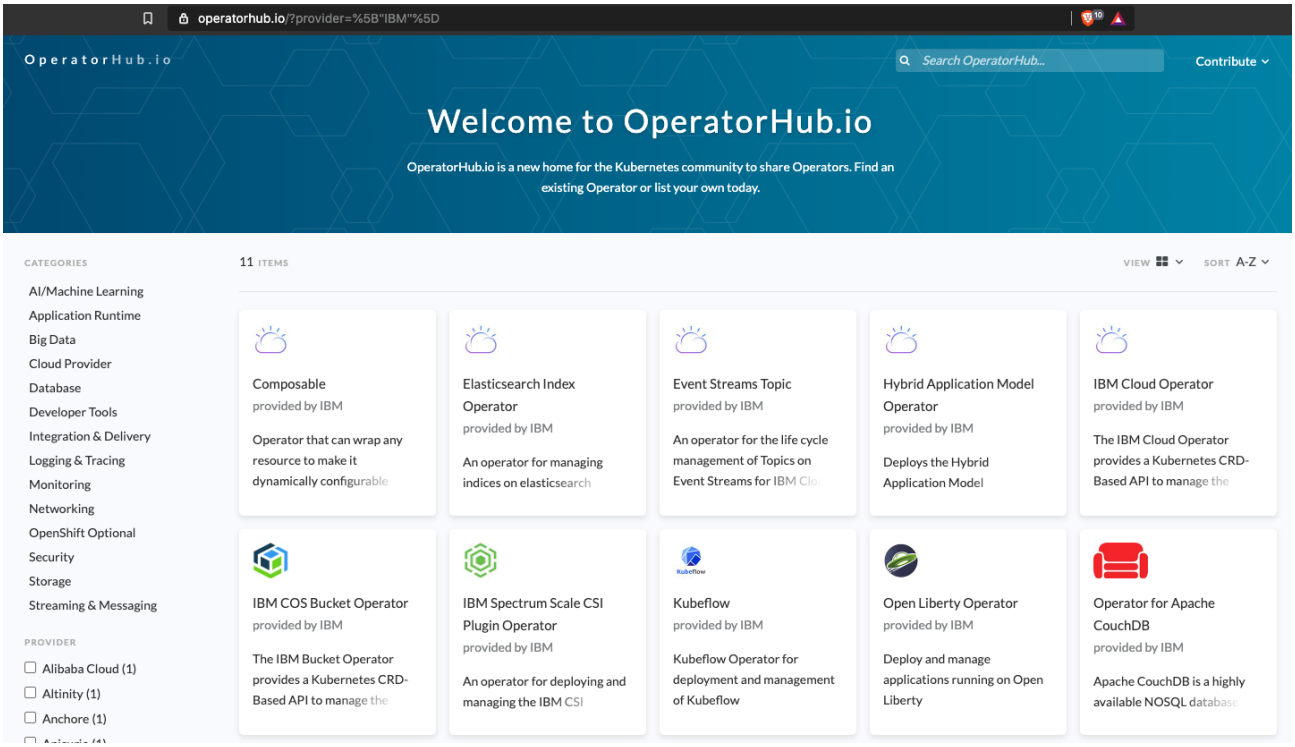
To write applications that use the Kubernetes REST API, you can use one of the following supported client libraries:

- Go,
- Python,
- Java,
- CSharp dotnet,
- JavaScript,
- Haskell.

In addition, there are many community-maintained client libraries.

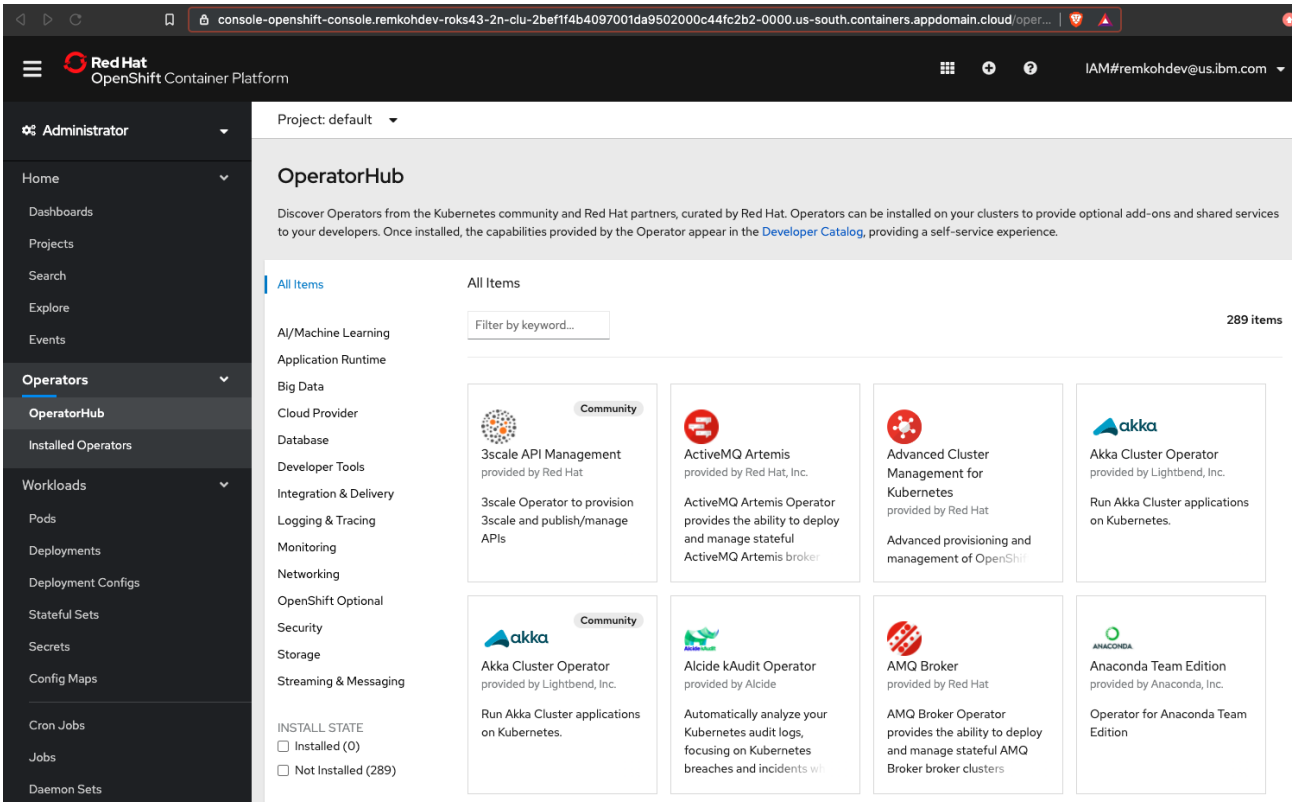
Ready made operators

At the Kubernetes OperatorHub.io, you find ready to use operators written by the community.



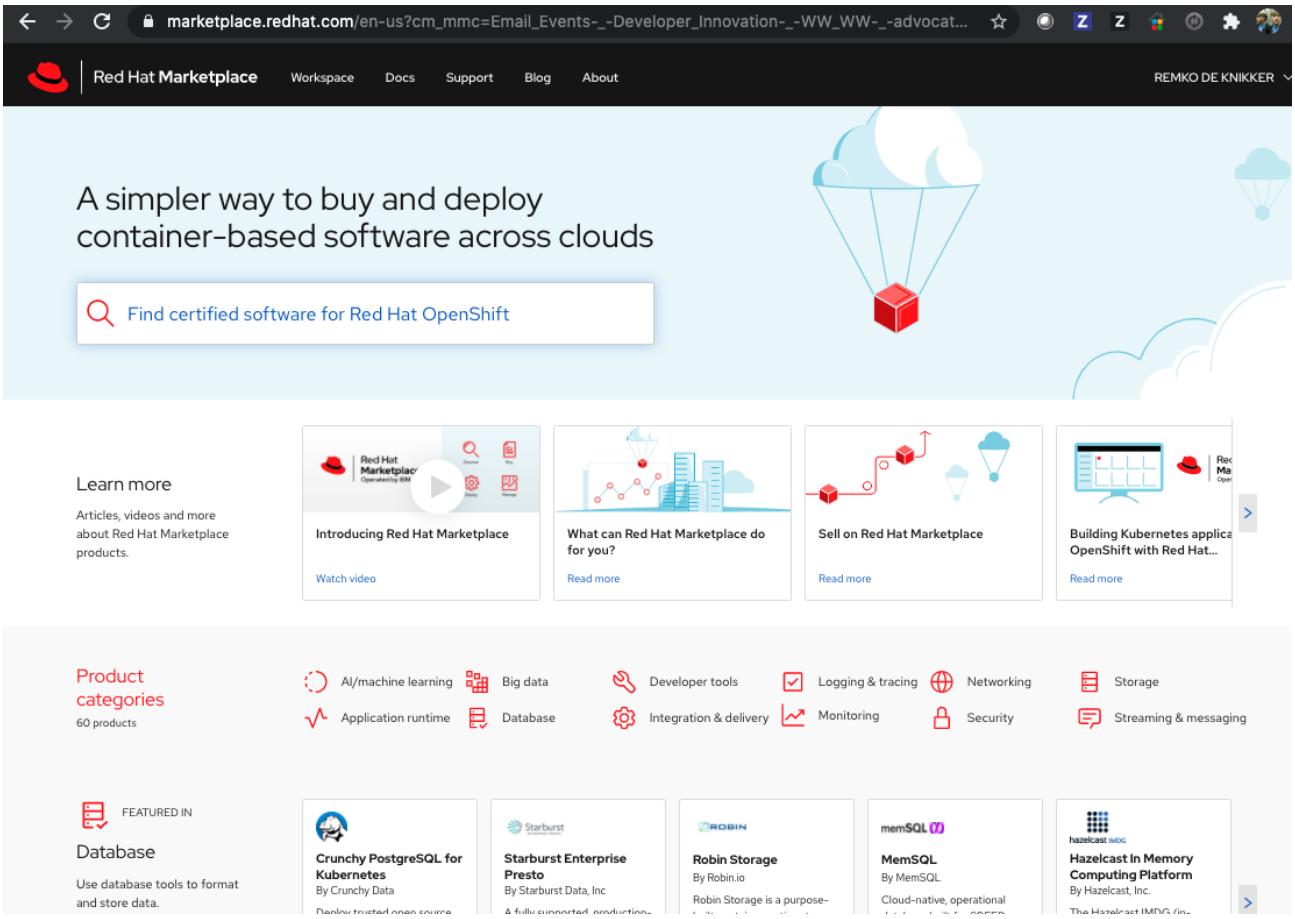
OperatorHub.io

For a Red Hat curated list of Kubernetes Operators you can go to the OPenShift console > Operators > OperatorHub.



Red Hat OperatorHub

For a list of Red Hat certified partner software go to the Red Hat Marketplace at http://ibm.biz/clientrhm



Red Hat Marketplace

Create a Custom Resource and Operator using the Operator SDK

To write your own operator you can use existing tools:

- KUDO (Kubernetes Universal Declarative Operator),
- kubebuilder,
- Metacontroller using custom WebHooks,
- the Operator Framework.

The Operator SDK provides the following workflow to develop a new Operator:

The following workflow is for a new Go operator:

1. Create a new operator project using the SDK Command Line Interface(CLI)
2. Define new resource APIs by adding Custom Resource Definitions(CRD)
3. Define Controllers to watch and reconcile resources
4. Write the reconciling logic for your Controller using the SDK and controller-runtime APIs
5. Use the SDK CLI to build and generate the operator deployment manifests

Install sdk-operator

The following section uses the `sdk-operator` cli, which depends on `Go` to be installed. Check if both tools are installed in your web-terminal,

```
go version
operator-sdk version
```

If you see a `command not found` error, install both now.

For detailed installation instructions go here.

To install the Operator SDK in Ubuntu, you need to install the Go tools and the Operator SDK.

```
curl -LO https://golang.org/dl/go1.14.4.linux-amd64.tar.gz
tar -C /usr/local -xzf go1.14.4.linux-amd64.tar.gz
export PATH=$PATH:/usr/local/go/bin
go version
```

Install the Operator SDK,

```
curl -LO https://github.com/operator-framework/operator-sdk/releases/downl
chmod +x operator-sdk-v0.18.2-x86_64-linux-gnu
sudo mkdir -p /usr/local/bin/
sudo cp operator-sdk-v0.18.2-x86_64-linux-gnu /usr/local/bin/operator-sdk
```

```
rm operator-sdk-v0.18.2-x86_64-linux-gnu
operator-sdk version
```

Check again,

```
go version
operator-sdk version
```

Create the Operator

1. Create a New Project

Create a new Operator project,

```
export DOCKER_USERNAME=<your-docker-username>
```

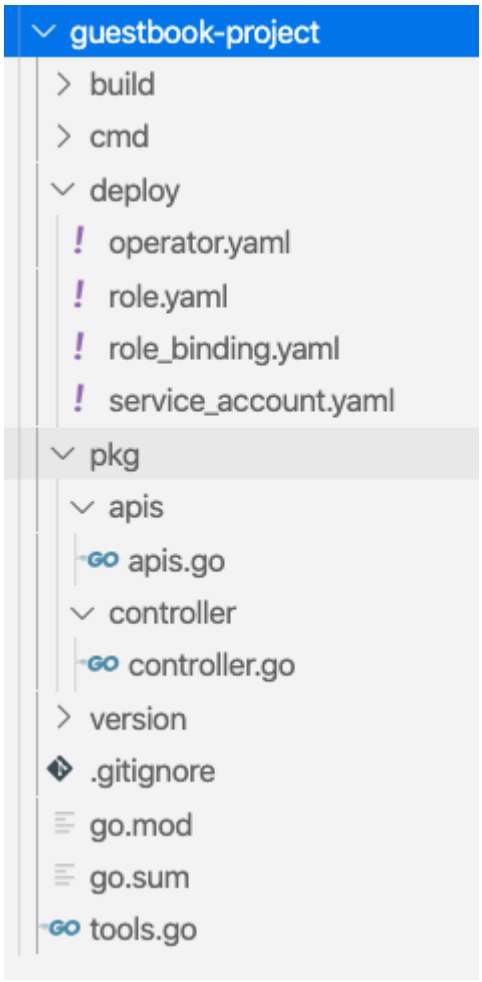
and

```
export OPERATOR_NAME=guestbook-operator
export OPERATOR_PROJECT=guestbook-project
export OPERATOR_GROUP=guestbook.ibm.com
export OPERATOR_VERSION=v1
export CRD_KIND=Guestbook
```

Create a new Operator project,

```
operator-sdk new $OPERATOR_PROJECT --type go --repo github.com/$DOCKER_USE

cd $OPERATOR_PROJECT
```

The scaffolding of a new project will create an operator, an api and a controller.



