
Enterprise
Kubernetes on
AWS
Red Hat

Table of Contents

Red Hat OpenShift on AWS Workshop

In this lab, you'll go through a set of tasks that will help you understand some of the concepts of deploying and securing container based applications on top of Red Hat OpenShift.

You can use this guide as an OpenShift tutorial and as study material to help you get started to learn OpenShift.

Some of the things you'll be going through:

- Creating a project on the Red Hat OpenShift Web Console
- Deploying a MongoDB container that uses AWS EBS for persistent storage
- Deploying a Node JS API and frontend app from Git Hub using Source-To-Image (S2I)
- Exposing the web application frontend using Routes
- Creating a network policy to control communication between the different tiers in the application

You'll be doing the majority of the labs using the OpenShift CLI, but you can also accomplish them using the Red Hat OpenShift web console.

Basic concepts

Source-To-Image (S2I)

Source-to-Image (S2I) is a toolkit and workflow for building reproducible container images from source code. S2I produces ready-to-run images by injecting source code into a container image and letting the container prepare that source code for execution. By creating self-assembling builder images, you can version and control your build environments exactly like you use container images to version your runtime environments.

How it works

For a dynamic language like Ruby, the build-time and run-time environments are typically the same. Starting with a builder image that describes this environment - with Ruby, Bundler, Rake, Apache, GCC, and other packages needed to set up and run a Ruby application installed - source-to-image performs the following steps:

1. Start a container from the builder image with the application source injected into a known directory
2. The container process transforms that source code into the appropriate runnable setup - in this case, by installing dependencies with Bundler and moving the source code into a directory where Apache has been preconfigured to look for the Ruby config.ru file.
3. Commit the new container and set the image entrypoint to be a script (provided by the builder image) that will start Apache to host the Ruby application.

For compiled languages like C, C++, Go, or Java, the dependencies necessary for compilation might dramatically outweigh the size of the actual runtime artifacts. To keep runtime images slim, S2I enables a multiple-step build processes, where a binary artifact such as an executable or Java WAR file is created in the first builder image, extracted, and injected into a second runtime image that simply places the executable in the correct location for execution.

For example, to create a reproducible build pipeline for Tomcat (the popular Java webserver) and Maven:

1. Create a builder image containing OpenJDK and Tomcat that expects to have a WAR file injected

2. Create a second image that layers on top of the first image Maven and any other standard dependencies, and expects to have a Maven project injected
3. Invoke source-to-image using the Java application source and the Maven image to create the desired application WAR
4. Invoke source-to-image a second time using the WAR file from the previous step and the initial Tomcat image to create the runtime image

By placing our build logic inside of images, and by combining the images into multiple steps, we can keep our runtime environment close to our build environment (same JDK, same Tomcat JARs) without requiring build tools to be deployed to production.

Goals and benefits

Reproducibility

Allow build environments to be tightly versioned by encapsulating them within a container image and defining a simple interface (injected source code) for callers. Reproducible builds are a key requirement to enabling security updates and continuous integration in containerized infrastructure, and builder images help ensure repeatability as well as the ability to swap runtimes.

Flexibility

Any existing build system that can run on Linux can be run inside of a container, and each individual builder can also be part of a larger pipeline. In addition, the scripts that process the application source code can be injected into the builder image, allowing authors to adapt existing images to enable source handling.

Speed

Instead of building multiple layers in a single Dockerfile, S2I encourages authors to represent an application in a single image layer. This saves time during creation and deployment, and allows for better control over the output of the final image.

Security

Dockerfiles are run without many of the normal operational controls of containers, usually running as root and having access to the container network. S2I can be used to control what permissions and privileges are available to the builder image since the build is launched in a single container. In concert with platforms like OpenShift, source-to-image can enable admins to tightly control what privileges developers have at build time.

Routes

An OpenShift `Route` exposes a service at a host name, like www.example.com, so that external clients can reach it by name. When a `Route` object is created on OpenShift, it gets picked up by the built-in HAProxy load balancer in order to expose the requested service and make it externally available with the given configuration. You might be familiar with the Kubernetes `Ingress` object and might already be asking "what's the difference?". Red Hat created the concept of `Route` in order to fill this need and then contributed the design principles behind this to the community; which heavily influenced the `Ingress` design. Though a `Route` does have some additional features as can be seen in the chart below.

Feature	Ingress on OpenShift	Route on OpenShift
Standard Kubernetes object	X	
External access to services	X	X
Persistent (sticky) sessions	X	X
Load-balancing strategies (e.g. round robin)	X	X
Rate-limit and throttling	X	X
IP whitelisting	X	X
TLS edge termination for improved security	X	X
TLS re-encryption for improved security		X
TLS passthrough for improved security		X
Multiple weighted backends (split traffic)		X
Generated pattern-based hostnames		X
Wildcard domains		X

NOTE: DNS resolution for a host name is handled separately from routing; your administrator may have configured a cloud domain that will always correctly resolve to the router, or if using an unrelated host name you may need to modify its DNS records independently to resolve to the router.

Also of note is that an individual route can override some defaults by providing specific configuration in its annotations. See here for more details: https://docs.openshift.com/container-platform/4.7/networking/routes/route-configuration.html

ImageStreams

An ImageStream stores a mapping of tags to images, metadata overrides that are applied when images are tagged in a stream, and an optional reference to a Docker image repository on a registry.

What are the benefits

Using an ImageStream makes it easy to change a tag for a container image. Otherwise to change a tag you need to download the whole image, change it locally, then push it all back. Also promoting applications by having to do that to change the tag and then update the deployment object entails many steps. With ImageStreams you upload a container image once and then you manage it's virtual tags internally in OpenShift. In one project you may use the `dev` tag and only change reference to it internally, in prod you may use a `prod` tag and also manage it internally. You don't really have to deal with the registry!

You can also use ImageStreams in conjunction with DeploymentConfigs to set a trigger that will start a deployment as soon as a new image appears or a tag changes its reference.

See here for more details: https://blog.openshift.com/image-streams-faq/
OpenShift Docs:
https://docs.openshift.com/container-platform/4.7/openshift_images/managing_images/managing-images-overview.html
ImageStream and Builds:
https://cloudowski.com/articles/why-managing-container-images-on-openshift-is-better-than-on-kubernetes/

Builds

A build is the process of transforming input parameters into a resulting object. Most often, the process is used to transform input parameters or source code into a runnable image. A BuildConfig object is the definition of the entire build process.

OpenShift Container Platform leverages Kubernetes by creating Docker-formatted containers from build images and pushing them to a container image registry.

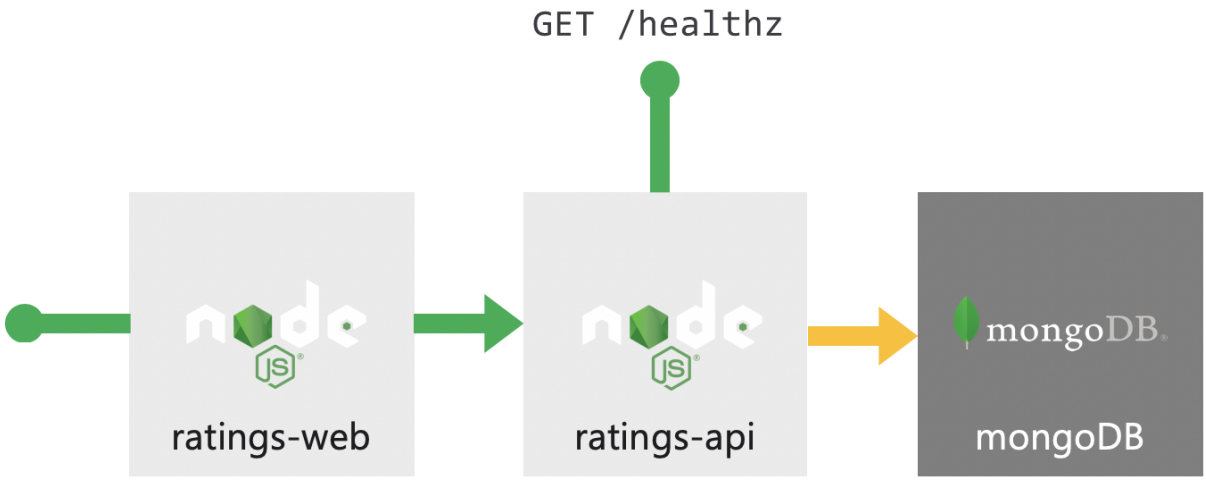
Build objects share common characteristics: inputs for a build, the need to complete a build process, logging the build process, publishing resources from successful builds, and publishing the final status of the build. Builds take advantage of resource restrictions, specifying limitations on resources such as CPU usage, memory usage, and build or pod execution time.

Lab 1 - Go Microservices

Now that you have your environment provisioned and the prerequisites fulfilled, it is time to start working on the labs.

Application Overview

You will be deploying a ratings application on Red Hat OpenShift.

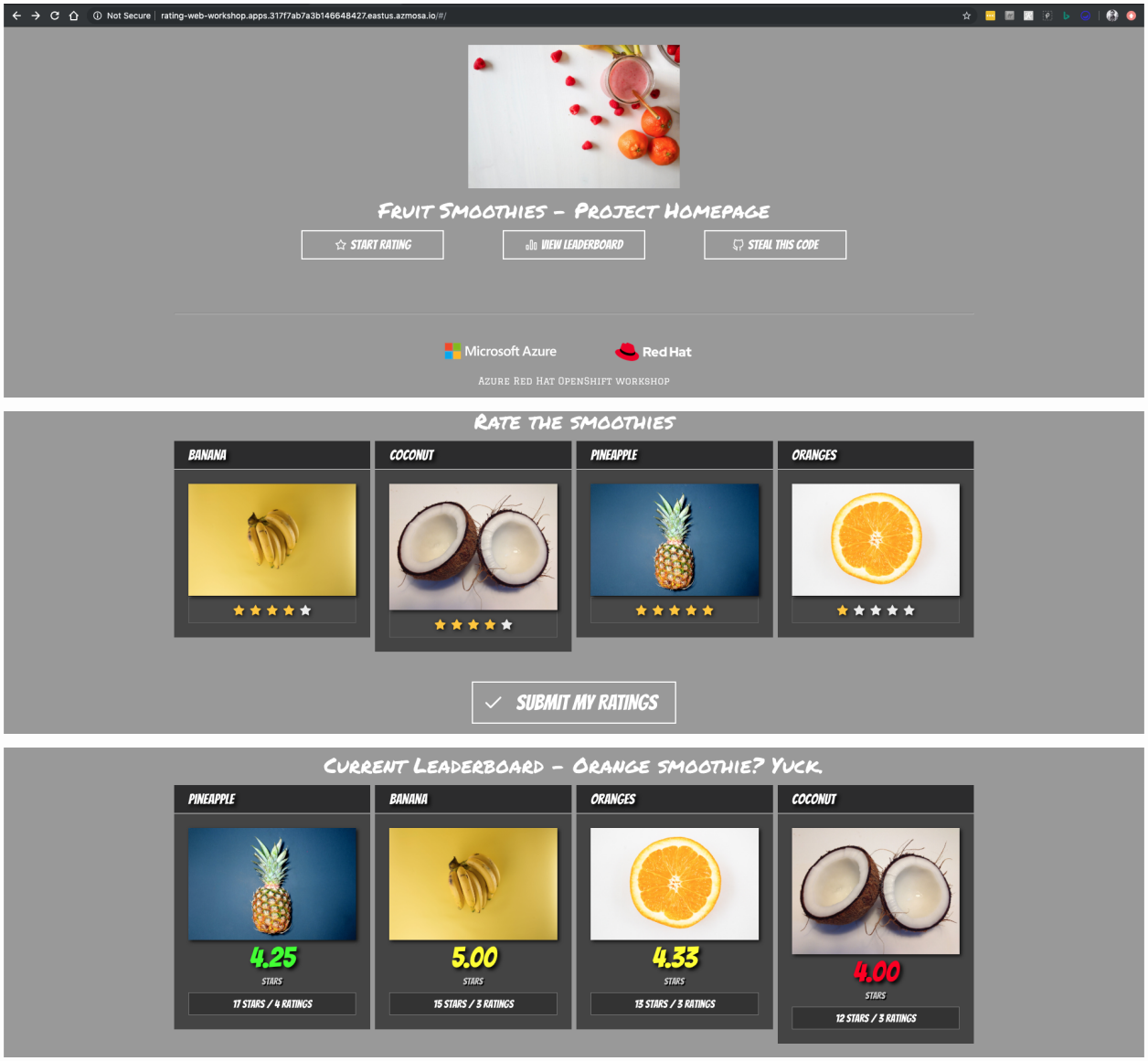


The application consists of 3 components:

Component	Link
A public facing API `rating-api`	GitHub repo

A public facing web frontend `rating-web`	GitHub repo
A MongoDB with pre-loaded data	Data

Once you're done, you'll have an experience similar to the below.



Create Project

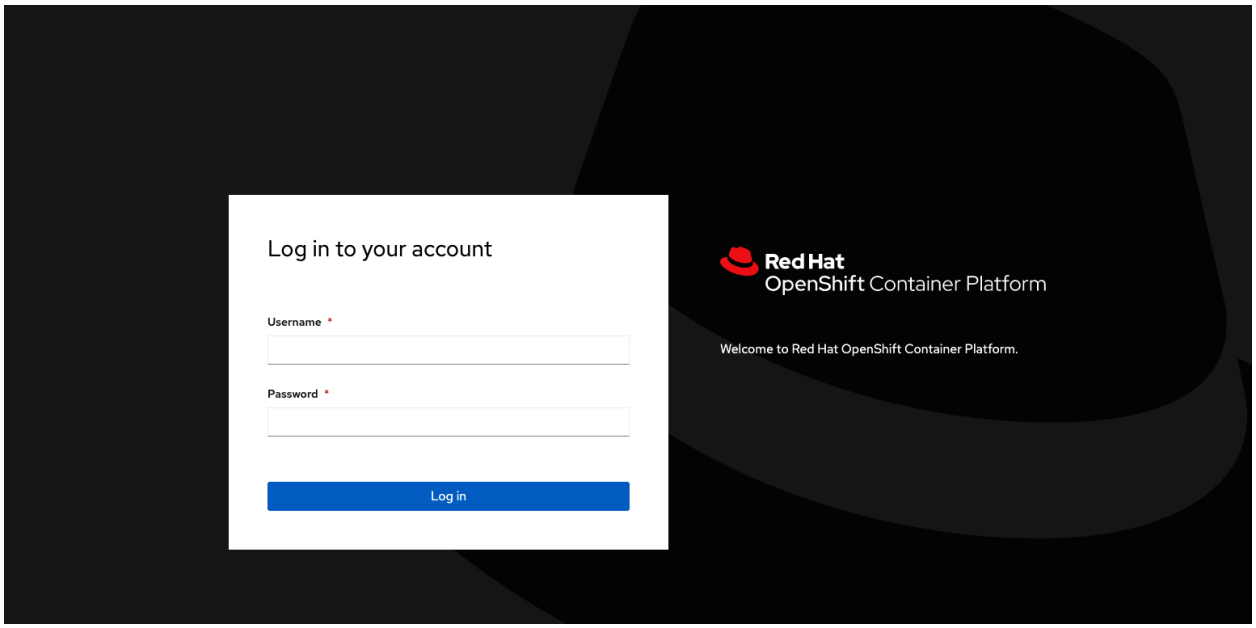
The cluster web console's URL will be listed as part of the Lab Instructions.

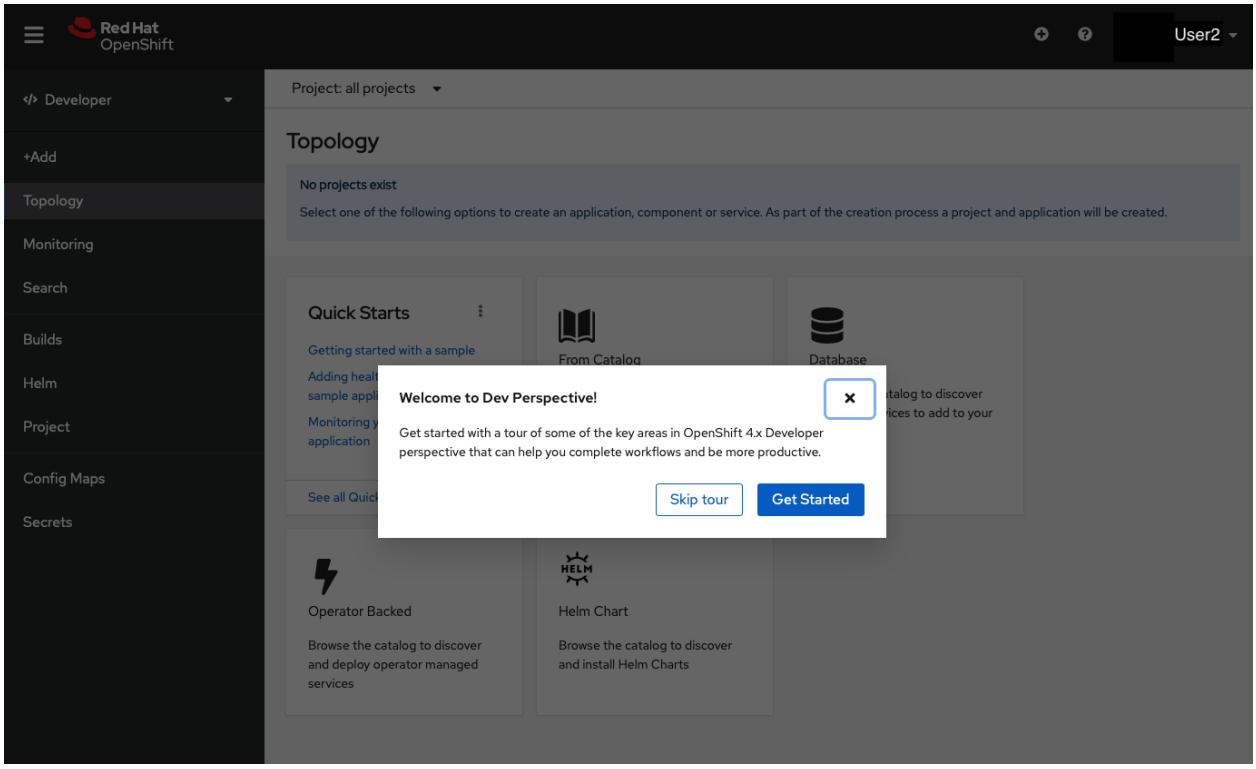
Open that link in a new browser tab and login with the `user<ID>` you have been given and password retrieved earlier.

After logging in, you should be able to see the Red Hat OpenShift Web Console

To login (we will run the example with user2 in those screenshots), do the following.

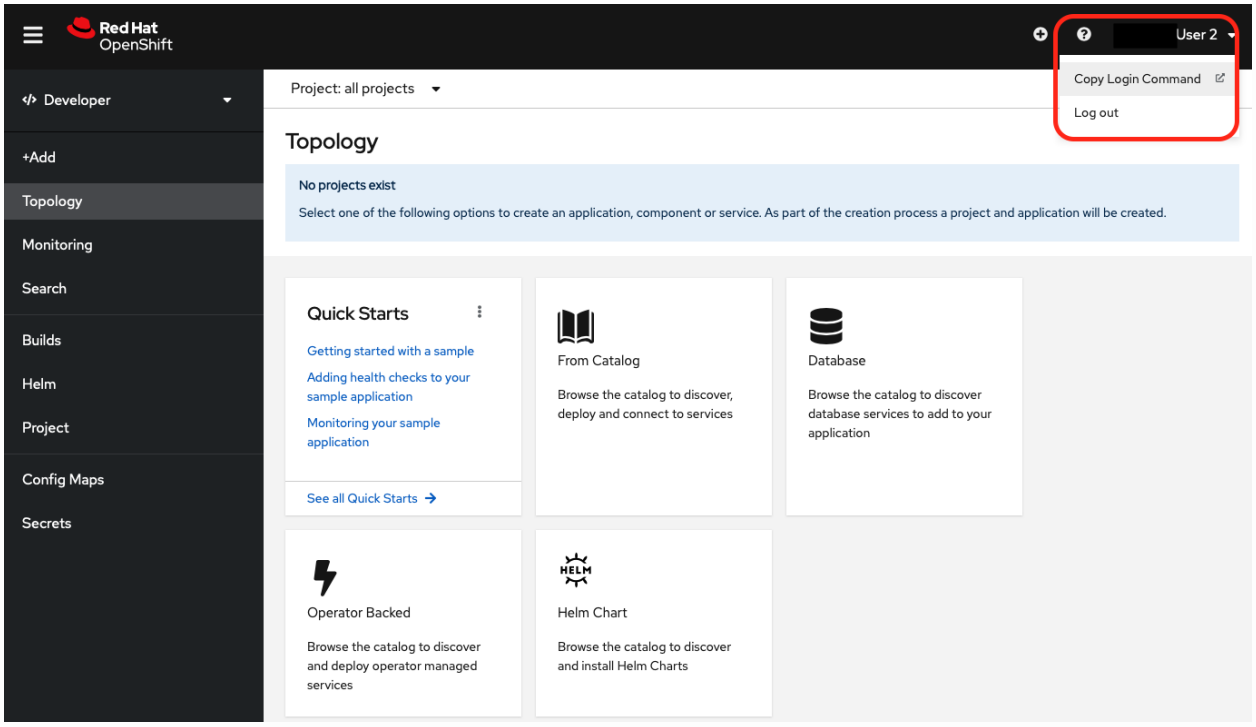
Browse to the web console's URL and enter your credentials and then you should see the OpenShift Web Console.



