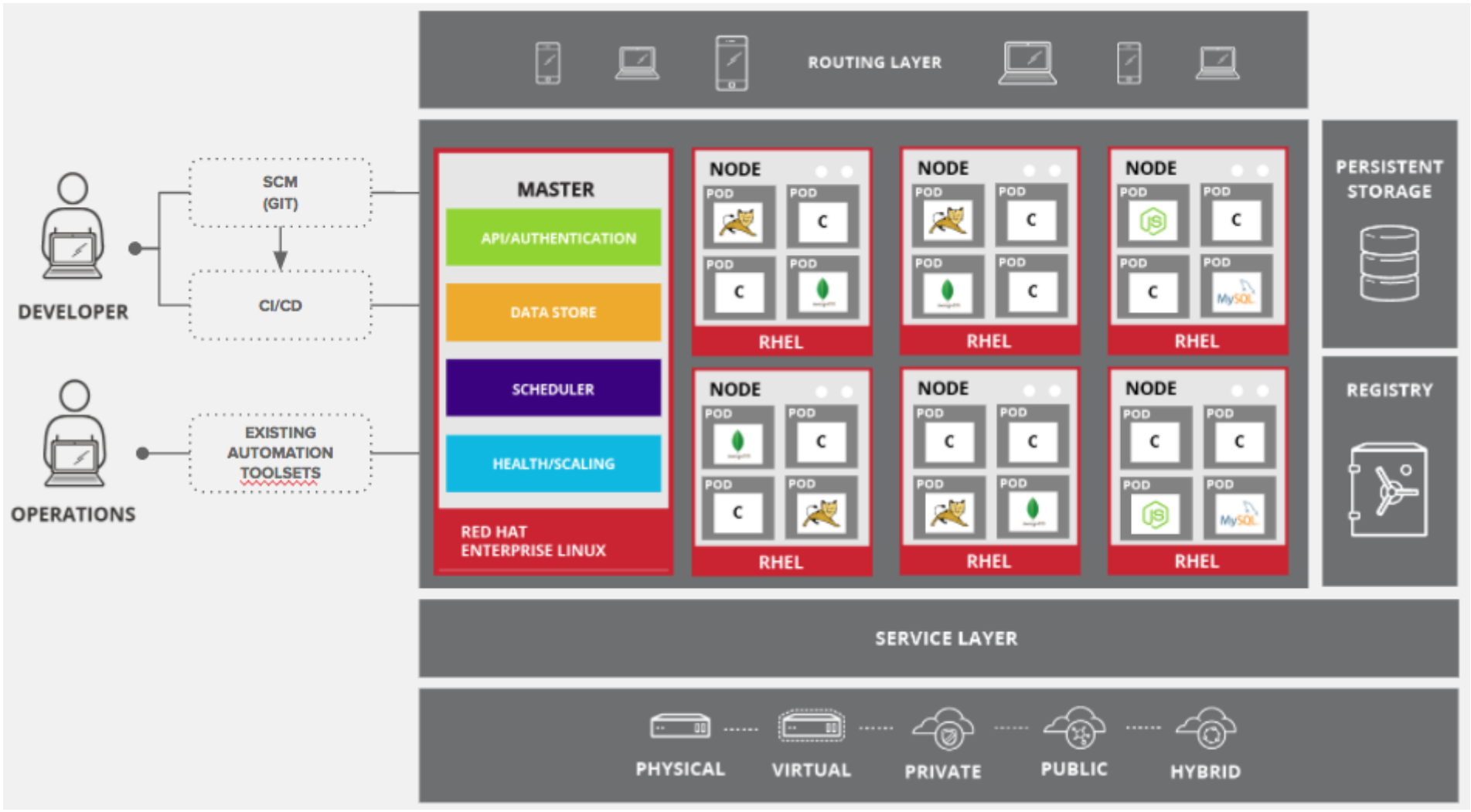
Environment Overview

Environment Overview



You will be interacting with an OpenShift environment that is running on . The environment consists of the following systems:

- 1 master nodes
- 1 infrastructure nodes
- 1 "application" nodes

The infrastructure node is providing several services:

- gitlab git server
- This lab manual
- The OpenShift Docker registry
- The OpenShift router

Additionally, 0 user accounts have been provisioned.

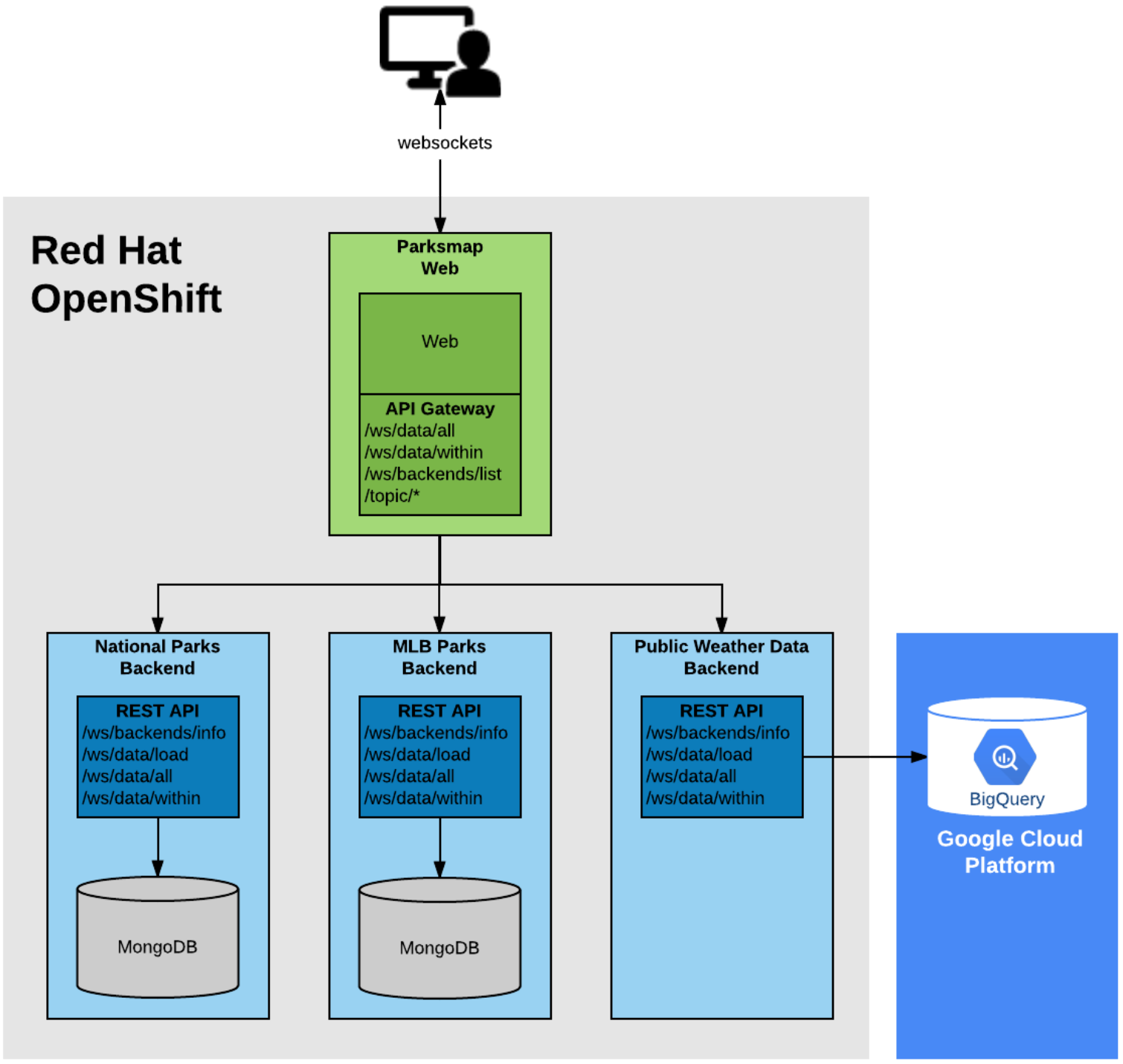
You will see various code and command blocks throughout these exercises. Some of the command blocks can be copy/pasted directly. Others will require modification of the command before execution. If you see a command block with a red border (see below), the command will require modification. Typically this is a substitution of your username or user number:

```
some command for userXX
```

Architecture Overview of the ParksMap Application

Lab: Architecture Overview of the ParksMap Application

This lab introduces you to the architecture of the ParksMap application used throughout this workshop, to get a better understanding of the things you'll be doing from a developer perspective. ParksMap is a polyglot geo-spatial data visualization application built using the microservices architecture and is composed of a set of services which are developed using different programming languages and frameworks.



The main service is a web application which has a server-side component in charge of aggregating the geo-spatial APIs provided by multiple independent backend services and a client-side component in Javascript that is responsible for visualizing the geo-spatial data on the map. The client-side component which runs in your browser communicates with the server-side via WebSockets protocol in order to update the map in real-time.

There will be a set of independent backend services deployed that will provide different mapping and geo-spatial information. The set of available backend services that provide geo-spatial information are:

- WorldWide National Parks
- Major League Baseball Stadiums in the US
- International Airports
- Earthquakes epicenters in New Zealand in November 2017
- Aussie Dunnies

The original source code for this applications is located here (https://github.com/openshift-roadshow/)

The server-side component of the ParksMap web acts as a communication gateway to all the available backends. These backends will be dynamically discovered by using service discovery mechanisms provided by OpenShift which will be discussed in more details in the following labs.

Installing the *oc* client tool

Lab: Installing the OpenShift CLI

Command Line Interface

OpenShift ships with a feature rich web console as well as command line tools to provide users with a nice interface to work with applications deployed to the platform. The OpenShift tools are a single executable written in the Go programming language and is available for the following operating systems:

- Microsoft Windows
- Apple OS X
- Linux

Downloading the tools

During this lab, we are going to download the client tool and add them to our operating system $PATH environment variables so the executable is accessible from any directory on the command line.

The first thing we want to do is download the correct executable for your operating system as linked below:

- Microsoft Windows (https://github.com/openshift/origin/releases/download/v3.7.1/openshift-origin-client-tools-v3.7.1-ab0f056-windows.zip)
- Apple OS X (https://github.com/openshift/origin/releases/download/v3.7.1/openshift-origin-client-tools-v3.7.1-ab0f056-mac.zip)
- Linux 64 (https://github.com/openshift/origin/releases/download/v3.7.1/openshift-origin-client-tools-v3.7.1-ab0f056-linux-64bit.tar.gz)

Once the file has been downloaded, you will need to extract the contents as it is a compressed archive. I would suggest saving this file to the following directories:

Windows:

```BASH
C:\OpenShift
```

OS X:

```BASH
~/OpenShift
```

Linux:

```BASH
~/OpenShift
```

Extracting the tools

Once you have the tools downloaded, you will need to extract the contents:

Windows:

In order to extract a zip archive on Windows, you will need a zip utility installed on your system. With newer versions of Windows (greater than XP), this is provided by the operating system. Just right click on the downloaded file using file explorer and select to extract the contents.

OS X:

Open up a terminal window and change to the directory where you downloaded the file. Once you are in the directory, enter in the following command:

```BASH
$ unzip oc-macosx.tar.gz
```

The zip file name needs to be replaced by the entire name that was downloaded in the previous step.

Linux:

Open up a terminal window and change to the directory where you downloaded the file. Once you are in the directory, enter in the following command:

```BASH
$ tar zxvf oc-linux.tar.gz
```

The tar.gz file name needs to be replaced by the entire name that was downloaded in the previous step.

Adding `oc` to your PATH

Windows:

Because changing your PATH on Windows varies by version of the operating system, we will not list each operating system here. However, the general workflow is right click on your computer name inside of the file explorer. Select Advanced system settings. I guess changing your PATH is considered an advanced task? :) Click on the advanced tab, and then finally click on Environment variables. Once the new dialog opens, select the Path variable and add ";C:\OpenShift" at the end. For an easy way out, you could always just copy it to C:\Windows or a directory you know is already on your path. For more detailed instructions:

Windows XP (https://support.microsoft.com/en-us/kb/310519)

Windows Vista (http://banagale.com/changing-your-system-path-in-windows-vista.htm)

Windows 7 (http://geekswithblogs.net/renso/archive/2009/10/21/how-to-set-the-windows-path-in-windows-7.aspx)

Windows 8 (http://www.itechtics.com/customize-windows-environment-variables/)

Windows 10 - Follow the directions above.

OS X:

```BASH
$ export PATH=$PATH:~/OpenShift
```

Linux:

```
$ export PATH=$PATH:~/OpenShift
```
BASH

Working with proxies

It might happen that you're behind a corporate proxy to access the internet. In this case, you'll need to set some additional environment variables for the oc command line to work.

Windows: Follow previous section's instructions on how to set an Environment Variable on Windows. The variables you'll need to set are:

```
https_proxy=http://proxy-server.mycorp.com:3128/
HTTPS_PROXY=http://proxy-server.mycorp.com:3128/
```
BASH

OS X:

```
$ export https_proxy=http://proxy-server.mycorp.com:3128/
$ export HTTPS_PROXY=http://proxy-server.mycorp.com:3128/
```
BASH

Linux:

```
$ export https_proxy=http://proxy-server.mycorp.com:3128/
$ export HTTPS_PROXY=http://proxy-server.mycorp.com:3128/
```
BASH

If the proxy is secured, make sure to use the following URL pattern, replacing the contents with the appropriate values:

export https_proxy=http://USERNAME:PASSOWRD@proxy-server.mycorp.com:3128/

Special Characters: If your password contains special characters, you must replace them with ASCII codes, for example the at sign @ must be replaced by the %40 code, e.g. p@ssword = p%40ssword.

Verify

At this point, we should have the oc tool available for use. Let's test this out by printing the version of the oc command:

```
$ oc version
```
BASH

You should see the following (or something similar):

```
oc v3.7.1+ab0f056
kubernetes v1.7.6+a08f5eeb62
```
BASH

If you get an error message, you have not updated your path correctly. If you need help, raise your hand and the instructor will assist.

Tab Completion

The OpenShift command line tool supports the ability to use tab completion for the popular zsh and bash shells. This suits the needs of users using either Linux or OS X. If you are using Microsoft Windows, never fear, we will discuss some ways to get tab completion working on that operating system as well.

If you are on the Mac operating sytem, you will need to ensure that you have the **bash-completion** project installed. This can be accomplished using the popular brew system:

```
$ brew install bash-completion
```
BASH

To enable tab completion in your shell, you can simply enter in the following command from your terminal

```
$ oc completion bash >> oc_completion.sh
$ source oc_completion.sh
```
BASH

Alternatively, you can add this to your .bashrc file.

If you are using zsh, you can run the following command:

```
$ source <(oc completion zsh)
```
BASH

Alternatively, you can add this to your .zshrc file.

For Windows users, things become a bit more tricky. You could of course use the Linux Subsystem for Windows but you may want to consider using a combination of babun and cmder. For a full list of instructions, you can check out the following blog post:

- https://blog.openshift.com/openshift-3-tab-completion-for-windows/ (https://blog.openshift.com/openshift-3-tab-completion-for-windows/)

Exploring the CLI and Web Console

Lab: Exploring the CLI and Web Console

Command Line

The first thing we want to do to ensure that our `oc` command line tools was installed and successfully added to our path is to log in to the OpenShift environment that has been provided for this Roadshow session. In order to log in, we will use the `oc` command and then specify the server that we want to authenticate to. Issue the following command:

```BASH
$ oc login 10.2.2.2:8443
```

You may see the following output:

```BASH
The server uses a certificate signed by an unknown authority.
You can bypass the certificate check, but any data you send to the server could be intercepted by others.
Use insecure connections? (y/n):
```

Enter in **Y** to use a potentially insecure connection. The reason you received this message is because we are using a self-signed certificate for this workshop, but we did not provide you with the CA certificate that was generated by OpenShift. In a real-world scenario, either OpenShift's certificate would be signed by a standard CA (eg: Thawte, Verisign, StartSSL, etc.) or signed by a corporate-standard CA that you already have installed on your system.

On some versions of Microsoft Windows, you may get an error that the server has an invalid x.509 certificate. If you receive this error, enter in the following command:

```BASH
$ oc login 10.2.2.2:8443 --insecure-skip-tls-verify=true
```

Once you issue the `oc login` command, you will be prompted for the username and password combination for your user account. Use the username and password combination provided to you by the instructor of this workshop:

```BASH
Username: dev01
Password: dev
```

Once you have authenticated to the OpenShift server, you will see the following confirmation message:

```BASH
Login successful.

Using project "project-userXY".
```

Congratulations, you are now authenticated to the OpenShift server. The OpenShift master includes a built-in OAuth server. Developers and administrators obtain OAuth access tokens to authenticate themselves to the API. By default your authorization token will last for 24 hours. There is more information about the login command and its configuration in the OpenShift Documentation (https://<url>/cli_reference/get_started_cli.html#basic-setup-and-login).

Using a project

Projects are a top level concept to help you organize your deployments. An OpenShift project allows a community of users (or a user) to organize and manage their content in isolation from other communities. Each project has its own resources, policies (who can or cannot perform actions), and constraints (quotas and limits on resources, etc). Projects act as a "wrapper" around all the application services and endpoints you (or your teams) are using for your work.

During this lab, we are going to use a few different commands to make sure that things in the environment are working as expected. Don't worry if you don't understand all of the terminology as we will cover it in detail in later labs.