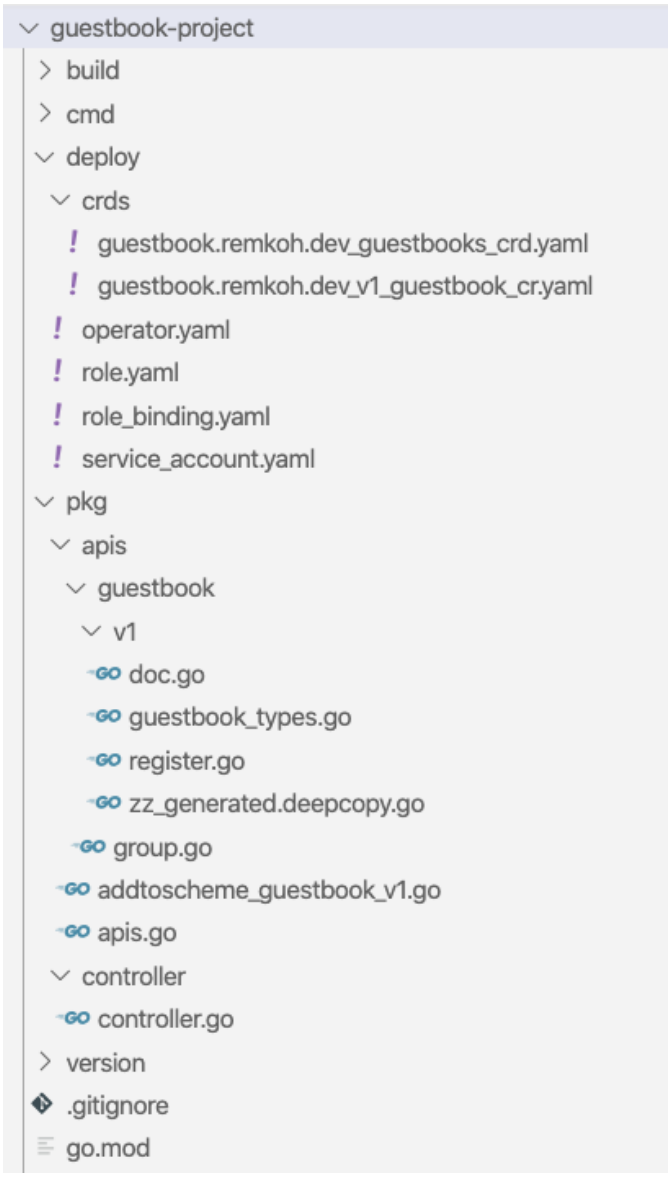
new project directory structure

2. Create a new API

Add a new API definition for a new Custom Resource under `pkg/apis` and generate the Custom Resource Definition (CRD) and Custom Resource (CR) files under `deploy/crds`.

```
operator-sdk add api --api-version=$OPERATOR_GROUP/$OPERATOR_VERSION --kind=
```

The command will create a new API, a Custom Resource (CR), a Custom Resource Definition (CRD).



new project directory structure

One file is created in `pkg/apis` called `addtoscheme_guestbook_v1.go` that registers the new schema. One new file is created in `pkg/apis/guestbook` called `group.go` that defines the package. Four new files are created in `pkg/apis/guestbook/v1`:

- doc.go,
- guestbook_types.go,
- register.go,
- zz_generated.deepcopy.go.

The `guestbook_types.go` file,

```go
package v1

import (
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// GuestbookSpec defines the desired state of Guestbook
type GuestbookSpec struct {
}

// GuestbookStatus defines the observed state of Guestbook
type GuestbookStatus struct {
}

type Guestbook struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   GuestbookSpec   `json:"spec,omitempty"`
    Status GuestbookStatus `json:"status,omitempty"`
}

// GuestbookList contains a list of Guestbook
type GuestbookList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata,omitempty"`
    Items           []Guestbook `json:"items"`
}

func init() {
    SchemeBuilder.Register(&Guestbook{}, &GuestbookList{})
}
```

The Custom Resource (CR) in file `deploy/crds/guestbook.remkoh.dev_v1_guestbook_cr`,

```yaml
apiVersion: guestbook.remkoh.dev/v1
kind: Guestbook
metadata:
  name: example-guestbook
spec:
  # Add fields here
  size: 3
```

The Custom Resource Definition (CRD) in file `deploy/crds/guestbook.remkoh.dev_guestbooks_crd.yaml`,

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: guestbooks.guestbook.remkoh.dev
spec:
  group: guestbook.remkoh.dev
  names:
    kind: Guestbook
    listKind: GuestbookList
    plural: guestbooks
    singular: guestbook
  scope: Namespaced
  versions:
  - name: v1
    schema:
      openAPIV3Schema:
        description: Guestbook is the Schema for the guestbooks API
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned schema of this
              of an object. Servers should convert recognized schemas to t
              internal value, and may reject unrecognized values. More inf
            type: string
          kind:
            description: 'Kind is a string value representing the REST res
              object represents. Servers may infer this from the endpoint
              submits requests to. Cannot be updated. In CamelCase. More i
            type: string
          metadata:
            type: object
          spec:
            description: GuestbookSpec defines the desired state of Guestb
            type: object
          status:
            description: GuestbookStatus defines the observed state of Gue
            type: object
        type: object
    served: true
    storage: true
    subresources:
      status: {}
```

3. Create a new Controller

Add a new controller under `pkg/controller/<kind>` .

```
operator-sdk add controller --api-version=$OPERATOR_GROUP/$OPERATOR_VERSION
```

This command creates two files in `pkg/controller` :

- `add_guestbook.go` , which registers the new controller, and
- `guestbook/guestbook_controller.go` , which is the actual custom controller logic.

The file `guestbook/guestbook_controller.go` defines the `Reconcile` function,

```
// Reconcile reads state of the cluster for a Guestbook object and makes c
// TODO(user): User must modify this Reconcile function to implement their
func (r *ReconcileGuestbook) Reconcile(request reconcile.Request) (reconci
    ...
    // Fetch the Guestbook instance
    instance := &guestbookv1.Guestbook{}
    ...
    // Define a new Pod object
    pod := newPodForCR(instance)
    ...
}
```

4. Compile and Build the Code

The operator-sdk build command compiles the code and builds the executables. fter you built the image, push it to your image registry, e.g. Docker hub.

```
operator-sdk build docker.io/$DOCKER_USERNAME/$OPERATOR_NAME
```

Login in to the container registry,

```
docker login docker.io -u $DOCKER_USERNAME
```

And push the build image to your image repository,

```
docker push docker.io/$DOCKER_USERNAME/$OPERATOR_NAME
```

5. Deploy the Operator

First replace the image attribute in the operator resource with the built image,

Inspect the deploy/operator.yaml file,

```
cat deploy/operator.yaml
```

And replace the `REPLACE_IMAGE` value,

```
$ sed -i "s|REPLACE_IMAGE|docker.io/$DOCKER_USERNAME/$OPERATOR_NAME|g" deplo
```

Make sure you are connected to the OpenShift cluster (see above how to connect), and deploy the operator with the following template code.

```
oc create sa $OPERATOR_PROJECT
oc create -f deploy/role.yaml
oc create -f deploy/role_binding.yaml
oc create -f deploy/crds/${OPERATOR_GROUP}_${CRD_KIND,,}s_crd.yaml
oc create -f deploy/operator.yaml
oc create -f deploy/crds/${OPERATOR_GROUP}_${OPERATOR_VERSION}_${CRD_KIND,
```

Verify the deployment,

```
oc get deployment $OPERATOR_PROJECT
oc get pod -l app=example-${CRD_KIND,,}
oc describe ${CRD_KIND,,}s.${OPERATOR_GROUP} example-${CRD_KIND,,}
```

For our example Guestbook project the above templates should resolve as follows,

```
$ oc create sa guestbook-project
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/crds/guestbook.remkoh.dev_guestbooks_crd.yaml
$ oc create -f deploy/operator.yaml
$ oc create -f deploy/crds/guestbook.remkoh.dev_v1_guestbook_cr.yaml
$ oc get deployment guestbook-project
$ oc get pod -l app=example-guestbook
$ oc describe guestbooks.guestbook.remkoh.dev example-guestbook
```

Cleanup

```
oc delete sa $OPERATOR_PROJECT
oc delete role $OPERATOR_PROJECT
oc delete rolebinding $OPERATOR_PROJECT
oc delete customresourcedefinition ${CRD_KIND,,}s.${OPERATOR_GROUP}
oc delete deployment $OPERATOR_PROJECT
```

Application CRD

The Application CRD (Custom Resource Definition) and Controller provide the following:

- Describe an applications metadata,
- A point to connect the infrastructure, such as Deployments, to as a root object.
-
- Application level health checks.

This could be used by:

- Application operators,
- Tools, such as Helm, and
- Dashboards.

```yaml
apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: "guestbook"
  labels:
    app.kubernetes.io/name: "guestbook"
spec:
  selector:
    matchLabels:
     app.kubernetes.io/name: "guestbook"
  componentKinds:
    - group: v1
      kind: Deployment
    - group: v1
      kind: Service
  descriptor:
    type: "guestbook"
    keywords:
      - "gb"
      - "guestbook"
    links:
      - description: Github
        url: "https://github.com/IBM/guestbook"
    version: "0.1.0"
    description: "The Guestbook application is an example app to demonstra
    maintainers:
      - name: IBM Developer
        email: developer@ibm.com
    owners:
      - name: IBM Developer
        email: developer@ibm.com
```

Day 2

Lab - S2I Open Liberty

Source To Image Builder for Open Liberty Applications on OpenShift

This project contains a Source-to-Image (S2I) builder image and a S2I runtime image which creates an image running Java web applications on Open Liberty.

Source-to-Image (S2I) is an open source toolkit for building reproducible container images from source code. S2I produces ready-to-run images by injecting source code into a container image.

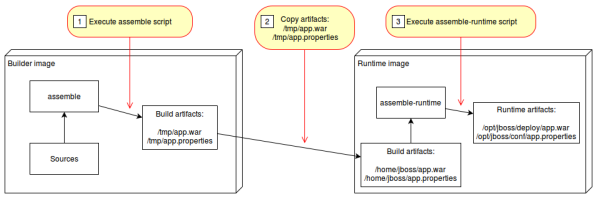
The Open Liberty builder can be used in two different environments:

- Local Docker runtime via 's2i',
- Deployment to OpenShift'.

With interpreted languages like python and javascript, the runtime container is also the build container. For example, with a node.js application the 'npm install' is run to build the application and then 'npm start' is run in the same container in order to start the application.

However, with compiled languages like Java, the build and runtime processes can be separated. This will allow for slimmer runtime containers for faster application starts and less bloat in the application image.

This lab will focus on the second scenario of using a builder image along with a runtime image.



(source: https://github.com/openshift/source-to-image/blob/master/docs/runtime_image.md)

Prerequisites

The following prerequisites are needed:

- A Docker Hub account
- GitHub Account
- IBM Cloud Account
- Have followed these steps to get a cluster
    - You can stop after step 11

Setup

For this lab we will need to use a docker-in-docker environment so that we can build our images. For this scenario we will be using the IBM Skills Network.

1. Follow the instructions here to create your environment.
2. Clone this repository locally and navigate to the newly cloned directory.

```
git clone https://github.com/IBM/s2i-open-liberty-workshop.git -b con
cd s2i-open-liberty-workshop
```

3. Then we need to install Source to Image. Run the following command to start the installation script.

```
chmod +x setup.sh
./setup.sh
```

Add the binary to the `PATH` variable,

```
    echo 'export PATH=/home/theia/s2i:$PATH' > .bash_profile
    source .bash_profile
    s2i version
```

1. To make things easier, we are going to set some environment variables that we can reuse in later commands.
   Note: Replace *Your Username* with your actual docker hub username. If you do not have one, go here to create one.

```
export ROOT_FOLDER=$(pwd)
export DOCKER_USERNAME=<your-docker-username>
```

Build the builder image

In this section we will create the first of our two S2I images. This image will be responsible for taking in our source code and building the application binary with Maven.

1. Navigate to the builder image directory

```
cd ${ROOT_FOLDER}/builder-image
```

2. Review the ./Dockerfile

```
cat Dockerfile
```

   - The image uses a Redhat certified Universal Base Image (UBI) from the public container registry at Redhat,

```
FROM registry.access.redhat.com/ubi8/ubi:8.1
```

3. You can customize the builder image further, e.g. change the `LABEL` for `maintainer` to your name,

```
LABEL maintainer="<your-name>"
```

4. Now build the builder image.

```
docker build -t $DOCKER_USERNAME/s2i-open-liberty-builder:0.1.0 .
```

5. Log in to your Dockerhub account. After running the below command, you will be asked to enter your docker username and password.

```
docker login
```

6. Push the builder image out to Docker hub.

```
docker push $DOCKER_USERNAME/s2i-open-liberty-builder:0.1.0
```

With that done, you can now build your runtime image.

Build the runtime image

In this section you will create the second of our two S2I images. The runtime image will be responsible for taking the compiled binary from the builder image and serving it with the Open Liberty application server.

1. Navigate to the runtime image directory

```
cd $ROOT_FOLDER/runtime-image
```

2. Review the ./Dockerfile

```
cat Dockerfile
```

3. Build the runtime image

```
docker build -t $DOCKER_USERNAME/s2i-open-liberty:0.1.0 .
```

4. Push the runtime image to Docker hub.

```
docker push $DOCKER_USERNAME/s2i-open-liberty:0.1.0
```

Now we are ready to build our application with S2I.

Use S2I to build the application container

In this section, we will use S2I to build our application container image and then we will run the image locally using Docker.

1. Use the builder image and runtime image to build the application image

```
cd $ROOT_FOLDER/web-app
```

2. Run a multistage S2I build, to build the application.

```
~/s2i/s2i build . $DOCKER_USERNAME/s2i-open-liberty-builder:0.1.0 autho
```

Let's break down the above command:

- s2i build . - Use `s2i build` in the current directory to build the Docker image by combining the builder image and sources
- $DOCKER_USERNAME/s2i-open-liberty-builder:0.1.0 - This is the builder image used to build the application
- authors - name of our application image
- –runtime-image $DOCKER_USERNAME/s2i-open-liberty:0.1.0 - Take the output of the builder image and run it in this container.
- -a /tmp/src/target -a /tmp/src/server.xml - The `runtime-artifact` flag specifies a file or directory to be copies from builder to runtime image. The runtime-artifact is where the builder output is located. These files will be passed into the runtime image.

3. Run the newly built image to start the application on your local machine in the background,

```
docker run -d --rm -p 9080:9080 authors
```

4. Retrieve the authors using curl,

```
curl -X GET "http://localhost:9080/api/v1/getauthor" -H "accept: applic
```

5. Or open up your browser and navigate to http://localhost:9080/openapi/ui to view your deployed microservice.

Deployment to OpenShift

In the following steps we will be using two deployment strategies:

- deploy as a traditional Kubernetes `Deployment`, and
- build and deployment using templates, OpenShift `BuildConfig`, and `DeploymentConfig`.

Now that we have the application running locally and have verified that it works, let's deploy it to an OpenShift environment.

1. Log in with your OpenShift Cluster.
    1. Open your `OpenShift web console` and from the profile dropdown `Copy Login Command`.
    2. Paste the login command to login, e.g.

```
oc login --token=<login-token> --server=https://<cluster-subdomain>:
```

Deploying as a traditional Kubernetes deployment

For this method, we will deploy our application by creating a kubernetes deployment along with a service and a route.

1. Tag the image that was created in the previous section.