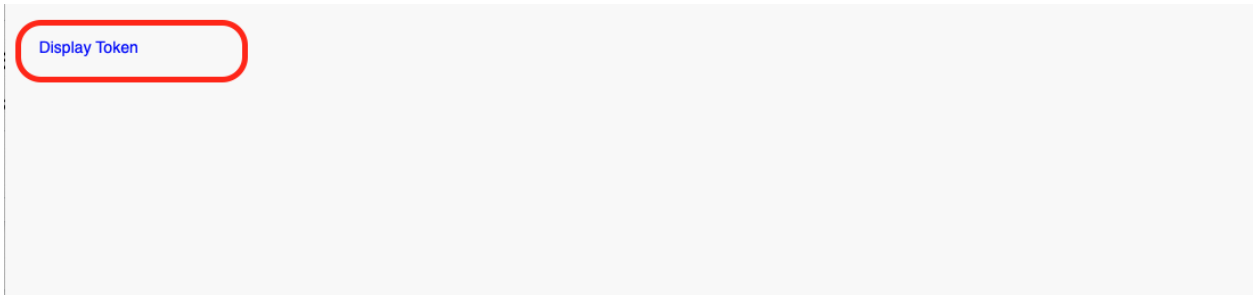


Retrieve the login command and token

Once you're logged into the Web Console, click on the username on the top right, then click **Copy login command.**



You will then need to click on the Display Token link



And you can finally copy the command highlighted below

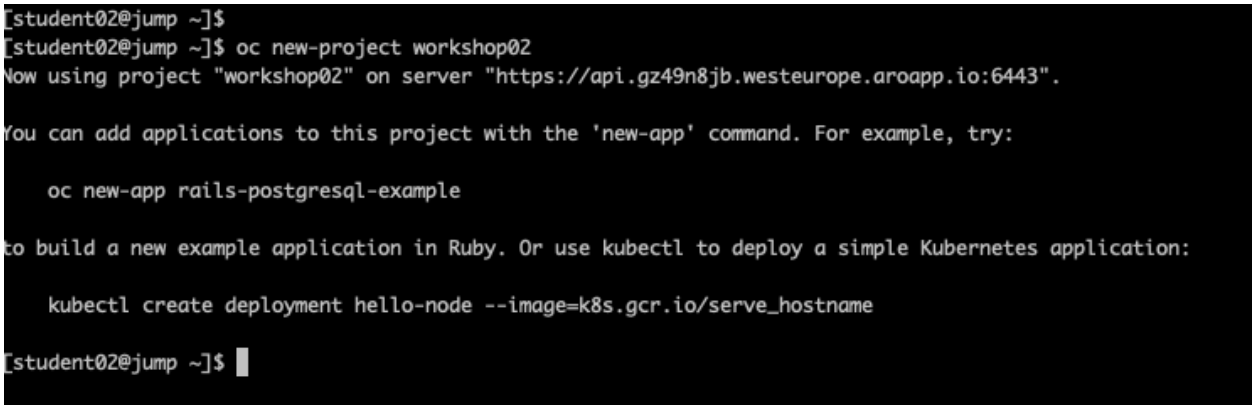
Paste this command in the Jumphost and you will be then connected to the OpenShift cluster via the CLI.

Create a project

A project allows a community of users to organize and manage their content in isolation from other communities. Type the following command in your ssh session

oc new-project workshop<userId>.

For example, if you are user 2, type in oc new-project workshop02



Deploy MongoDB

Create mongoDB from template

Red Hat OpenShift provides many container images and templates to make creating new applications & services easy.

Templates provide parameter fields to define all the mandatory environment variables (user, password, database name, etc) with predefined defaults including auto-generation of password values.

It also defines both a deployment configuration and a service.

For this exercise we will use the following template:

- `mongodb-persistent` uses a persistent volume store for the database data which means the data will survive a pod restart. Using persistent volumes requires a persistent volume pool be defined in the Red Hat OpenShift deployment.

Hint You can retrieve a list of templates using the command below. The templates are preinstalled in the `openshift` namespace.

```
oc get templates -n openshift
```

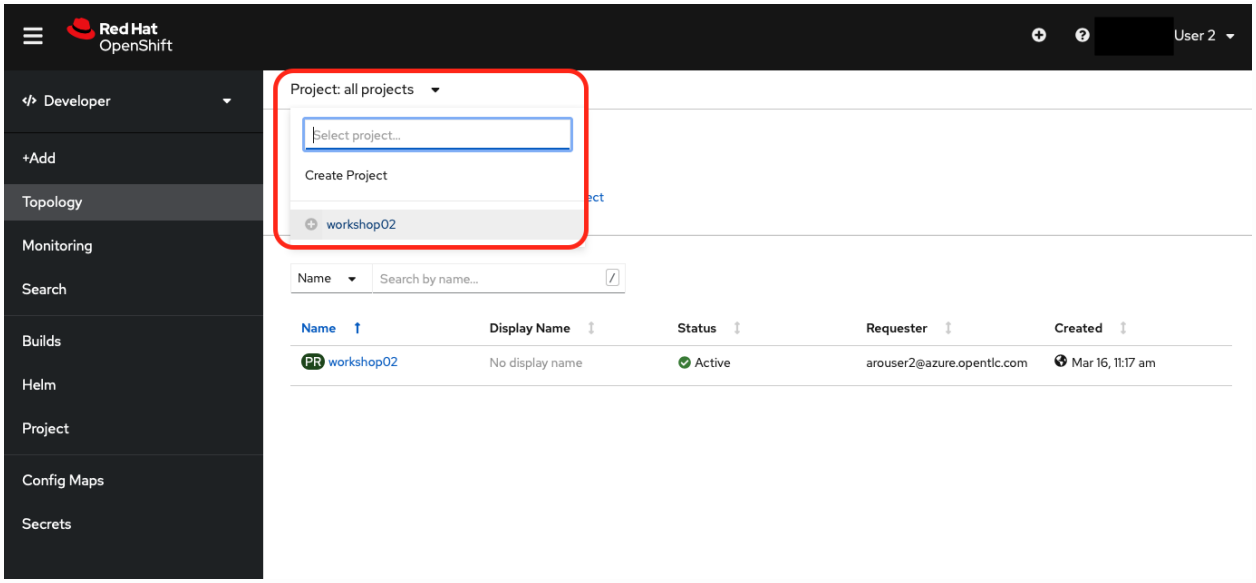
Create a mongoDB deployment using the `mongodb-persistent` template. You're passing in the values to be replaced (username, password and database) which generates a YAML/JSON file. You then pipe it to the `oc create` command.

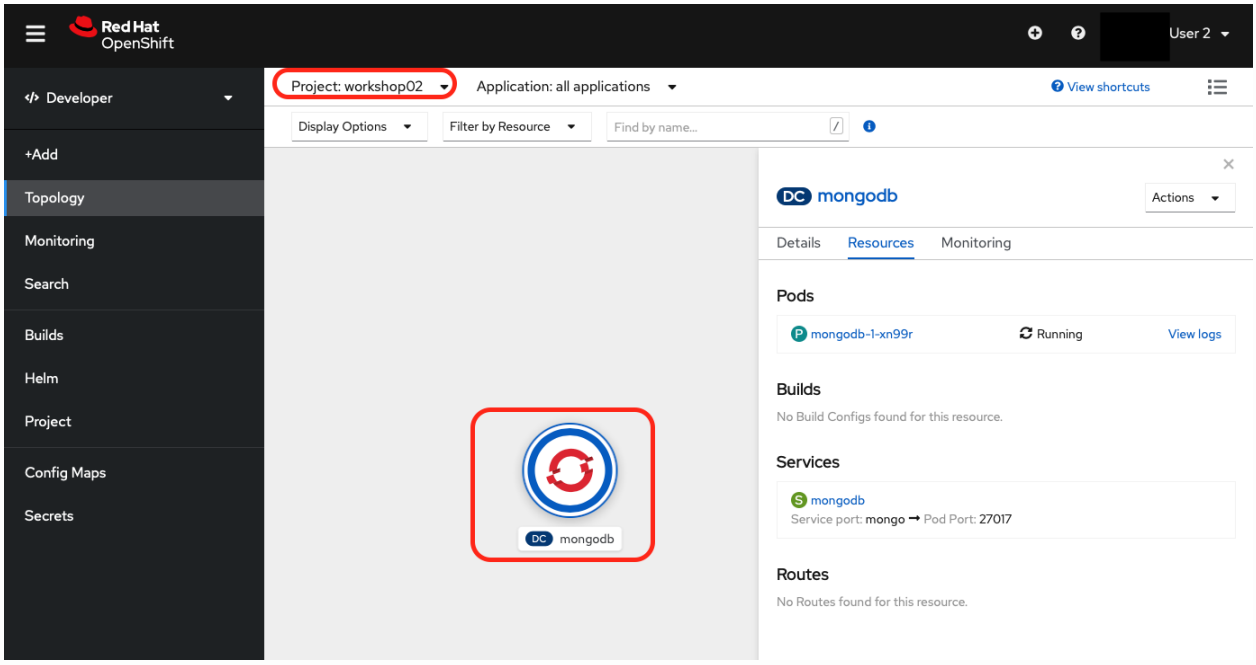
```
oc process openshift//mongodb-persistent \
    -p MONGODB_USER=ratingsuser \
    -p MONGODB_PASSWORD=ratingspassword \
    -p MONGODB_DATABASE=ratingsdb \
    -p MONGODB_ADMIN_PASSWORD=ratingspassword | oc create -f -
```

This is what you should see in your console (CLI)

```
[student02@jump ~]$
[student02@jump ~]$
[student02@jump ~]$
[student02@jump ~]$ oc process openshift//mongodb-persistent \
>      -p MONGODB_USER=ratingsuser \
>      -p MONGODB_PASSWORD=ratingspassword \
>      -p MONGODB_DATABASE=ratingsdb \
>      -p MONGODB_ADMIN_PASSWORD=ratingspassword | oc create -f -
secret/mongodb created
service/mongodb created
persistentvolumeclaim/mongodb created
deploymentconfig.apps.openshift.io/mongodb created
[student02@jump ~]$
```

If you now head back to the web console, and switch to the **workshop<userID>** project (workshop02 in this example), you should see a new deployment for mongoDB.





Verify if the mongoDB pod was created successfully

Run the `oc get all` command to view the status of the new application and verify if the deployment of the mongoDB template was successful.

```
oc get all
```

```
[student02@jump ~]$ oc get all
NAME                  READY   STATUS      RESTARTS   AGE
pod/mongodb-1-deploy  0/1     Completed   0          9m22s
pod/mongodb-1-xn99r   1/1     Running     0          9m17s

NAME                            DESIRED   CURRENT   READY   AGE
replicationcontroller/mongodb-1   1         1         1       9m22s

NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
service/mongodb   ClusterIP   172.30.48.154   <none>        27017/TCP   9m26s

NAME                                    REVISION   DESIRED   CURRENT   TRIGGERED BY
deploymentconfig.apps.openshift.io/mongodb   1          1         1         config,image(mongodb:3.6)
[student02@jump ~]$
```

Retrieve mongoDB service hostname

Find the mongoDB service.

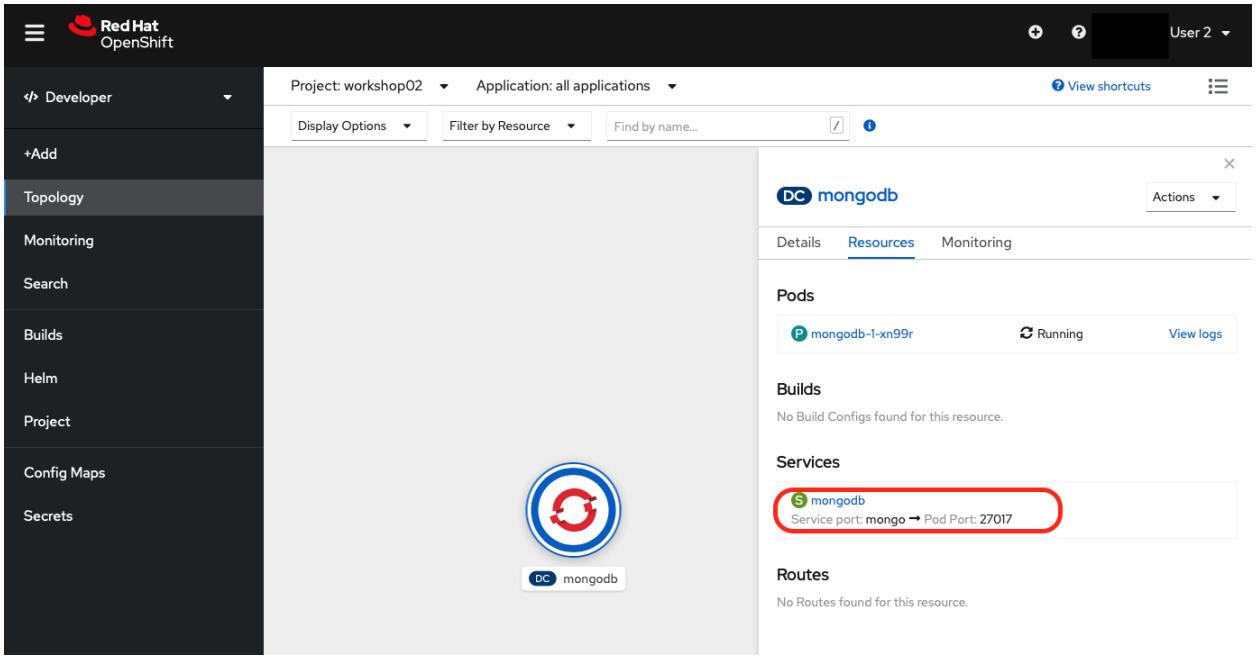
```
oc get svc mongodb
```

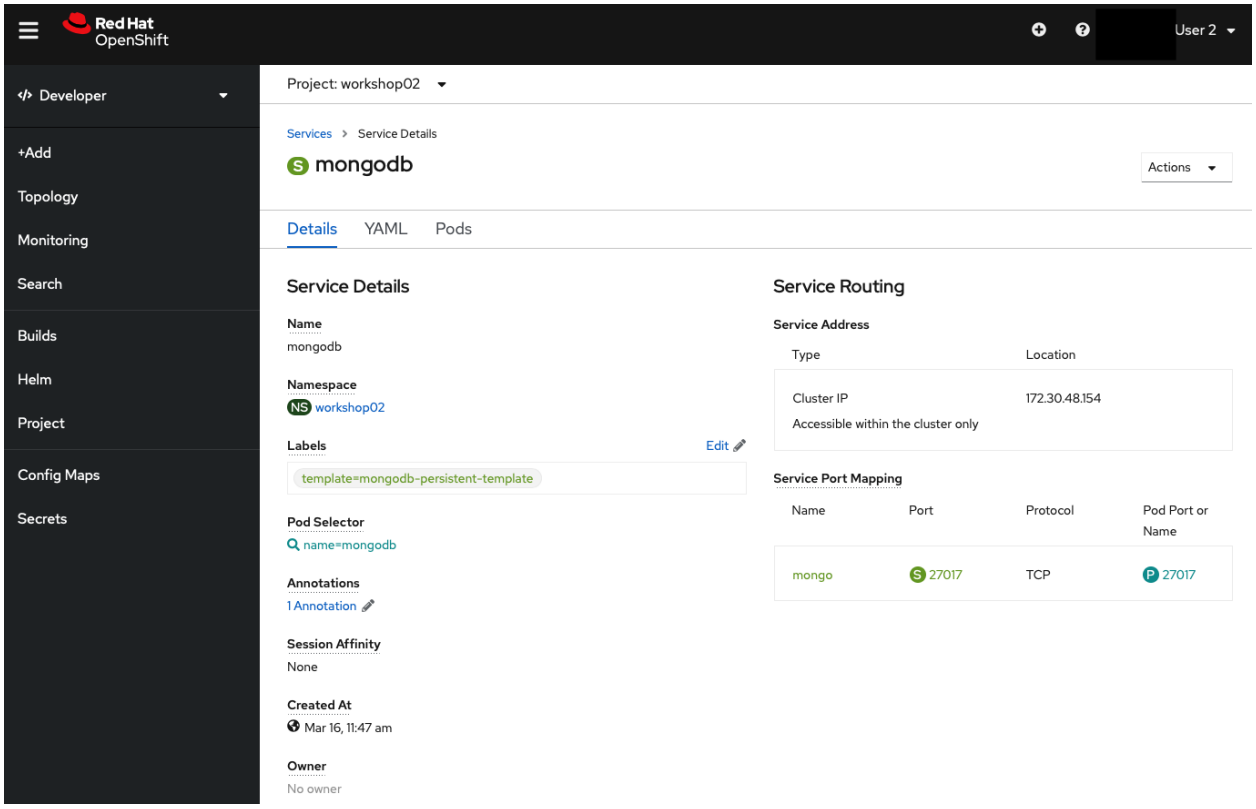
```
[student02@jump ~]$ oc get svc mongodb
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
mongodb   ClusterIP   172.30.48.154   <none>        27017/TCP   12m
[student02@jump ~]$
```

The service will be accessible at the following DNS name: `mongodb.workshop<userID>.svc.cluster.local` which is formed of `[service name].[project name].svc.cluster.local`. This resolves only within the cluster.

You can also retrieve this from the web console. To do this, click on the service link in the console and then you will see the overall service details for the mongodb service

You'll need this hostname to configure the `rating-api`.





Deploy Ratings API

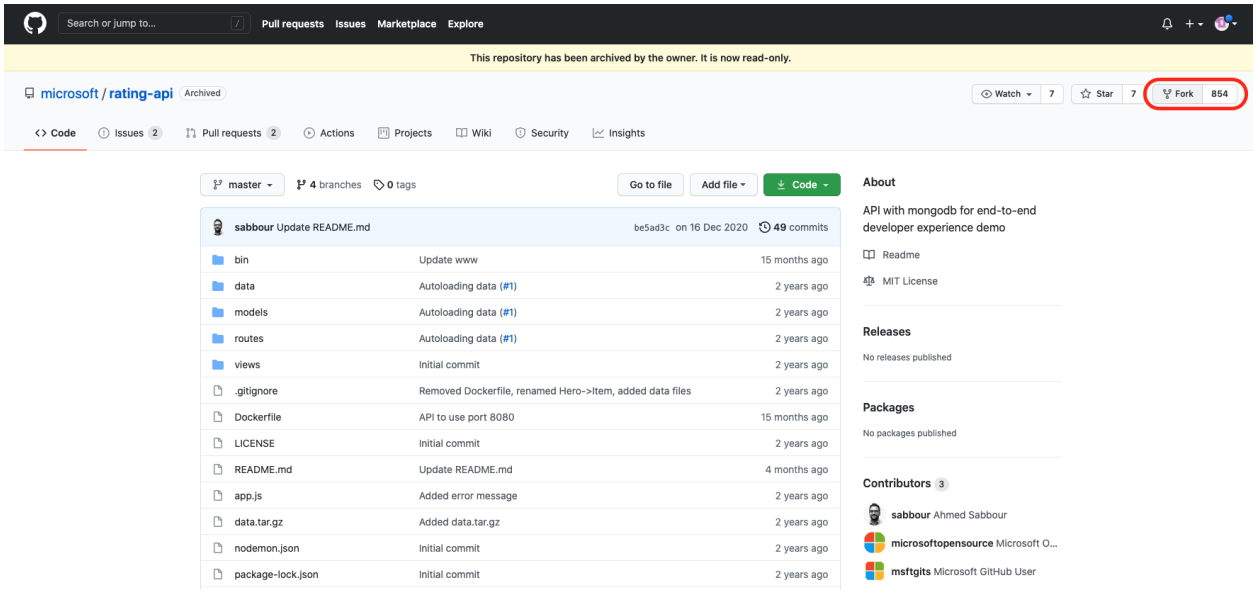
The `rating-api` is a NodeJS application that connects to mongoDB to retrieve and rate items. Below are some of the details that you'll need to deploy this.