The first thing we want to do is switch to the **project-userXY** project. You can do this with the following command:

```BASH
$ oc project project-userXY
```

You will see the following confirmation message:

```BASH
Now using project "project-userXY" on server "https://10.2.2.2:8443".
```

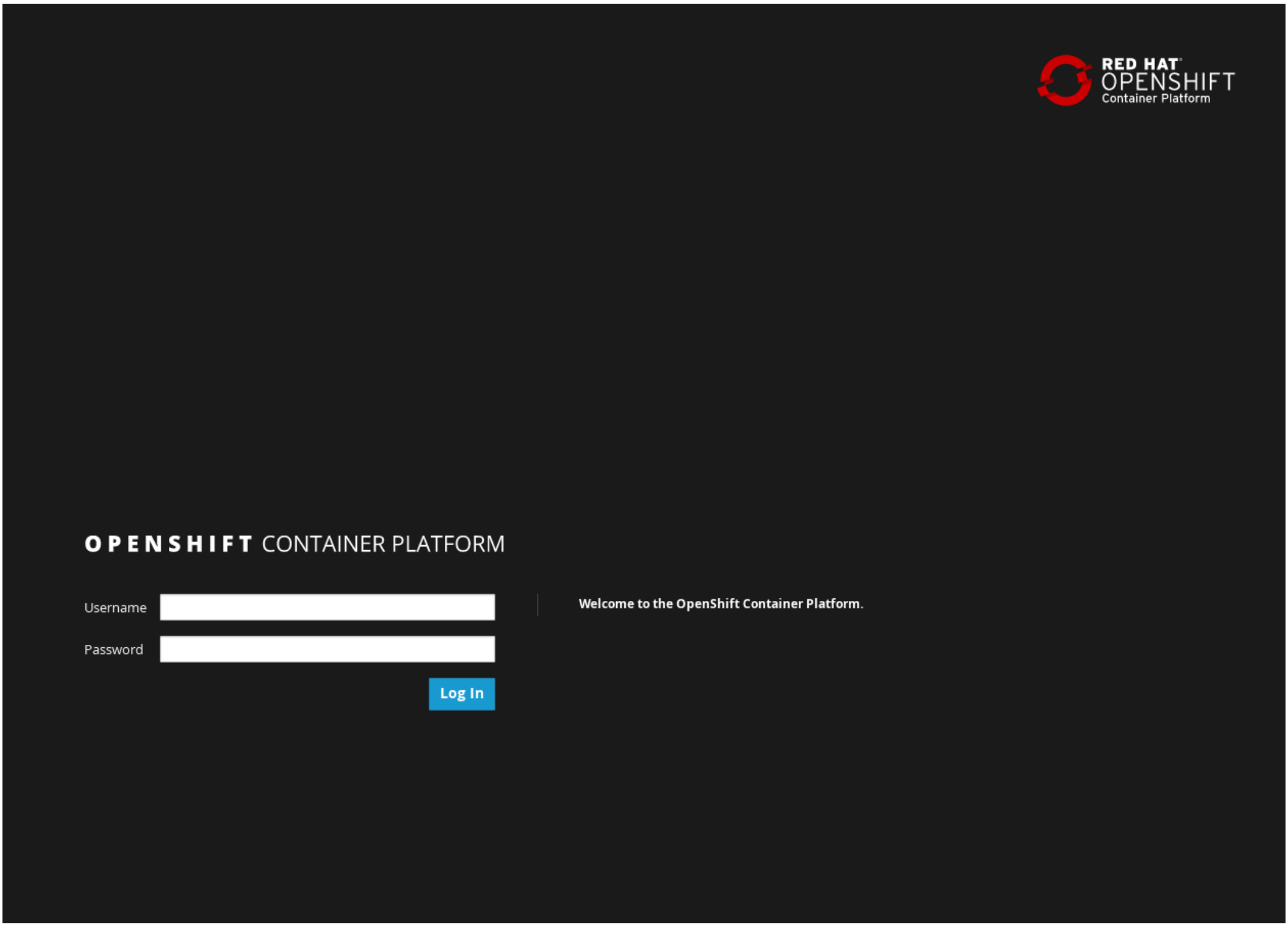
The Web Console

OpenShift ships with a web-based console that will allow users to perform various tasks via a browser. To get a feel for how the web console works, open your browser and go to the following URL:

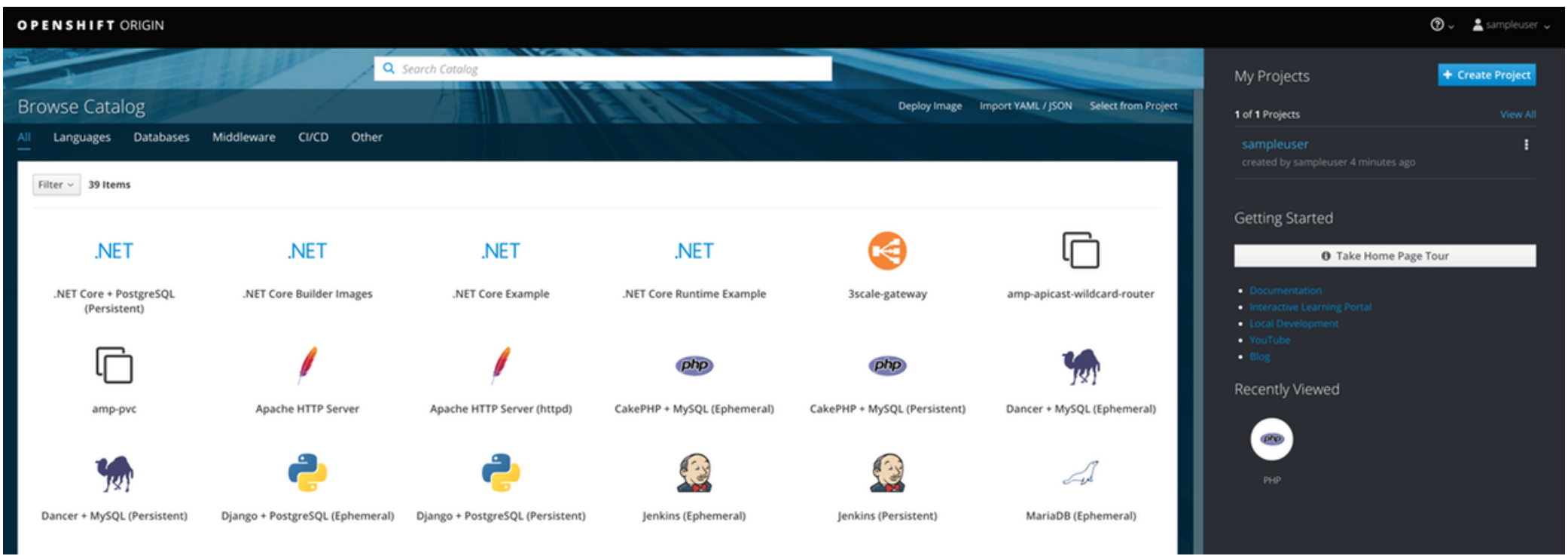
https://10.2.2.2:8443

The first screen you will see is the authentication screen. Enter in the following credentials:

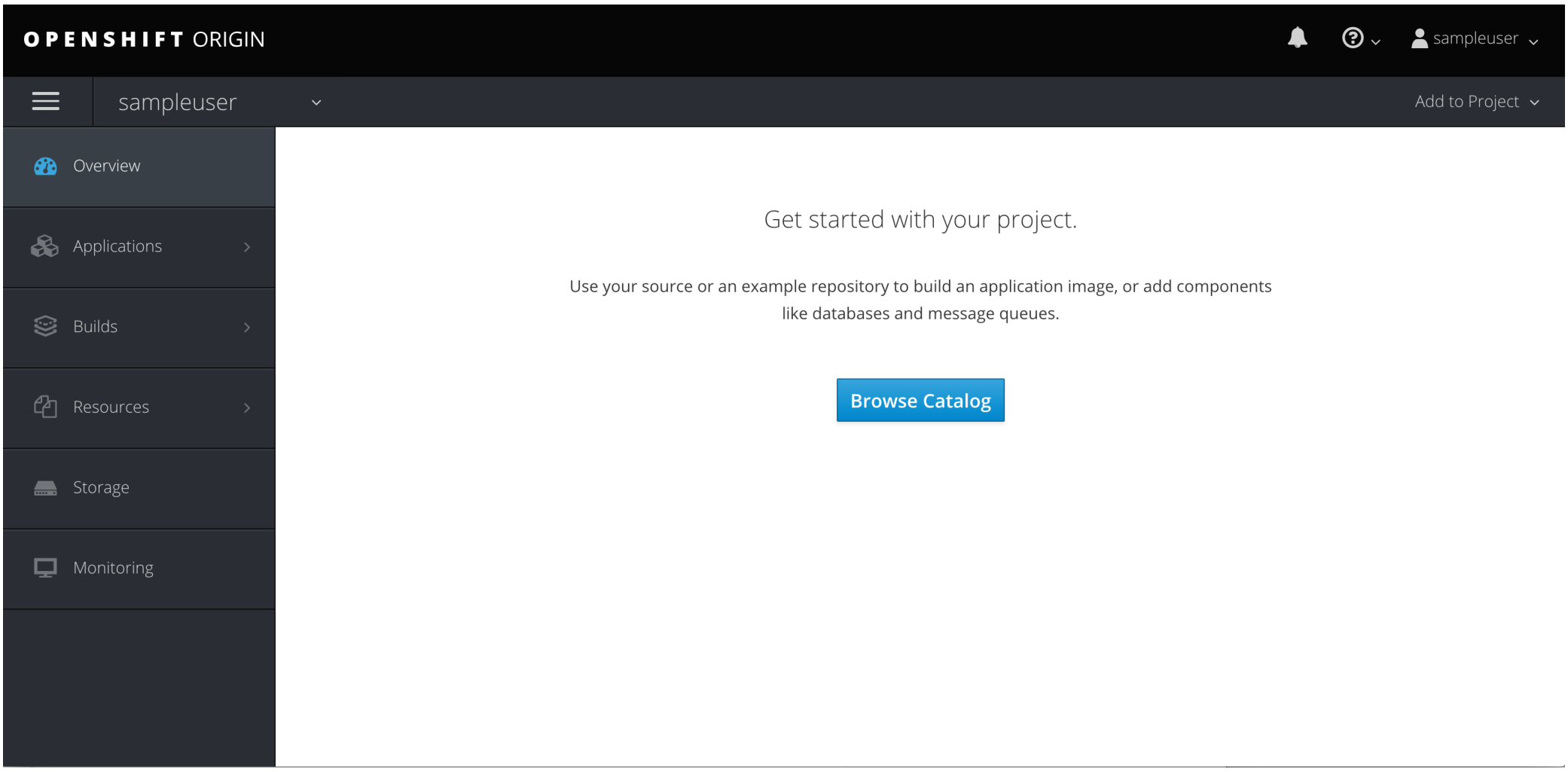
```BASH
Username: dev01
Password: dev
```



After you have authenticated to the web console, you will be presented with a list of items that you can add to your project. This is called the service catalog. You will see something that looks like the following image:



Click on the **project-userXY** project on the right hand side of the screen. When you click on the **project-userXY** project, you will be taken to the project overview page which will list all of the routes, services, deployments, and pods that you have running as part of your project. There's nothing there now, but that's about to change.



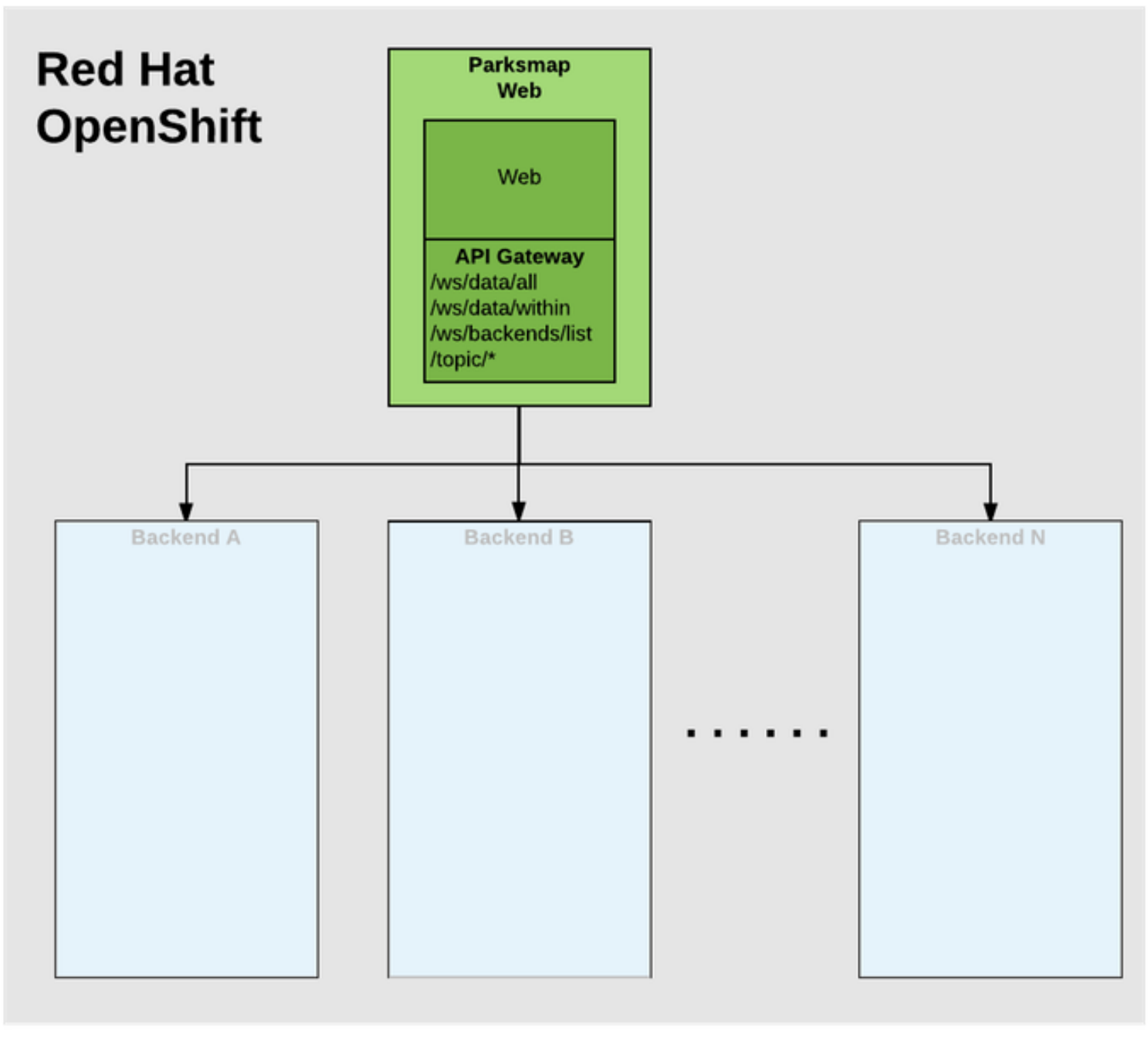
We will be using a mix of command line tooling and the web console for the labs. Get ready!

Deploying our First Docker Image

Lab: Deploy a Docker Image

Application description

In this lab, we're going to deploy the web component of the ParksMap application which is also called `parksmap` and uses OpenShift service discovery mechanism to discover the backend services deployed and shows their data on the map.



Exercise: Deploying your first Image

Let's start by doing the simplest thing possible - get a plain old Docker-formatted image to run on OpenShift. This is incredibly simple to do. With OpenShift Container Platform 3.5 it can be done directly from the web console.

Return to the web console:

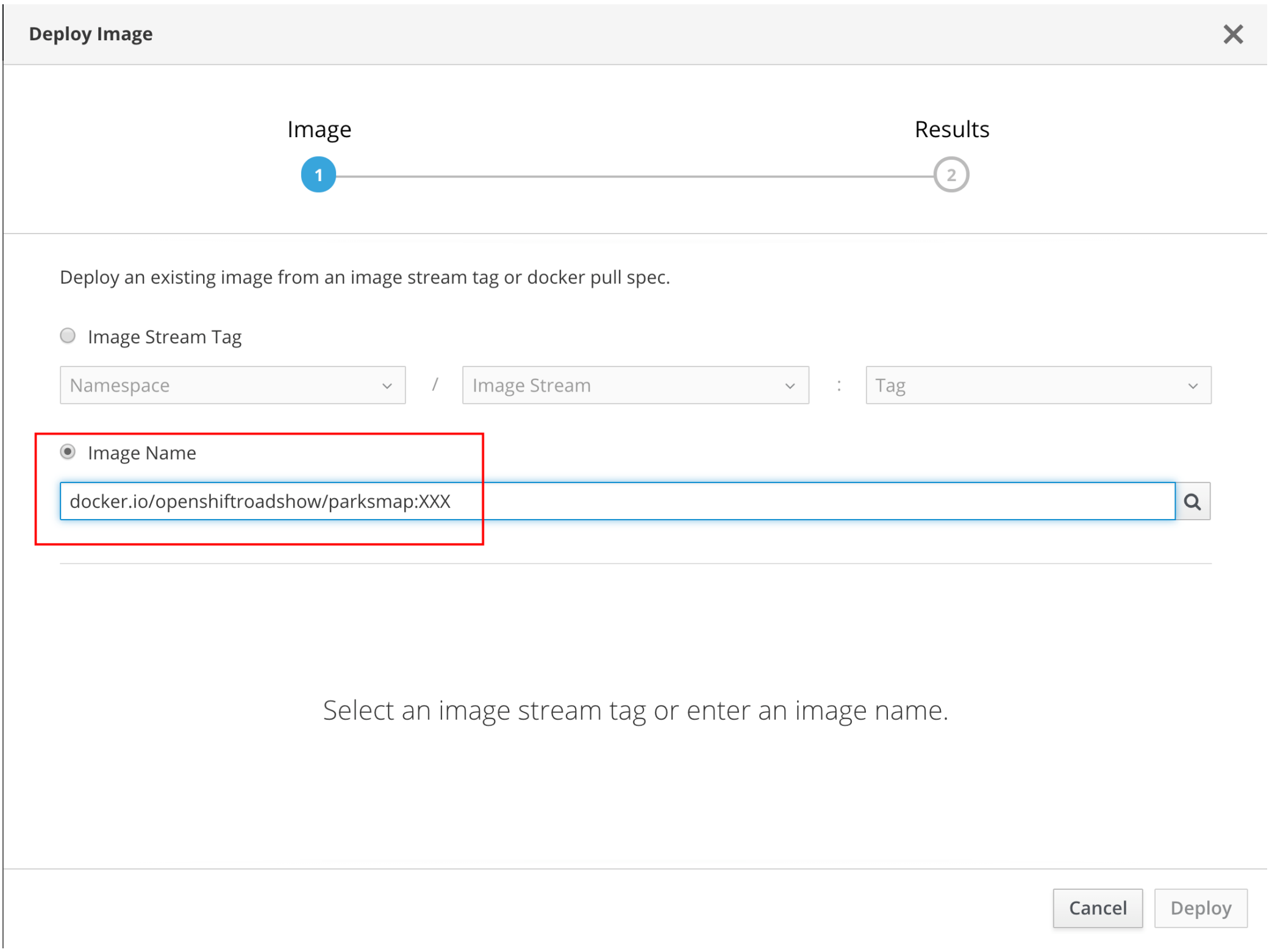
https://10.2.2.2:8443

Find your **project-userXY** project and click it. Next, click on the `Deploy Image` in the **"Add to Project"** drop down. This will open a dialog that will allow you to specify the information for the image you want to deploy.