```
export IMAGE=docker.io/$DOCKER_USERNAME/authors:latest
echo $IMAGE
docker tag authors $IMAGE
```

2. Push the image that we built locally using s2i to the OpenShift image registry.

```
docker push $IMAGE
```

3. Go back to the root folder,

```
cd $ROOT_FOLDER
```

4. Review the `application.yaml` file,

```
sed "s|APPLICATION_IMAGE|$IMAGE|" application.yaml
```

This command will add your newly pushed authors image to the deployment yaml file.

```
sed -i "s|APPLICATION_IMAGE|$IMAGE|" application.yaml
```

5. Apply the `application.yaml` file using the `oc` cli to create our Deployment, Service, and Route.

```
oc apply -f application.yaml
```

6. Now let's visit the deployed application. Run the following to get the route to access the application.

```
oc get routes -l app=authors -o go-template='{{range .items}}{{.spec.ho
```

7. Copy and paste the output of the previous command to set a variable $APP_URL,

```
APP_URL=<get-routes-output>
```

8. Test the application using curl

```
curl -X GET "http://$APP_URL/api/v1/getauthor" -H "accept: application/
```

9. Or use the route to your app and paste it into your web browser and add the following to the end of the route:

```
/openapi/ui
```

So your route should appear like the following but without the (...):

```
authors-route-default...appdomain.cloud/openapi/ui
```

It sometimes takes a minute to fully deploy so if you get a message about the application not being available, try again.

Using Templates, BuildConfigs, and DeploymentConfigs

For this section, you will explore how to build and deploy our application using OpenShift concepts known as templates, build configs and deployment configs. With build configs, the s2i builds are actually happening on the cluster rather than locally.

1. Create a push secret to push the application image to your DockerHub account. **You must** modify the following command by replacing **** with your DockerHub password and **** with your email address for DockerHub.

```
kubectl create secret docker-registry regcred --docker-server=https://i
```

With the secret created we can now give the *builder* service account access to the secret so that it can push the newly created application image to your account.

```
kubectl patch sa builder --type='json' -p='[{"op":"add","path":"/secret
```

2. Then, we need to create a builder template that contains our builder image build config. See Using Templates.
3. Review the `buildTemplate.yaml` file, which contains the BuildConfig object for the builder image,

```
cat buildTemplate.yaml
```

4. Next, let's create the build config and start the first build:

```
oc process -f buildTemplate.yaml -p DOCKER_USERNAME=$DOCKER_USERNAME |
```

This command will not only create our build config but also kick off the first build to create our builder image and push it to DockerHub.
To view the status of the build you can run:

```
oc describe build open-liberty-builder
```

Or view the Build section of the OpenShift console.
5. Now let's take a look at our runtime image build config:

```
cat runtimeTemplate.yaml
```

Then, create the build config for our runtime image and start the first build:

```
oc process -f runtimeTemplate.yaml -p DOCKER_USERNAME=$DOCKER_USERNAME
```

To take a look at the build status, run:

```
oc describe build open-liberty-app
```

Or view the Build section of the OpenShift console.

Now with our builds run, we can deploy our application. Previously we used a kubernetes deployment object to do this however this time we will use an OpenShift deployment configuration.

Both objects are similar and will accomplish the same goals however with deployment configs you have greater control of your application's deployment behavior. You also have the option to set automated triggers to kick off builds and deploys based on image, configuration, or git repository changes. Due to a limitation in our workshop environment, we will not be exploring triggers and utilizing image streams with the integrated OpenShift registry, however, if you would like to take a look at a template that automates the entire build and deploy process with triggers, check out the this example template found in the workshop repo.

1. Let's check out the deployment config template:

```
cat deploymentConfig.yaml
```

2. Then, create the deploymentConfig

```
oc apply -f deploymentConfig.yaml
```

```
oc new-app --template authors-app -p DOCKER_USERNAME=$DOCKER_USERNAME
```

3. To verify that your app is deployed, run:

```
```

```
oc get pods
```

You should see a pod named `authors2-1-` followed by 5 random characters, in the example below they are `x58fk`. This is the application pod.

```
oc get pods
NAME                                READY   STATUS      RESTARTS   A
authors-deployment-69ff497df6-vz75c 1/1     Running     0          1
authors2-1-deploy                   0/1     Completed   0          (
authors2-1-x58fk                    1/1     Running     0          (
open-liberty-app-1-build            0/1     Completed   0          1
open-liberty-builder-1-build        0/1     Completed   0          1
```

You can also view these on the OpenShift dashboard if you navigate to `Workloads` > `Pods` and look for your new pod. It should start with `authors2-`.

4. Once you have verified that the new pod is running, enter the following command to view the application routes.

```
oc get routes
```

```
oc get routes

NAME    HOST/PORT    PATH    SERVICES    PORT    TERMINATION    WILDC
authors-route    authors-route-default.your-roks-43-1n-cl-2bef1f4b409
authors2    authors2-default.your-roks-43-1n-cl-2bef1f4b4097001da9502
```

5. Copy the route named `authors2` and add it to the end of the command below.

```
export API_URL=
```

View the sample below, however, you will have a different route.

```
export API_URL=authors2-default.osgdcw01-0e3e0ef4c9c6d831e8aa6fe01f33bf
```

Then test your application with the command below:

```
curl -X GET "http://$API_URL/api/v1/getauthor" -H "accept: application/
```

You should then get a response back from the application that we just deployed:

```
curl -X GET "http://$API_URL/api/v1/getauthor" -H "accept: applicatio
{"name":"Oliver Rodriguez","twitter":"https://twitter.com/heres__olli
```

Optional: Customizing your application

In this optional section we will create our own copy of the code push the changes to OpenShift.

1. First we need to create your own version of the code repo by creating a fork. This will copy the repo into your GitHub account. Navigate to the lab repo at https://github.com/IBM/s2i-open-liberty-workshop and click on the **Fork** button in the upper right of the page.
2. When the repo is done forking, click on the green **Clone or download** button and copy your git repo url.
   To make it easier, create an environment variable to hold your repo url. Copy the following command and replace `<repo url>` with your actual repo url:

```
export REPOSITORY_URL=<repo url>
```

3. Now that we have our own copy, let's push a change and test it out. From your browser, navigate to the GitHub repo that you forked. Click on the **Branch** dropdown and select **conference**.
4. Then navigate to the file at `web-app/src/main/java/com/ibm/authors/GetAuthor.java`
5. Click on the pencil icon in the upper right of the code to enter editing mode.
6. On lines 56-59 edit the name, twitter, and blog to your own information or fake information if you'd like.

```
Author author = new Author();
    author.name = "Oliver Rodriguez";
    author.twitter = "https://twitter.com/heres__ollie";
    author.blog = "http://developer.ibm.com";
```

7. Scroll down and click on `Commit changes`.
8. With the changes pushed, we can now rebuild and redeploy the application. Follow the following steps:
   Build the builder image:

```
oc process -f buildTemplate.yaml -p DOCKER_USERNAME=$DOCKER_USERNAME -p
```

```
oc start-build open-liberty-builder
```

   Ensure that the previous build is finished, then start the runtime build:

```
oc start-build open-liberty-app
```

   Ensure that the runtime image build is finished, then deploy the app with the following command:

```
oc new-app --template authors-app -p DOCKER_USERNAME=$DOCKER_USERNAME -
```

9. Once you have verified that the application is deployed and the new pod is running, enter the following command to view the application routes.

```
oc get routes

NAME    HOST/PORT    PATH    SERVICES    PORT    TERMINATION    WILDC
authors-route    authors-route-default.your-roks-43-1n-cl-2bef1f4b409
authors2    authors2-default.your-roks-43-1n-cl-2bef1f4b4097001da9502
```

10. Copy the route named `authors-3` and add it to the end of the command below.

```
export API_URL=
```

View the sample below, however, you will have a different route.

```
export API_URL=authors2-default.osgdcw01-0e3e0ef4c9c6d831e8aa6fe01f33bf
```

Then test your application with the command below:

```
curl -X GET "http://$API_URL/api/v1/getauthor" -H "accept: application/
```

You should then see the info that you edited in the file earlier.

Conclusion

In this lab we have explored building our own custom s2i images for building containerized application from source code. We utilized a multi stage s2i process that separated the build environment from the runtime environment which allowed for us to have a slimmer application image. Then, we deployed the application as a traditional Kubernetes deployment. Lastly, we explored how to build and deploy the application using templates, OpenShift build configs, and deployment configs.

Lab - CI/CD Pipeline with Jenkins on OpenShift

Lab 1 - Deploy application to OpenShift via s2i

Lab - Deploy "Plants by WebSphere Java application" to OpenShift

---

Overview

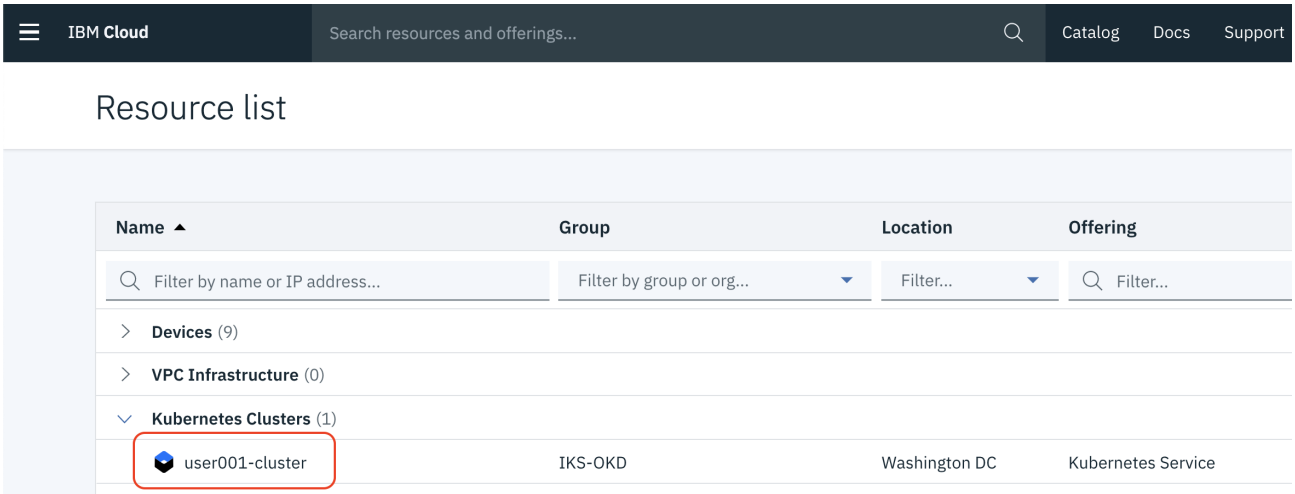
S2I in a tool deployed in OpenShift that provides a repeatable method to generate application images from source/binary code Templates provide a parameterized set of objects that can be processed by OpenShift. Templates provide a parameterized set of objects that can be processed by OpenShift.

In this lab you'll use these capabilities can be used to deploy a small legacy Java EE app to OpenShift on the IBM Cloud Kubernetes Service.

Step 1: Logon into the OpenShift Web Console and to the OpenShift CLI

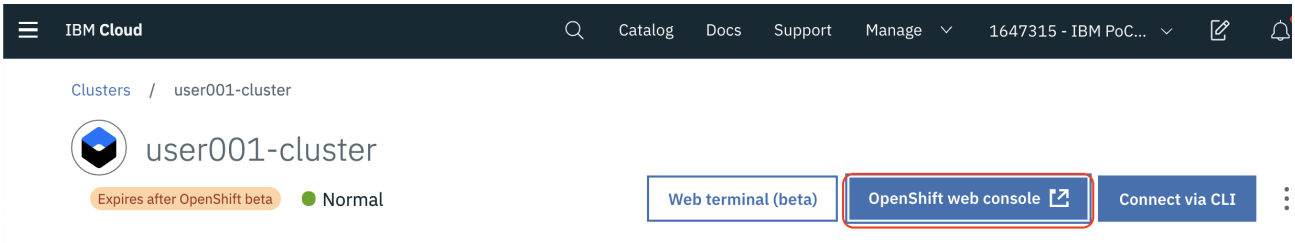
1.1 Go to your IBM Cloud resource list https://cloud.ibm.com/resources

1.2 Click on your designated OpenShift cluster



Your designated cluster

1.3 Click on **OpenShift web console**



Web console

1.4 From the OpenShift web console click on your username in the upper right and select **Copy Login Command**

Copy Login Command

1.5 Click `Display Token` link.

1.6 Copy the login command in the `Log in with this token` field.

1.7 Paste the login command in a terminal window and run it (Note: leave the web console browser tab open as you'll need it later on in the lab)

Step 2: Clone the WebSphere Liberty S2I image source, create a Docker image, and push it to the OpenShift internal registry

2.1 Clone the the WebSphere Liberty S2I image source by issuing the following commands in the terminal window you just used to login via the CLI

```
git clone https://github.com/IBMAppModernization/s2i-liberty-javaee7.gi
cd s2i-liberty-javaee7
```

2.2 In order to deploy to OpenShift, you need to push container images to your cluster's internal registry. First, run the following commands to authenticate with your OpenShift image registry.