- `rating-api` on GitHub: https://github.com/microsoft/rating-api
- The container exposes port 8080
- MongoDB connection is configured using an environment variable called `MONGODB_URI`

Fork the application to your own GitHub repository

To be able to setup CI/CD webhooks, you'll need to fork the application into your personal GitHub repository.

Go to the following gitRepo https://github.com/microsoft/rating-api/ and click on the Fork button (in the top right corner).



Use the OpenShift CLI to deploy the `rating-api`

You're going to be using source-to-image (S2I) as a build strategy.

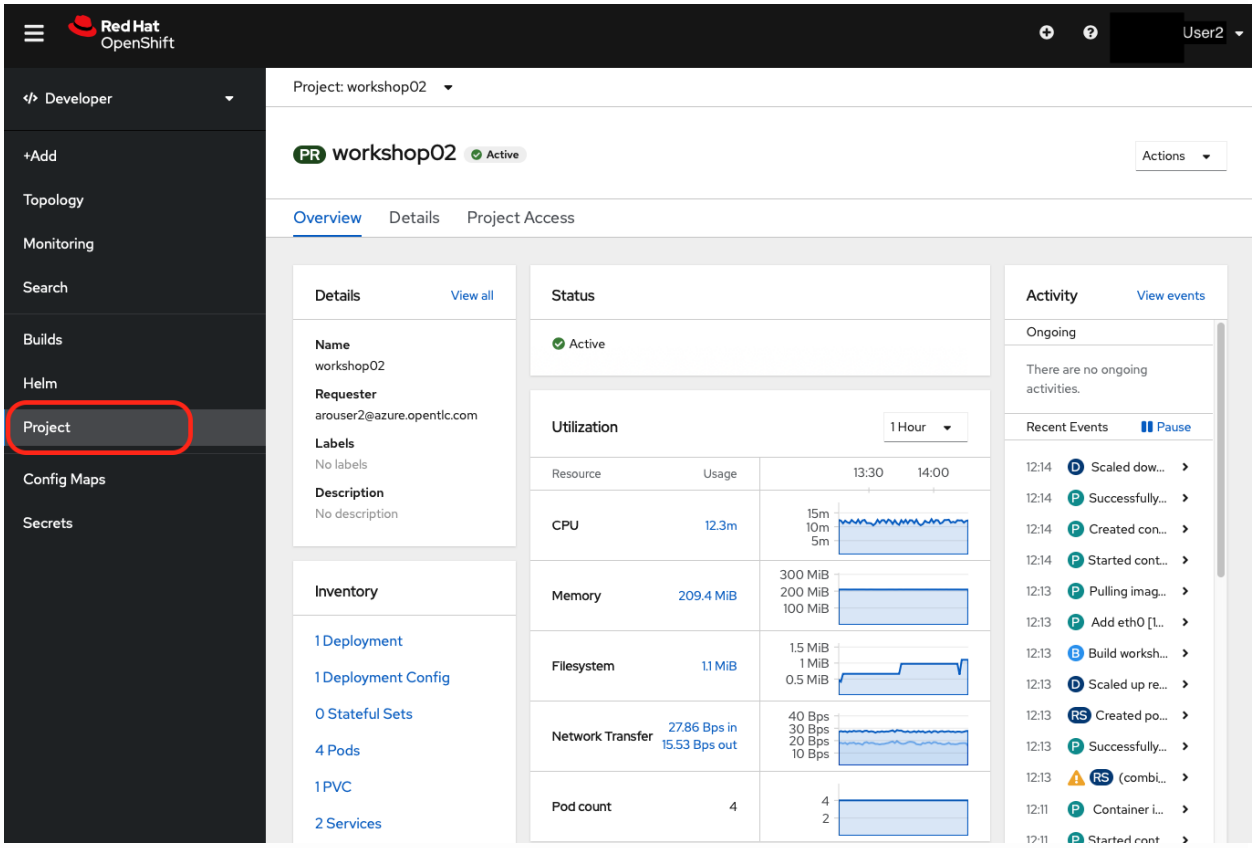
```
oc new-app https://github.com/<your GitHub username>/rating-api
--strategy=source
```

Configure the required environment variables

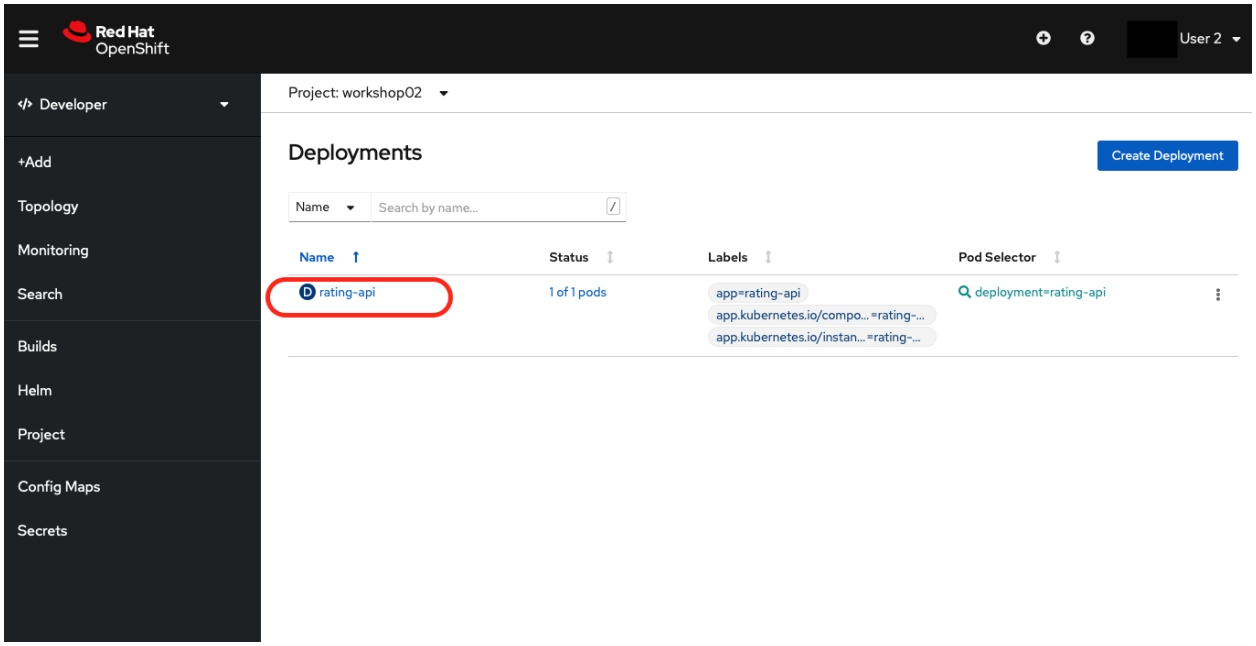
Create the `MONGODB_URI` environment variable. This URI should look like `mongodb://[username]:[password]@[endpoint]:27017/ratingsdb`. You'll need to replace the `[usernaame]` and `[password]` with the ones you used when creating the database.

You'll also need to replace the `[endpoint]` with the hostname acquired in the previous step.

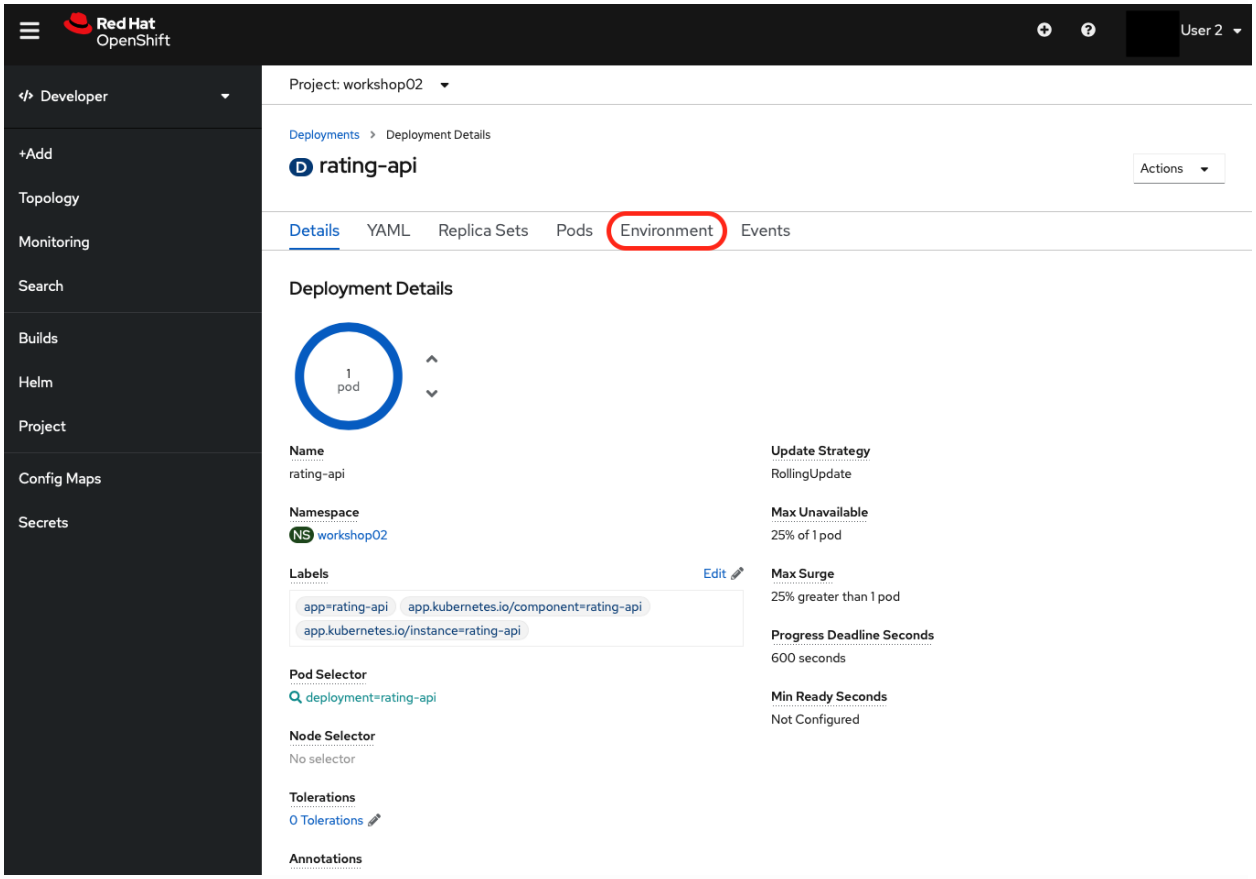
To do this, first go to the Project tab on the left and select the workshop<userID> project (workshop02 in this example).



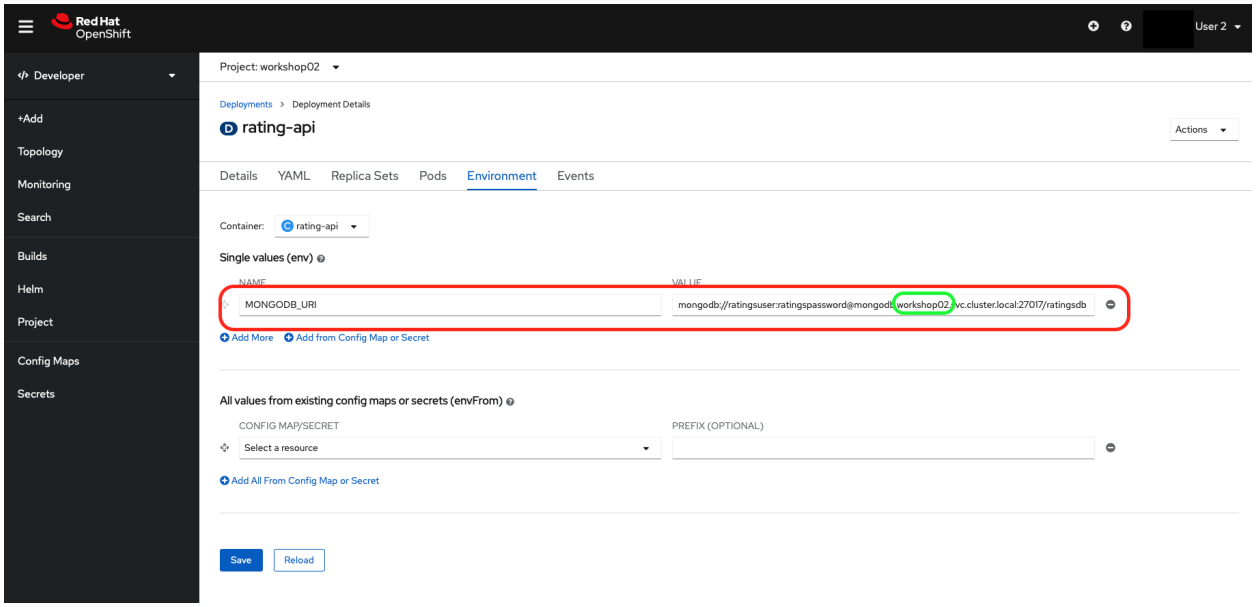
Then select the Deployment object under inventory and click on it.



Click on the rating-api object and then browse to the Environment tab



Finally enter the relevant values and hit Save when done.

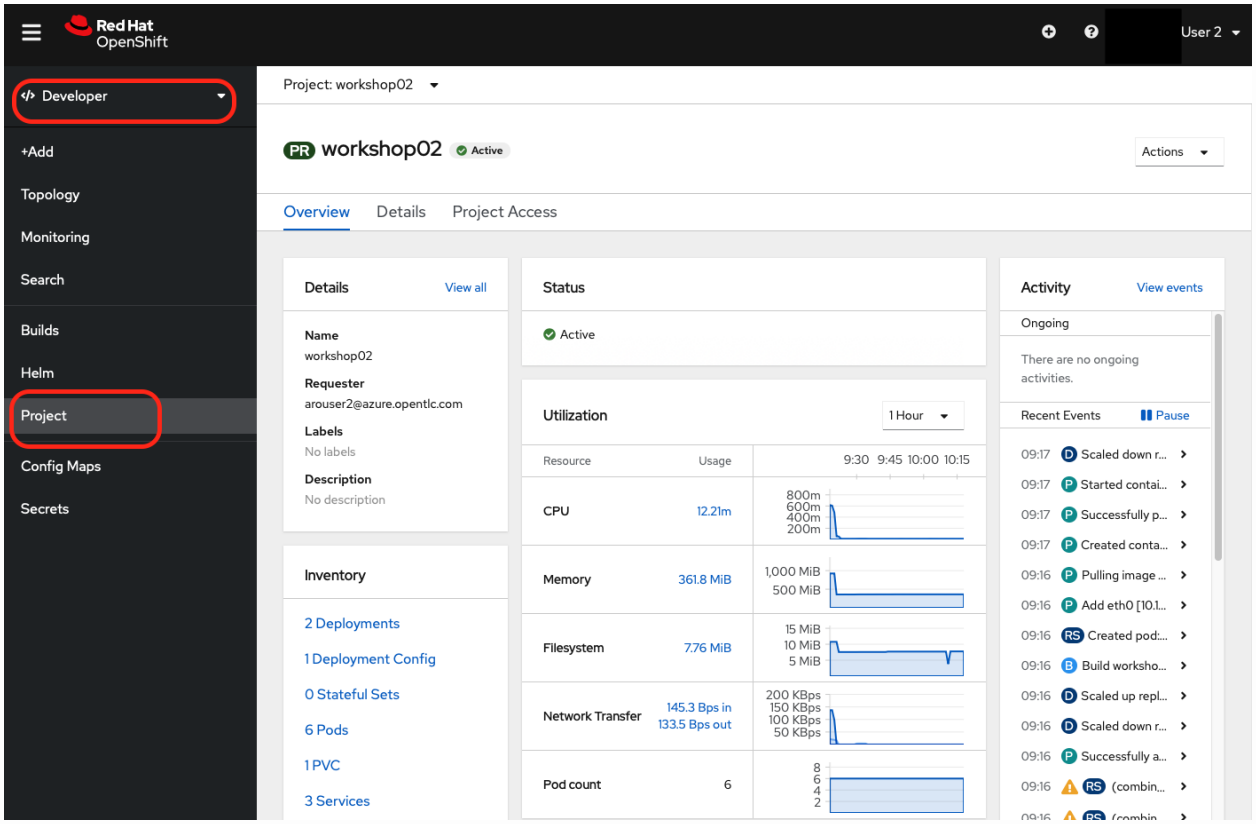


It can also be done with CLI

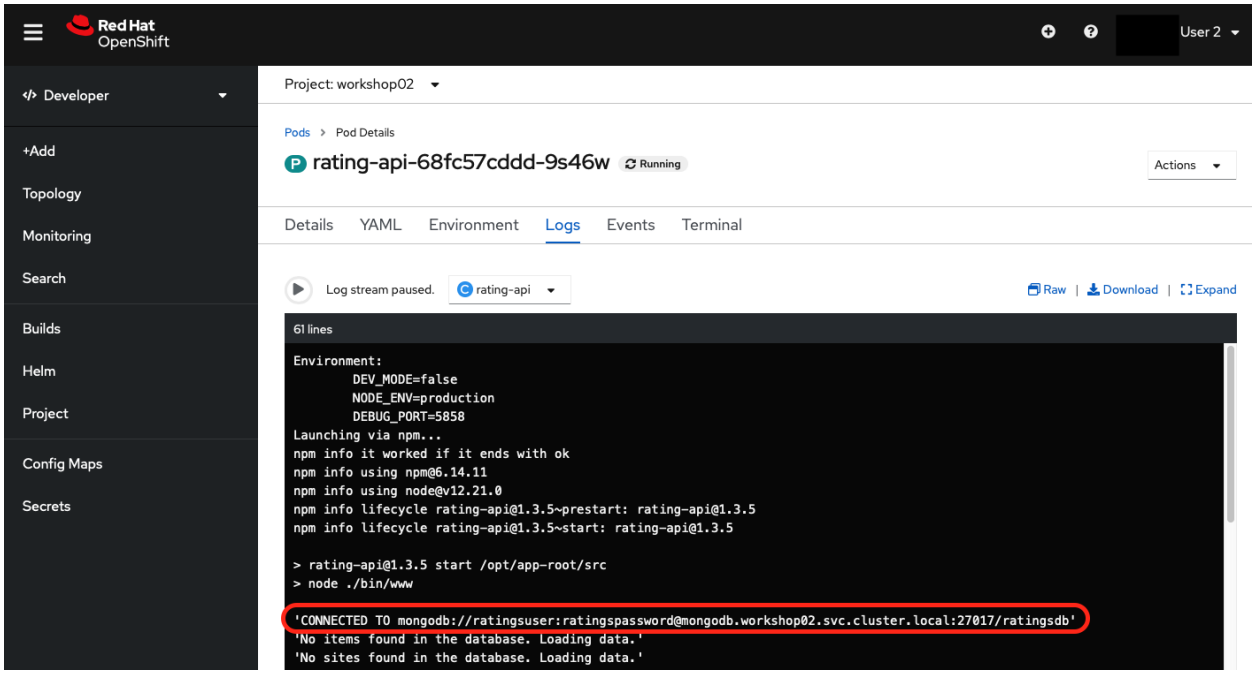
```
oc set env deploy/rating-api
MONGODB_URI=mongodb://ratingsuser:ratingspassword@mongodb.workshop<userID>.svc.
cluster.local:27017/ratingsdb
```

Verify that the service is running

If you navigate (by clicking on the Developer view, then to the project tab on the left, then onto PoDs and select the rating-api - see screenshot below)



to the logs of the `rating-api` deployment (by clicking on PoDs within the Project view), you should see a log message confirming the code can successfully connect to the mongoDB. For that, in the deployment's details screen, click on *Pods* tab, then on one of the pods



Retrieve `rating-api` service hostname

Find the `rating-api` service.

```
oc get svc rating-api
```

The service will be accessible at the following DNS name over port 8080: `ating-api.workshop<userId>.svc.cluster.local:8080` which is formed of `[service name].[project name].svc.cluster.local`. This resolves only within the cluster.