There are several options, but we are only concerned with "Deploy Image". Click it. We will learn more about image streams and image stream tags later. For now, select the "Image Name" option, and copy/paste the following into the box:

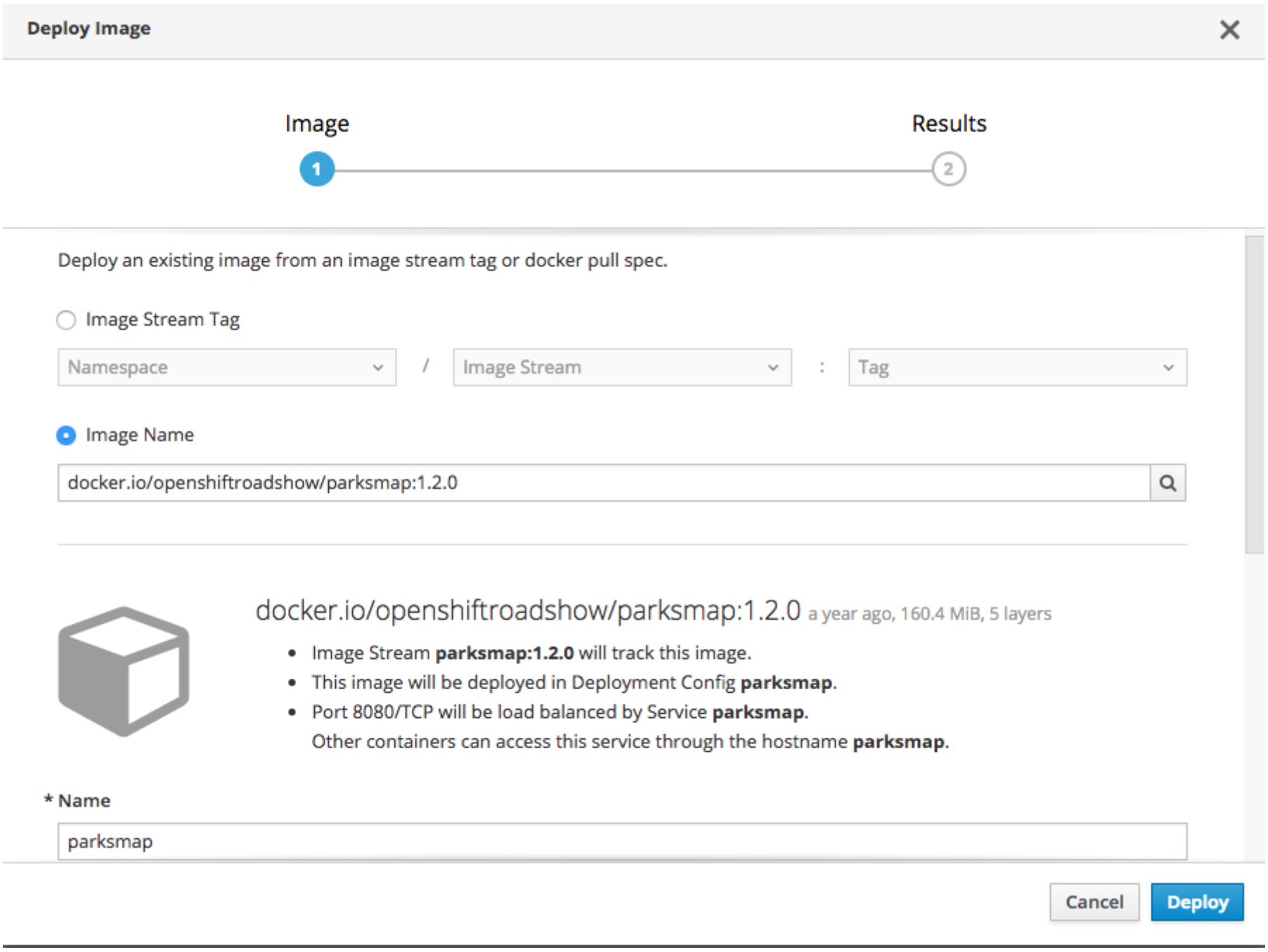
```
docker.io/openshiftroadshow/<image name>:<version>
```

Your screen will end up looking something like this:



Either press **enter** or click on the magnifying glass. OpenShift will then go out to the Docker registry specified and interrogate the image. You then are presented with some options to add things like environment variables, labels, and etc. — which we will learn about later.

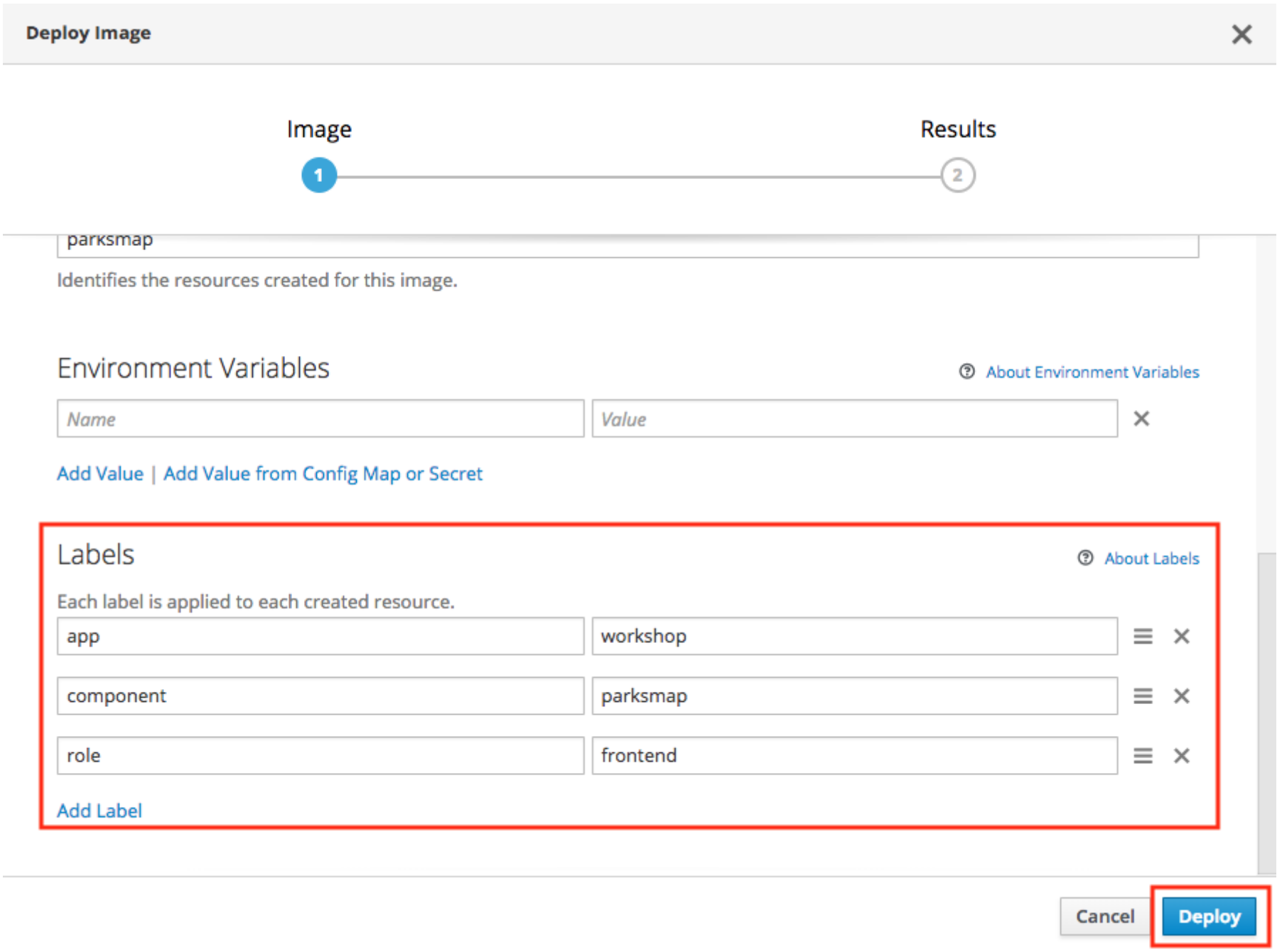
Make sure to have the correct application name:



Let's scroll to the bottom of this page and add some labels to better identify this deployment later. Labels will help us identify and filter components in the web UI and in the command line.

We will add 3 labels:

- ***app***=**workshop** (the name we will be giving to the app)
- ***component***=**parksmap** (the name of this deployment)
- ***role***=**frontend** (the role this component plays in the overall application)



Hit the blue "Deploy" button at the bottom of the dialog and then click the "Continue to overview" link or the "Close" button. Take a moment to look at the various messages that you now see on the overview page.

WINNING! These few steps are the only ones you need to run to get a Docker-formatted image deployed on OpenShift. This should work with any Docker-formatted image that follows best practices, such as defining an EXPOSE port, not needing to run specifically as the **root user** or other user name, and a single non-exiting CMD to execute on start.

Adding the labels is desired when deploying complex applications for organization purposes.

Background: Containers and Pods

Before we start digging in we need to understand how containers and **Pods** are related. We will not be covering the background on these technologies in this lab but if you have questions please inform the instructor. Instead, we will dive right in and start using them.

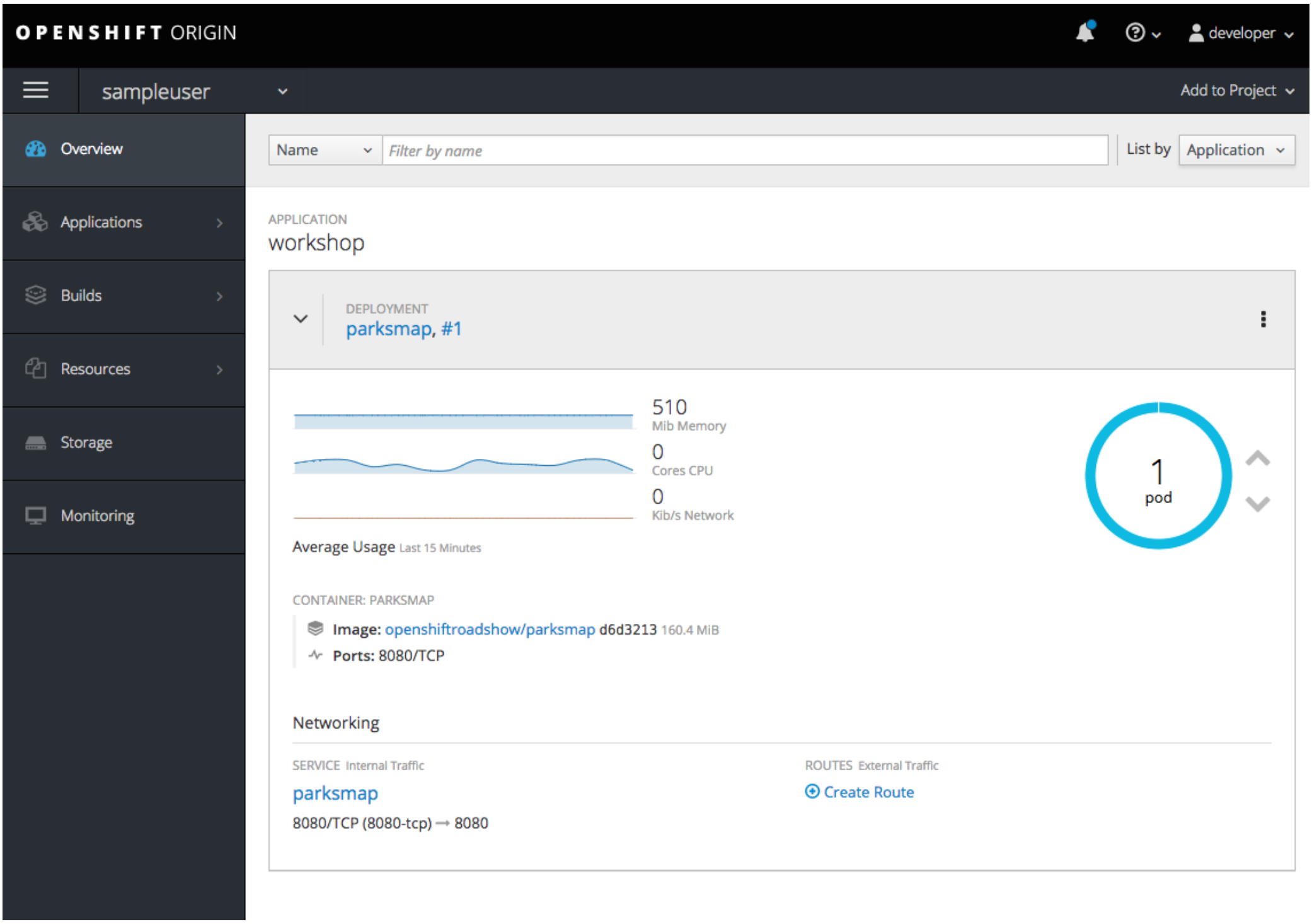
In OpenShift, the smallest deployable unit is a **Pod**. A **Pod** is a group of one or more Docker containers deployed together and guaranteed to be on the same host. From the official OpenShift documentation:

> *Each pod has its own IP address, therefore owning its entire port space, and containers within pods can share storage. Pods can be "tagged" with one or more labels, which are then used to select and manage groups of pods in a single operation.*

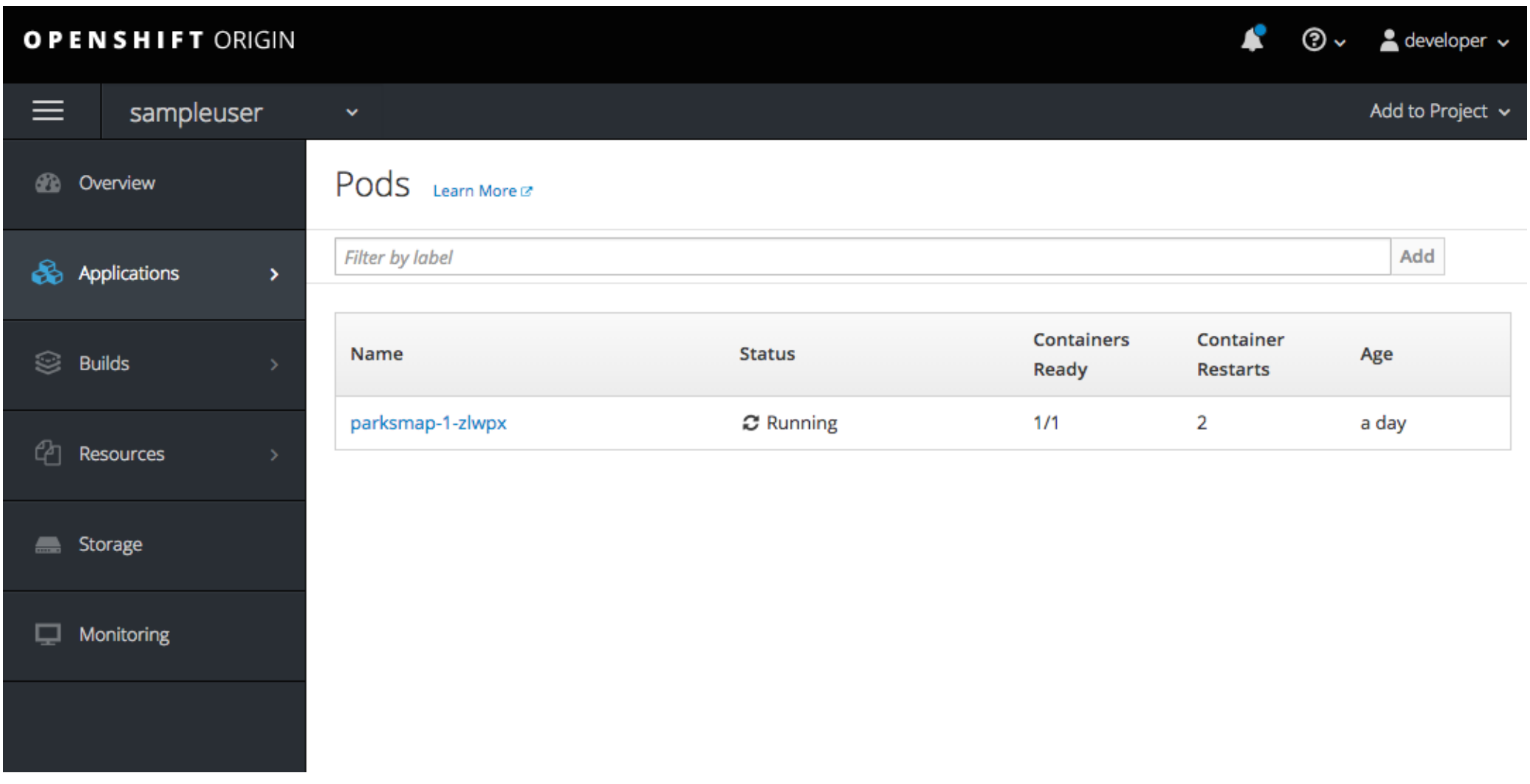
Pods can contain multiple Docker instances. The general idea is for a Pod to contain a "server" and any auxiliary services you want to run along with that server. Examples of containers you might put in a **Pod** are, an Apache HTTPD server, a log analyzer, and a file service to help manage uploaded files.

Exercise: Examining the Pod

In the web console's overview page you will see that there is a single **Pod** that was created by your actions.



You can also get a list of all the pods created within your project, by navigating to "Applications → Pods"



This **Pod** contains a single container, which happens to be the parks map application - a simple Spring Boot/Java application.