```
oc patch configs.imageregistry.operator.openshift.io/cluster --patch '{

export INTERNAL_REG_HOST=$(oc get route default-route -n openshift-imag
echo $INTERNAL_REG_HOST
```

2.3 Create a new OpenShift project for this lab

```
oc new-project pbw-liberty-mariadb
```

2.4 Build the S2I Liberty image and tag it appropriately for the internal registry

```
docker build -t $INTERNAL_REG_HOST/`oc project -q`/s2i-liberty-javaee7:1.
```

2.5 Login to the internal registry

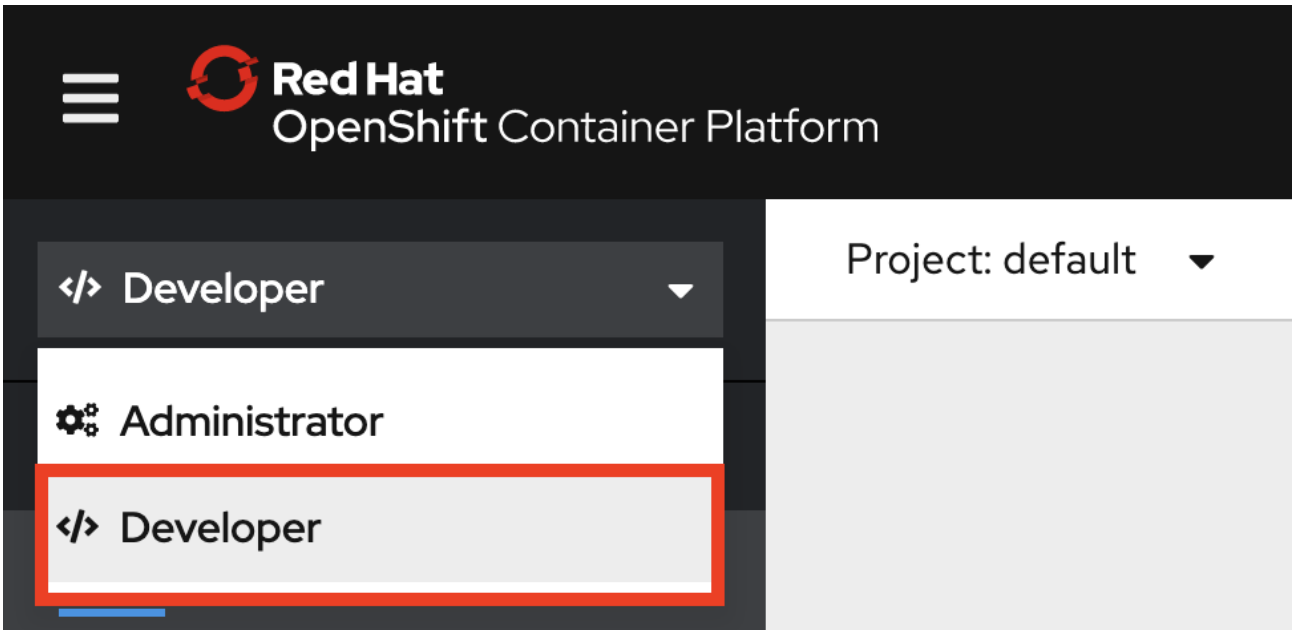
```
docker login -u `oc whoami` -p `oc whoami -t` $INTERNAL_REG_HOST
```

2.6 Push the S2I Liberty image to the internal registry

```
docker push $INTERNAL_REG_HOST/`oc project -q`/s2i-liberty-javaee7:1.0
```

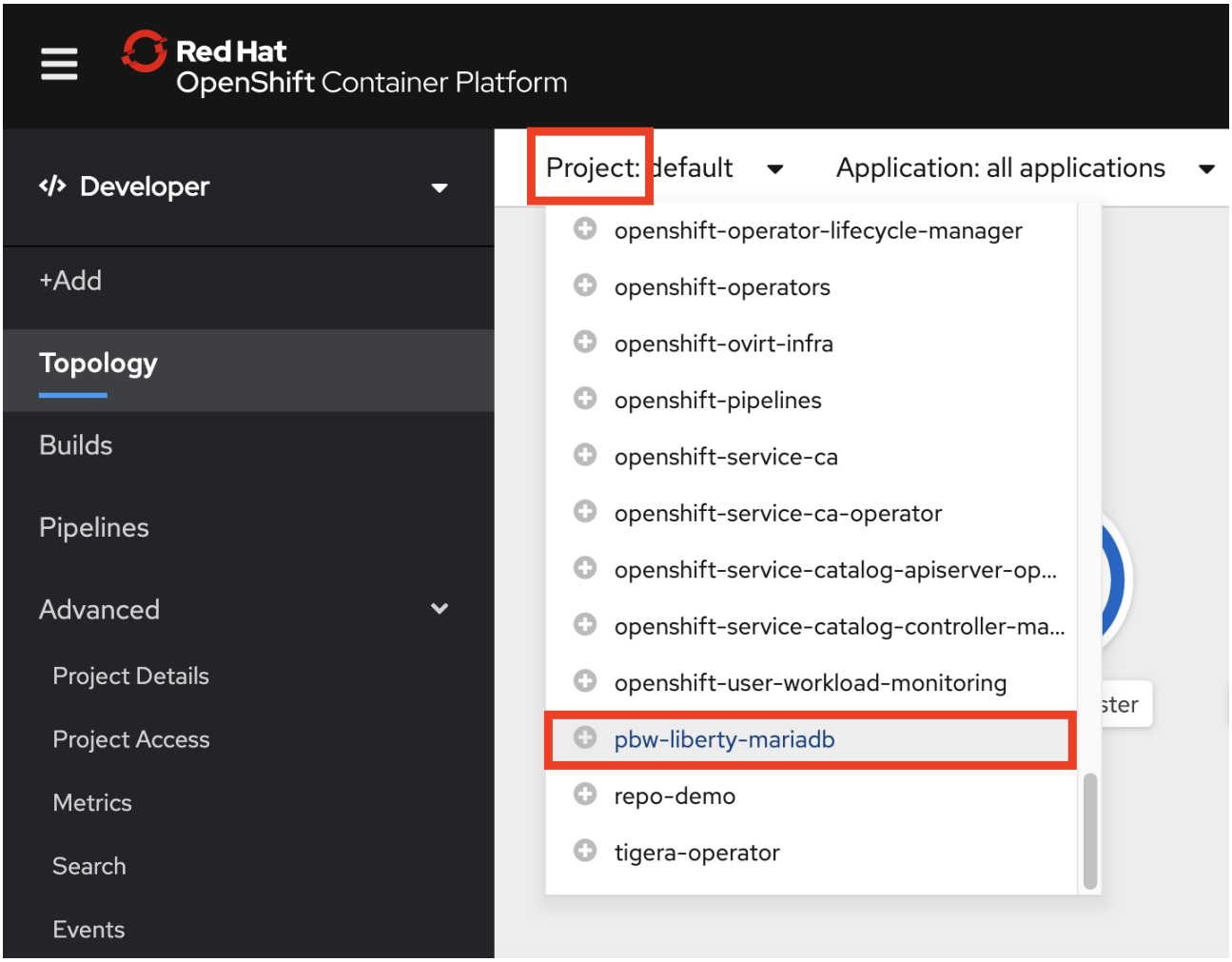
Step 3: Install MariaDB from the OpenShift template catalog

3.1 In your OpenShift Web console, switch to `Developer` view.



View All

3.2 From the `Project` dropdown list, select **pbw-liberty-mariadb** project.

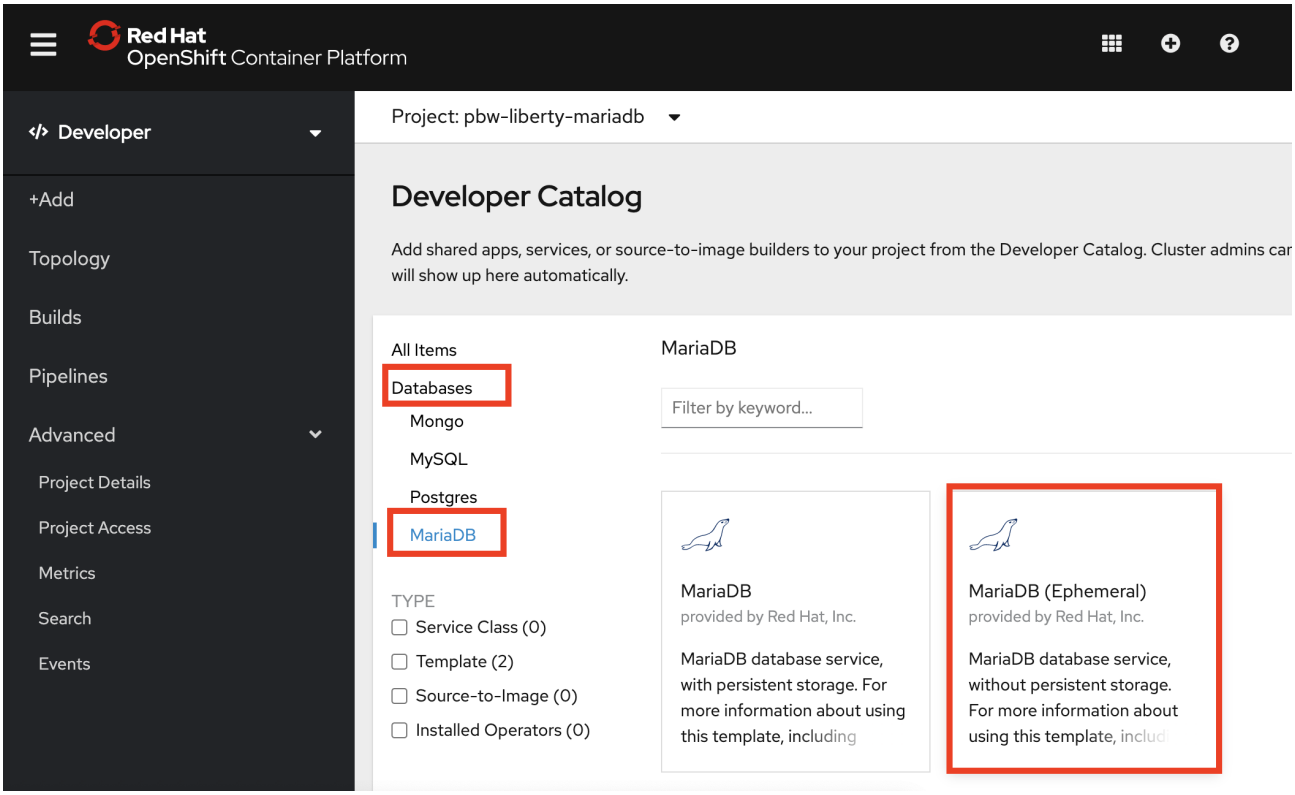


View All

3.3 Click on **From Catalog** tile.

3.4 Under `All IItems` , select the **Databases** category, then choose **MariaDB**.

3.5 Select **MariaDB (Ephemeral)** tile on the right.



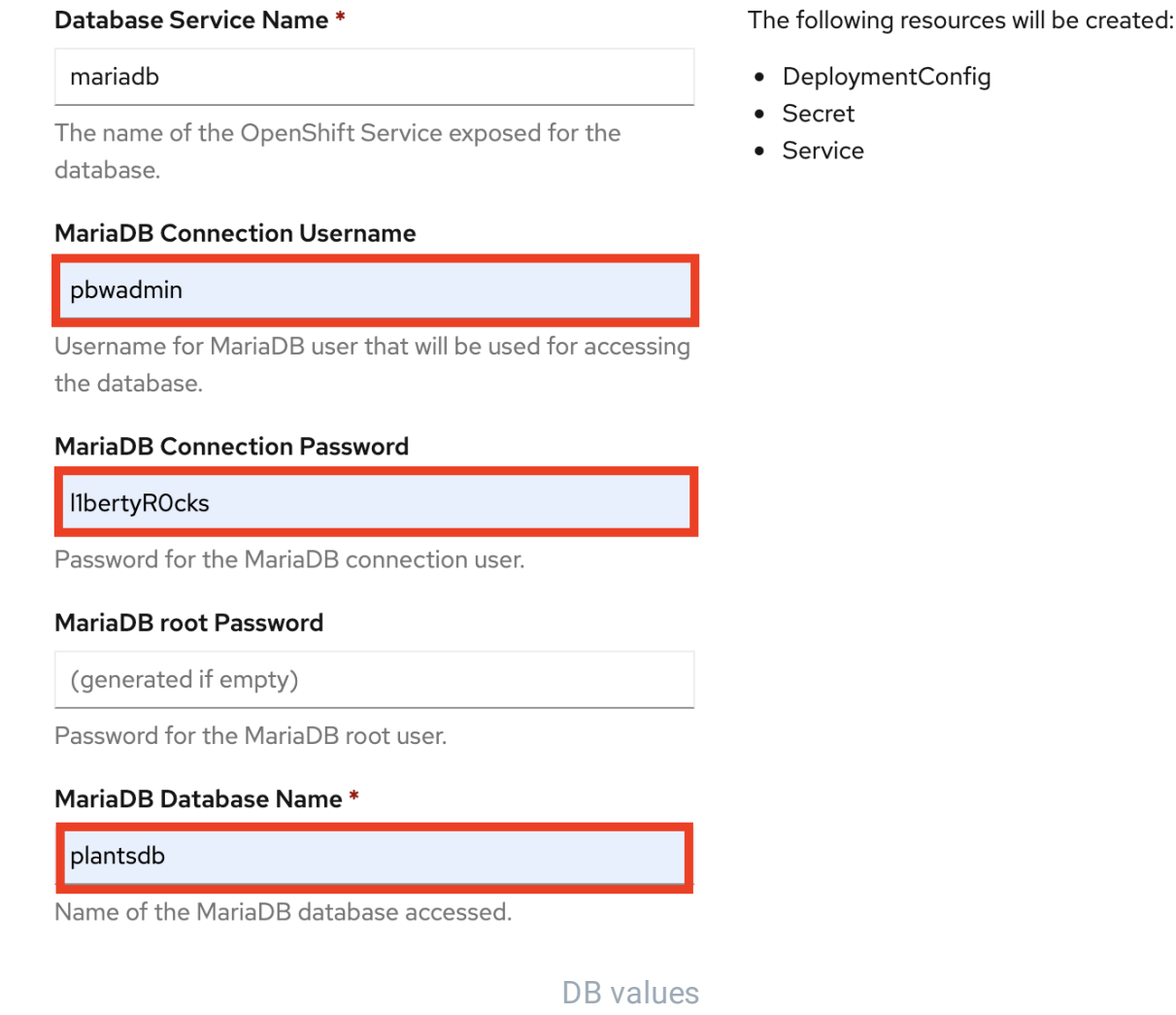
Create MariaDB

3.6 Click **Instantiate Template**.

3.7 Enter the following values for the fields indicated below (leave remaining values at their default values)

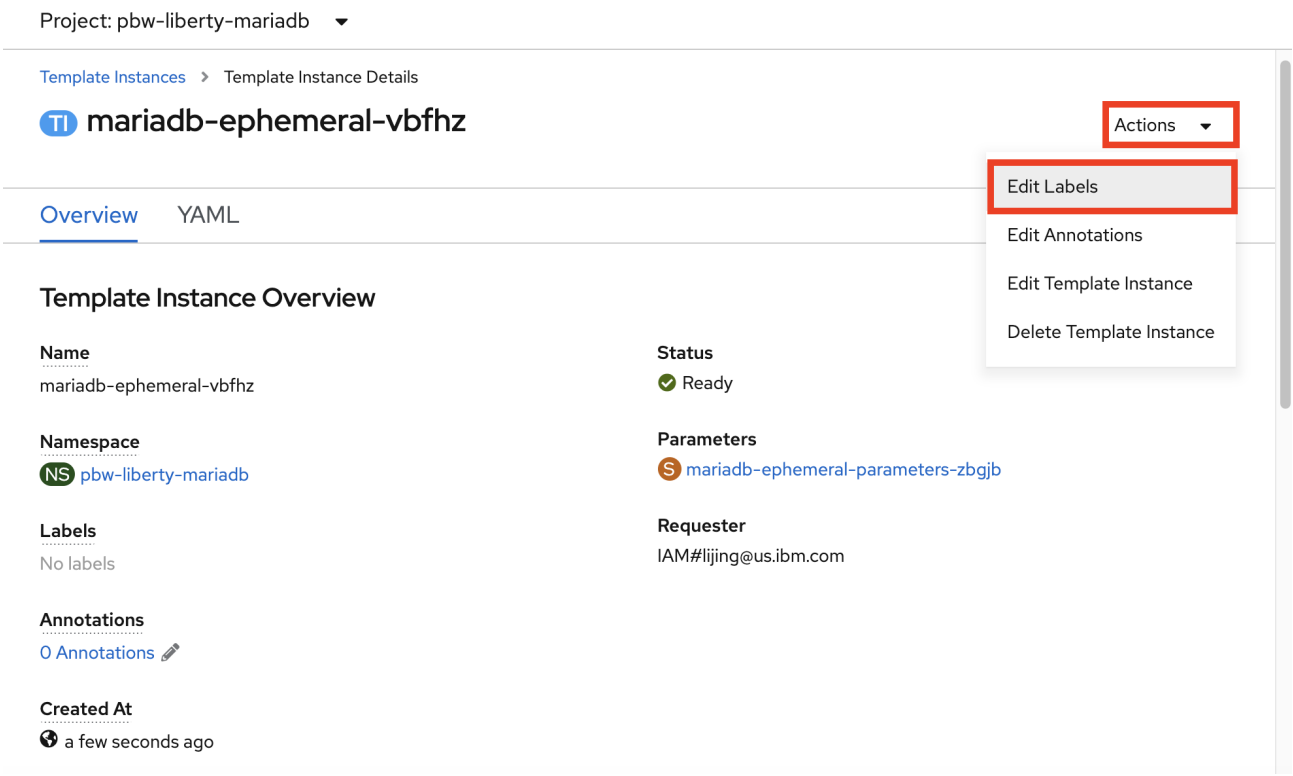
Field name	Value
MariaDB Connection Username	`pbwadmin`
MariaDB Connection Password	`l1bertyR0cks`
MariaDB Database Name	`plantsdb`

When you're done the dialog should look like the following:



3.8 Scroll down and click **Create**.

3.9 From the `Actions` dropdown mnenu, select the **Edit Labels**.

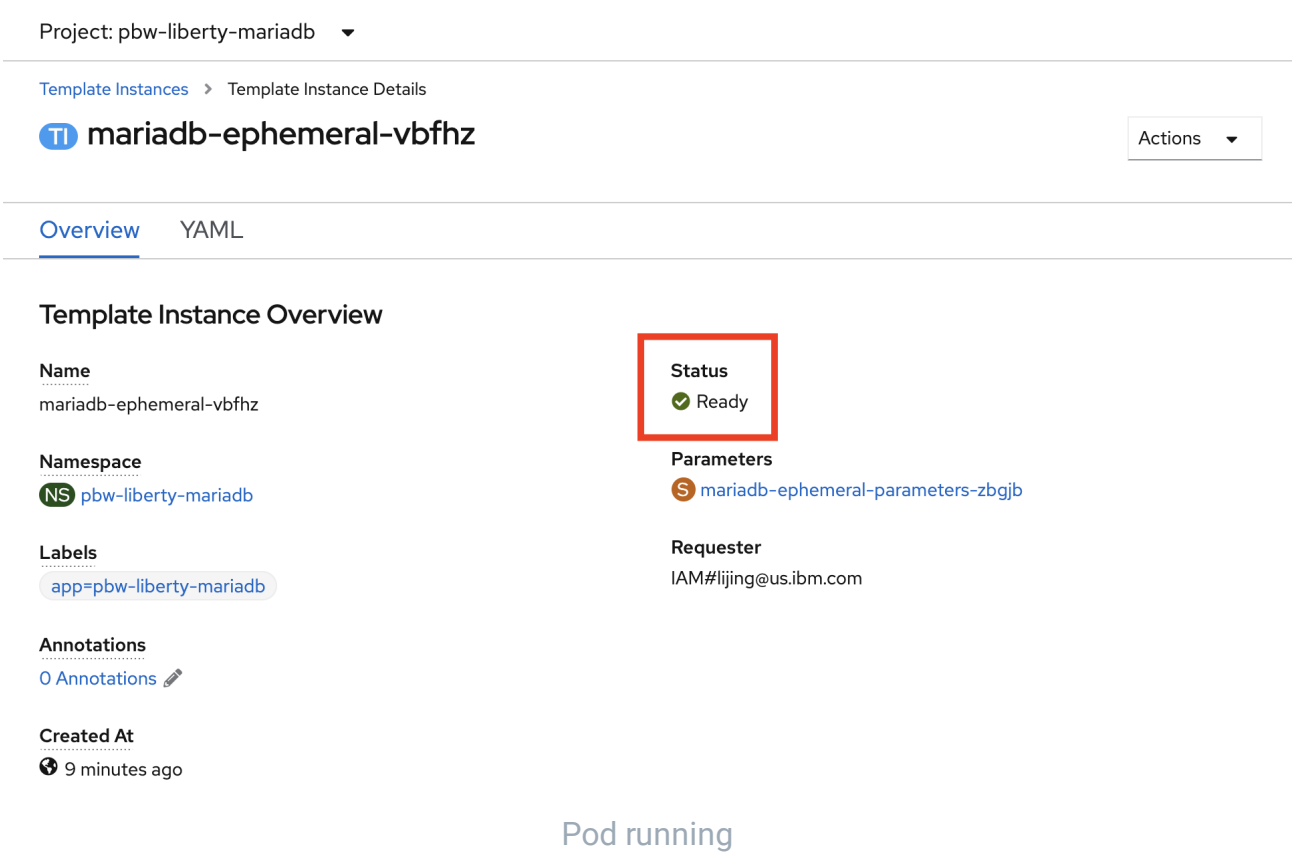


App label

3.10 Enter `app=pbw-liberty-mariadb` in field `Labels for TI mariadb-ephemeral-vbfhz`.

3.11 `Save`.

3.12 It may take couple of minutes for the new database instance to be ready. Verify the status of MariaDB instance before moving on to the next step.

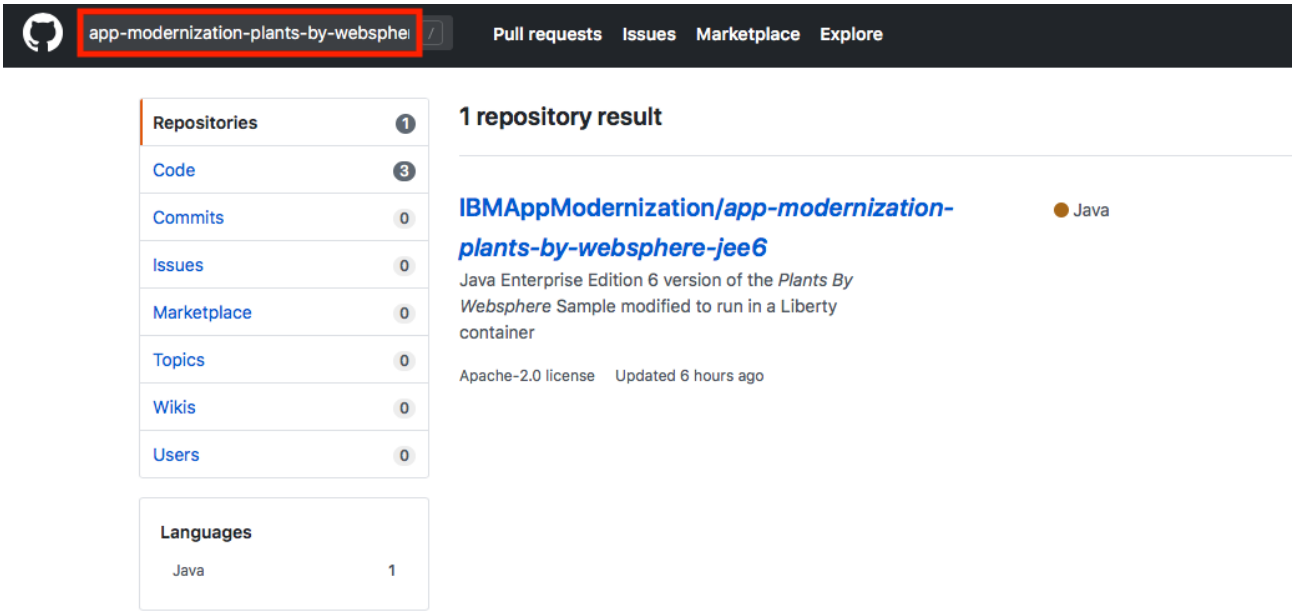


Step 4: Clone the Github repo that contains the code for the Plants by WebSphere app

4.1 Login in your Github account

4.2 In the search bar at the top left, type in

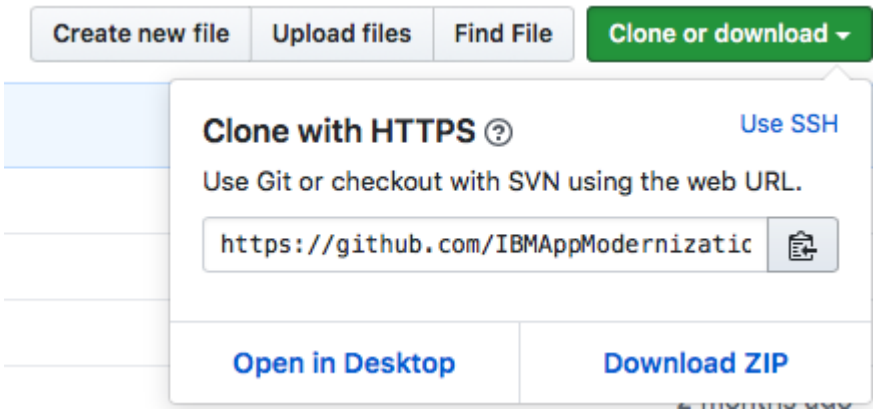
```
app-modernization-plants-by-websphere-jee6
```



Search results

4.3 Select the repository `IBMAppModernization/app-modernization-plants-by-websphere-jee6` and then click on the **Fork** icon

4.4 Click the **Clone or download** button from your copy of the forked repo and copy the HTTPS URL to your clipboard



Clone URL

4.5 From your terminal go back to your home directory