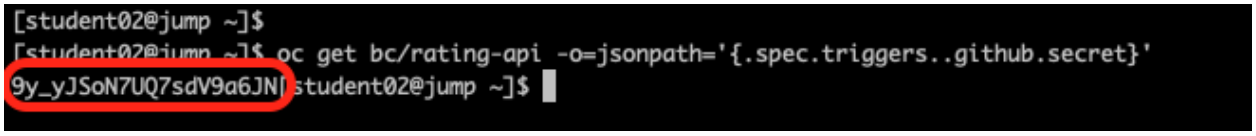
Setup GitHub webhook

To trigger S2I builds when you push code into your GitHib repo, you'll need to setup the GitHub webhook.

Retrieve the GitHub webhook trigger secret. You'll need use this secret in the GitHub webhook URL.

```
oc get bc/rating-api -o=jsonpath='{.spec.triggers..github.secret}'
```

You'll get back something similar to the below. Make note the secret key in the red box as you'll need it in a few steps.



Retrieve the GitHub webhook trigger URL from the build configuration.

```
oc describe bc/rating-api
```

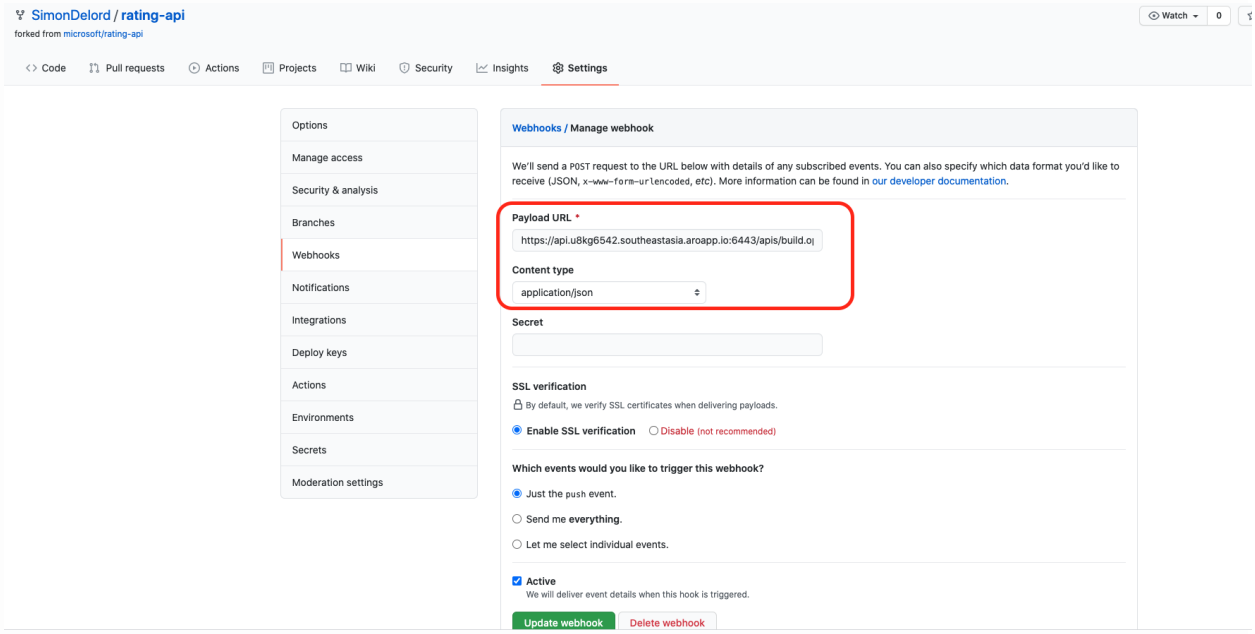
Replace the `<secret>` placeholder with the secret you retrieved in the previous step to have a URL similar to `https://api.u8kg6542.southeastasia.aroapp.io:6443/apis/build.openshift.io/v1/namespaces/workshop02/buildconfigs/rating-api/webhooks/SECRETSTRING/github`. You'll use this URL to setup the webhook on your GitHub repository.

In your GitHub repository, select Add Webhook from Settings → Webhooks.

Paste the URL output (similar to above) into the Payload URL field.

Change the Content Type from GitHub's default application/x-www-form-urlencoded to application/json.

Click Add webhook.



You should see a message from GitHub stating that your webhook was successfully configured.

Now, whenever you push a change to your GitHub repository, a new build will automatically start, and upon a successful build a new deployment will start.

Deploy Ratings frontend

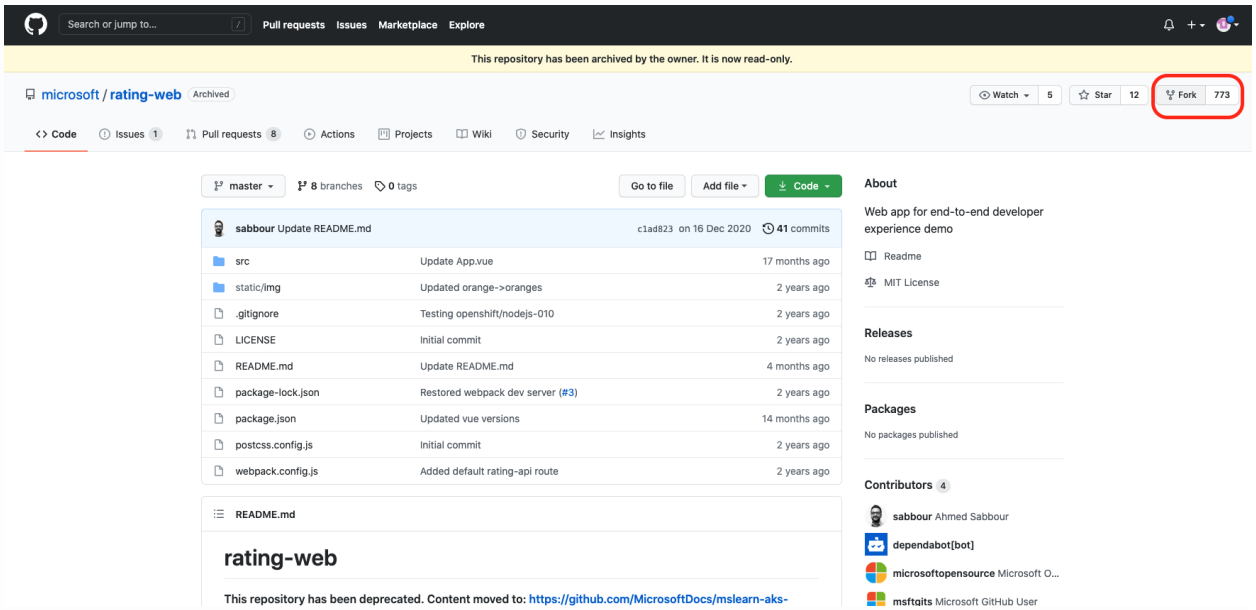
The `rating-web` is a NodeJS application that connects to the `rating-api`. Below are some of the details that you'll need to deploy this.

- `rating-web` on GitHub: https://github.com/microsoft/rating-web
- The container exposes port 8080
- The web app connects to the API over the internal cluster DNS, using a proxy through an environment variable named `API`

Fork the application to your own GitHub repository

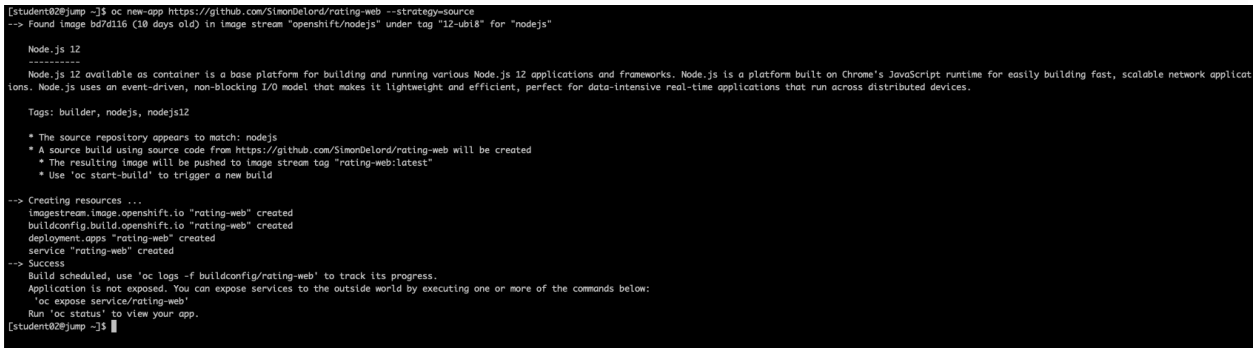
To be able to setup CI/CD webhooks, you'll need to fork the application into your personal GitHub repository.

Go to the following gitRepo https://github.com/microsoft/rating-web/ and click on the Fork button (in the top right corner).



Use the OpenShift CLI to deploy the `rating-web`

```
oc new-app https://github.com/<your GitHub username>/rating-web
--strategy=source
```



Configure the required environment variables

Create the `API` environment variable for `rating-web` Deployment Config. The value of this variable is going to be the hostname/port of the `rating-api` service.

Instead of setting the environment variable through the Red Hat OpenShift Web Console, you can set it through the OpenShift CLI.

```
oc set env deploy rating-web API=http://rating-api:8080
```

Expose the `rating-web` service using a Route

Expose the service.

```
oc expose svc/rating-web
```

Find out the created route hostname

```
oc get route rating-web
```

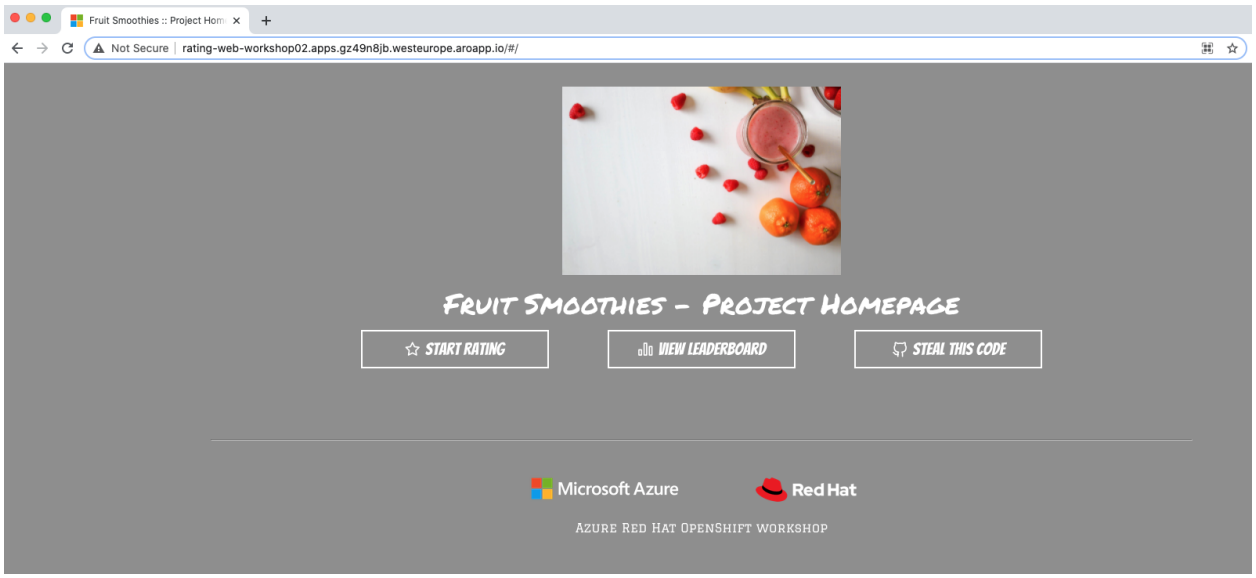
You should get a response similar to the below.

```
[student02@jump ~]$
[student02@jump ~]$ oc expose svc/rating-web
route.route.openshift.io/rating-web exposed
[student02@jump ~]$
[student02@jump ~]$ oc get route rating-web
NAME         HOST/PORT                                                PATH   SERVICES     PORT       TERMINATION   WILDCARD
rating-web   rating-web-workshop02.apps.gz49n8jb.westeurope.aroapp.io         rating-web   8080-tcp                 None
[student02@jump ~]$
```

Notice the fully qualified domain name (FQDN) is comprised of the application name and project name by default. The remainder of the FQDN, the subdomain, is your Red Hat OpenShift cluster specific apps subdomain.

Try the service

Open the hostname in your browser, you should see the rating app page. Play around, submit a few votes and check the leaderboard.



Setup GitHub webhook

To trigger S2I builds when you push code into your GitHib repo, you'll need to setup the GitHub webhook.

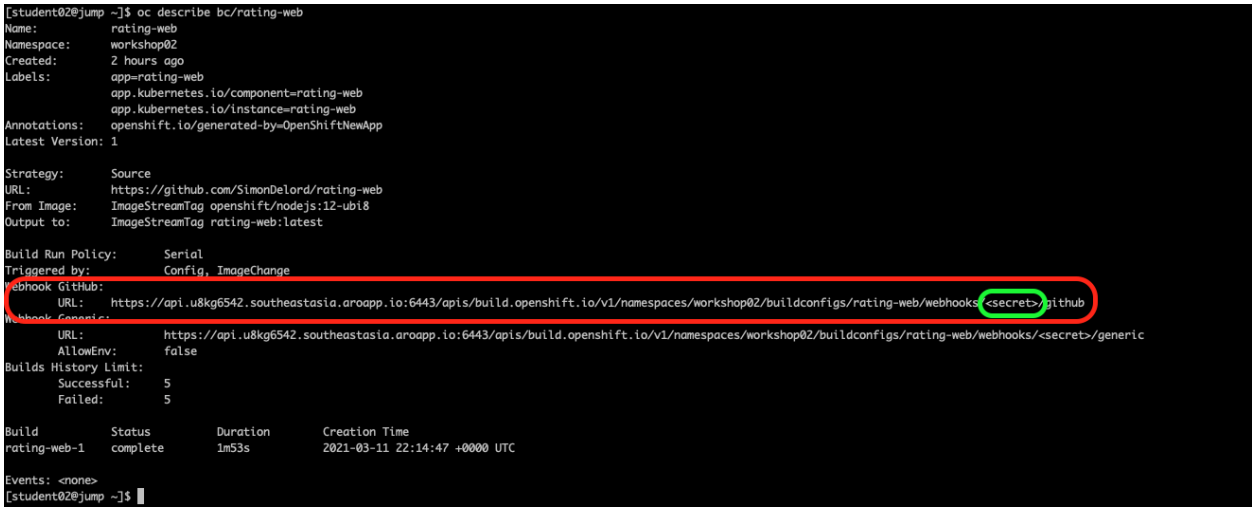
Retrieve the GitHub webhook trigger secret. You'll need use this secret in the GitHub webhook URL.

```
oc get bc/rating-web -o=jsonpath='{.spec.triggers..github.secret}'
```

You'll get back something similar to the below. Make note the secret key in the red box as you'll need it in a few steps.

Retrieve the GitHub webhook trigger URL from the build configuration.

```
oc describe bc/rating-web
```



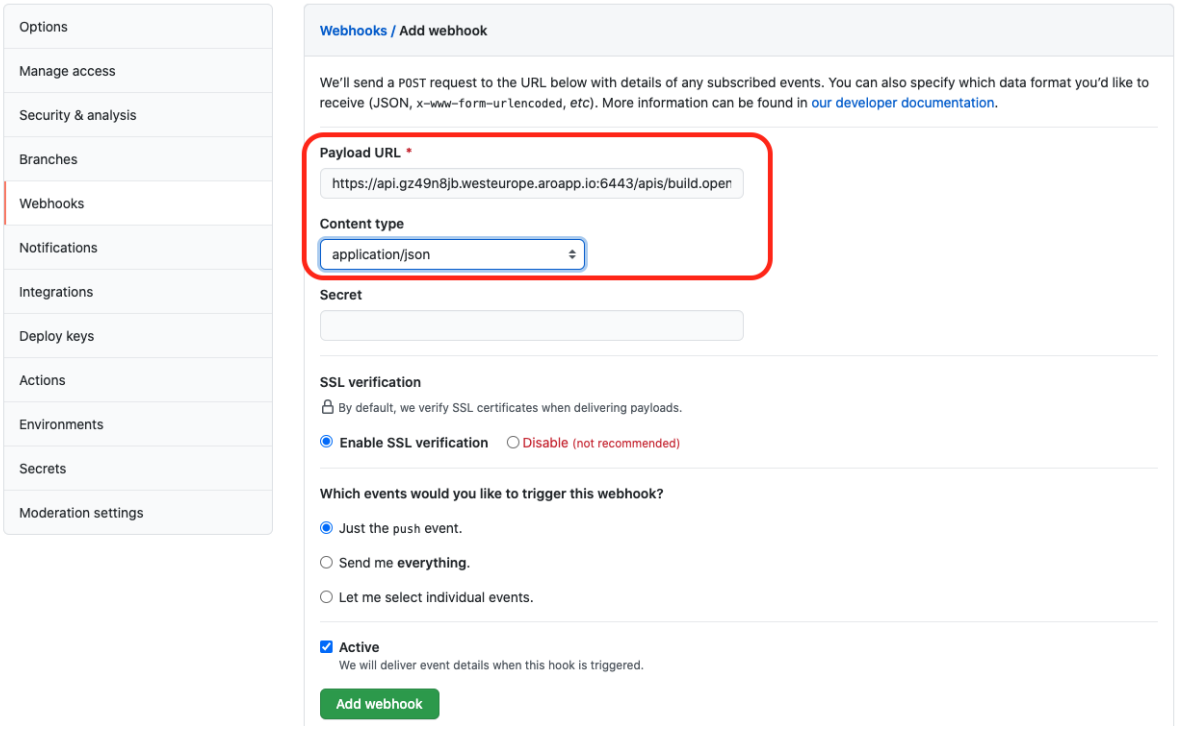
Replace the `<secret>` placeholder with the secret you retrieved in the previous step to have a URL similar to `https://api.gz49n8jb.westeurope.aroapp.io:6443/apis/build.openshift.io/v1/namespaces/workshop/buildconfigs/rating-web/webhooks/SECRETSTRING/github`. You'll use this URL to setup the webhook on your GitHub repository.

In your GitHub repository, select Add Webhook from Settings → Webhooks.

Paste the URL output (similar to above) into the Payload URL field.

Change the Content Type from GitHub's default application/x-www-form-urlencoded to application/json.

Click Add webhook.



You should see a message from GitHub stating that your webhook was successfully configured.