You can also examine **Pods** from the command line:

```
$ oc get pod
```
BASH

You should see output that looks similar to:

```
BASH
NAME                READY     STATUS    RESTARTS   AGE
parksmap-1-hx0kv    1/1       Running   0          2m
```

The above output lists all of the **Pods** in the current **Project**, including the **Pod** name, state, restarts, and uptime. Once you have a **Pod**'s name, you can get more information about the **Pod** using the **oc get** command. To make the output readable, I suggest changing the output type to **YAML** using the following syntax:

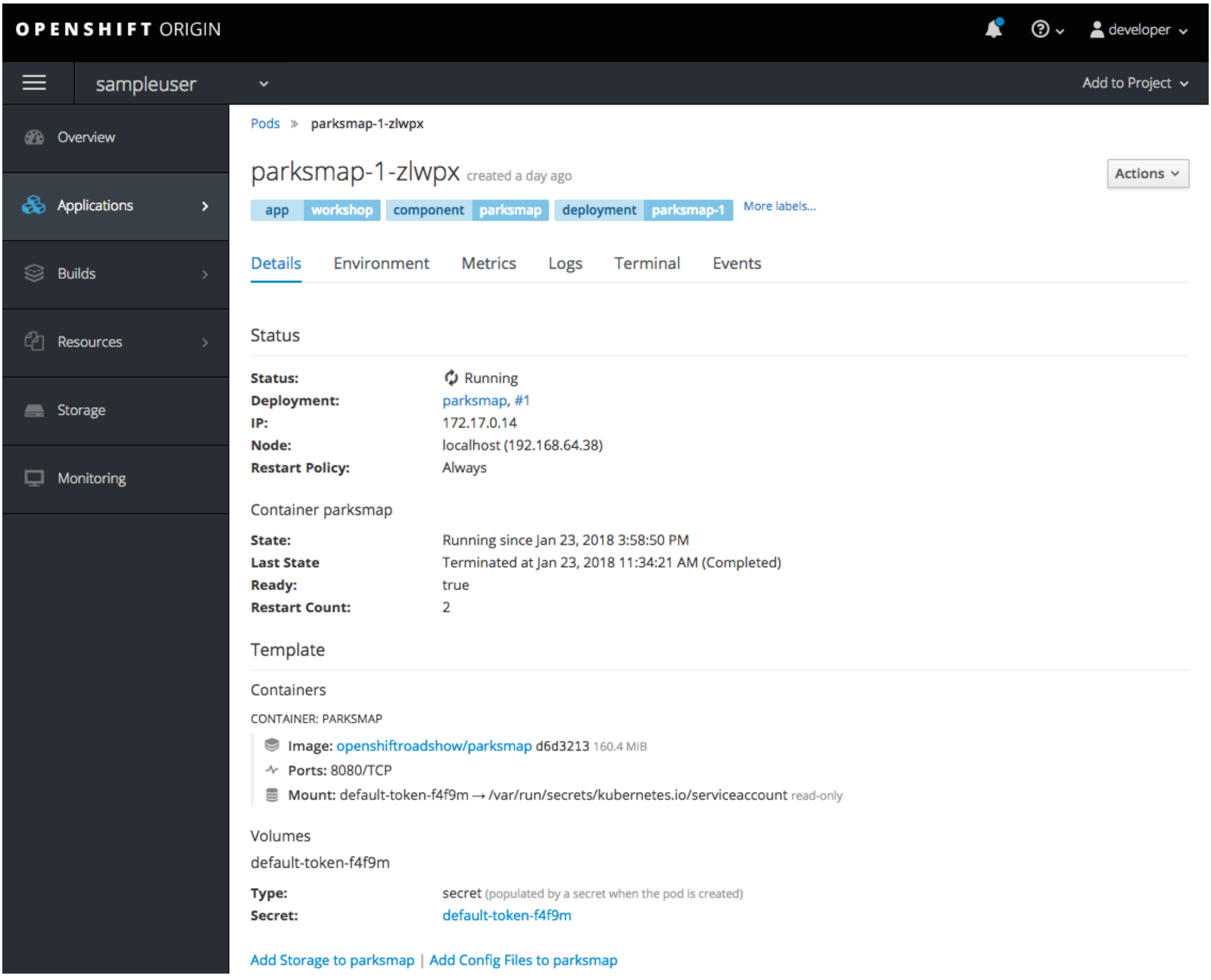
Make sure you use the correct **Pod** name from your output.

```
BASH
$ oc get pod parksmap-1-hx0kv -o yaml
```

You should see something like the following output (which has been truncated due to space considerations of this workshop manual):

```
BASH
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: |
      {"kind":"SerializedReference","apiVersion":"v1","reference":{"kind":"ReplicationController","namespace":"explore-00","name":"parksmap-
1","uid":"f1b37b1b-e3e2-11e6-81a2-0696d1181070","apiVersion":"v1","reso
urceVersion":"36222"}}
    kubernetes.io/limit-ranger: 'LimitRanger plugin set: cpu, memory request for container
      parksmap; cpu, memory limit for container parksmap'
    openshift.io/deployment-config.latest-version: "1"
    openshift.io/deployment-config.name: parksmap
    openshift.io/deployment.name: parksmap-1
    openshift.io/generated-by: OpenShiftWebConsole
    openshift.io/scc: restricted
  creationTimestamp: 2017-01-26T16:17:36Z
  generateName: parksmap-1-
  labels:
    app: parksmap
    deployment: parksmap-1
    deploymentconfig: parksmap
  name: parksmap-1-bvaz6
...............
```

The web interface also shows a lot of the same information on the **Pod** details page. If you click in the **Pod** circle, you will find the details page. You can also get there by clicking "Applications", then "Pods", at the left, and then clicking the **Pod** name.



Getting the parks map image running may take a little while to complete. Each OpenShift node that is asked to run the image has to pull (download) it if the node does not already have it cached locally. You can check on the status of the image download and deployment in the **Pod** details page, or from the command line with the `oc get pods` command that you used before.

Background: A Little About the Docker Daemon

Whenever OpenShift asks the node's Docker daemon to run an image, the Docker daemon will check to make sure it has the right "version" of the image to run. If it doesn't, it will pull it from the specified registry.

There are a number of ways to customize this behavior. They are documented in specifying an image (https://<url>/dev_guide/application_lifecycle/new_app.html#specifying-an-image) as well as image pullpolicy (https://<url>/dev_guide/managing_images.html#image-pull-policy).

Background: Services

Services provide a convenient abstraction layer inside OpenShift to find a group of like **Pods**. They also act as an internal proxy/load balancer between those **Pods** and anything else that needs to access them from inside the OpenShift environment. For example, if you needed more parks map servers to handle the load, you could spin up more **Pods**. OpenShift automatically maps them as endpoints to the **Service**, and the incoming requests would not notice anything different except that the **Service** was now doing a better job handling the requests.

When you asked OpenShift to run the image, it automatically created a **Service** for you. Remember that services are an internal construct. They are not available to the "outside world", or anything that is outside the OpenShift environment. That's OK, as you will learn later.

The way that a **Service** maps to a set of **Pods** is via a system of **Labels** and **Selectors**. **Services** are assigned a fixed IP address and many ports and protocols can be mapped.

There is a lot more information about Services (https://<url>/architecture/core_concepts/pods_and_services.html#services), including the YAML format to make one by hand, in the official documentation.

Now that we understand the basics of what a **Service** is, let's take a look at the **Service** that was created for the image that we just deployed. In order to view the **Services** defined in your **Project**, enter in the following command:

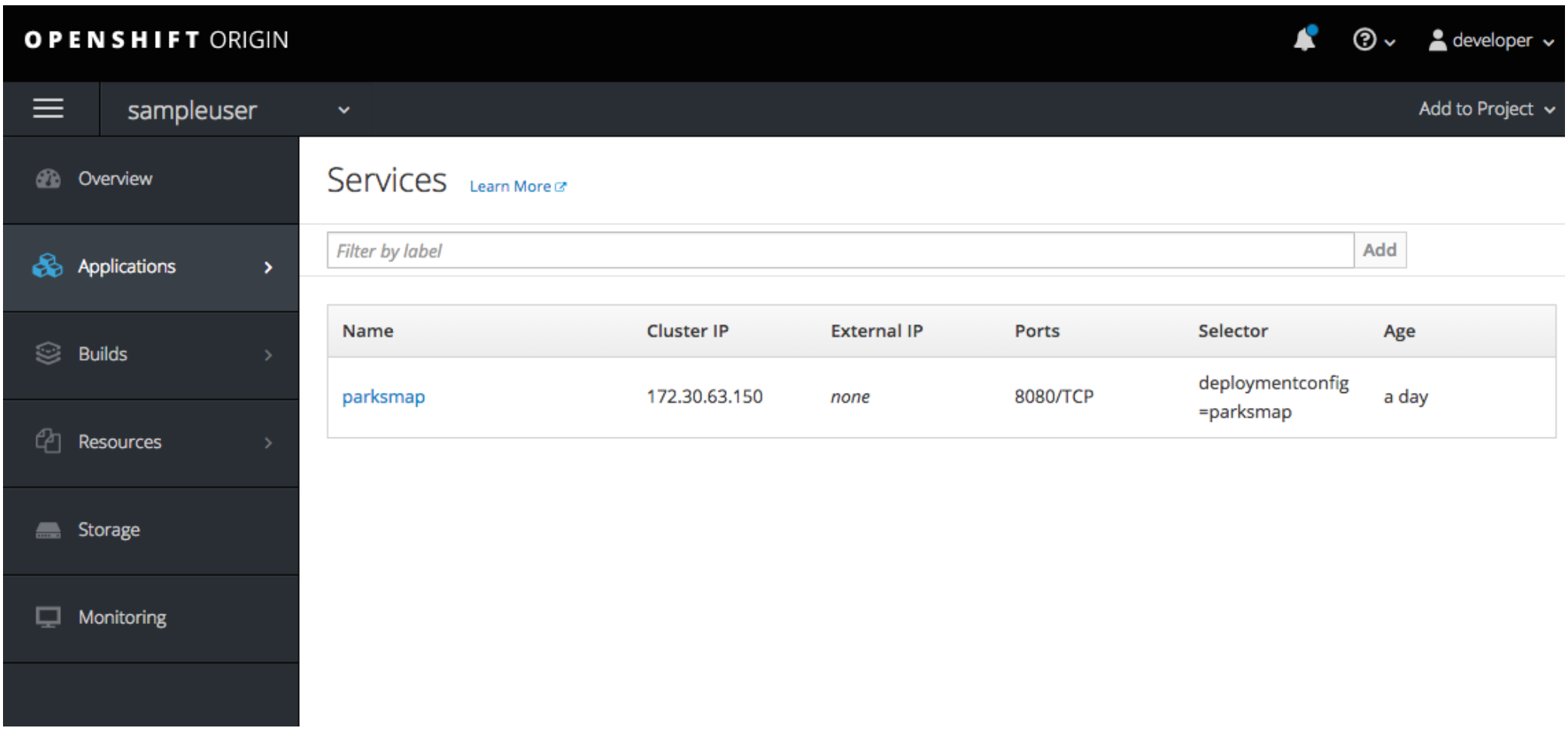
```
$ oc get services
```
BASH

You should see output similar to the following:

```
BASH
NAME        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
parksmap    172.30.169.213   <none>        8080/TCP   3h
```

In the above output, we can see that we have a **Service** named `parksmap` with an IP/Port combination of 172.30.169.213/8080TCP. Your IP address may be different, as each **Service** receives a unique IP address upon creation. **Service** IPs are fixed and never change for the life of the **Service**.

In the web console, service information is available by clicking "Applications" and then clicking "Services".



You can also get more detailed information about a **Service** by using the following command to display the data in YAML:

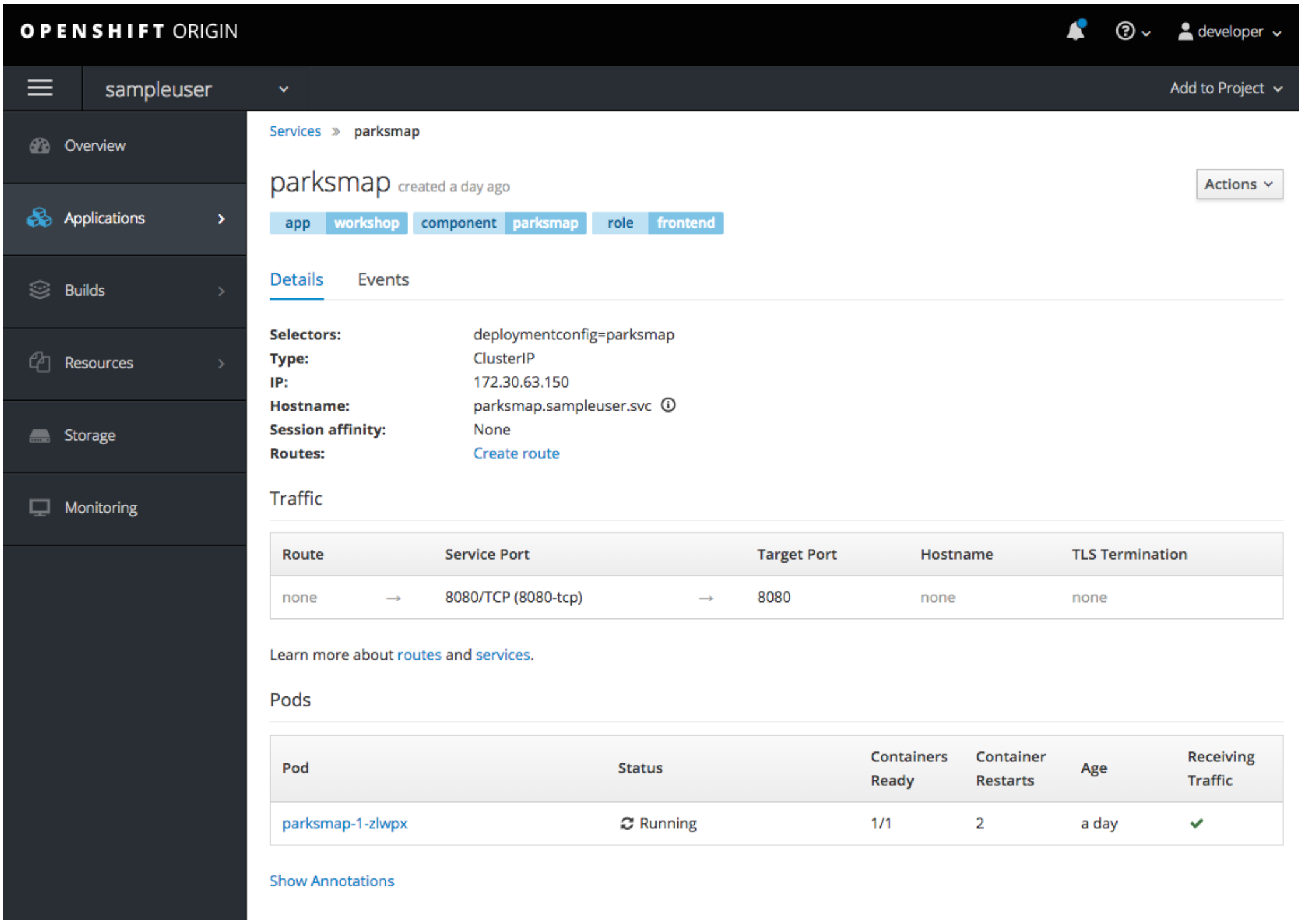
```
BASH
$ oc get service parksmap -o yaml
```

You should see output similar to the following:

```
BASH
apiVersion: v1
kind: Service
metadata:
  annotations:
    openshift.io/generated-by: OpenShiftWebConsole
  creationTimestamp: 2016-10-03T15:33:17Z
  labels:
    app: parksmap
  name: parksmap
  namespace: project-userXY
  resourceVersion: "6893"
  selfLink: /api/v1/namespaces/project-userXY/services/parksmap
  uid: b51260a9-897e-11e6-bdaa-2cc2602f8794
spec:
  clusterIP: 172.30.169.213
  ports:
  - name: 8080-tcp
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    deploymentconfig: parksmap
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

Take note of the `selector` stanza. Remember it.

Alternatively, you can use the web console to view information about the **Service**.



It is also of interest to view the YAML of the **Pod** to understand how OpenShift wires components together. For example, run the following command to get the name of your `parksmap` **Pod**:

```
$ oc get pods
```
BASH

You should see output similar to the following:

```
NAME                READY     STATUS    RESTARTS   AGE
parksmap-1-hx0kv    1/1       Running   0          3h
```
BASH

Now you can view the detailed data for your **Pod** with the following command:

```
$ oc get pod parksmap-1-hx0kv -o yaml
```
BASH

Under the `metadata` section you should see the following:

```
  labels:
    app: workshop
    component: parksmap
    deployment: parksmap-1
    deploymentconfig: parksmap
    role: frontend
```
BASH

- The **Service** has `selector` stanza that refers to `deploymentconfig=parksmap`.
- The **Pod** has multiple **Labels**:
  - `app=workshop`
  - `component=parksmap`
  - `role=frontend`
  - `deploymentconfig=parksmap`
  - `deployment=parksmap-1`

Labels are just key/value pairs. Any **Pod** in this **Project** that has a **Label** that matches the **Selector** will be associated with the **Service**. To see this in action, issue the following command:

```BASH
$ oc describe service parksmap
```

You should see something like the following output:

```BASH
Name:                   parksmap
Namespace:              project-userXY
Labels:                 app=workshop
                        component=parksmap
                        role=frontend
Selector:               deploymentconfig=parksmap
Type:                   ClusterIP
IP:                     172.30.169.213
Port:                   8080-tcp        8080/TCP
Endpoints:              10.1.2.5:8080
Session Affinity:       None
No events.
```

You may be wondering why only one end point is listed. That is because there is only one **Pod** currently running. In the next lab, we will learn how to scale an application, at which point you will be able to see multiple endpoints associated with the **Service**.

Scaling

Lab: Scaling and Self Healing

Background: Deployment Configurations and Replication Controllers

While **Services** provide routing and load balancing for **Pods**, which may go in and out of existence, **ReplicationControllers** (RC) are used to specify and then ensure the desired number of **Pods** (replicas) are in existence. For example, if you always want your application server to be scaled to 3 **Pods** (instances), a **ReplicationController** is needed. Without an RC, any **Pods** that are killed or somehow die/exit are not automatically restarted. **ReplicationControllers** are how OpenShift "self heals".

A **DeploymentConfiguration** (DC) defines how something in OpenShift should be deployed. From the deployments documentation (https://<url>/architecture/core_concepts/deployments.html#deployments-and-deployment-configurations):

> *Building on replication controllers, OpenShift adds expanded support for the software development and deployment lifecycle with the concept of deployments. In the simplest case, a deployment just creates a new replication controller and lets it start up pods. However, OpenShift deployments also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the replication controller.*

In almost all cases, you will end up using the **Pod**, **Service**, **ReplicationController** and **DeploymentConfiguration** resources together. And, in almost all of those cases, OpenShift will create all of them for you.

There are some edge cases where you might want some **Pods** and an **RC** without a **DC** or a **Service**, and others, so feel free to ask us about them after the labs.

Exercise: Exploring Deployment-related Objects

Now that we know the background of what a **ReplicatonController** and **DeploymentConfig** are, we can explore how they work and are related. Take a look at the **DeploymentConfig** (DC) that was created for you when you told OpenShift to stand up the `parksmap` image:

```BASH
$ oc get dc

NAME       REVISION   DESIRED   CURRENT   TRIGGERED BY
parksmap   1          1         1         config,image(parksmap:<version>)
```

To get more details, we can look into the **ReplicationController** (**RC**).

Take a look at the **ReplicationController** (RC) that was created for you when you told OpenShift to stand up the `parksmap` image:

```BASH
$ oc get rc

NAME          DESIRED   CURRENT   READY     AGE
parksmap-1    1         1         0         4h
```

This lets us know that, right now, we expect one **Pod** to be deployed (`Desired`), and we have one **Pod** actually deployed (`Current`). By changing the desired number, we can tell OpenShift that we want more or less **Pods**.

OpenShift's **HorizontalPodAutoscaler** effectively monitors the CPU usage of a set of instances and then manipulates the RCs accordingly.

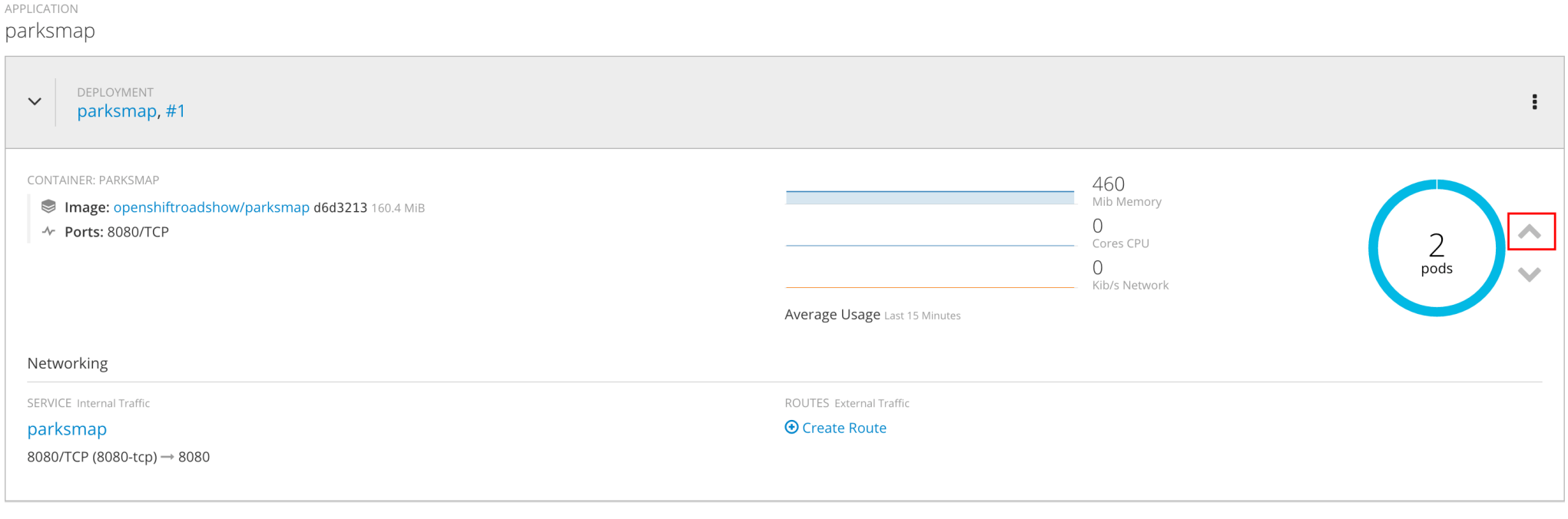
You can learn more about the CPU-based Horizontal Pod Autoscaler here (https://<url>/dev_guide/pod_autoscaling.html)

Exercise: Scaling the Application

Let's scale our parksmap "application" up to 2 instances. We can do this with the `scale` command. You could also do this by clicking the "up" arrow next to the **Pod** in the OpenShift web console on the overview page. It's your choice.

```
$ oc scale --replicas=2 dc/parksmap
```
BASH

You can also scale up to two pods by clicking the up arrow in the web console as shown in the following image:



To verify that we changed the number of replicas, issue the following command:

```
$ oc get rc

NAME          DESIRED   CURRENT   READY     AGE
parksmap-1    2         2         0         4h
```
BASH

You can see that we now have 2 replicas. Let's verify the number of pods with the `oc get pods` command:

```
$ oc get pods

NAME                READY     STATUS    RESTARTS   AGE
parksmap-1-8g6lb    1/1       Running   0          1m
parksmap-1-hx0kv    1/1       Running   0          4h
```
BASH

And lastly, let's verify that the **Service** that we learned about in the previous lab accurately reflects two endpoints:

```
$ oc describe svc parksmap
```
BASH

You will see something like the following output:

```
Name:           parksmap
Namespace:      project-userXY
Labels:         app=parksmap
Selector:       deploymentconfig=parksmap
Type:           ClusterIP
IP:         172.30.169.213
Port:           8080-tcp    8080/TCP
Endpoints:      10.1.0.5:8080,10.1.1.5:8080
Session Affinity:   None
No events.
```
BASH

Another way to look at a **Service**'s endpoints is with the following:

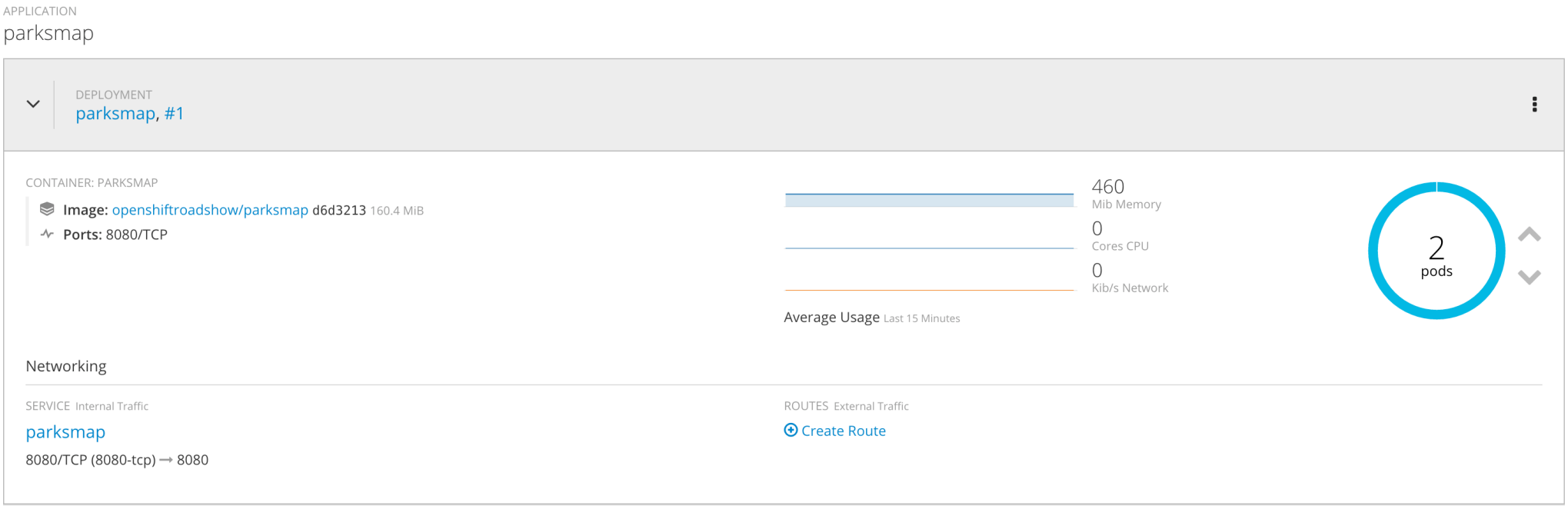
```
$ oc get endpoints parksmap
```
BASH

And you will see something like the following:

```
NAME       ENDPOINTS                                  AGE
parksmap   10.1.0.5:8080,10.1.1.5:8080                4h
```
BASH

Your IP addresses will likely be different, as each pod receives a unique IP within the OpenShift environment. The endpoint list is a quick way to see how many pods are behind a service.

You can also see that both **Pods** are running using the web console:



Overall, that's how simple it is to scale an application (**Pods** in a **Service**). Application scaling can happen extremely quickly because OpenShift is just launching new instances of an existing image, especially if that image is already cached on the node.

Application "Self Healing"

Because OpenShift's **RCs** are constantly monitoring to see that the desired number of **Pods** actually is running, you might also expect that OpenShift will "fix" the situation if it is ever not right. You would be correct!

Since we have two **Pods** running right now, let's see what happens if we "accidentally" kill one. Run the `oc get pods` command again, and choose a **Pod** name. Then, do the following:

```BASH
$ oc delete pod parksmap-1-hx0kv && oc get pods

pod "parksmap-1-h45hj" deleted
NAME               READY     STATUS              RESTARTS   AGE
parksmap-1-h45hj   1/1       Terminating         0          4m
parksmap-1-q4b4r   0/1       ContainerCreating   0          1s
parksmap-1-vdkd9   1/1       Running             0          32s
```

Did you notice anything? There is a container being terminated (the one we deleted), and there's a new container already being created.

Also, the names of the **Pods** are slightly changed. That's because OpenShift almost immediately detected that the current state (1 **Pod**) didn't match the desired state (2 **Pods**), and it fixed it by scheduling another **Pod**.

Additionally, OpenShift provides rudimentary capabilities around checking the liveness and/or readiness of application instances. If the basic checks are insufficient, OpenShift also allows you to run a command inside the container in order to perform the check. That command could be a complicated script that uses any installed language.

Based on these health checks, if OpenShift decided that our `parksmap` application instance wasn't alive, it would kill the instance and then restart it, always ensuring that the desired number of replicas was in place.

More information on probing applications is available in the Application Health (https://<url>/dev_guide/application_health.html) section of the documentation.

Exercise: Scale Down

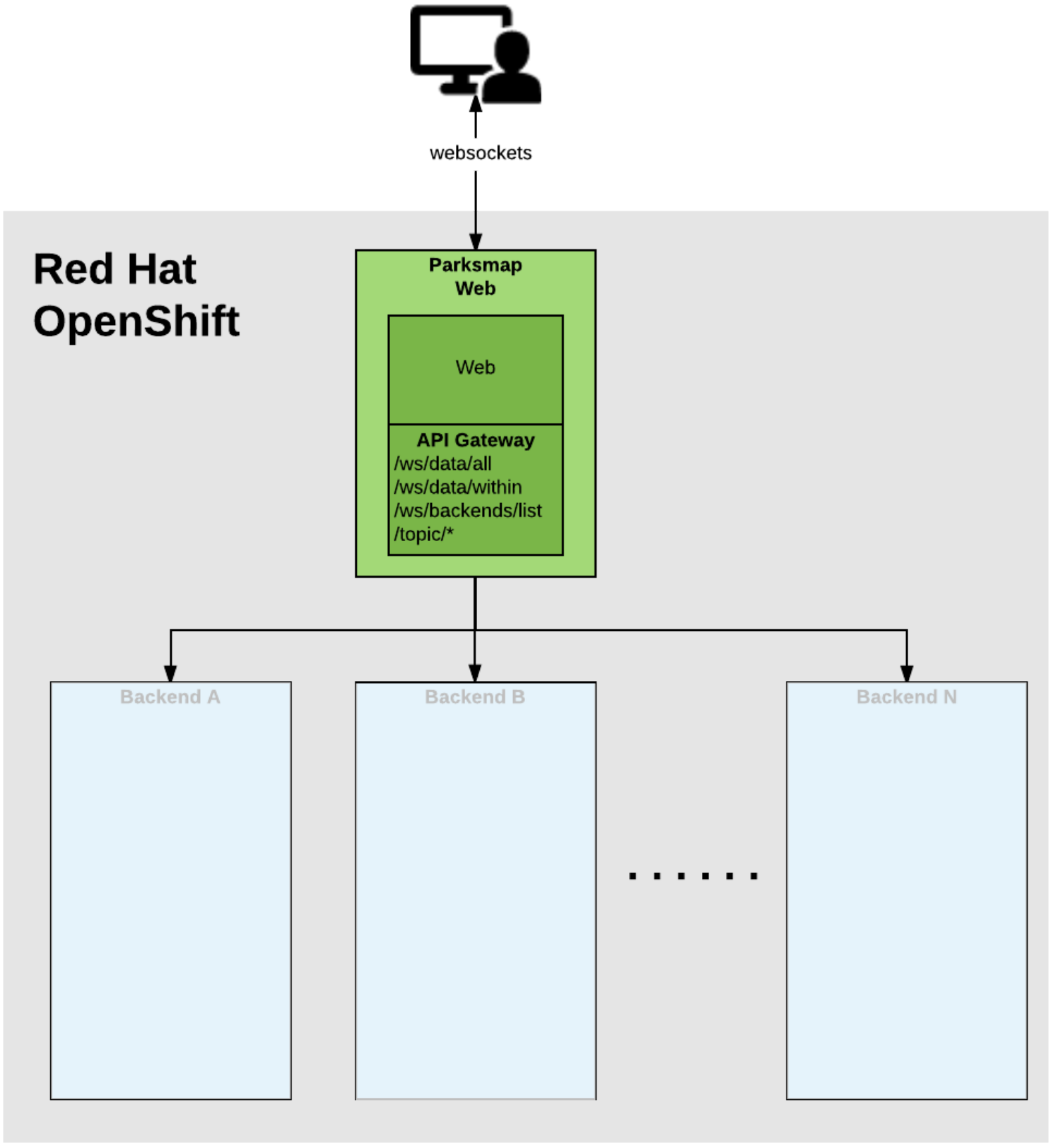
Before we continue, go ahead and scale your application down to a single instance. Feel free to do this using whatever method you like.

Creating Routes

Lab: Creating Routes by Exposing Services

Application description

In this lab, we're going to make our application visible to the end users, so they can access it.



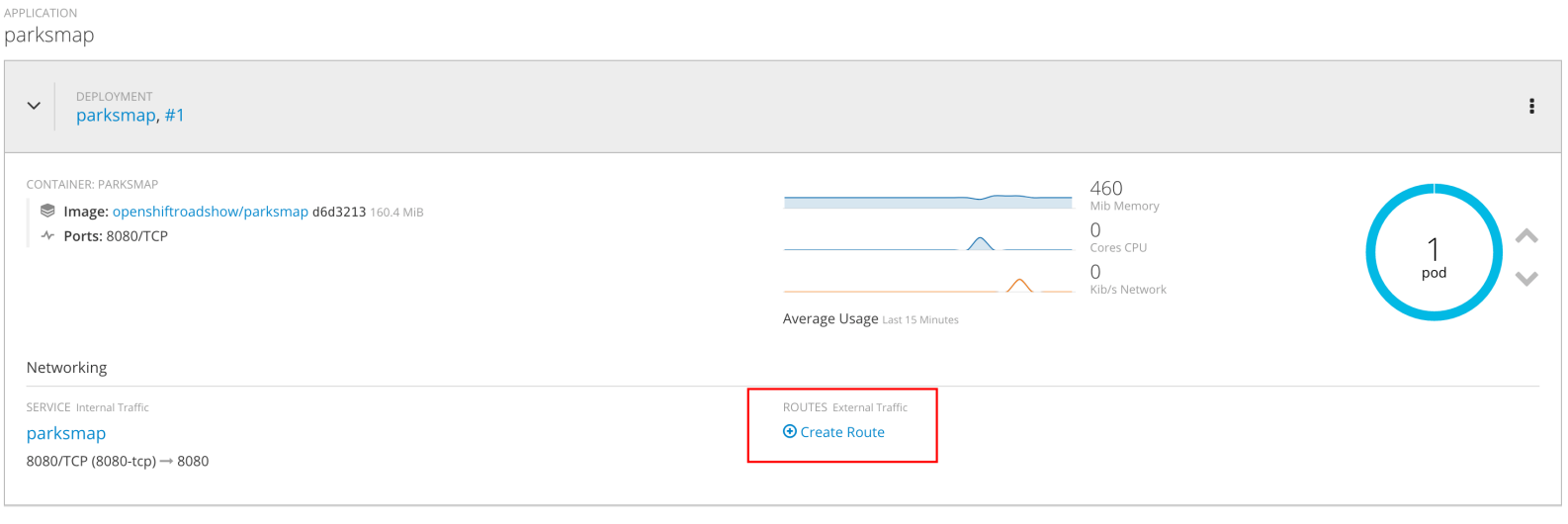
Background: Routes

While **Services** provide internal abstraction and load balancing within an OpenShift environment, sometimes clients (users, systems, devices, etc.) **outside** of OpenShift need to access an application. The way that external clients are able to access applications running in OpenShift is through the OpenShift routing layer. And the data object behind that is a **Route**.

The default OpenShift router (HAProxy) uses the HTTP header of the incoming request to determine where to proxy the connection. You can optionally define security, such as TLS, for the **Route**. If you want your **Services**, and, by extension, your **Pods**, to be accessible to the outside world, you need to create a **Route**.

Exercise: Creating a Route

Fortunately, creating a **Route** is a pretty straight-forward process. You simply `expose` the **Service** via the command line. Or, via the web console, just click the "Create Route" button associated with the service and choose the defaults as highlighted in the following image:



First we want to verify that we don't already have any existing routes:

```
$ oc get routes
No resources found.
```

Now we need to get the **Service** name to expose:

```
$ oc get services

NAME       CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
parksmap   172.30.169.213   <none>        8080/TCP   5h
```

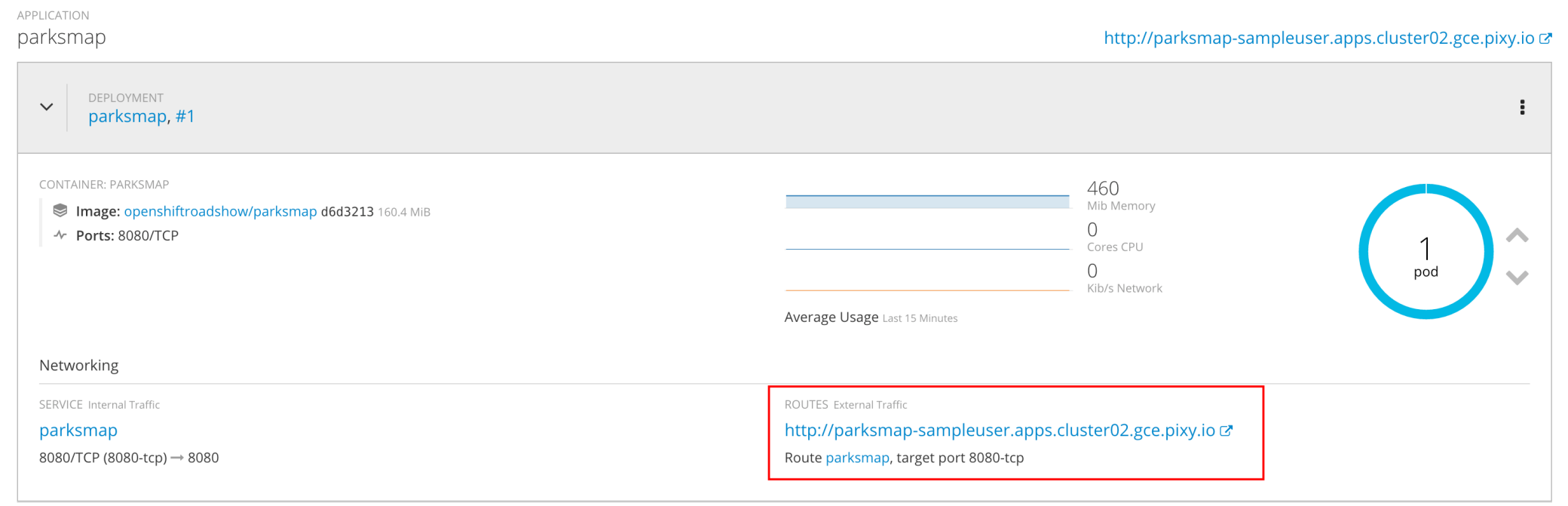
Once we know the **Service** name, creating a **Route** is a simple one-command task:

```
$ oc expose service parksmap
route "parksmap" exposed
```

Verify the **Route** was created with the following command:

```
$ oc get route
NAME       HOST/PORT                                       PATH      SERVICES   PORT       TERMINATION
parksmap   parksmap-project-userXY.apps.10.2.2.2.xip.io              parksmap   8080-tcp
```

You can also verify the **Route** by looking at the project in the OpenShift web console:



Pretty nifty, huh? This application is now available at the URL shown in the web console. Click the link and you will see:

Exploring OpenShift's Logging Capabilities

Lab: Exploring OpenShift’s Logging Capabilities

OpenShift provides some convenient mechanisms for viewing application logs. First and foremost is the ability to examine a **Pod**'s logs directly from the web console or via the command line.

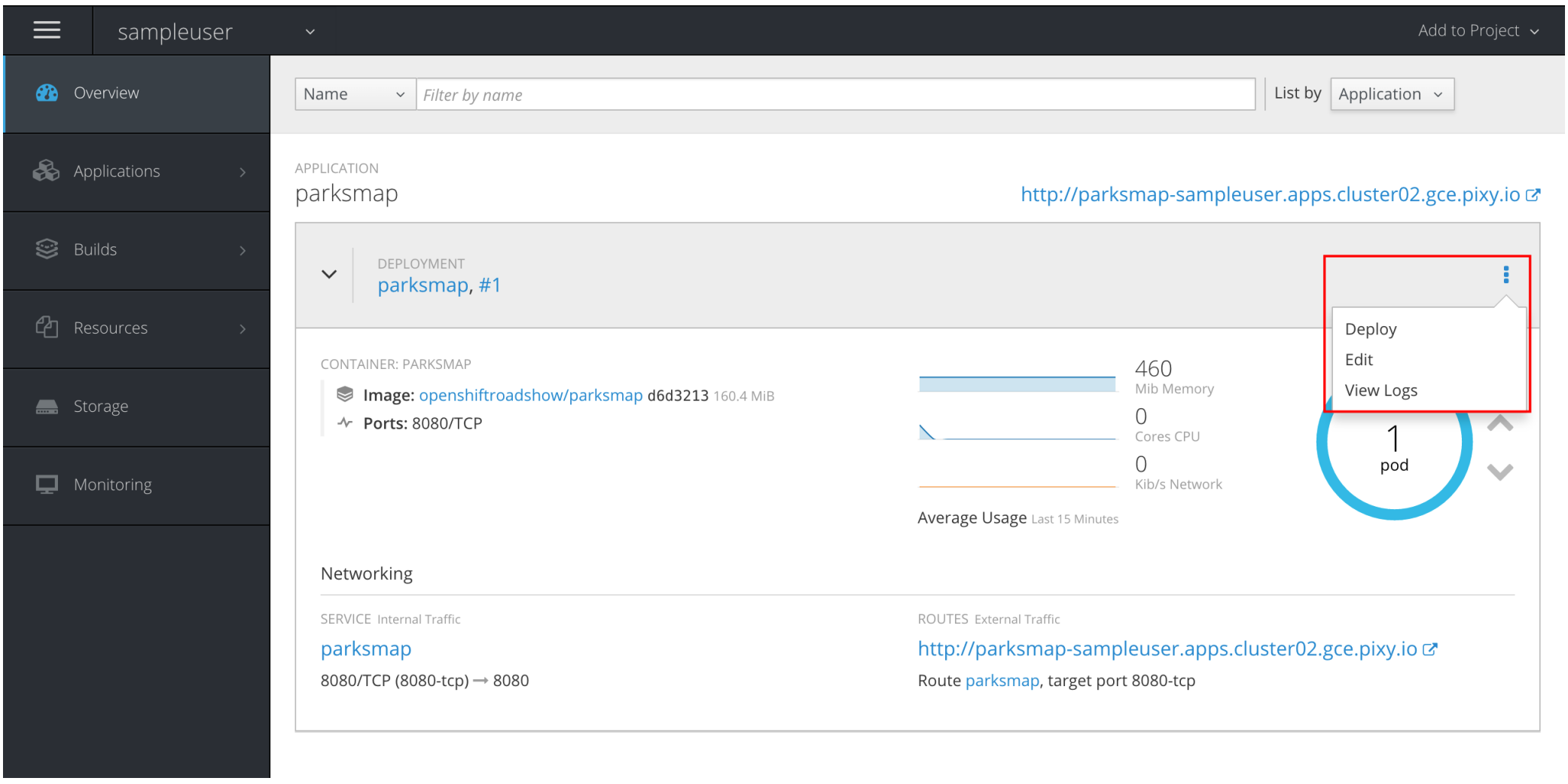
Background: Container Logs

OpenShift is constructed in such a way that it expects containers to log all information to `STDOUT`. In this way, both regular and error information is captured via standardized Docker mechanisms. When exploring the **Pod**'s logs directly, you are essentially going through the Docker daemon to access the container's logs, through OpenShift's API. Neat!

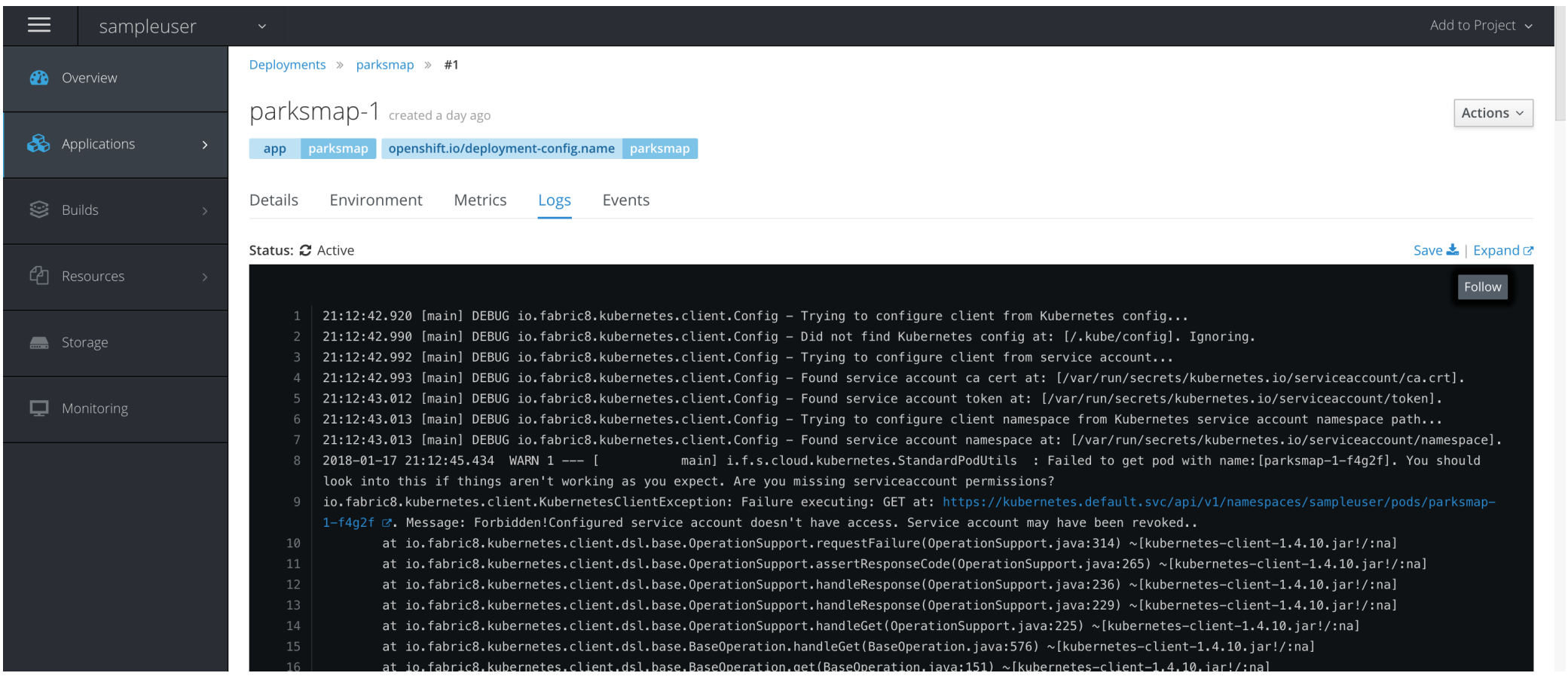
In some cases, applications may not have been designed to send all of their information to `STDOUT` and `STDERR`. In many cases, multiple local log files are used. While OpenShift cannot parse any information from these files, nothing prevents them from being created, either. In other cases, log information is sent to some external system. Here, too, OpenShift does not prohibit these behaviors. If you have an application that does not log to `STDOUT`, either because it already sends log information to some "external" system or because it writes various log information to various files, fear not.

Exercise: Examining Logs

Since we already deployed our application, we can take some time to examine its logs. In the web console `Overview` page, click on the kebab menu (three squares) at the far right of the deployment and then click on the `View Logs` item.



You should see a nice view of the **Pod**'s logs:



It appears there are some errors in the log, and that's OK. We'll remedy those in a little bit.

You also have the option of viewing logs from the command line. Get the name of your **Pod**:

```BASH
$ oc get pods
NAME                READY     STATUS    RESTARTS   AGE
parksmap-1-hx0kv    1/1       Running   0          5h
```

And then use the `logs` command to view this **Pod**'s logs:

```
$ oc logs parksmap-1-hx0kv
```

You will see all of the application logs scroll on your screen:

```
15:34:25.844 [main] DEBUG io.fabric8.kubernetes.client.Config - Trying to configure client from Kubernetes config...
15:34:25.937 [main] DEBUG io.fabric8.kubernetes.client.Config - Did not find Kubernetes config at: [/.kube/config]. Ignoring.
15:34:25.937 [main] DEBUG io.fabric8.kubernetes.client.Config - Trying to configure client from service account...
15:34:25.938 [main] DEBUG io.fabric8.kubernetes.client.Config - Found service account ca cert at:
[/var/run/secrets/kubernetes.io/serviceaccount/ca.crt].
15:34:25.960 [main] DEBUG io.fabric8.kubernetes.client.Config - Found service account token at:
[/var/run/secrets/kubernetes.io/serviceaccount/token].
15:34:25.961 [main] DEBUG io.fabric8.kubernetes.client.Config - Trying to configure client namespace from Kubernetes service account namespace
path...
15:34:25.962 [main] DEBUG io.fabric8.kubernetes.client.Config - Found service account namespace at:
[/var/run/secrets/kubernetes.io/serviceaccount/namespace].
....
```

You may have noticed an error that mentions a service account. What's that? Never fear, we will cover that shortly.

Role-Based Access Control

Lab: OpenShift Role-Based Access Control

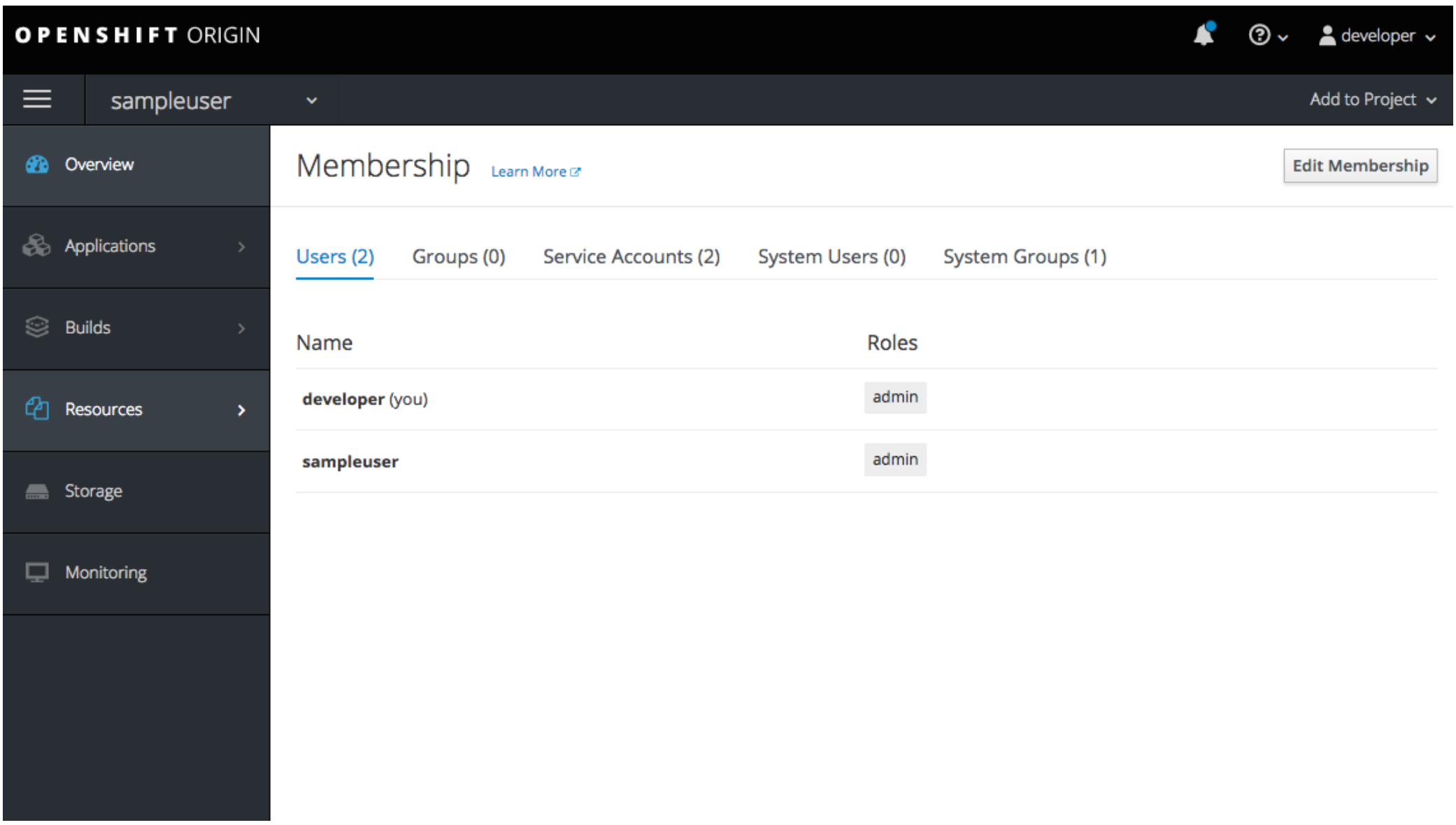
Background

Almost every interaction with an OpenShift environment that you can think of requires going through the master's API. All API interactions are both authenticated (AuthN - who are you?) and authorized (AuthZ - are you allowed to do what you are asking?).

Just like a user has permissions (AuthZ), sometimes we may wish for non-users to be able to perform actions against the API. These "non-users" are referred to as service accounts.

OpenShift automatically creates a few special service accounts in every project. The **default** service account has its credentials automatically injected into every pod that is launched. By changing the permissions for that service account, we can do interesting things.

You can view current permissions in the web UI, by navigating to "Resources → Membership" page.



Exercise: Grant Service Account View Permissions

The parksmap application wants to talk to the OpenShift API to learn about other **Pods**, **Services**, and resources within the **Project**. You'll soon learn why!