```
cd ~
```

4.6 From the client terminal window clone the Git repo with the following commands appending the HTTPS URL from your clipboard

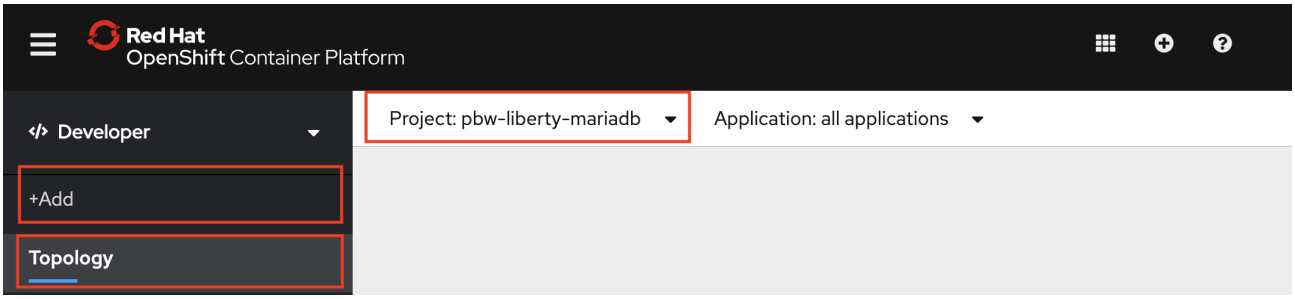
```
git clone [HTTPS URL for NEW REPO]
cd app-modernization-plants-by-websphere-jee6
```

Step 5: Install the Plants by WebSphere Liberty app using a template that utilizes S2I to build the app image

5.1 Add the Plants by WebSphere Liberty app template to your OpenShift cluster in the terminal window.