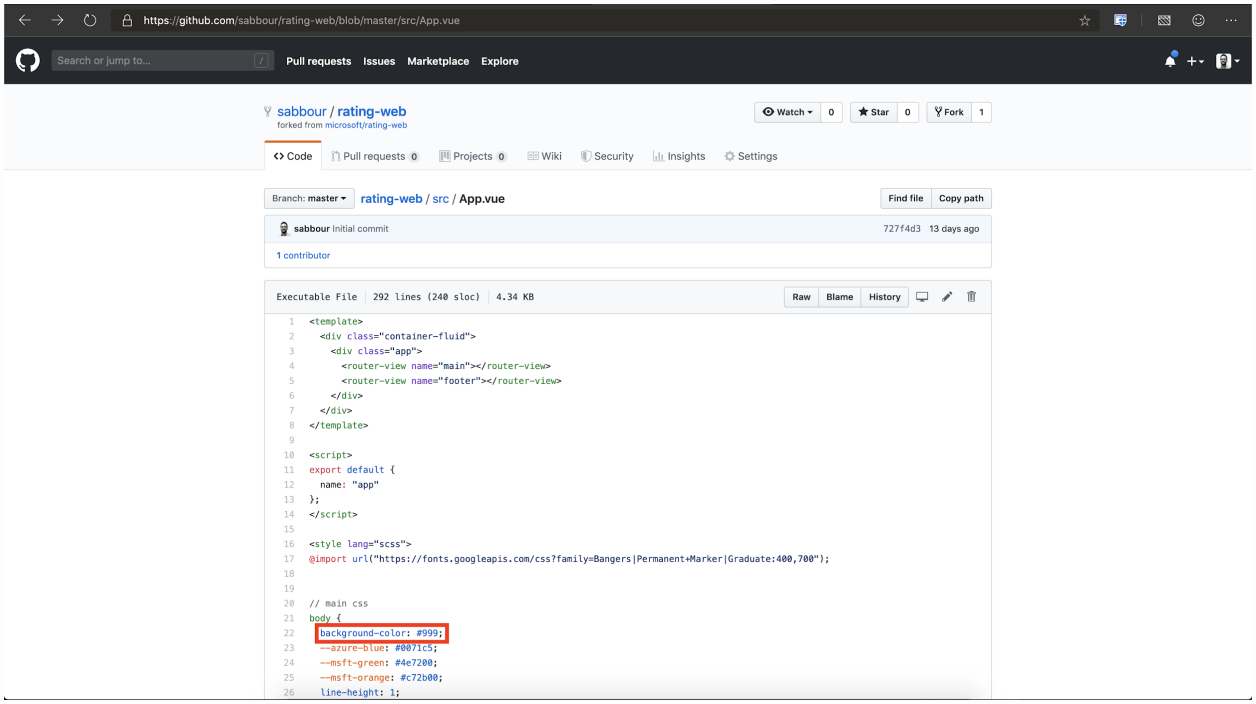
Now, whenever you push a change to your GitHub repository, a new build will automatically start, and upon a successful build a new deployment will start.

Make a change to the website app and see the rolling update

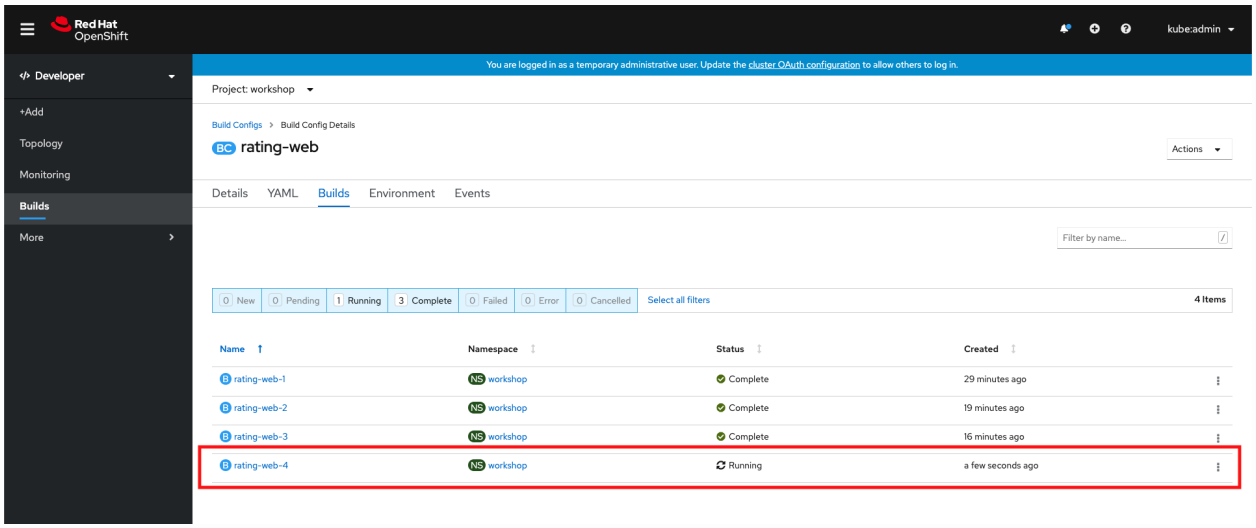
Go to the `https://github.com/<your GitHub username>/rating-web/blob/master/src/App.vue` file in your repository on GitHub.

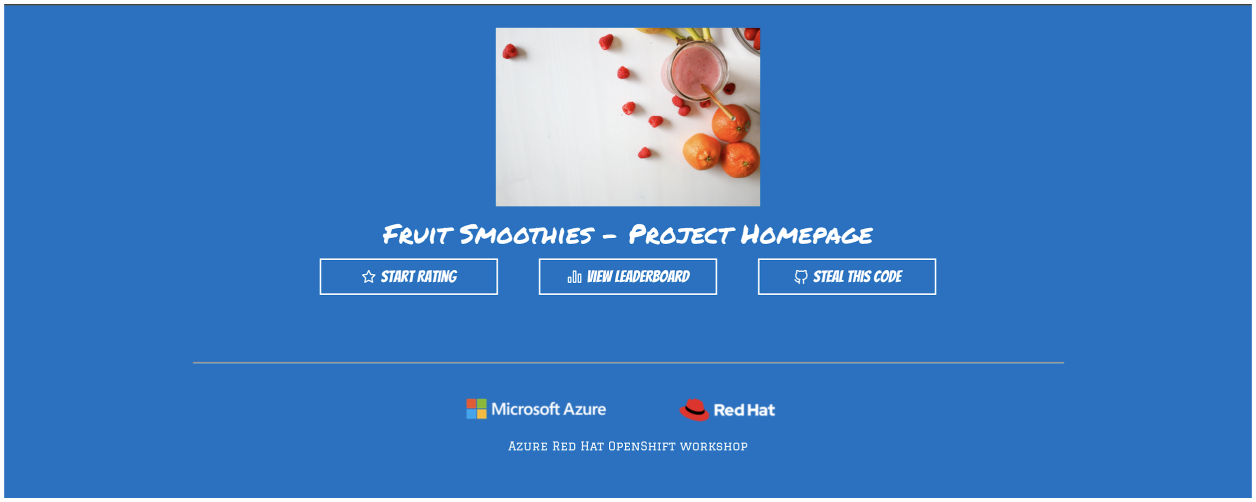
Edit the file, and change the `background-color: #999;` line to be `background-color: #0071c5`.

Commit the changes to the file into the `master` branch.



Immediately, go to the Builds tab in the OpenShift Web Console. You'll see a new build queued up which was triggered by the push. Once this is done, it will trigger a new deployment and you should see the new website color updated.



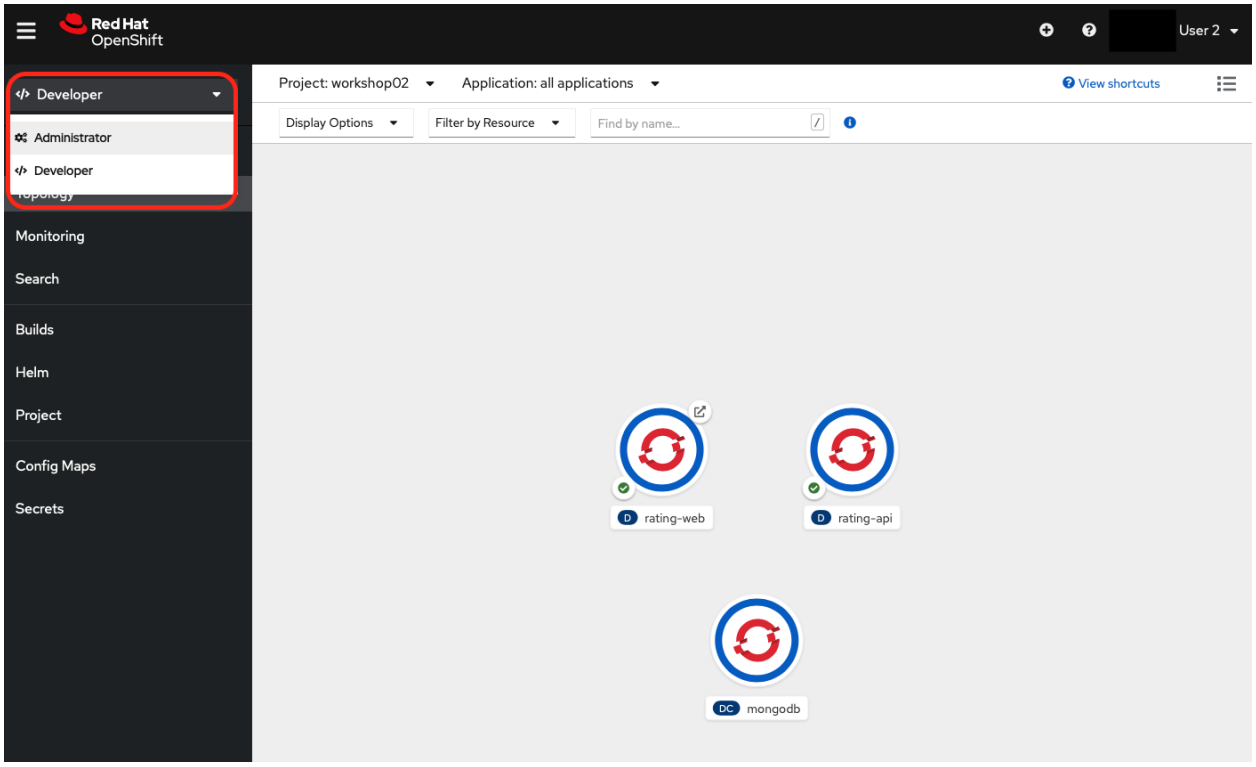


Create Network Policy

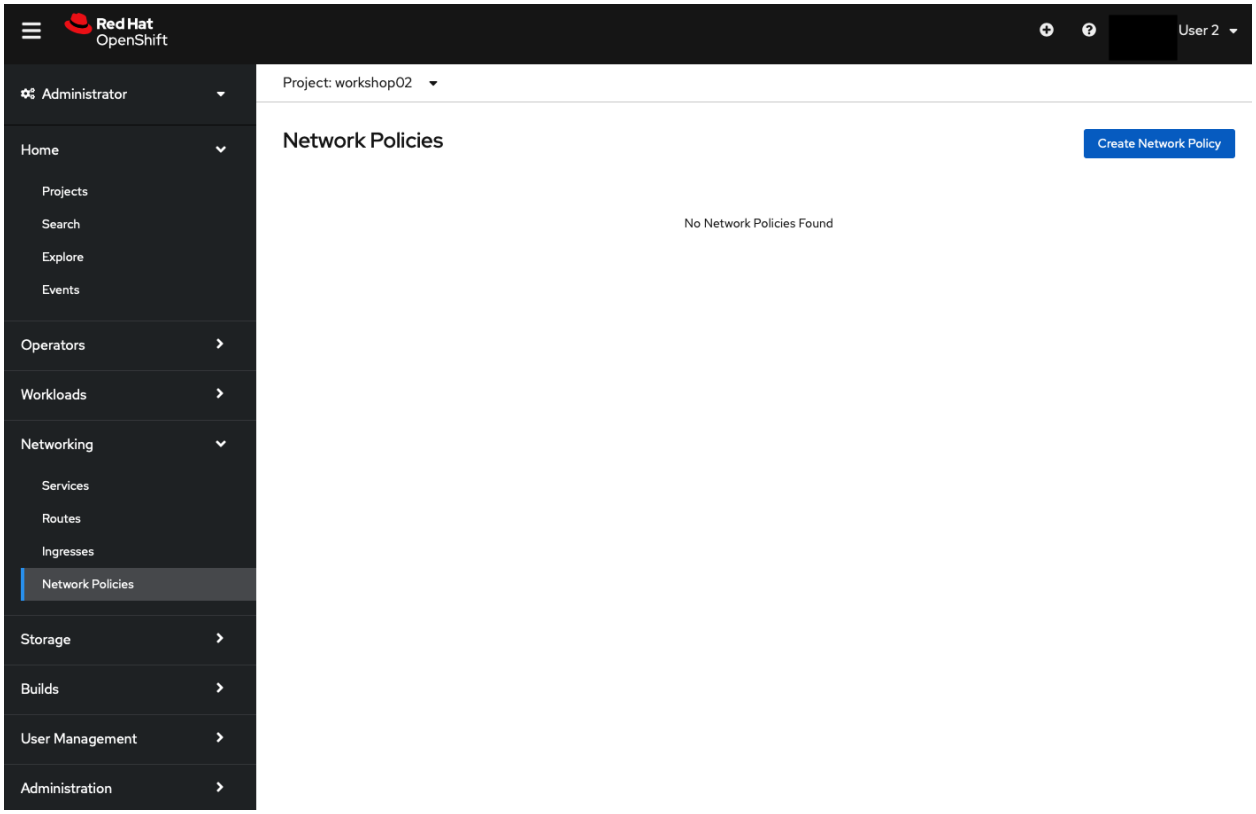
Now that you have the application working, it is time to apply some security hardening. You'll use network policies to restrict communication to the `rating-api`.

Switch to the Cluster Console

Switch to the Administrator console.



Make sure you're in the workshop project, expand Networking and click Create Network Policy.



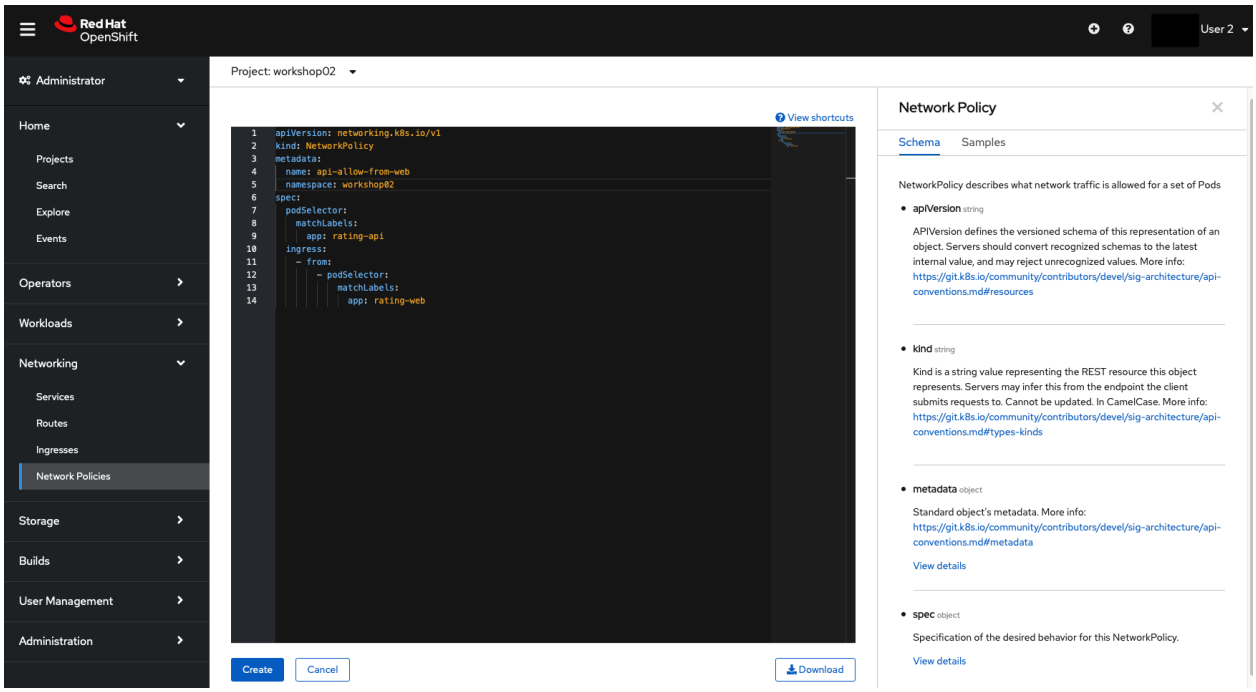
Create network policy

You will create a policy that applies to any pod matching the `app=rating-api` label. The policy will allow ingress only from pods matching the `app=rating-web` label.

Use the YAML below in the editor, and make sure you're targeting the workshop<userID> project.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-allow-from-web
  namespace: workshop<userID>
spec:
```

```
  podSelector:
    matchLabels:
      app: rating-api
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: rating-web
```



Lab 2 - OpenShift Internals

Application Overview

Resources

- The source code for this app is available here: https://github.com/openshift-cs/ostoy
- OSToy front-end container image: https://quay.io/repository/ostoylab/ostoy-frontend?tab=tags
- OSToy microservice container image: https://quay.io/repository/ostoylab/ostoy-microservice?tab=tags
- Deployment Definition YAMLs:
    - ostoy-fe-deployment.yaml
    - ostoy-microservice-deployment.yaml

Note In order to simplify the deployment of the app (which you will do next) we have included all the objects needed in the above YAMLs as “all-in-one” YAMLs. In reality though, an enterprise would most likely want to have a different yaml file for each Kubernetes object.

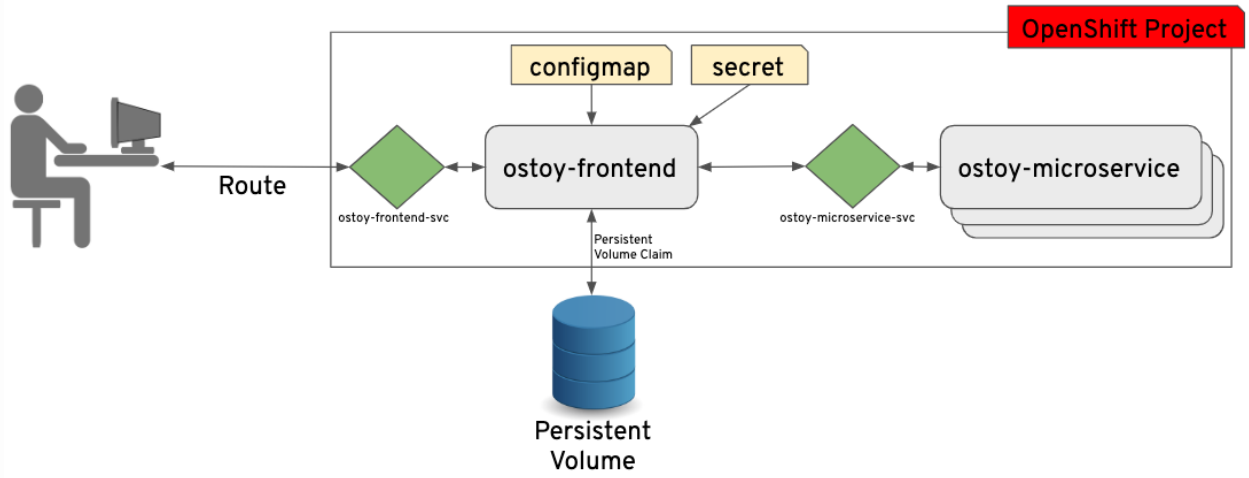
About OSToy

OSToy is a simple Node.js application that we will deploy to Red Hat OpenShift. It is used to help us explore the functionality of Kubernetes. This application has a user interface which you can:

- write messages to the log (stdout / stderr)
- intentionally crash the application to view self-healing
- toggle a liveliness probe and monitor OpenShift behavior
- read config maps, secrets, and env variables
- if connected to shared storage, read and write files
- check network connectivity, intra-cluster DNS, and intra-communication with an included microservice

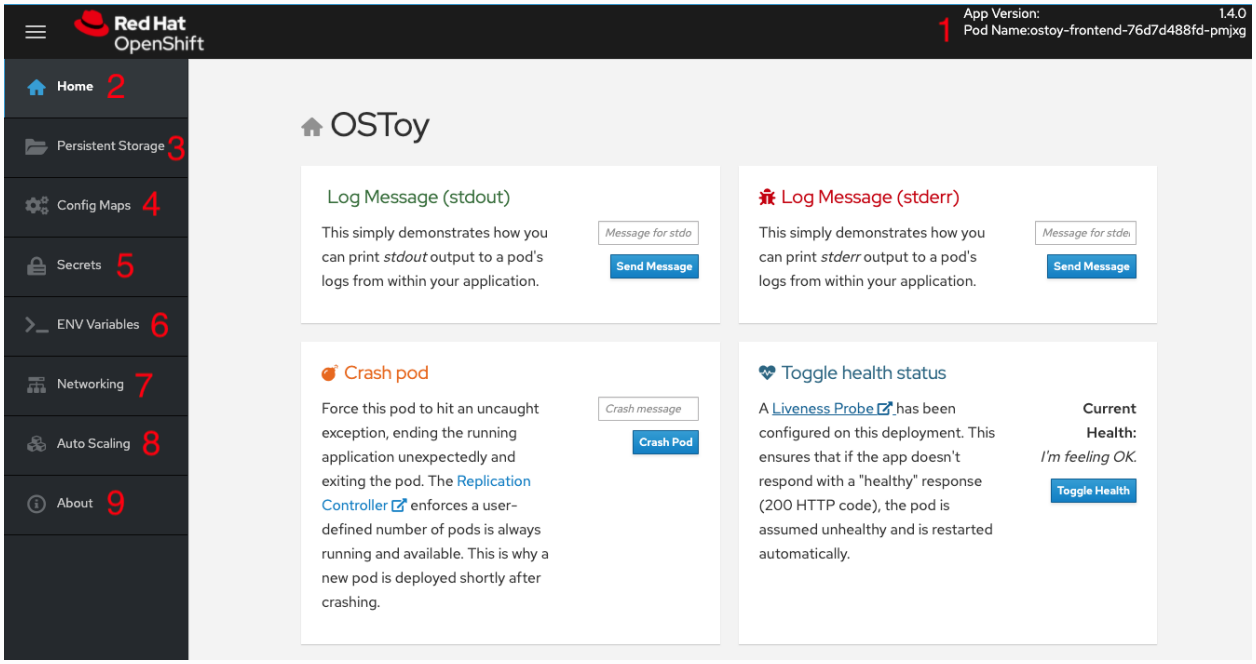
- increase the load to view automatic scaling of the pods to handle the load (via the Horizontal Pod Autoscaler)

OSToy Application Diagram



Familiarization with the Application UI

1. Shows the pod name that served your browser the page.
2. Home: The main page of the application where you can perform some of the functions listed which we will explore.
3. Persistent Storage: Allows us to write data to the persistent volume bound to this application.
4. Config Maps: Shows the contents of configmaps available to the application and the key:value pairs.
5. Secrets: Shows the contents of secrets available to the application and the key:value pairs.
6. ENV Variables: Shows the environment variables available to the application.
7. Auto Scaling: Explore the Horizontal Pod Autoscaler to see how increased loads are handled.
8. Networking: Tools to illustrate networking within the application.
9. Shows some more information about the application.



Application Deployment

Create new project

Create a new project called “OSToy<userID>” in your cluster. In the rest of this document we will use the OSToy02 project.

Use the following command