```
oc create -f openshift/templates/s2i/pbw-liberty-template.yaml
```

5.2 In your OpenShift coonsole, make sure you're in the **pbw-liberty-mariadb** project.



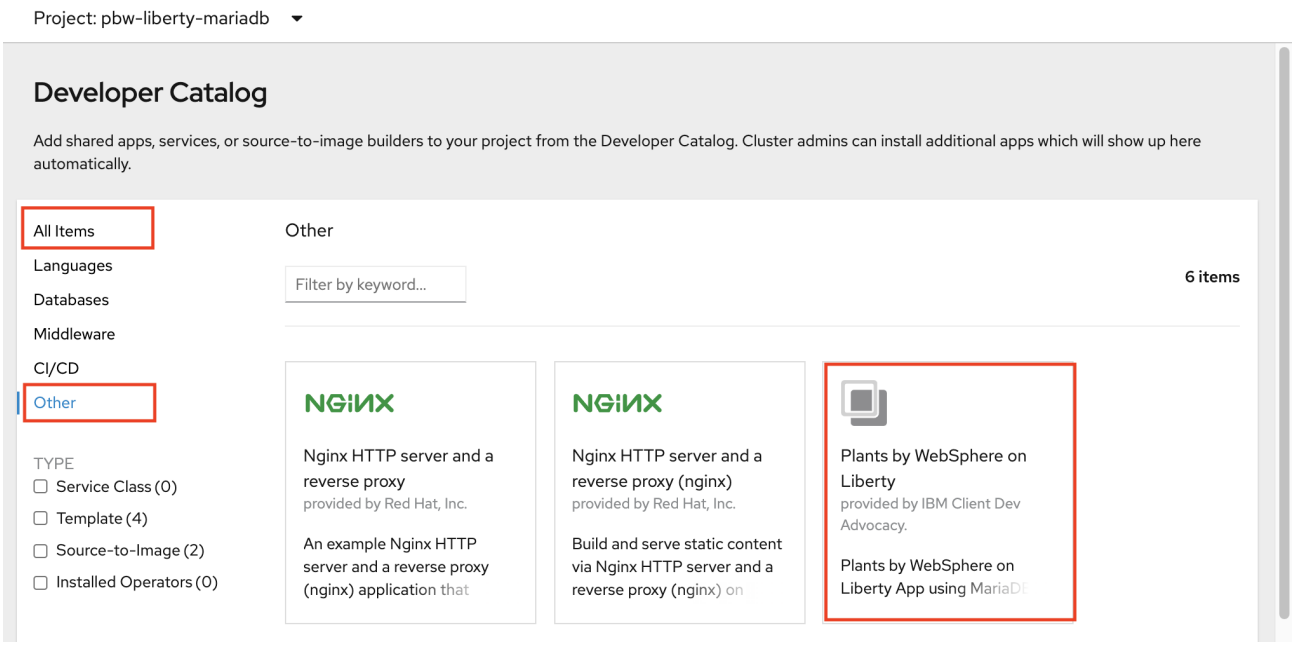
View All

5.3 Select `Topology` in the left pane to view your MariaDB instance.

5.4 Select **+Add** in the left pane.

5.5 Select `From Catalog` tile.

5.6 Select the **Other** category under `All Items` and then select **Plants by WebSphere on Liberty**.



View All

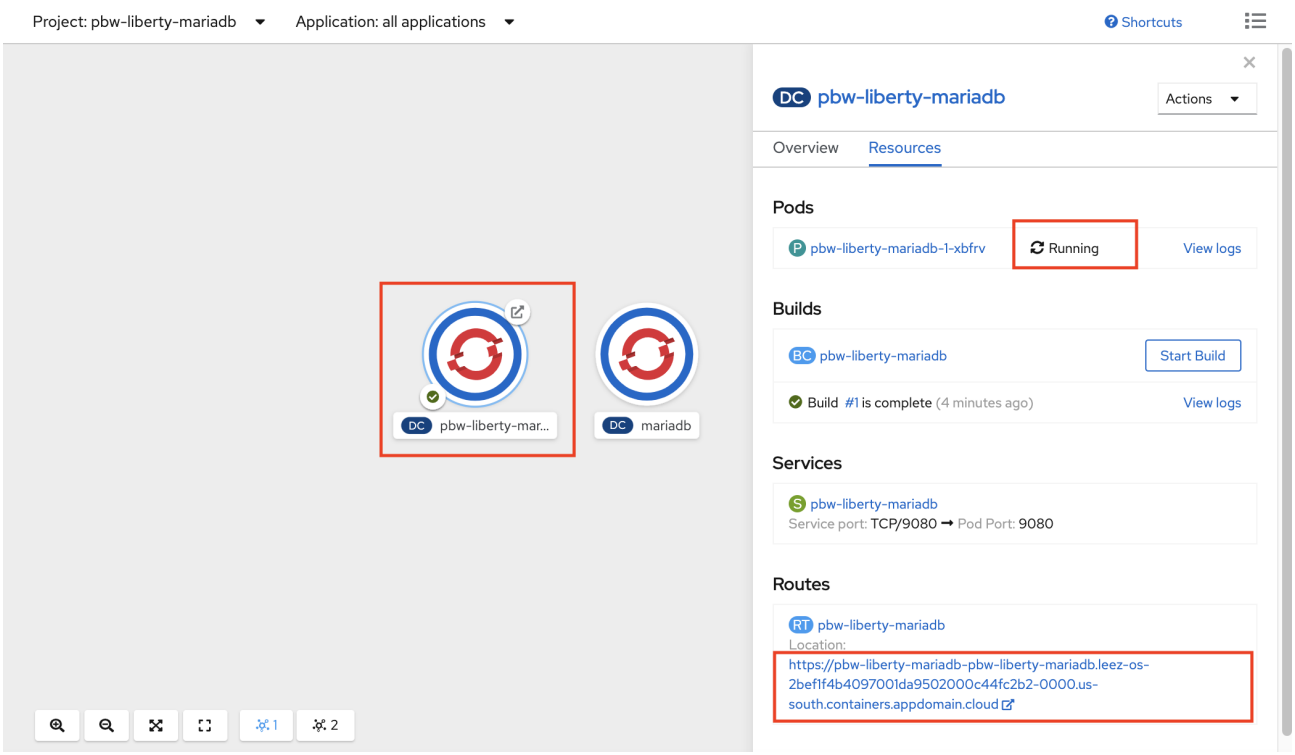
5.7 Select `Instantiate Template`.

5.8 Accept all the default values and click **Create**

5.9 Wait until the instance of the Plants by WebSphere app on Liberty shows as `Ready` status.

5.9 Select `Topology` in the left pane.

5.10 Select `pbw-liberty-mariadb` icon in the right pane.



View All

5.11 It should be in `Running` status.

5.12 Click the `Route` link to access the application.

> Note: The application may not be ready for you to access even deployment has been completed. It may take another couple of minutes when you are able to access it.

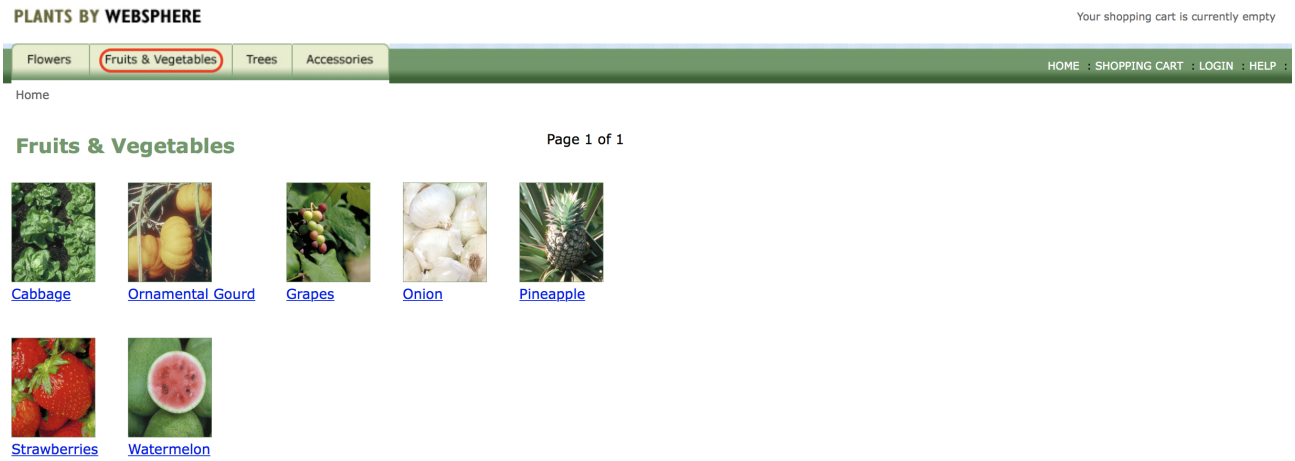
Step 6: Test the Plants by WebSphere app

6.1 From the Plants by WebSphere app UI, click on the **HELP** link

Running app

6.2 Click on **Reset database** to populate the MariaDB database with data

6.3 Verify that browsing different sections of the online catalog shows product descriptions and images.



Online catalog

Summary

With even small simple apps requiring multiple OpenShift objects, templates greatly simplify the process of distributing OpenShift apps. S2I allows you to reuse the same builder image for apps on the same app server, avoiding the effort of having to create unique images for each app.

Lab 2 - Deploy application to OpenShift via CI/CD Pipeline with Jenkins

Lab - Automated updates of containerized applications from SCM commits

Creating a CI/CD Pipeline for deployment to OpenShift using Jenkins

---

Overview

In this lab you will be connecting your Git repository with the Plants by WebSphere app to a Continuous Integration/Continuous Deployment pipeline built with Jenkins that will deploy to an OpenShift cluster.

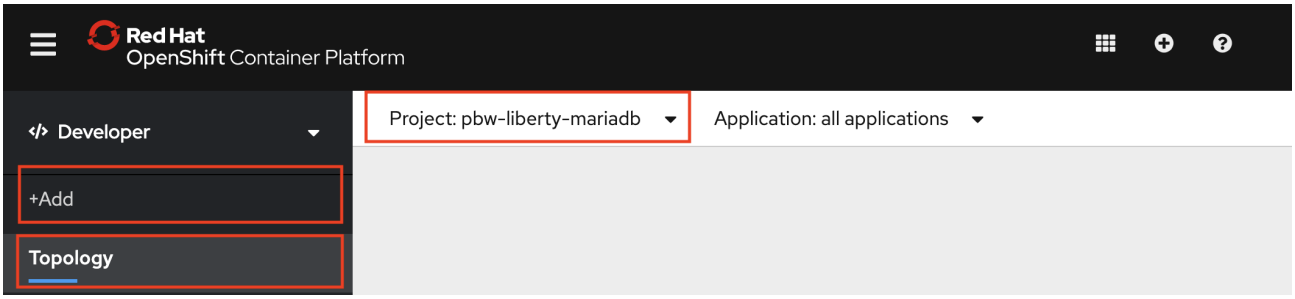
Setup

If you haven't already:

Complete the lab exercise *S2I and Template Lab for the App Modernization Dojo on OpenShift on IBM Cloud Kubernetes Service* by following the instructions here

Step 1: Install Jenkins in your OpenShift cluster

1.1 Open the OpenShift web console in your browser and mke sure you're in the **pbw-liberty-mariadb** project.

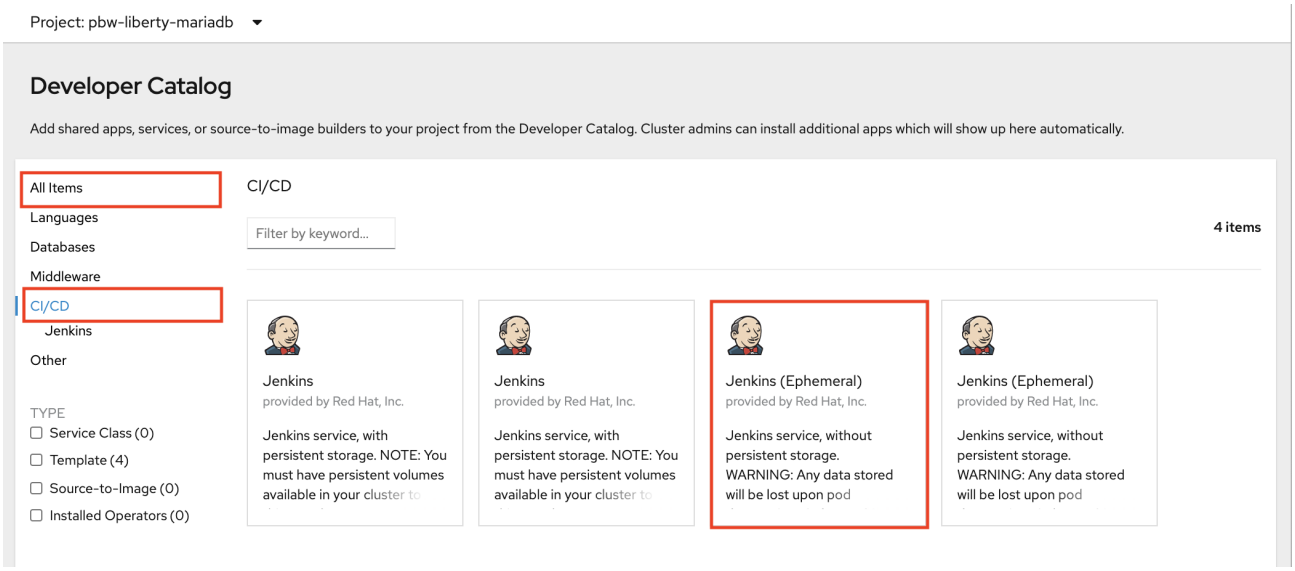


Select project

1.2 Select `+Add` in the left pane.

1.3 Select **From Catalog** tile in the window on the right.

1.4 Select the **CI/CD** category and then select on **Jenkins (Ephemeral)**

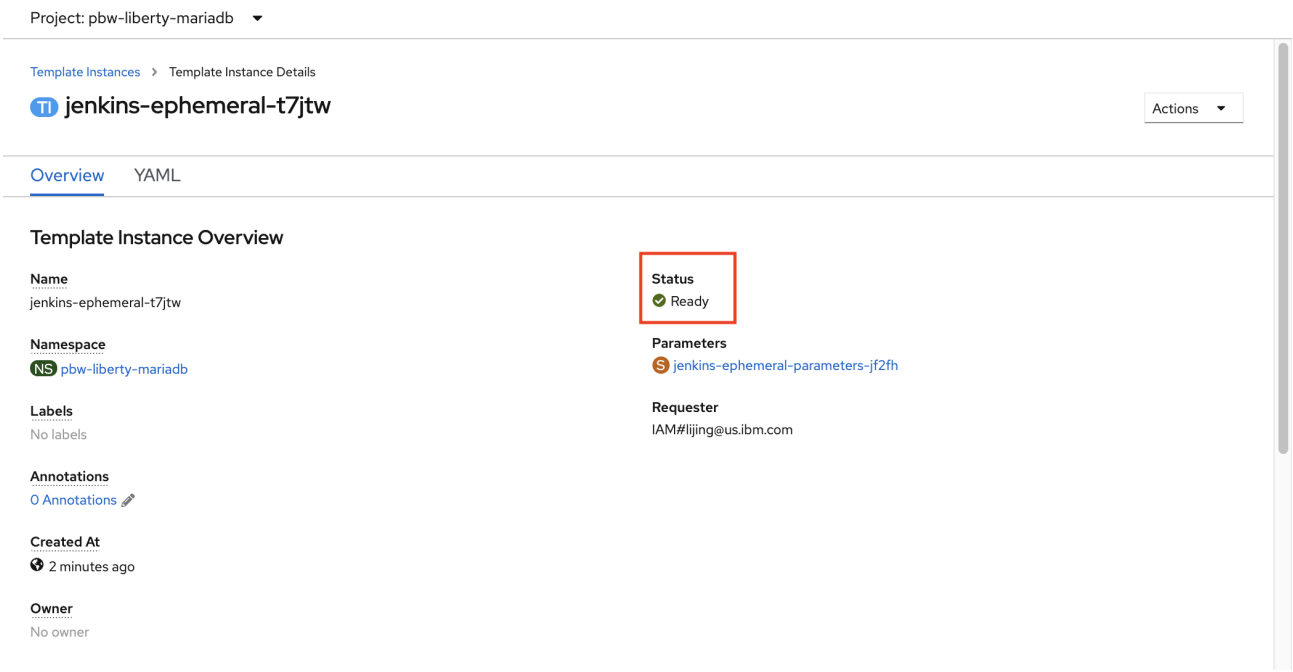


Jenkins

1.5 Select `Instantiate Template`.

1.6 Accept all default click **Create**.

1.7 Wait until the status of the Jenkins instance becomes `Ready` (note this may take a few minutes).



Running

1.8 In the terminal window, run the following command to give Jenkins Service Account push access to the internal container registry

```
oc policy add-role-to-user system:image-builder system:serviceaccount:pbw
```

Step 2: Create Pipeline from a template

2.1 From the terminal run the following command to install the Plants by WebSphere pipeline template (note: you need to be in the top level folder of the cloned Plants by WebSphereGitHub repo)

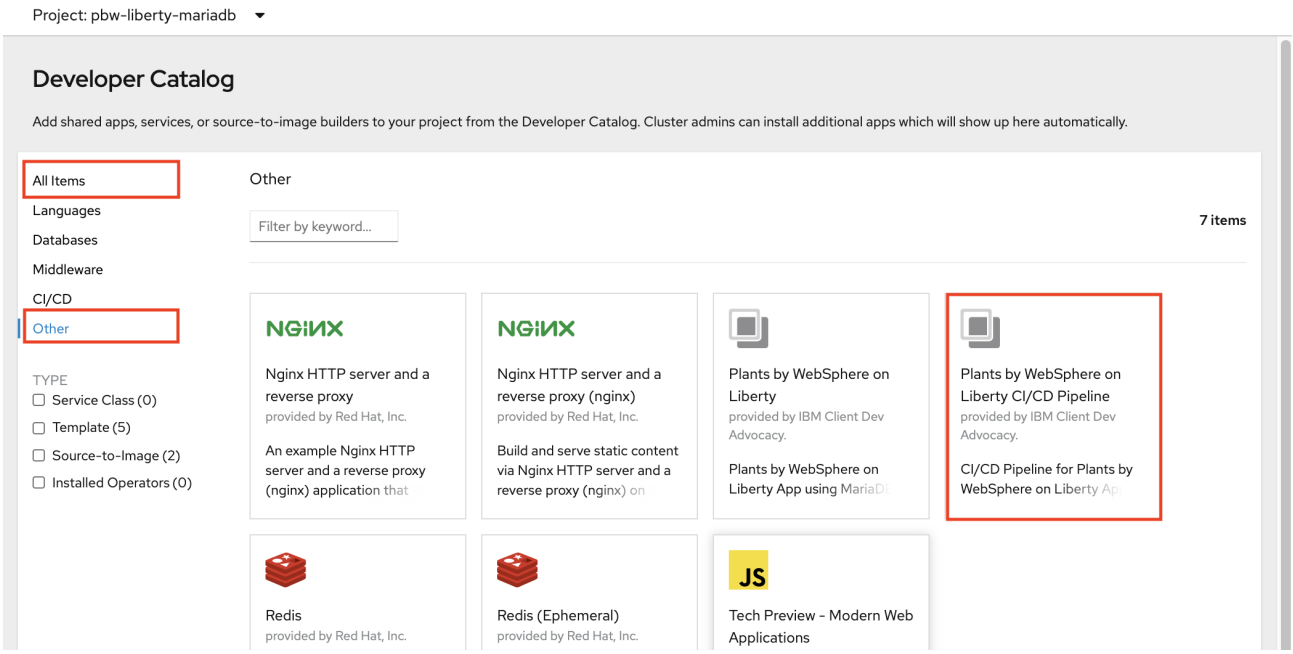
```
oc create -f openshift/templates/cicd/pbw-liberty-cicd-pipeline.yaml
```

2.2 In your Web console browser tab make sure you're in the **pbw-liberty-mariadb** project.

2.3 Select `+Add` in the left pane.

2.4 Select **From Catalog** tile in the window on the right.

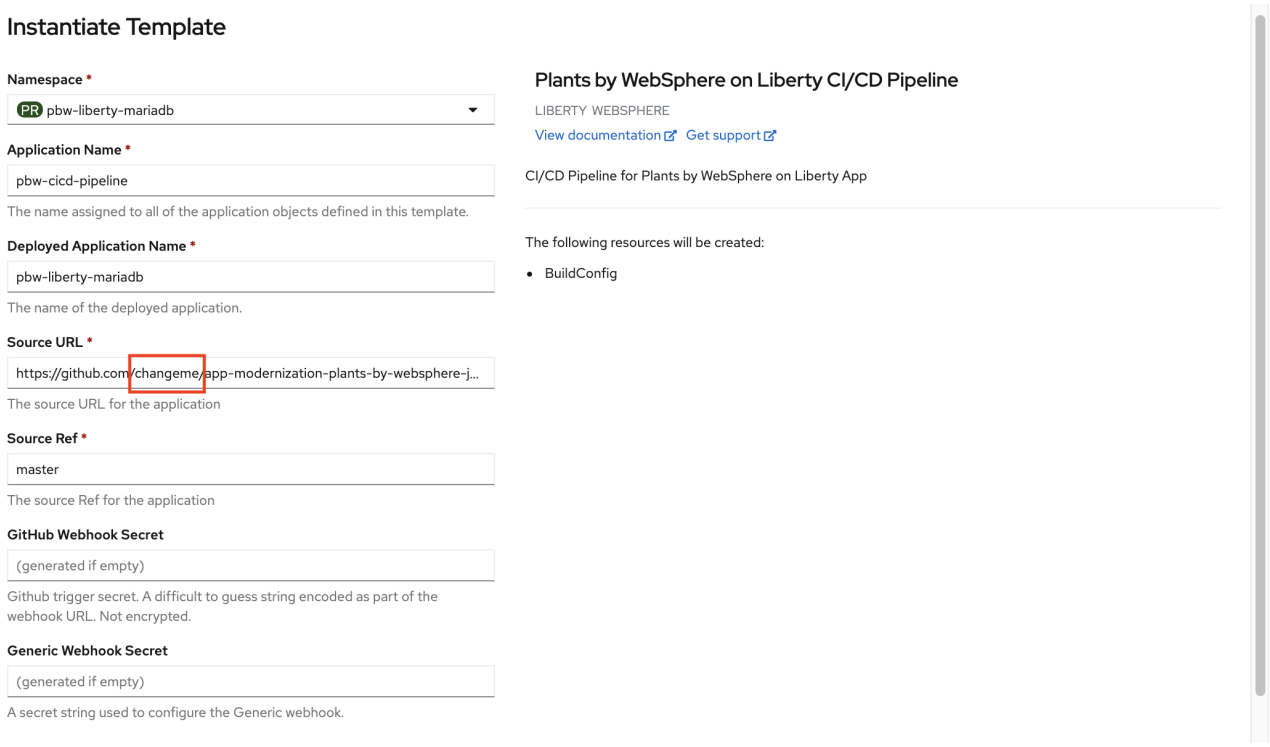
2.5 Select the **Other** category and then click **Plants by WebSphere on Liberty CI/CD Pipeline** tile.



Running

2.6 Select `Instantiate Template`.

2.7 Change the **Source URL** to the url of your clone of the Plants by WebSphere repo

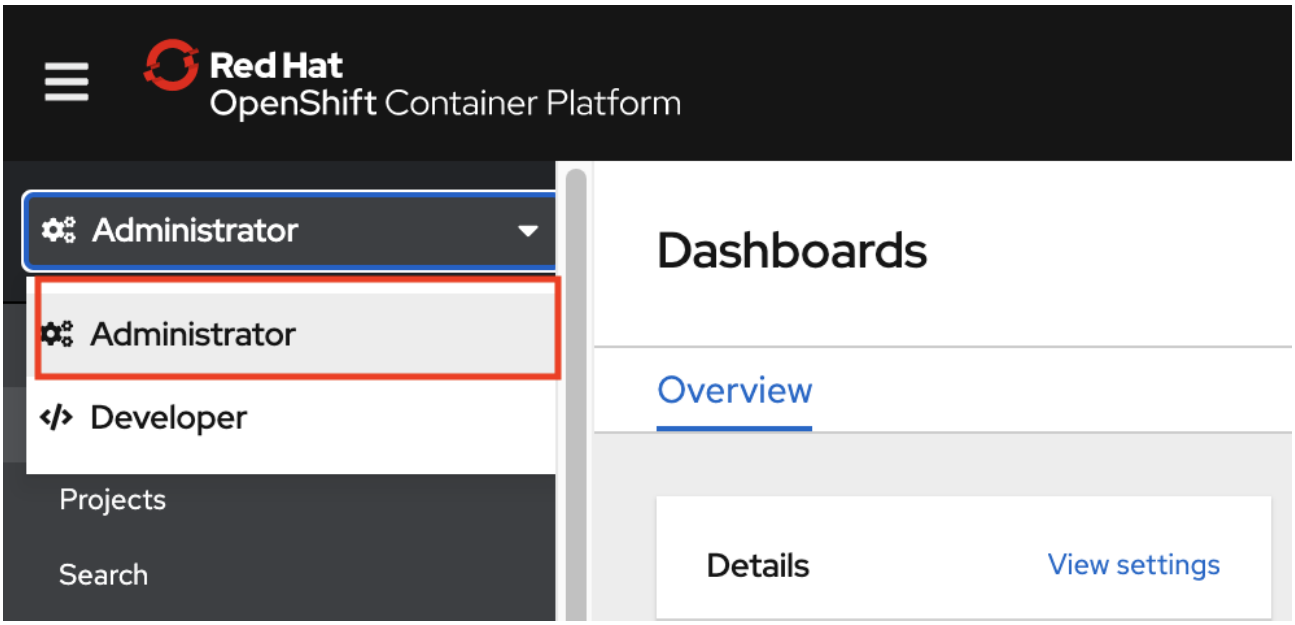


Source URL

2.8 Scroll down and select **Create**.

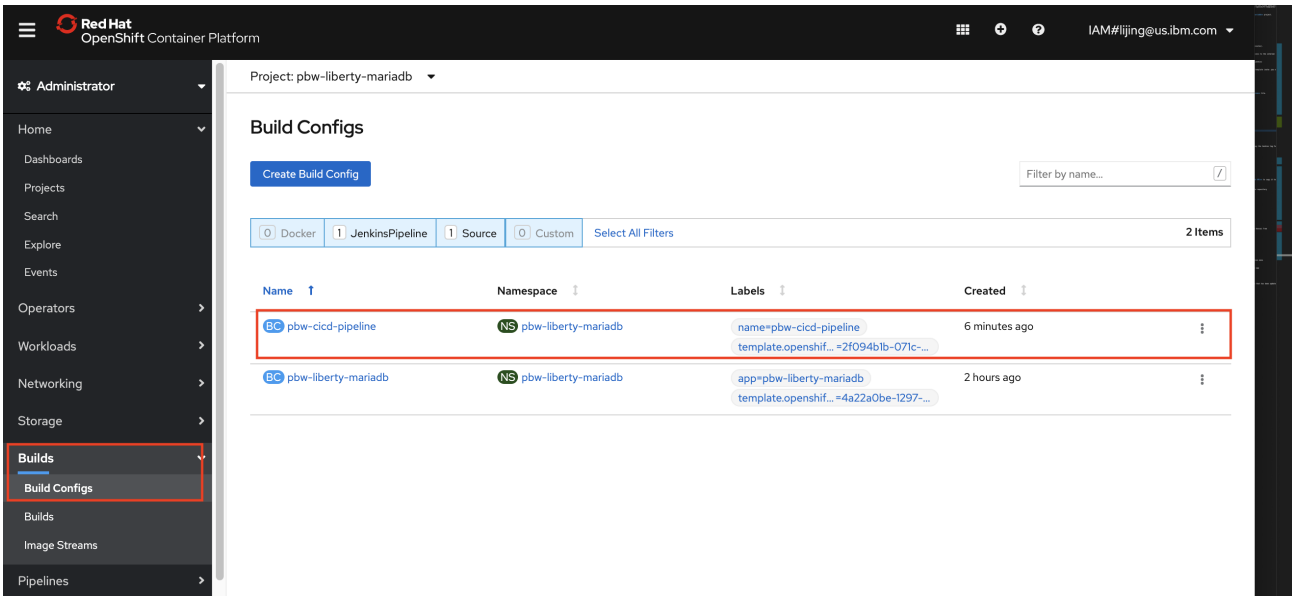
Step 3: Manually trigger a build to test pipeline

3.1 In your OpenShift Web console, switch to `Administrator` view.



Source URL

3.2 Select **Builds -> Build Configs** in the left pane.



Pipelines

3.3 Select `pbw-cicd-pipeline` to open the pipeline.

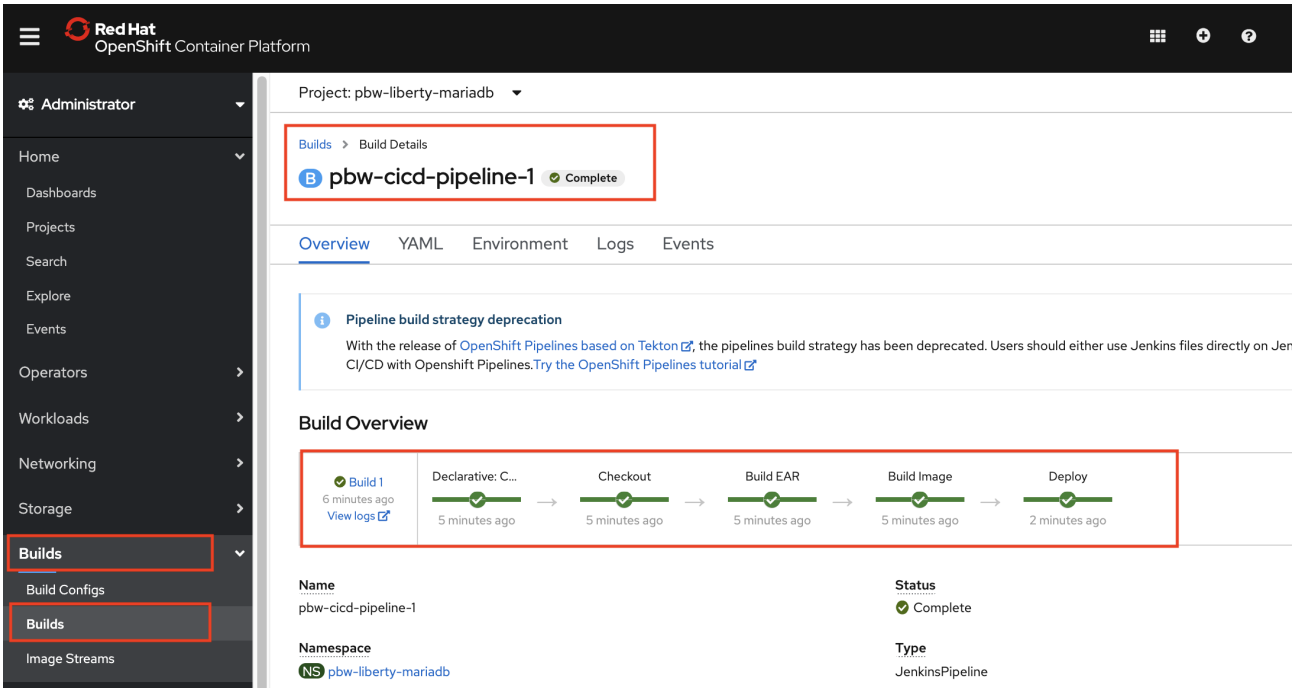
3.4 Select `Start Build` from the `Actions` menu.

3.5 Once the Pipeline starts, navigate to `Logs` tab and click on **View Log**. This will take you into Jenkins console and display the Jenkins log for the pipeline. (Note: you may be prompted to use your OpenShift credentials for Jenkins)

3.6 Verify that the pipeline runs without errors. In Jenkins console, you should see the following log entries when the pipeline execution completes:

Pipeline log

3.7 In the OpenShift console, you should see `Complete` status on the `Overview` tab of `Build Detail`.

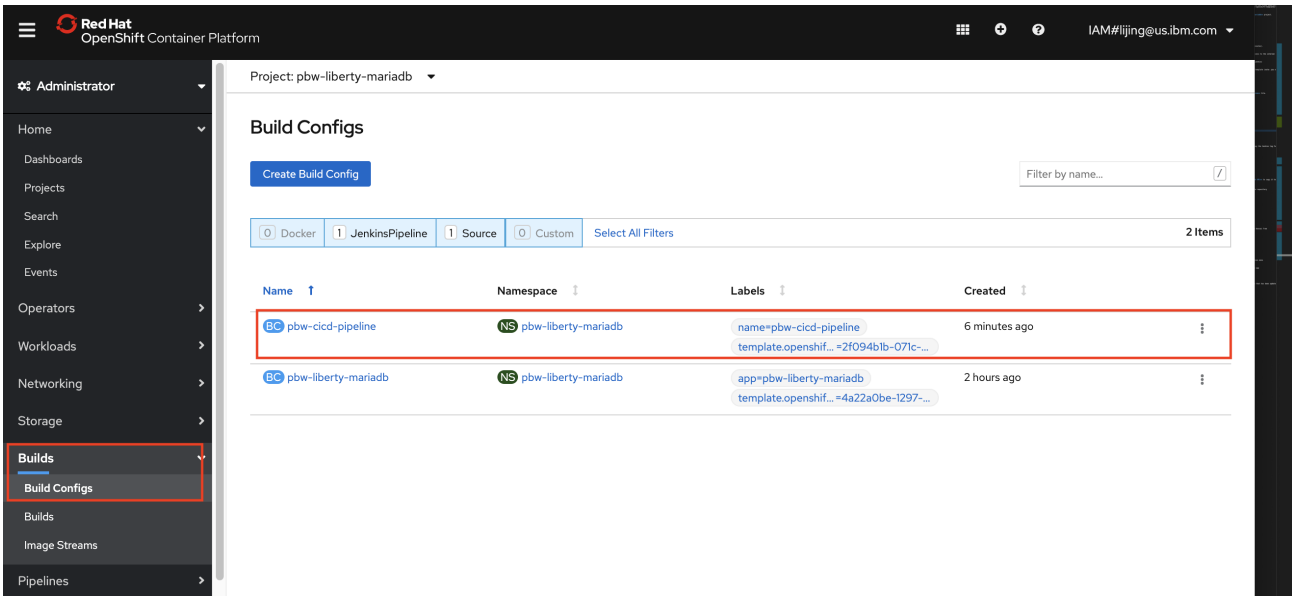


Pipeline log

Step 4: Trigger a build via a commit to Github

The BuildConfig for your pipeline is already configured to be triggered by a Github webhook

4.1 In your Web console, select **Builds -> Build Configs** in the left pane.

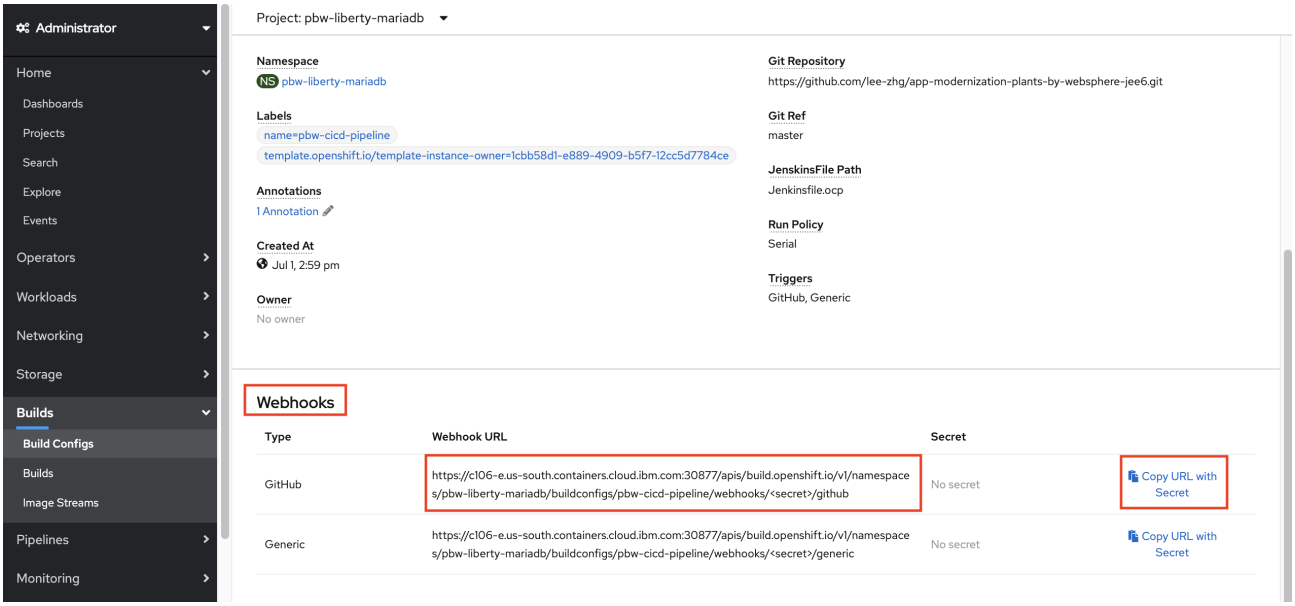


Pipelines

4.2 Select `pbw-cicd-pipeline` to open the pipeline.