```
oc new-project ostoy02
```

You should receive the following response

```
[student02@jump ~]$ oc new-project ostoy02

Now using project "ostoy02" on server
"https://api.gz49n8jb.westeurope.aroapp.io:6443".
```

```
You can add applications to this project with the 'new-app' command. For
example, try:

    oc new-app rails-postgresql-example

to build a new example application in Ruby. Or use kubectl to deploy a simple
Kubernetes application:

    kubectl create deployment hello-node --image=k8s.gcr.io/serve_hostname
```

Equivalently you can also create this new project using the web UI by selecting *Home* > *Projects* on the left menu, then clicking on “Create Project” button on the left.

View the YAML deployment objects

View the Kubernetes deployment object yamls. If you wish you can download them from the following locations to your Shell, in a directory of your choosing (just remember where you placed them for the next step). Or just use the direct link in the next step.

Feel free to open them up and take a look at what we will be deploying. For simplicity of this lab we have placed all the Kubernetes objects we are deploying in one “all-in-one” yaml file. Though in reality there are benefits to separating these out into individual yaml files.

ostoy-fe-deployment.yaml

Ostoy-microservice-deployment.yaml

Deploy backend microservice

The microservice application serves internal web requests and returns a JSON object containing the current hostname and a randomly generated color string.

In your command line deploy the microservice using the following command:

```
oc apply -f
https://raw.githubusercontent.com/microsoft/aroworkshop/master/yaml/ostoy-micro
service-deployment.yaml
```

You should see the following response:

```
$ oc apply -f
https://raw.githubusercontent.com/microsoft/aroworkshop/master/yaml/ostoy-micro
service-deployment.yaml

deployment.apps/ostoy-microservice created

service/ostoy-microservice-svc created
```

Deploy the front-end service

The frontend deployment contains the node.js frontend for our application along with a few other Kubernetes objects to illustrate examples.

If you open the *ostoy-fe-deployment.yaml* you will see we are defining:

- Persistent Volume Claim
- Deployment Object
- Service
- Route
- Configmaps
- Secrets

In your command line deploy the frontend along with creating all objects mentioned above by entering:

```
oc apply -f
https://raw.githubusercontent.com/microsoft/aroworkshop/master/yaml/ostoy-fe-de
ployment.yaml
```

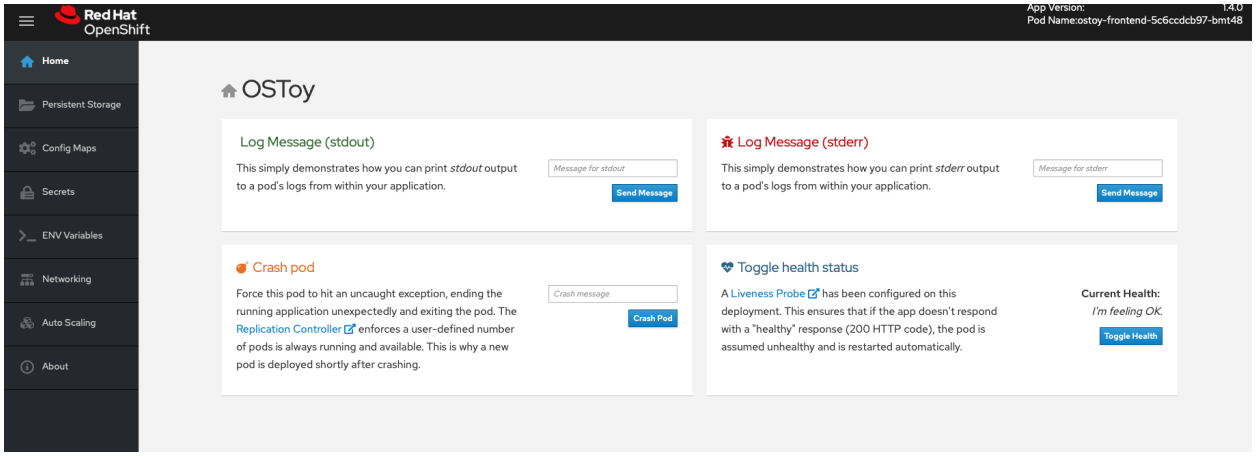
You should see all objects created successfully

```
$ oc apply -f
https://raw.githubusercontent.com/microsoft/aroworkshop/master/yaml/ostoy-fe-de
ployment.yaml

persistentvolumeclaim/ostoy-pvc created

deployment.apps/ostoy-frontend created

service/ostoy-frontend-svc created

route.route.openshift.io/ostoy-route created

configmap/ostoy-configmap-env created

secret/ostoy-secret-env created

configmap/ostoy-configmap-files created

secret/ostoy-secret created
```

Get route

Get the route so that we can access the application via `oc get route`

Copy `ostoy-route-ostoy02.apps.gz49n8jb.westeurope.aroapp.io` above and paste it into your browser and press enter. You should see the homepage of our application.

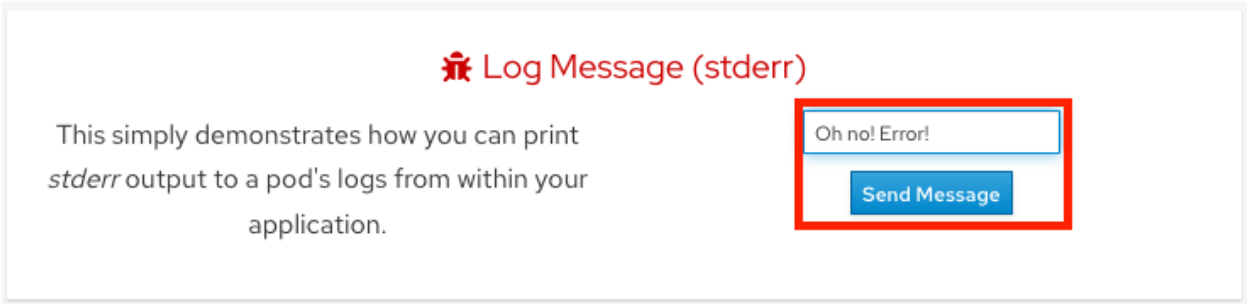


Logging

Assuming you can access the application via the Route provided and are still logged into the CLI (please go back to part 2 if you need to do any of those) we'll start to use this application. As stated earlier, this application will allow you to "push the buttons" of OpenShift and see how it works. We will do this to test the logs.

Click on the *Home* menu item and then click in the message box for "Log Message (stdout)" and write any message you want to output to the *stdout* stream. You can try "All is well!". Then click "Send Message".

Click in the message box for "Log Message (stderr)" and write any message you want to output to the *stderr* stream. You can try "Oh no! Error!". Then click "Send Message".



View logs directly from the pod

Go to the CLI and enter the following command to retrieve the name of your frontend pod which we will use to view the pod logs:

```
$ oc get pods -o name
```

```
pod/ostoy-frontend-679cb85695-5cn7x
```

```
pod/ostoy-microservice-86b4c6f559-p594d
```

So the pod name in this case is ostoy-frontend-679cb85695-5cn7x. Then run `oc logs ostoy-frontend-679cb85695-5cn7x` and you should see your messages:

```
$ oc logs ostoy-frontend-679cb85695-5cn7x
```

```
[...]
```

```
ostoy-frontend-679cb85695-5cn7x: server starting on port 8080
```

```
Redirecting to /home
```

```
stdout: All is well!
```

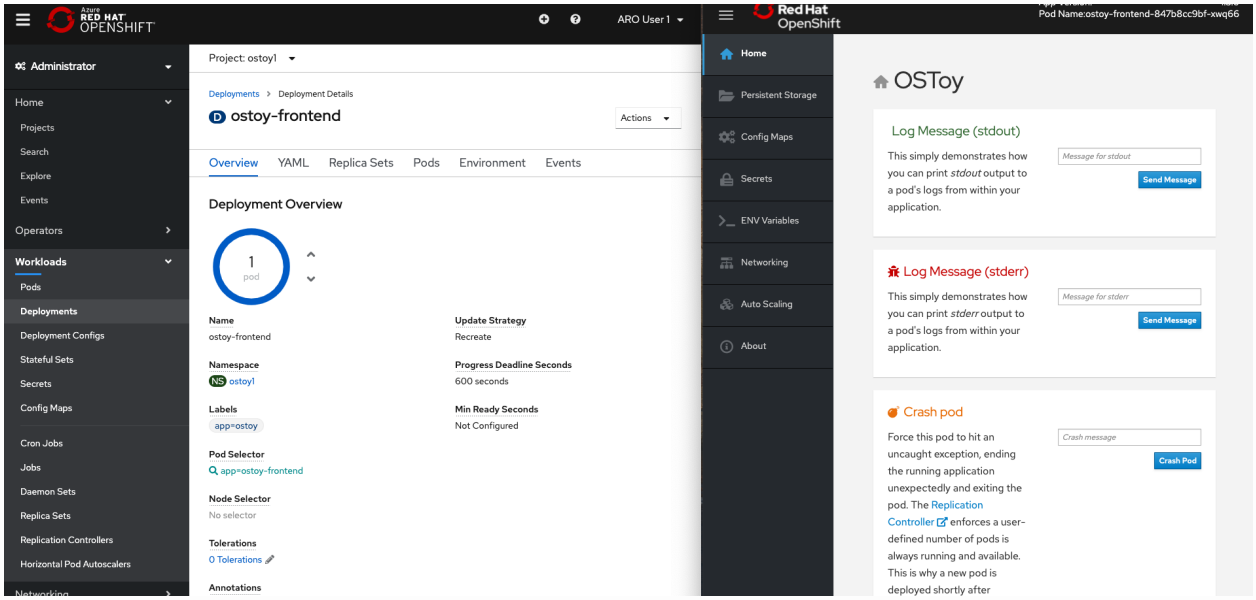
```
stderr: Oh no! Error!
```

You should see both the *stdout* and *stderr* messages.

Exploring Health Checks

In this section we will intentionally crash our pods as well as make a pod non-responsive to the liveness probes and see how Kubernetes behaves. We will first intentionally crash our pod and see that Kubernetes will self-heal by immediately spinning it back up. Then we will trigger the health check by stopping the response on the `/health` endpoint in our app. After three consecutive failures, Kubernetes should kill the pod and then recreate it.

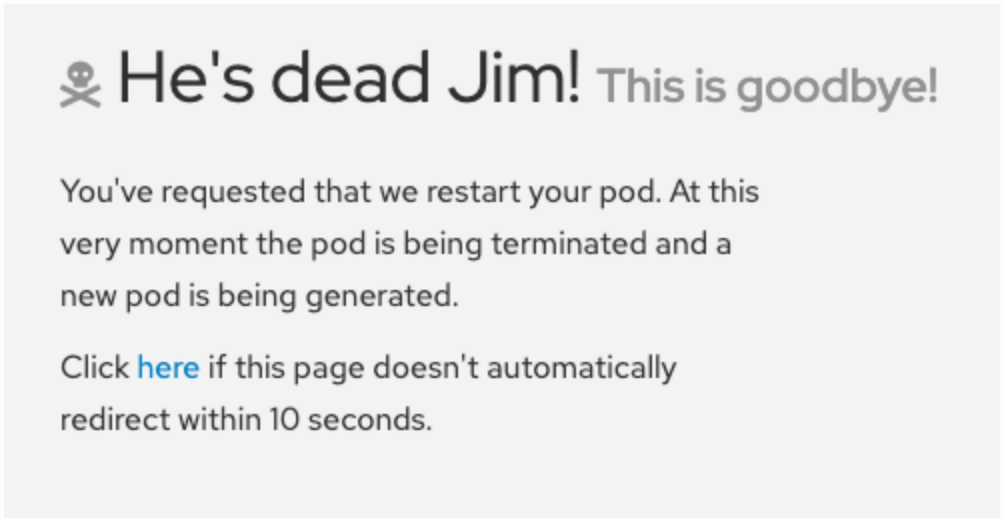
It would be best to prepare by splitting your screen between the OpenShift Web UI and the OSToy application so that you can see the results of our actions immediately.



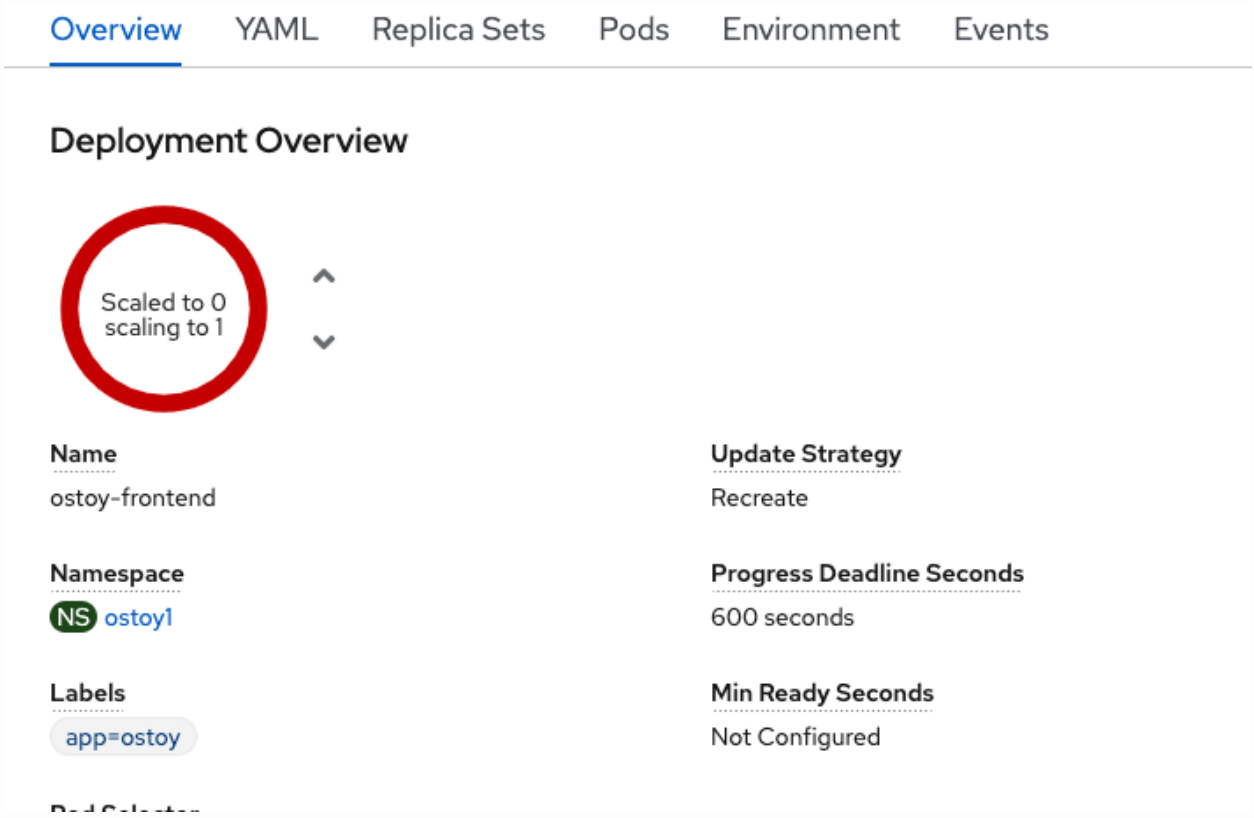
But if your screen is too small or that just won't work, then open the OSToy application in another tab so you can quickly switch to the OpenShift Web Console once you click the button. To get to this deployment in the OpenShift Web Console go to the left menu and click:

Workloads > Deployments > "ostoy-frontend"

Go to the browser tab that has your OSToy app, click on *Home* in the left menu, and enter a message in the "Crash Pod" tile (e.g.: "This is goodbye!") and press the "Crash Pod" button. This will cause the pod to crash and Kubernetes should restart the pod. After you press the button you will see:

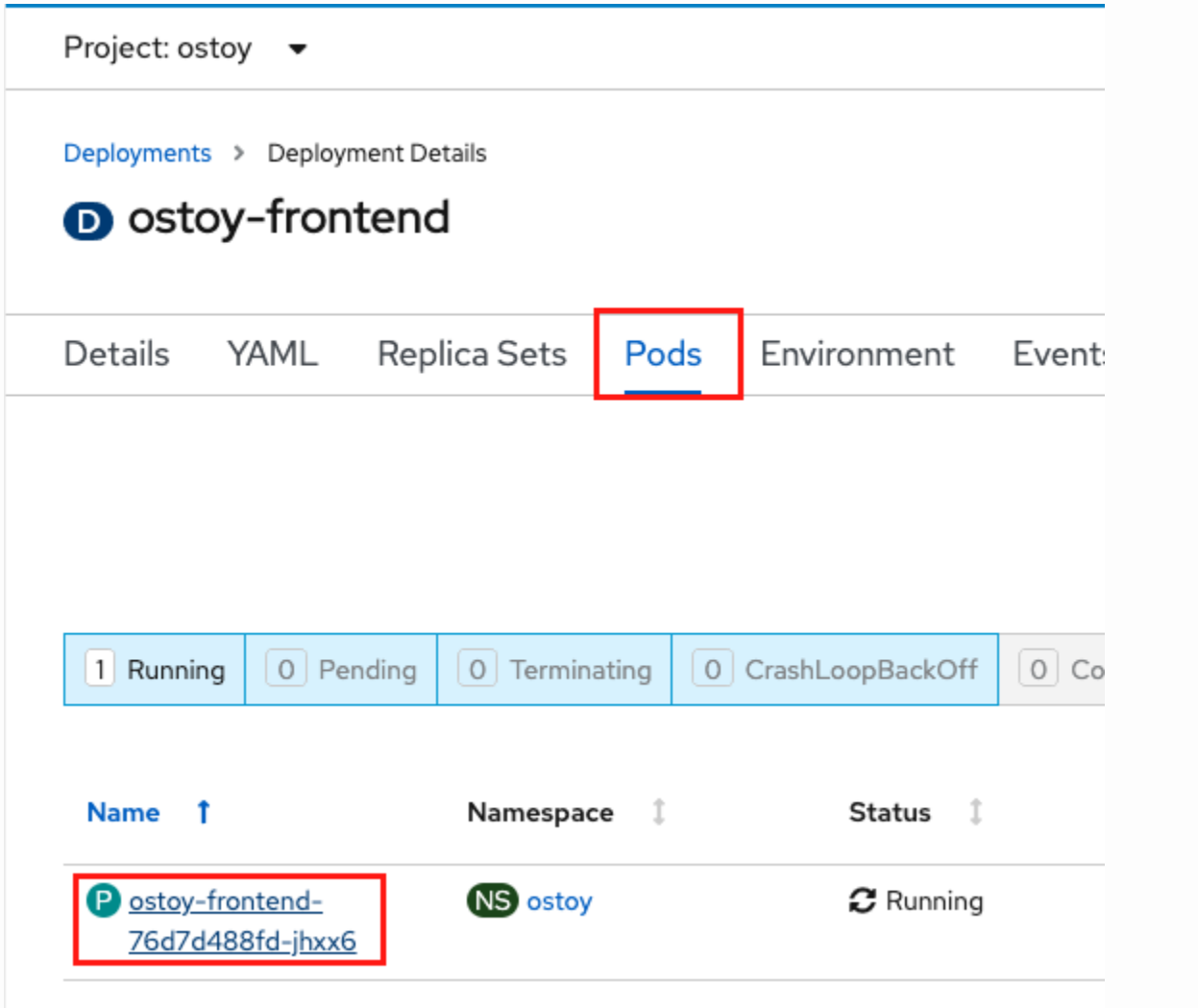


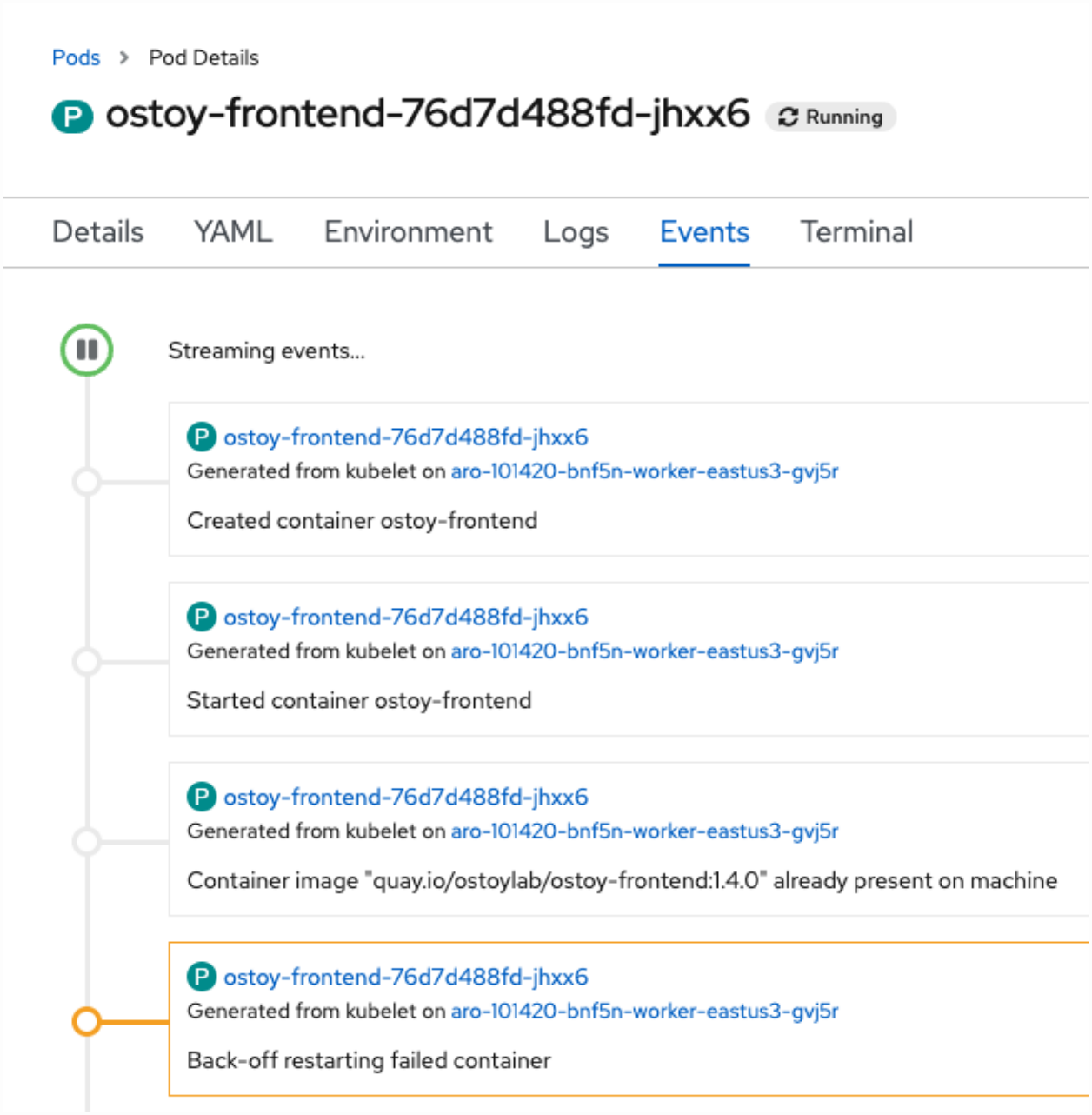
Quickly switch to the tab with the Deployment showing in the Web Console. You will see that the pod is red, meaning it is down but should quickly come back up and show blue. It does happen quickly so you might miss it.



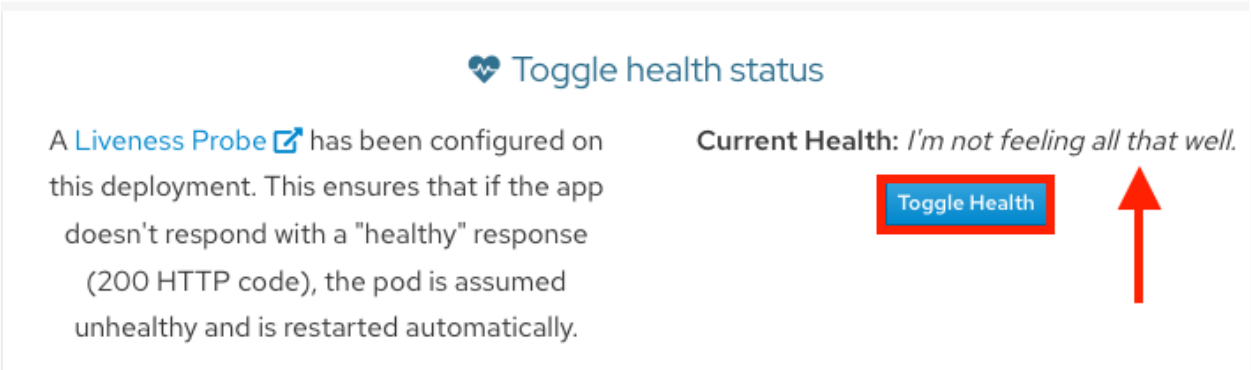
You can also check in the pod events and further verify that the container has crashed and been restarted.

Click on *Pods* > *[Pod Name]* > *Events*

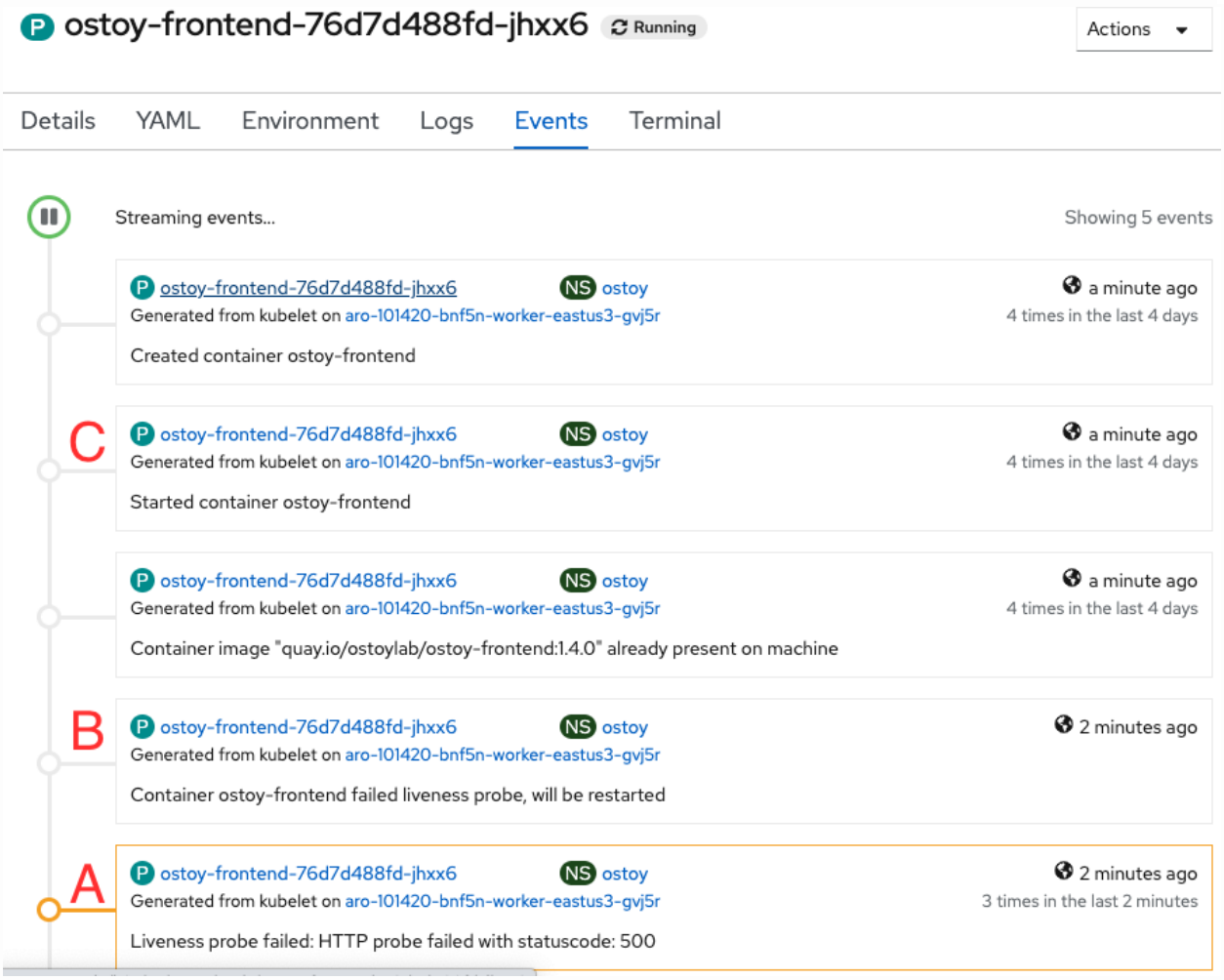




Keep the page from the pod events still open from the previous step. Then in the OSToy app click on the “Toggle Health” button, in the “Toggle Health Status” tile. You will see the “Current Health” switch to “I’m not feeling all that well”.



This will cause the app to stop responding with a “200 HTTP code”. After 3 such consecutive failures (“A”), Kubernetes will kill the pod (“B”) and restart it (“C”). Quickly switch back to the pod events tab and you will see that the liveness probe failed and the pod as being restarted.



Persistent Storage

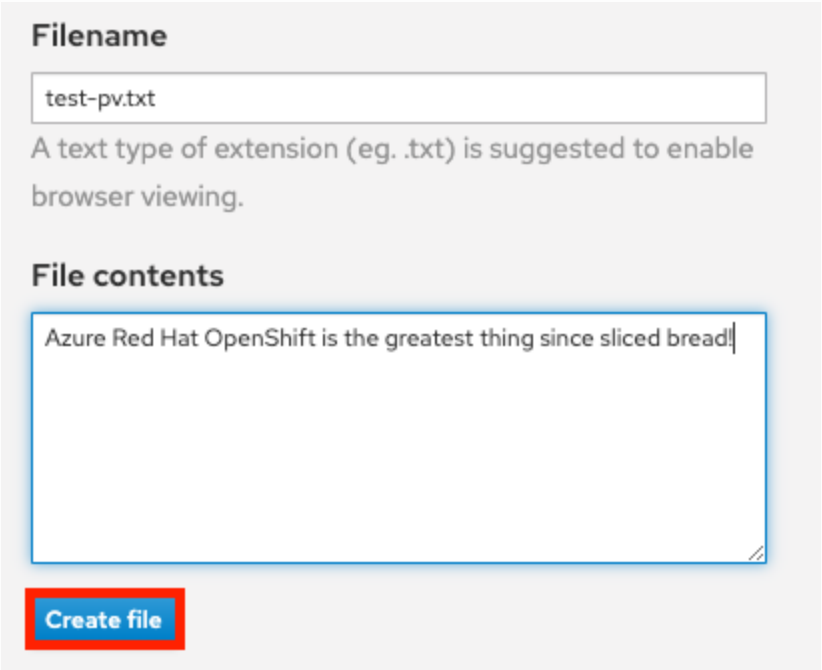
In this section we will execute a simple example of using persistent storage by creating a file that will be stored on a persistent volume in our cluster and then confirm that it will “persist” across pod failures and recreation.

Inside the OpenShift web UI click on *Storage* > *Persistent Volume Claims* in the left menu. You will then see a list of all persistent volume claims that our application has made. In this case there is just one called “ostoy-pvc”. If you click on it you will also see other pertinent information such as whether it is bound or not, size, access mode and creation time.

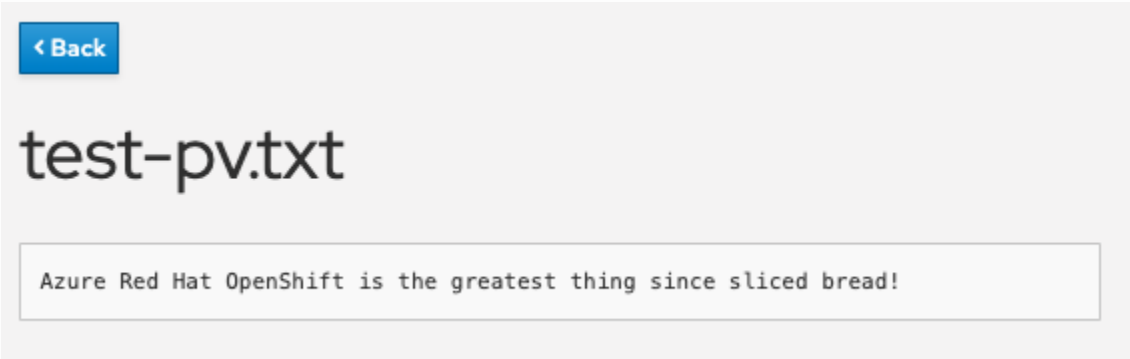
In this case the mode is RWO (Read-Write-Once) which means that the volume can only be mounted to one node, but the pod(s) can both read and write to that volume. The default in AWS is for Persistent Volumes to be backed by EBS, but it is possible to choose EFS Files so that you can use the RWX (Read-Write-Many) access mode. See here for more info on access modes

In the OSToy app click on *Persistent Storage* in the left menu. In the “Filename” area enter a filename for the file you will create. (e.g.: “test-pv.txt”)

Underneath that, in the “File Contents” box, enter text to be stored in the file. (e.g.: “ Red Hat OpenShift is the greatest thing since sliced bread!” or “test” :)). Then click “Create file”.



You will then see the file you created appear above under “Existing files”. Click on the file and you will see the filename and the contents you entered.

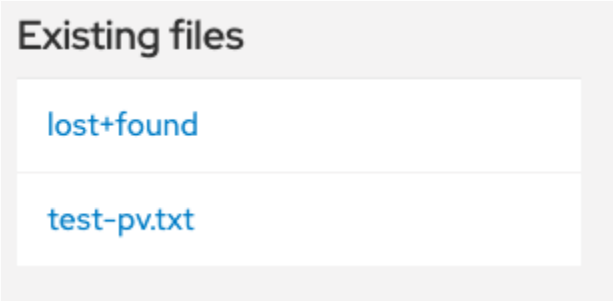


We now want to kill the pod and ensure that the new pod that spins up will be able to see the file we created. Exactly like we did in the previous section. Click on *Home* in the left menu.

Click on the “Crash pod” button. (You can enter a message if you’d like).

Click on *Persistent Storage* in the left menu

You will see the file you created is still there and you can open it to view its contents to confirm.



Now let's confirm that it's actually there by using the CLI and checking if it is available to the container. If you remember we mounted the directory `/var/demo_files` to our PVC. So get the name of your frontend pod

```
oc get pods
```

then get an SSH session into the container

```
oc rsh <pod name>
```

then `cd /var/demo_files`

if you enter `ls` you can see all the files you created. Next, let's open the file we created and see the contents

```
cat test-pv.txt
```

You should see the text you entered in the UI.