4.3 Scroll down to the **Webhooks** section.

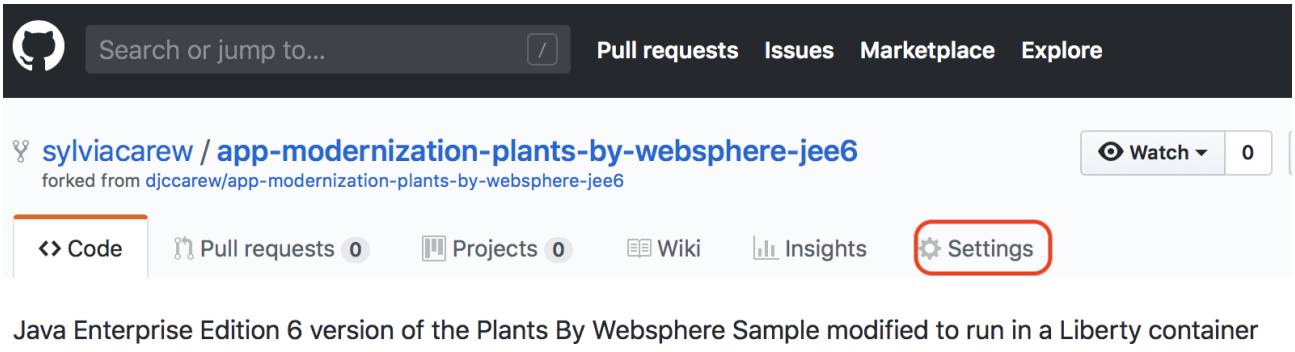
4.4 Copy the `Github Webhook URL` to the clipboard



Copy webhook

4.5 In another browser tab go to https://github.com and open your cloned `Plants by WebSphere` repository

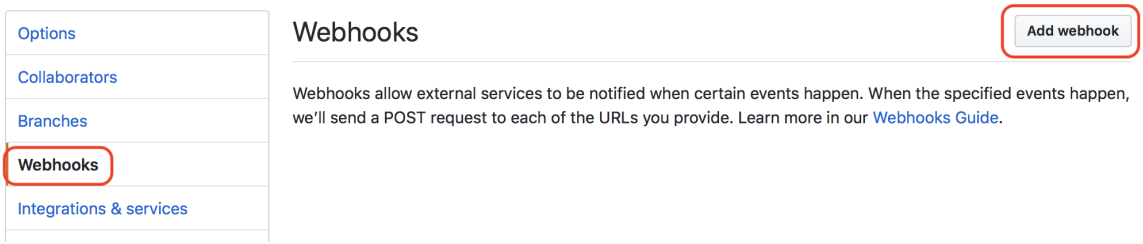
4.6 Navgate to the repository `Settings` tab



Settings

4.7 Navigate to the **Webhooks** subtab.

4.8 Click **Add webhook**

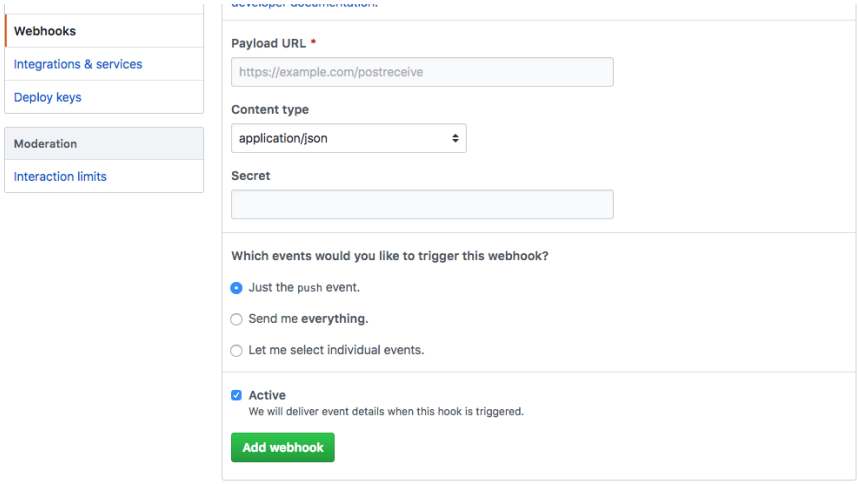


Add webhook

4.9 For the Payload URL, paste in the URL you copied to your clipboard in the previous step.

4.10 Change content type to **application/json**.

4.11 Accept the other defaults and click **Add webhook**



Add webhook

4.12 In the Github repo, drill down to *pbw-web/src/main/webapp/promo.xhtml*.

4.13 Click on the `pencil` icon to edit the file **promo.xhtml**.

4.14 At line 95, locate the price of the Bonsai Tree.

4.15 Change `$30.00 each` to `<strike>$30.00</strike> $25.00 each`

This will show the price of the Bonsai Tree as being reduced even more

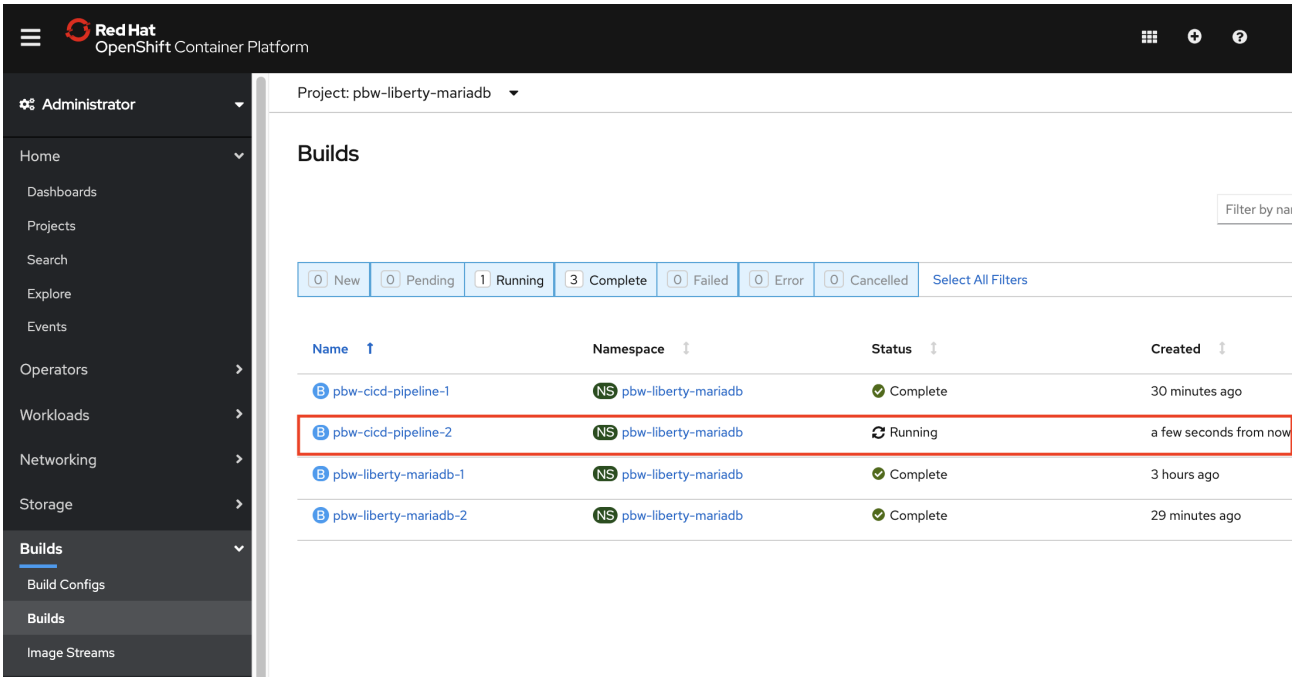
```
90                <h:commandLink styleClass="promos" action="#{shopping.performProductDetail}" >
91                  <f:param name="itemID" value="T0003" />
92                  <f:verbatim>
93                    Bonsai Tree
94                    <br />
95                          <strike>$30.00</strike> $25.00 each
96                  </f:verbatim>
97                </h:commandLink>
98              </p>
```

Reduce Bonsai price

4.16 At the bottom of the UI window add a commit message and click on **Commit changes**

4.17 Switch back to your OpenShift console and select **Builds -> Builds** in the left pane.

4.18 Verify that your pipeline is running.

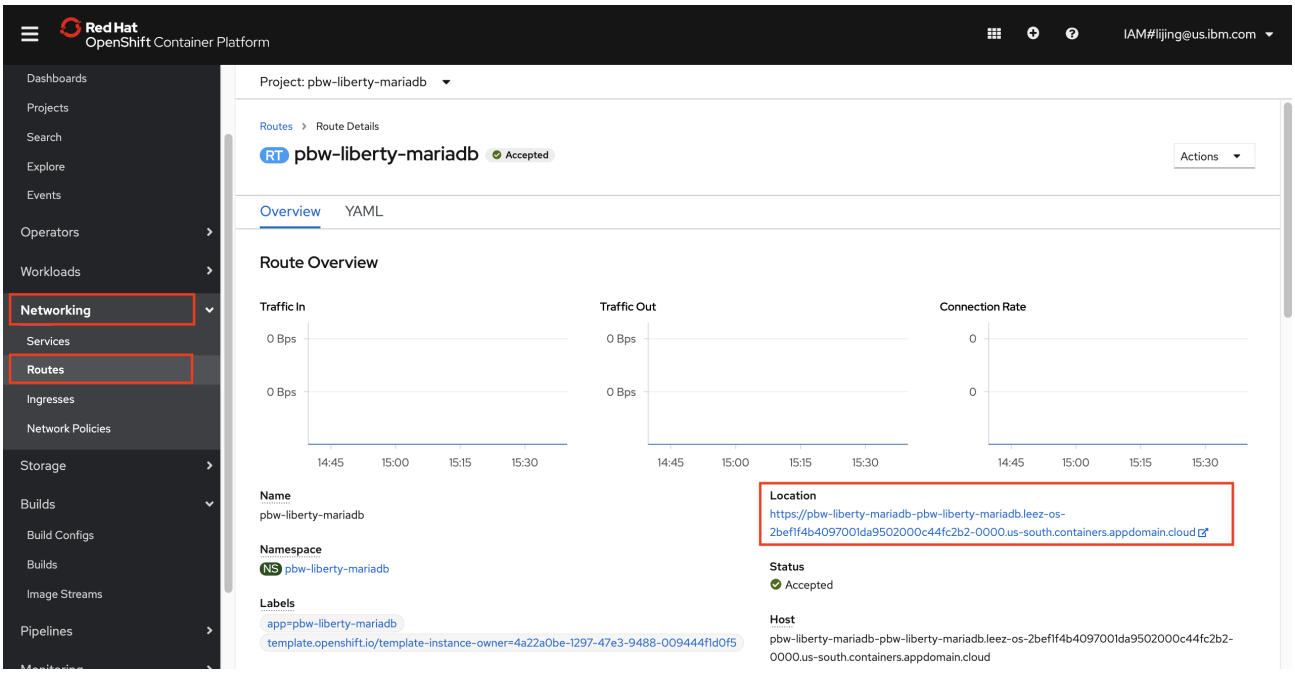


new build

4.19 Once the pipeline has completed, select **Networking -> Routes** in the left navigation pane.

4.20 Select `pbw-liberty-mariadb` .

4.21 The application route is available on this window.



Routes

4.22 Click on the `Location` link of **pbw-liberty-mariadb** to launch the Plants by WebSphere app.

4.23 Verify that the price of the bonzai tree has changed.

Price reduced

Summary

You created a Jenkins pipeline from within OpenShift to automatically build and deploy an app that has been updated in Github .

Resources