```
$ oc get pods

NAME                                READY     STATUS    RESTARTS   AGE

ostoy-frontend-5fc8d486dc-wsw24     1/1       Running   0          18m

ostoy-microservice-6cf764974f-hx4qm   1/1       Running   0          18m
```

```
$ oc rsh ostoy-frontend-5fc8d486dc-wsw24

/ $ cd /var/demo_files/



/var/demo_files $ ls

lost+found   test-pv.txt



/var/demo_files $ cat test-pv.txt

Red Hat OpenShift is the greatest thing since sliced bread!
```

Then exit the SSH session by typing `exit`. You will then be in your CLI.

Configuration

In this section we'll take a look at how OSToy can be configured using ConfigMaps, Secrets, and Environment Variables. This section won't go into details explaining each (the links above are for that), but will show you how they are exposed to the application.

Configuration using ConfigMaps

ConfigMaps allow you to decouple configuration artifacts from container image content to keep containerized applications portable.

Click on *Config Maps* in the left menu.

This will display the contents of the configmap available to the OSToy application. We defined this in the `ostoy-fe-deployment.yaml` here:

```
kind: ConfigMap

apiVersion: v1

metadata:

  name: ostoy-configmap-files

data:

  config.json:  '{ "default": "123" }'
```

Configuration using Secrets

Kubernetes Secret objects allow you to store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a secret is safer and more flexible than putting it, verbatim, into a Pod definition or a container image.

Click on *Secrets* in the left menu.

This will display the contents of the secrets available to the OSToy application. We defined this in the `ostoy-fe-deployment.yaml` here:

```
apiVersion: v1

kind: Secret

metadata:
```

```
  name: ostoy-secret

data:

  secret.txt:
VVNFUk5BTUU9bXlfdXNlcgpQQVNTV09SRD1AT3RCbCVYQXAhIzYzMlk1RndDQE1UUWsKU01UUD1sb2N
hbGhvc3QKU01UUF9QT1JUPTI1

type: Opaque
```

Configuration using Environment Variables

Using environment variables is an easy way to change application behavior without requiring code changes. It allows different deployments of the same application to potentially behave differently based on the environment variables, and OpenShift makes it simple to set, view, and update environment variables for Pods/Deployments.

Click on *ENV Variables* in the left menu.

This will display the environment variables available to the OSToy application. We added three as defined in the deployment spec of `ostoy-fe-deployment.yaml` here:

```
 env:

  - name: ENV_TOY_CONFIGMAP

    valueFrom:

      configMapKeyRef:

        name: ostoy-configmap-env
```

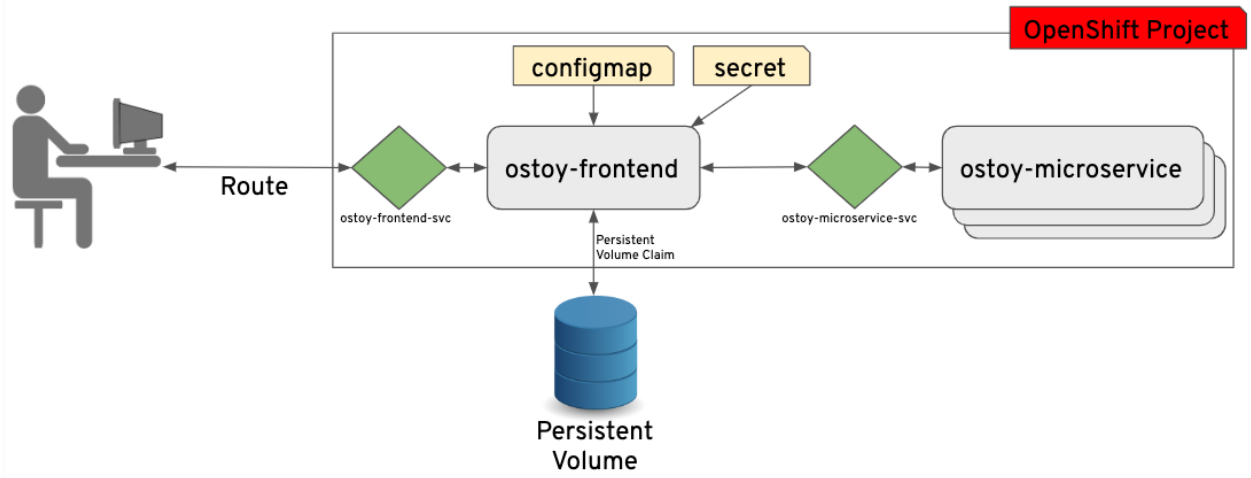
```
        key: ENV_TOY_CONFIGMAP

  - name: ENV_TOY_SECRET

    valueFrom:

      secretKeyRef:

        name: ostoy-secret-env

        key: ENV_TOY_SECRET

  - name: MICROSERVICE_NAME

    value: OSTOY_MICROSERVICE_SVC
```

The last one, `MICROSERVICE_NAME` is used for the intra-cluster communications between pods for this application. The application looks for this environment variable to know how to access the microservice in order to get the colors.

Networking and Scaling

In this section we'll see how OSToy uses intra-cluster networking to separate functions by using microservices and visualize the scaling of pods.

Let's review how this application is set up…



As can be seen in the image above we have defined at least 2 separate pods, each with its own service. One is the frontend web application (with a service and a publicly accessible route) and the other is the backend microservice with a service object created so that the frontend pod can communicate with the microservice (across the pods if more than one). Therefore this microservice is not accessible from outside this cluster, nor from other namespaces/projects (due to OpenShift's network policy, ovs-networkpolicy). The sole purpose of this microservice is to serve internal web requests and return a JSON object containing the current hostname and a randomly generated color string. This color string is used to display a box with that color displayed in the tile titled "Intra-cluster Communication".

Networking

Click on *Networking* in the left menu. Review the networking configuration.

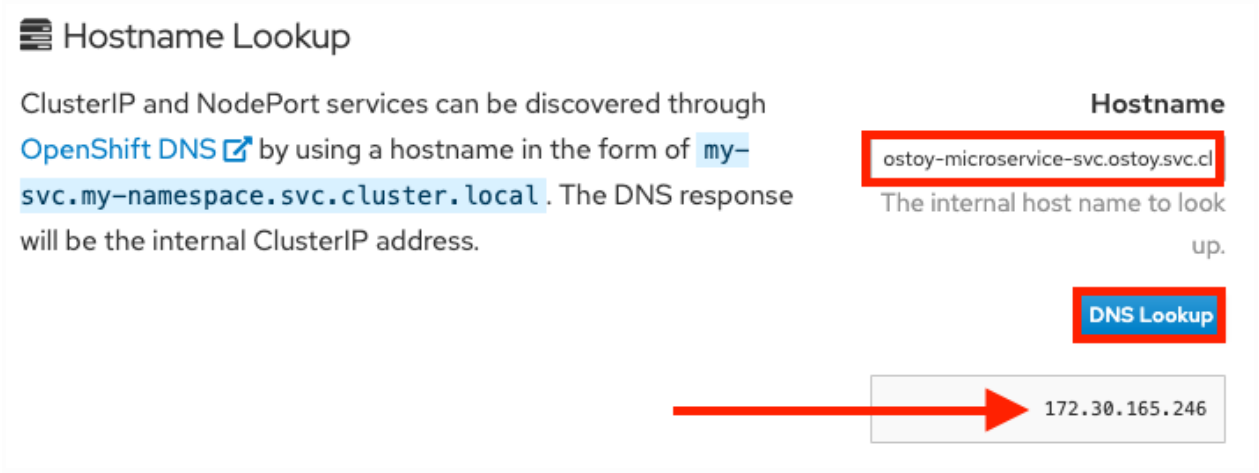
The right tile titled "Hostname Lookup" illustrates how the service name created for a pod can be used to translate into an internal ClusterIP address. Enter the name of the microservice following the format of `my-svc.my-namespace.svc.cluster.local` which we created in our `ostoy-microservice.yaml` which can be seen here:

```
apiVersion: v1
```

```
kind: Service

metadata:

  name: ostoy-microservice-svc

  labels:

    app: ostoy-microservice

spec:

  type: ClusterIP

  ports:

    - port: 8080

      targetPort: 8080

      protocol: TCP

  selector:

    app: ostoy-microservice
```

In this case we will enter: `ostoy-microservice-svc.ostoy.svc.cluster.local`

We will see an IP address returned. In our example it is `172.30.165.246`. This is the intra-cluster IP address; only accessible from within the cluster.



Scaling

OpenShift allows one to scale up/down the number of pods for each part of an application as needed. This can be accomplished by changing our *replicaset/deployment* definition (declarative), through the command line (imperative), or through the web UI (imperative). In our deployment definition (part of our `ostoy-fe-deployment.yaml`) we stated that we only want one pod for our microservice to start with. This means that the Kubernetes Replication Controller will always strive to keep one pod alive.

If we look at the tile on the left we should see one box randomly changing colors. This box displays the randomly generated color sent to the frontend by our microservice along with the pod name that sent it. Since we see only one box that means there is only one microservice pod. We will now scale up our microservice pods and will see the number of boxes change.

To confirm that we only have one pod running for our microservice, run the following command, or use the web UI.

```
$ oc get pods
```

```
NAME                                    READY     STATUS    RESTARTS   AGE

ostoy-frontend-679cb85695-5cn7x         1/1       Running   0          1h

ostoy-microservice-86b4c6f559-p594d     1/1       Running   0          1h
```

Let's change our microservice definition yaml to reflect that we want 3 pods instead of the one we see. Download the ostoy-microservice-deployment.yaml and save it on your local machine.

Open the file using your favorite editor. Ex: `vi ostoy-microservice-deployment.yaml`.

Find the line that states `replicas: 1` and change that to `replicas: 3`. Then save and quit.

It will look like this

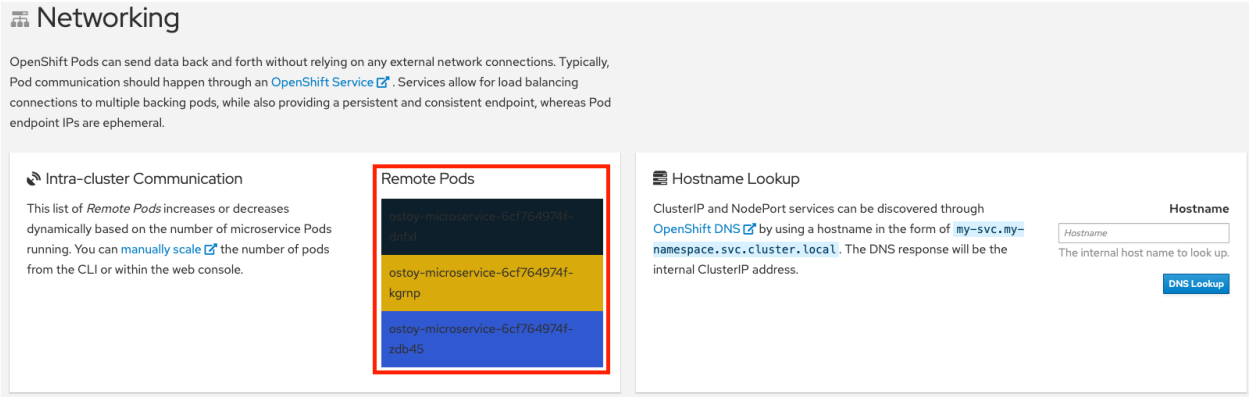
```
spec:

    selector:

      matchLabels:

        app: ostoy-microservice

    replicas: 3
```

Assuming you are still logged in via the CLI, execute the following command:

```
oc apply -f ostoy-microservice-deployment.yaml
```

Confirm that there are now 3 pods via the CLI (`oc get pods`) or the web UI (*Overview > expand "ostoy-microservice"*).

See this visually by visiting the OSToy app and seeing how many boxes you now see. It should be three.



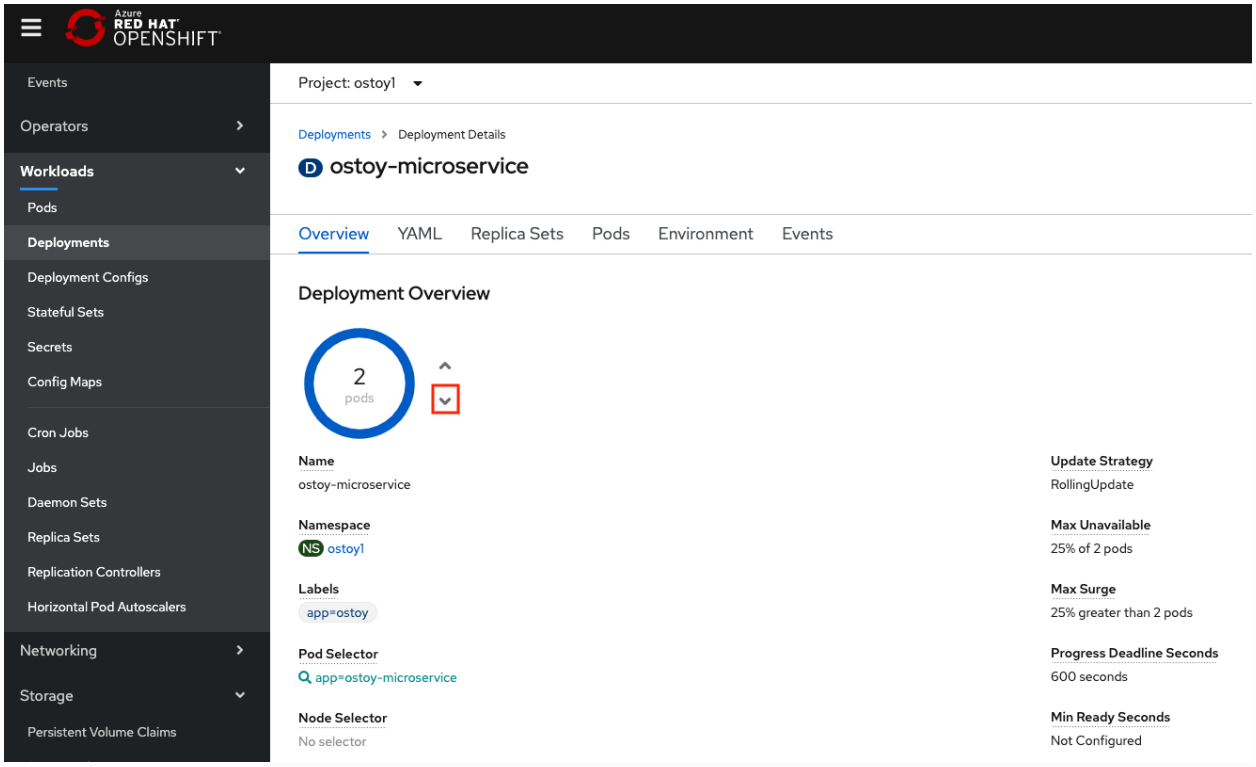
Now we will scale the pods down using the command line. Execute the following command from the CLI:

```
oc scale deployment ostoy-microservice --replicas=2
```

Confirm that there are indeed 2 pods, via the CLI (`oc get pods`) or the web UI.

See this visually by visiting the OSToy App and seeing how many boxes you now see. It should be two.

Lastly let's use the web UI to scale back down to one pod. In the project you created for this app (ie: "ostoy") in the left menu click *Overview > expand "ostoy-microservice"*. On the right you will see a blue circle with the number 2 in the middle. Click on the down arrow to the right of that to scale the number of pods down to 1.



See this visually by visiting the OSToy app and seeing how many boxes you now see. It should be one. You can also confirm this via the CLI or the web UI

Autoscaling

Autoscaling

In this section we will explore how the Horizontal Pod Autoscaler (HPA) can be used and works within Kubernetes/OpenShift.

As defined in the Kubernetes documentation:

Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization.

We will create an HPA and then use OSToy to generate CPU intensive workloads. We will then observe how the HPA will scale up the number of pods in order to handle the increased workloads.

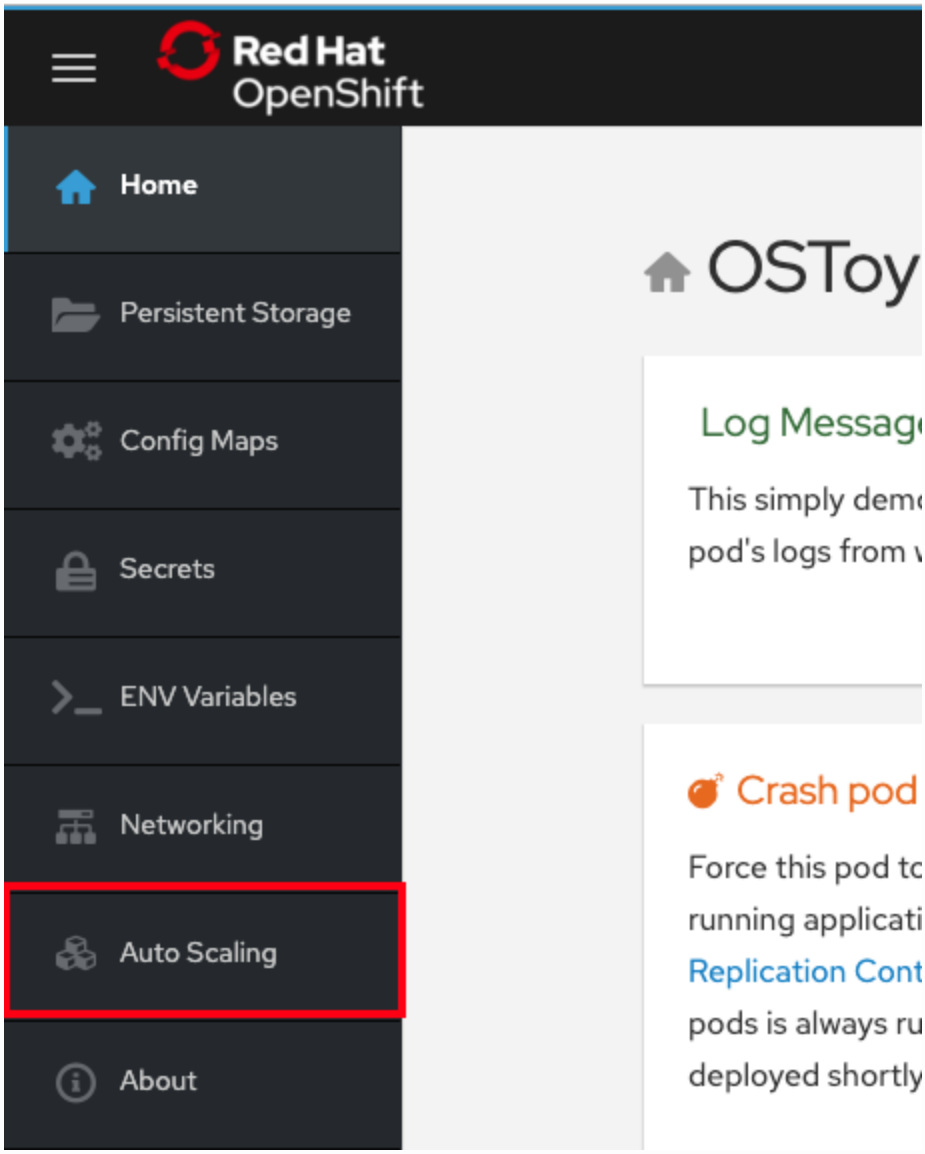
1. Create the Horizontal Pod Autoscaler

Run the following command to create the autoscaler. This will create an HPA that maintains between 1 and 10 replicas of the Pods controlled by the *ostoy-microservice* DeploymentConfig created. Roughly speaking, the HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 80% (since each pod requests 50 millicores, this means average CPU usage of 40 millicores)

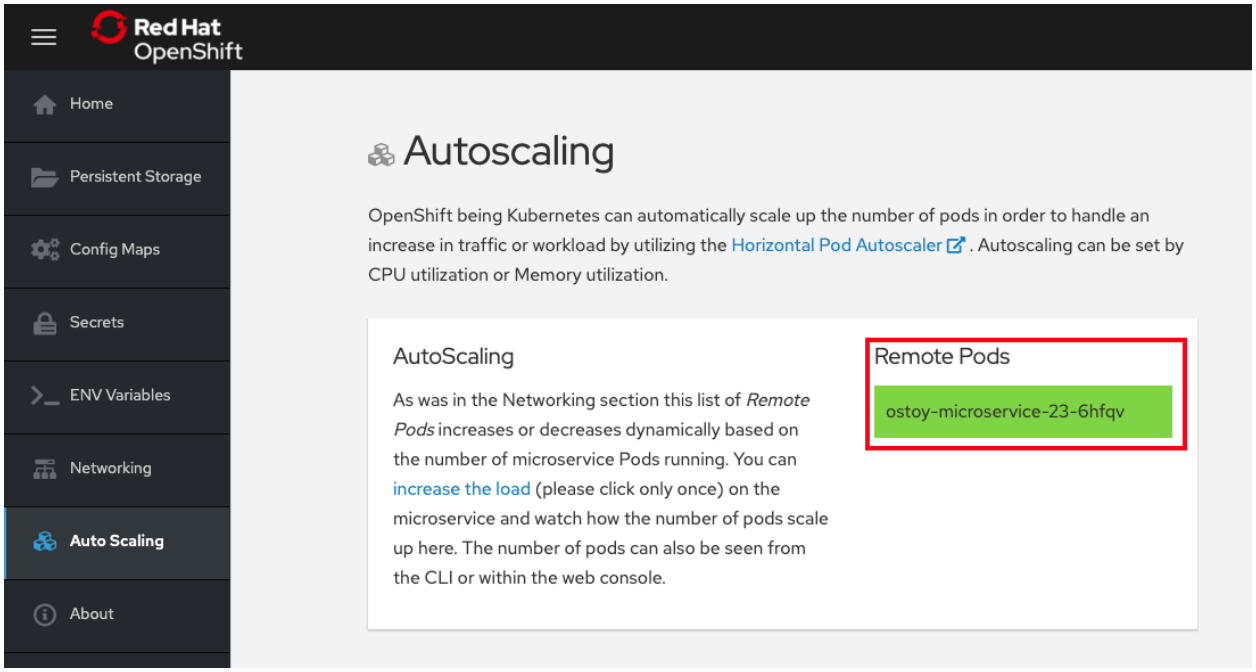
```
oc autoscale deployment/ostoy-microservice --cpu-percent=80 --min=1 --max=10
```

2. View the current number of pods

In the OSToy app in the left menu click on “Autoscaling” to access this portion of the workshop.



As was in the networking section you will see the total number of pods available for the microservice by counting the number of colored boxes. In this case we have only one. This can be verified through the web UI or from the CLI.



You can use the following command to see the running microservice pods only: `oc get pods --field-selector=status.phase=Running | grep microservice`

3. Increase the load

Now that we know that we only have one pod let's increase the workload that the pod needs to perform. Click the link in the center of the card that says "increase the load". Please click only *ONCE*!

This will generate some CPU intensive calculations. (If you are curious about what it is doing you can click here).

Note: The page may become slightly unresponsive. This is normal; so be patient while the new pods spin up.

4. See the pods scale up

After about a minute the new pods will show up on the page (represented by the colored rectangles). Confirm that the pods did indeed scale up through the OpenShift Web Console or the CLI (you can use the command above).

Note: The page may still lag a bit which is normal.