```
$ oc project project-userXY
```

Then:

```
$ oc policy add-role-to-user view -z default
```

The `oc policy` command above is giving a defined *role* (`view`) to a user. But we are using a special flag, `-z`. What does this flag do? From the `-h` output:

```BASH
-z, --serviceaccount=[]: service account in the current namespace to use as a user
```

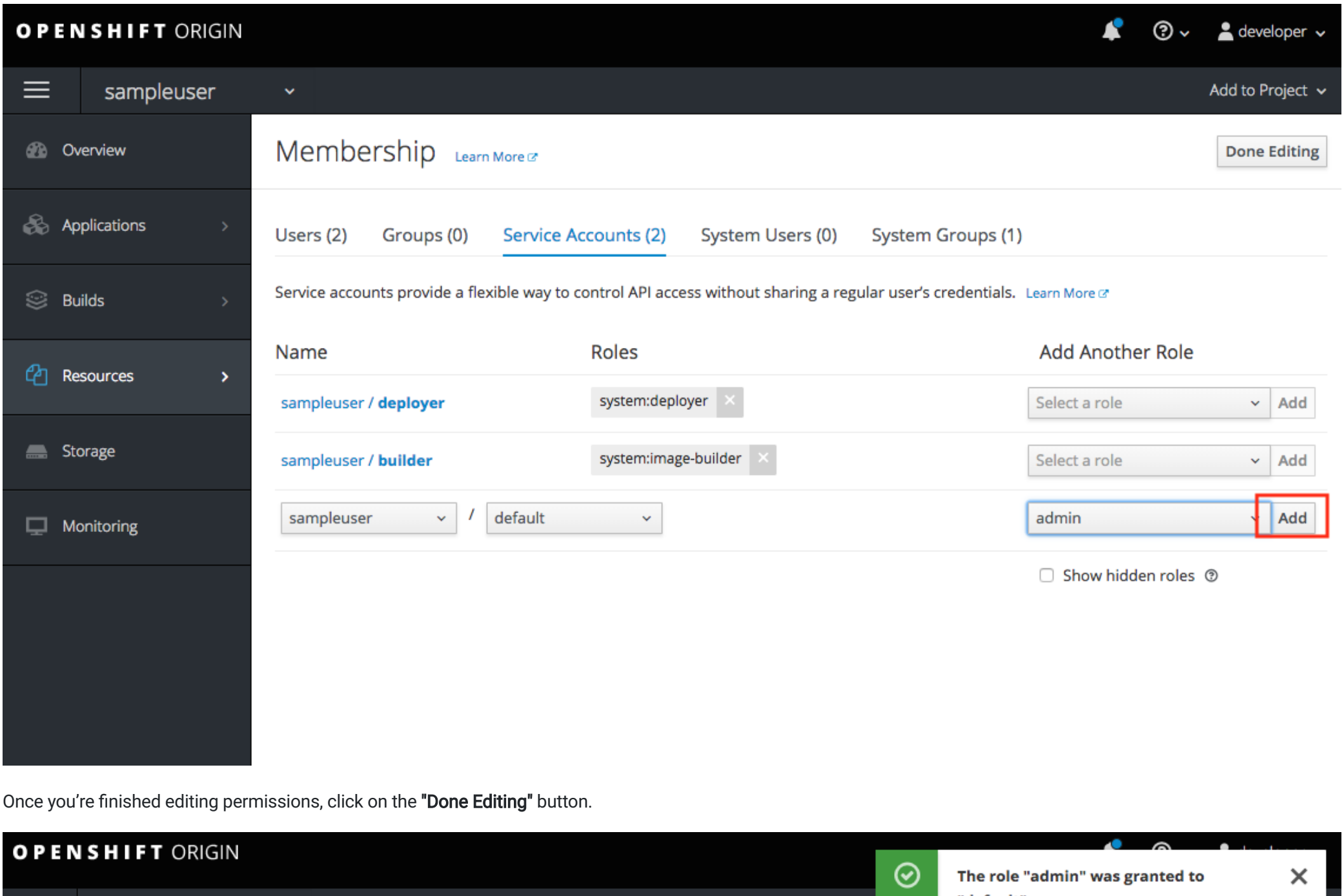
The `-z` syntax is a special one that saves us from having to type out the entire string, which, in this case, is `system:serviceaccount:project-userXY:default`. It's a nifty shortcut.

> The `-z` flag will only work for service accounts in the **current** project. If you're referring to a service account in a different project, use the `-n <project>`switch.

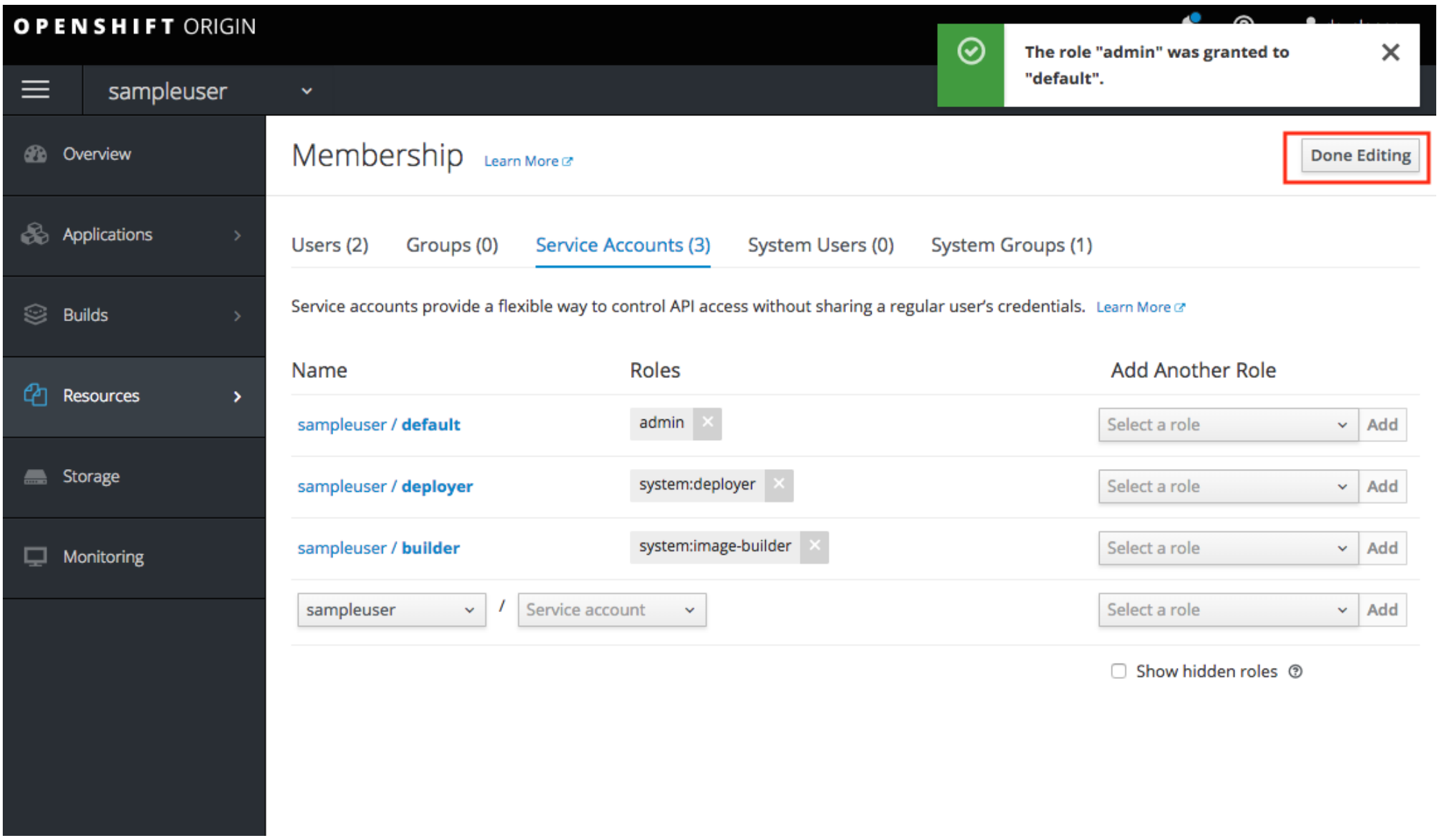
Now that the default **Service Account** has **view** access, it can query the API to see what resources are within the **Project**. This also has the added benefit of suppressing the error message! Although, in reality, we fixed the application.

Another way you could have done the same is by using the OpenShift console. Once you're on the **"Applications -> Membership"** page, click on Service Accounts, and then on **"Edit Membership"** button.

Add the admin role to the **project-userXY/default** Service Account, and then click the **"Add"** button.



Once you're finished editing permissions, click on the **"Done Editing"** button.



Exercise: Grant User View Permissions

If you create a project, you are that project's administrator. This means that you can grant access to other users, too. If you like, give your neighbor view access to your project using the following command:

```BASH
$ oc policy add-role-to-user view userXY
```

Have them go to the project view by clicking the **Projects** button and verify that they can see your project and its resources. This type of arrangement (view but not edit) might be ideal for a developer getting visibility into a production application's project.

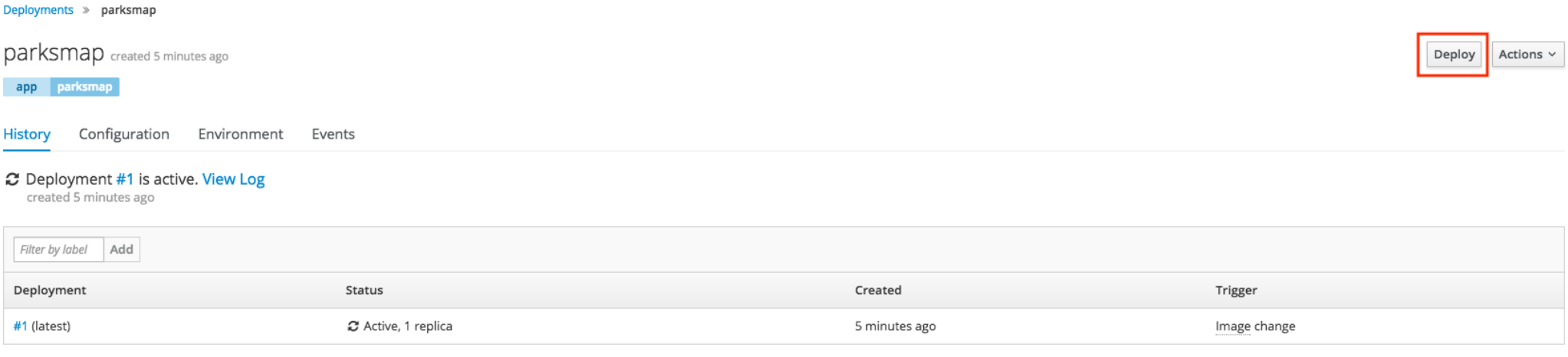
Exercise: Redeploy Application

One more step is required. We need to re-deploy the parks map application because it's given up trying to query the API.

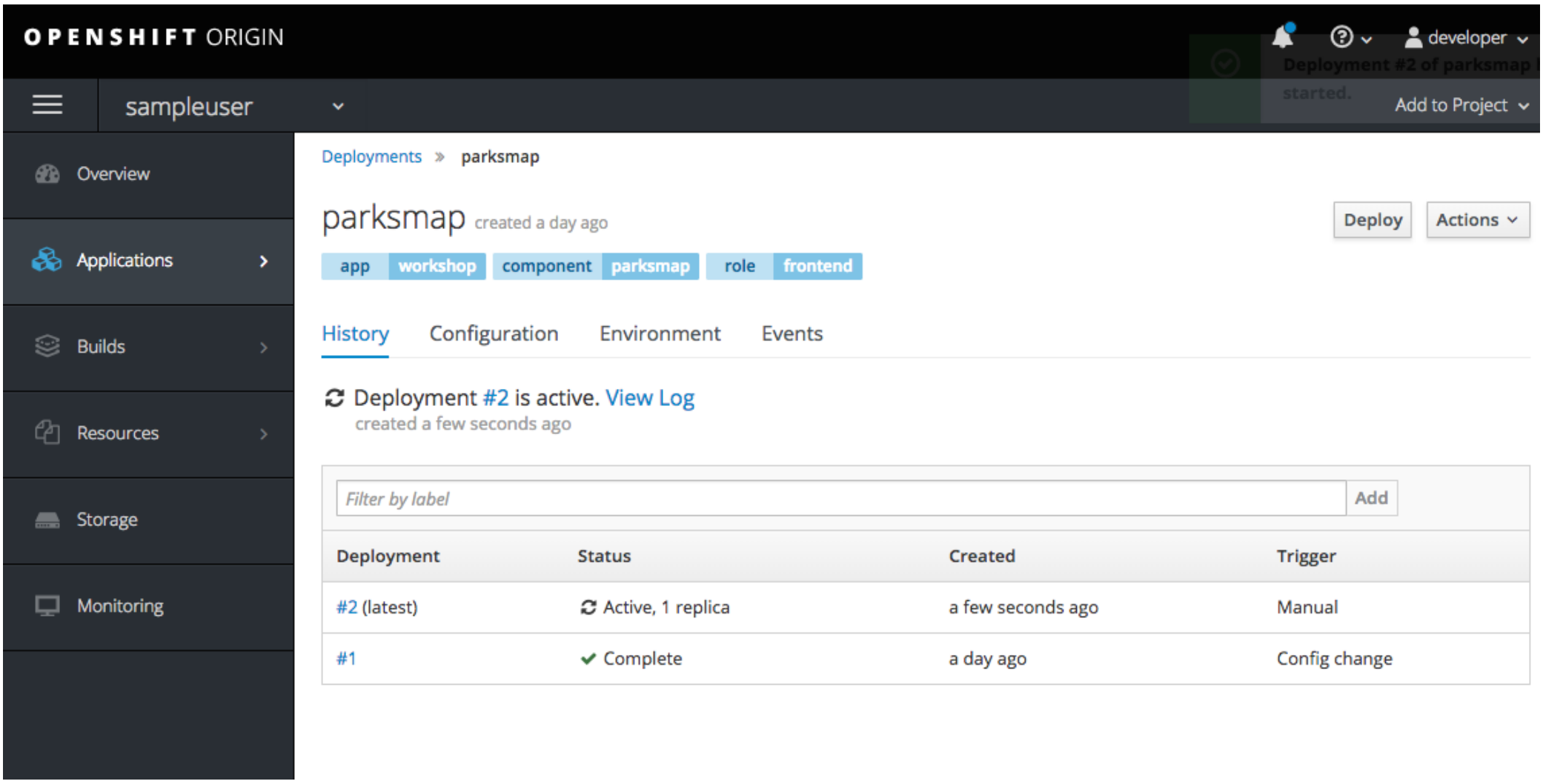
This time we'll use the web console. Find your `project-userXY` project, and then click "Applications" and then "Deployments". You'll see your only application, `parksmap`, listed. Click that.

The deployment screen tells you a lot about how the application will be deployed. At the top right, there is a button labeled "Deploy". This button will cause a new deployment (which you know creates a new **ReplicationController**, right?).

Click it.



You'll see that a new deployment is immediately started. Return to the overview page and watch it happen. You might not be fast enough! But it will be in the list of deployments.



If you look at the logs for the application now, you should see no errors. That's great.

Remote Operations

Lab: Remote Operations

Background

Containers are treated as immutable infrastructure and therefore it is generally not recommended to modify the content of a container through SSH or running custom commands inside the container. Nevertheless, in some use-cases, such as debugging an application, it might be beneficial to get into a container and inspect the application.

Exercise: Remote Shell Session to a Container Using the CLI

OpenShift allows establishing remote shell sessions to a container without the need to run an SSH service inside each container. In order to establish an interactive session inside a container, you can use the `oc rsh` command. First get the list of available pods:

```
$ oc get pods
```

You should an output similar to the following:

```
NAME                READY     STATUS    RESTARTS   AGE
parksmap-2-tegp4    1/1       Running   0          2m
```

Now you can establish a remote shell session into the pod by using the pod name:

```
$ oc rsh parksmap-2-tegp4
```

You would see the following output:

```
sh-4.2$
```

The default shell used by `oc rsh` is `/bin/sh`. If the deployed container does not have **sh** installed and uses another shell, (e.g. **A Shell**) the shell command can be specified after the pod name in the issued command.

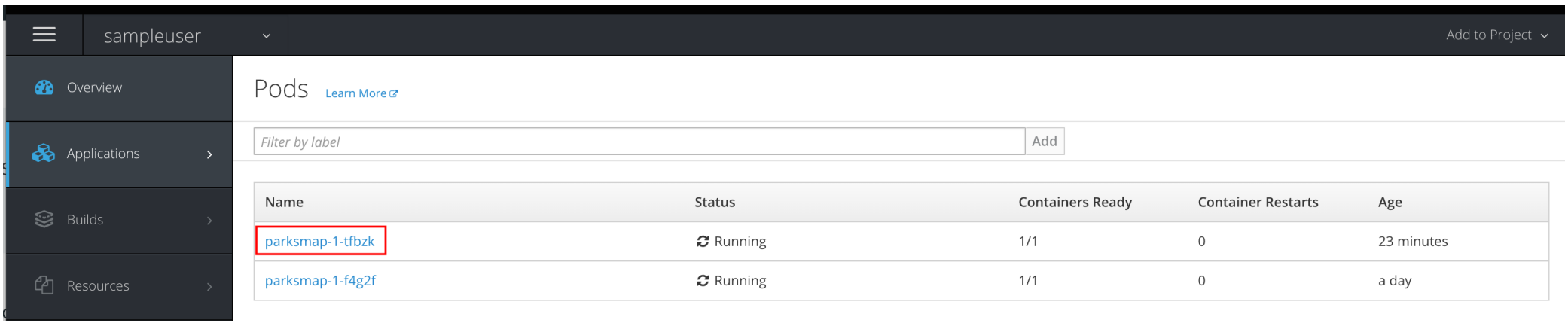
Run the following command to list the files in the top folder:

```
 $ ls /

anaconda-post.log  bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  parksmap.jar  proc  root  run  sbin  srv  sys  tmp  usr  var
```

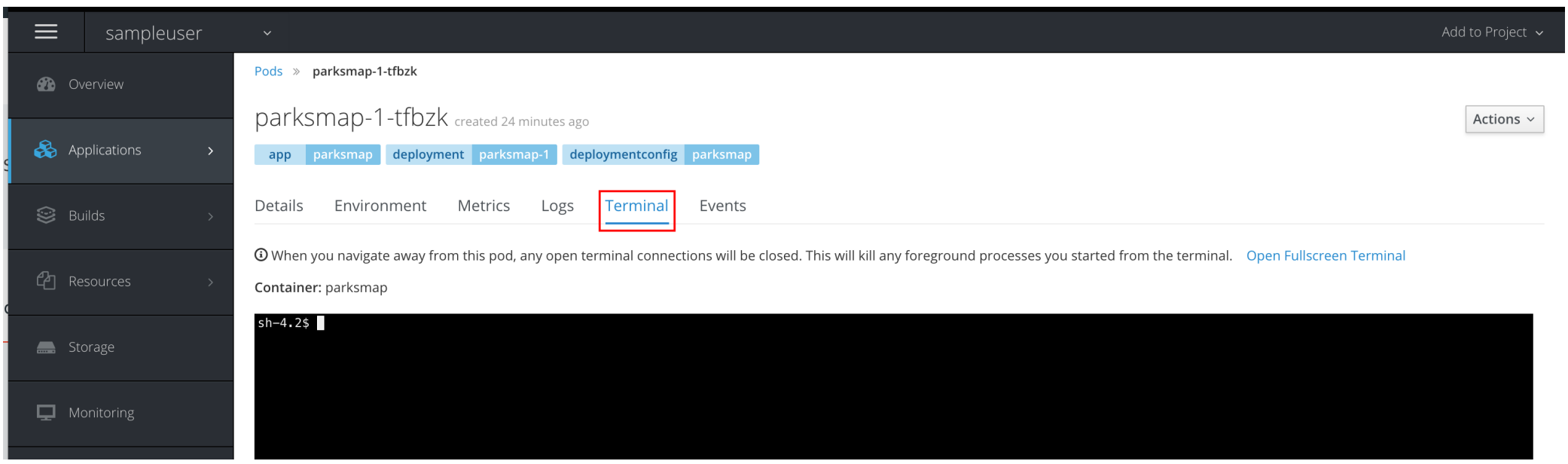
Exercise: Remote Shell Session to a Container Using the Web Console

The OpenShift Web Console also provides a convenient way to access a terminal session on the container without having to use the CLI.

In order to access a pods terminal via the Web Console, click on "Applications", then "Pods" and finally the pod name as shown in the following image:



Once you are viewing the information for the selected pod, click on the "Terminal" tab to open up a shell session.



Pretty nifty! Go ahead and execute the same commands you did when using the CLI to see how the Web Console based terminal behaves.

Exercise: Execute a Command in a Container

In addition to remote shell, it is also possible to run a command remotely in an already running container using the `oc exec` command. This does not require that a shell is installed, but only that the desired command is present and in the executable path.

In order to show just the JAR file, run the following:

```
$ oc exec parksmap-2-tegp4 -- ls -l /parksmap.jar
```

You would see something like the following:

```
-rw-r--r--. 1 root root 21753918 Nov 23 15:54 /parksmap.jar
```

The `--` syntax in the `oc exec` command delineates where exec's options end and where the actual command to execute begins. Take a look at `oc exec --help` for more details.

You can also specify the shell commands to run directly with the **oc rsh** command:

```
$ oc rsh parksmap-2-tegp4 whoami
```

You would see something like:

```
 whoami: cannot find name for user ID 1000060000
error: error executing remote command: error executing command in container: Error executing in Docker Container: 1
```

It is important to understand that, for security reasons, OpenShift does not run Docker containers as the user specified in the Dockerfile by default. In fact, when OpenShift launches a container its user is actually randomized.

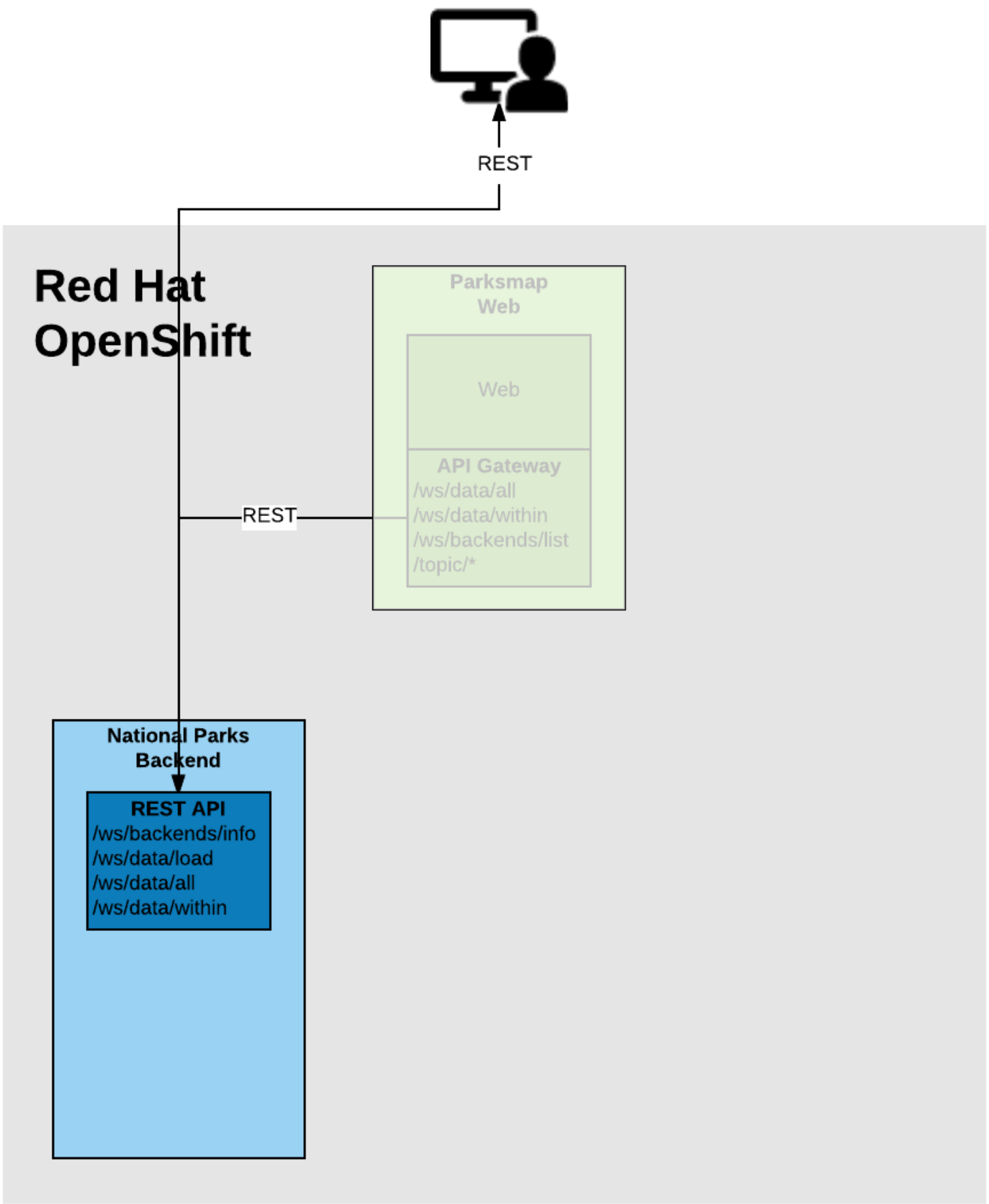
If you want or need to allow OpenShift users to deploy Docker images that do expect to run as root (or any specific user), a small configuration change is needed. You can learn more about the Docker guidelines (https://<url>/creating_images/guidelines.html) for OpenShift, or you can look at the section on enabling images to run with a USER in the dockerfile (https://<url>/admin_guide/manage_scc.html#enable-images-to-run-with-user-in-the-dockerfile).

Deploying Java Code

Lab: Deploying Java Code

Application Description

In this lab, we're going to deploy a backend service, developed in Java programming language that will expose 2 main REST endpoints to the visualizer application (`parksmap` web component that was deployed in the previous labs). The application will query for national parks information (including it's coordinates) that is stored in a Mongo database. This application will also provide an external access point, so that the API provided can be directly used by the end user.



Background: Source-to-Image (S2I)

In a previous lab, we learned how to deploy a pre-existing Docker-formatted image. Now we will expand on that by learning how OpenShift builds Docker images using source code from an existing repository. This is accomplished using the Source-to-Image project.

Source-to-Image (S2I) (https://github.com/openshift/source-to-image) is a open source project sponsored by Red Hat that has the following goal:

```
 Source-to-image (S2I) is a tool for building reproducible Docker images. S2I
produces ready-to-run images by injecting source code into a Docker image and
assembling a new Docker image which incorporates the builder image and built
source. The result is then ready to use with docker run. S2I supports
incremental builds which re-use previously downloaded dependencies, previously
built artifacts, etc.
```

OpenShift is S2I-enabled and can use S2I as one of its build mechanisms (in addition to building Docker images from Dockerfiles, and "custom" builds).

OpenShift runs the S2I process inside a special **Pod**, called a Build Pod, and thus builds are subject to quotas, limits, resource scheduling, and other aspects of OpenShift.

A full discussion of S2I is beyond the scope of this class, but you can find more information about it either in the OpenShift S2I documentation (https://<url>/creating_images/s2i.html) or on GitHub (https://github.com/openshift/source-to-image). The only key concept you need to remember about S2I is that it's magic.

Exercise: Creating a Java application

The backend service that we will be deploying as part of this exercise is called `nationalparks`. This is a Java Spring Boot application that performs 2D geo-spatial queries against a MongoDB database to locate and return map coordinates of all National Parks in the world. That was just a fancy way of saying that we are going to deploy a webservice that returns a JSON list of places.

Add to Project

Because the `nationalparks` component is a back-end to serve data that our existing front-end (parksmap) will consume, we are going to build it inside the existing project that we have been working with. To illustrate how you can interact with OpenShift via the CLI or the Web Console, we will deploy the nationalparks component using the web console.

Using application code on embedded Git server

OpenShift can work with any accessible Git repository. This could be GitHub, GitLab, or any other server that speaks Git. You can even register webhooks in your Git server to initiate OpenShift builds triggered by any update to the application code!

The repository that we are going to use is already cloned in the internal GitLab repository and located at the following URL:

```BASH
http://<gitlab>.apps.10.2.2.2.xip.io/userXY/nationalparks/tree/<version>
```

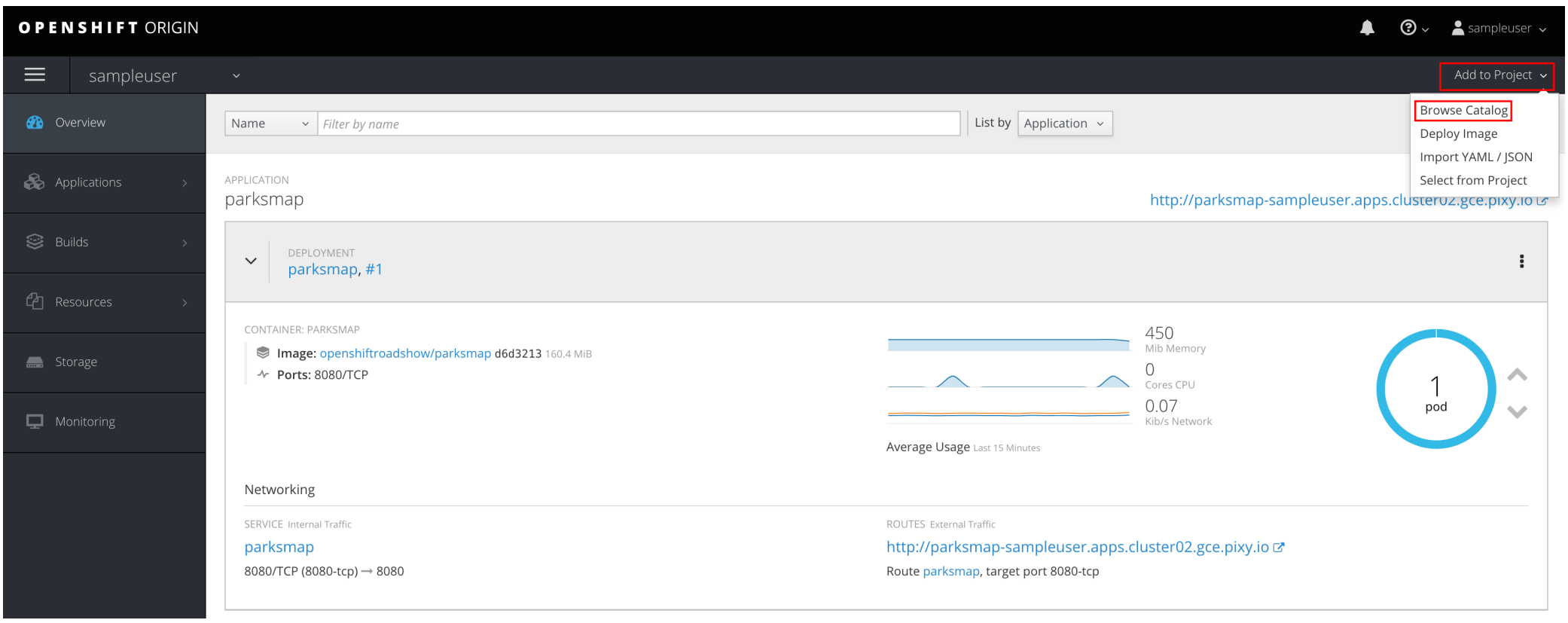
Your GitLab credentials are: **userXY/<password>**

Later in the lab, we want you to make a code change and then rebuild your application. This is a fairly simple Spring framework Java application.

Build the Code on OpenShift

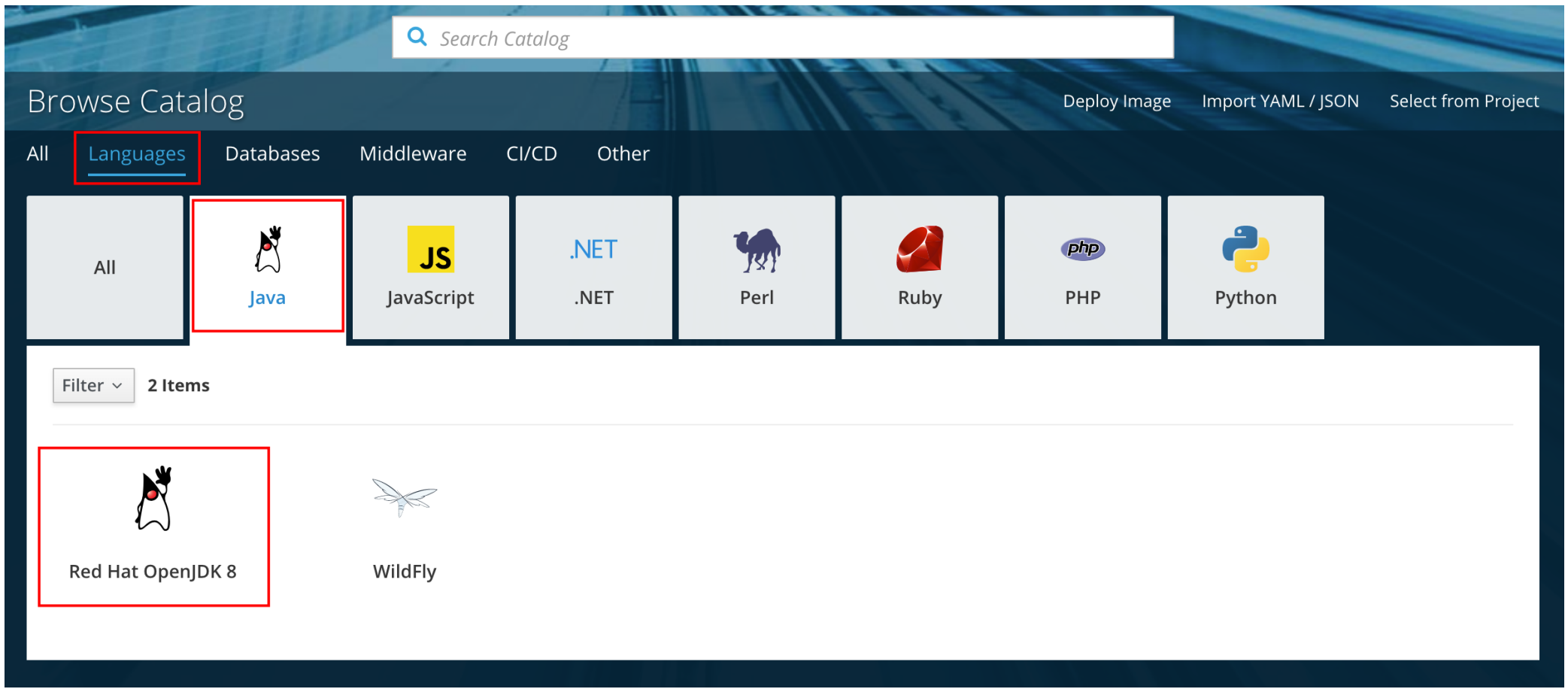
Similar to how we used "Add to project" before with a Docker-formatted image, we can do the same for specifying a source code repository. Since for this lab you have your own git repository, let's use it with a simple Java S2I image.

In the OpenShift web console, find your `project-userXY` project, and then click the **"Add to Project"** button and then the browse catalog link as highlighted in the following image:



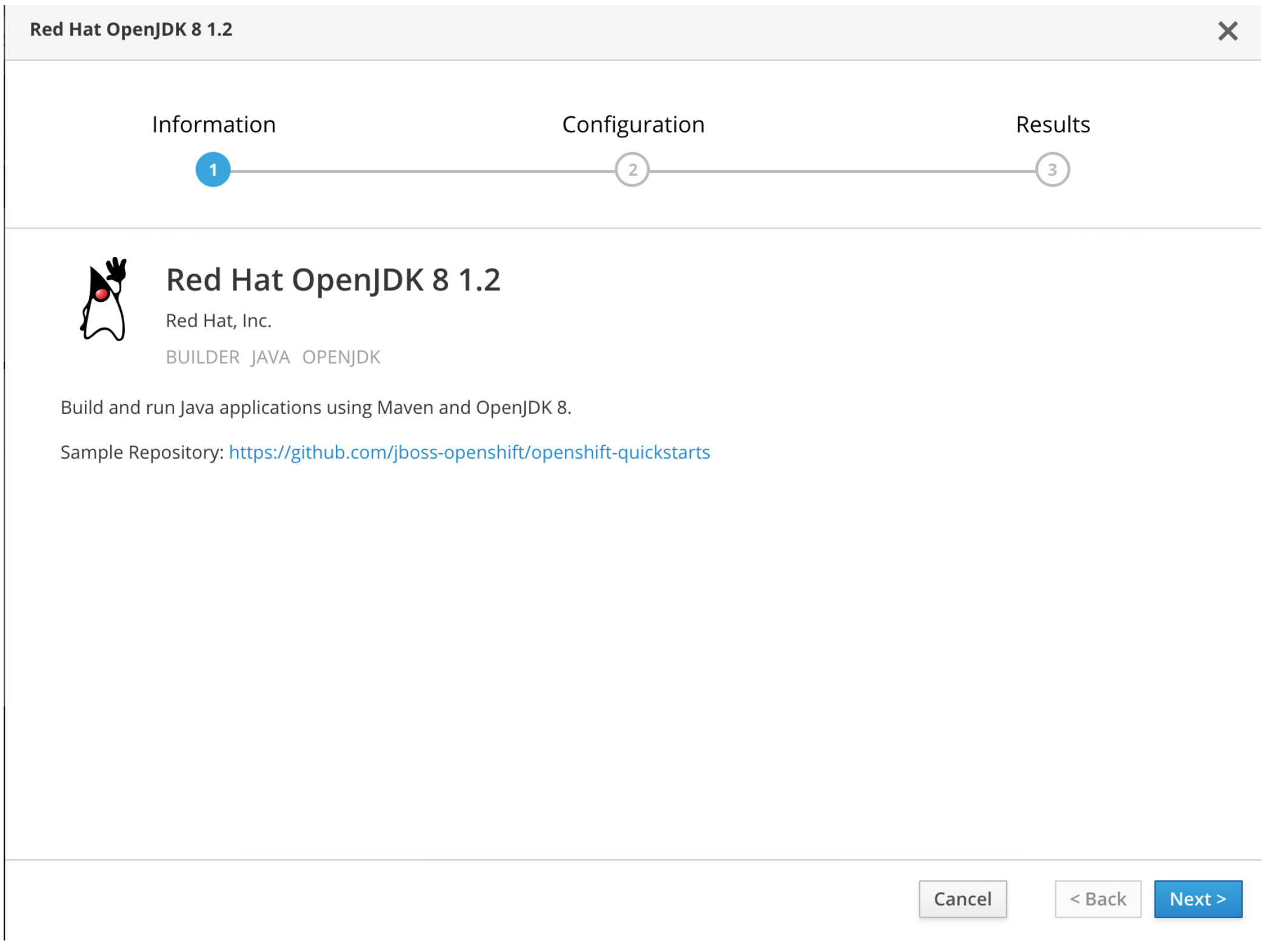
This is the service catalog which allows a user to select components they want to add to their application. In this case, we are using Spring Boot so we want to select a JDK without an application server.

Select "Languages" at the top of the service catalog, then "Java", and finally "Red Hat OpenJDK" as shown in the following image:



You could have also typed *"jdk"* into the search box, and then selected the item titled `Red Hat OpenJDK 8`.

After you click `Red Hat OpenJDK 8`, a dialog is presented as shown in the following image:

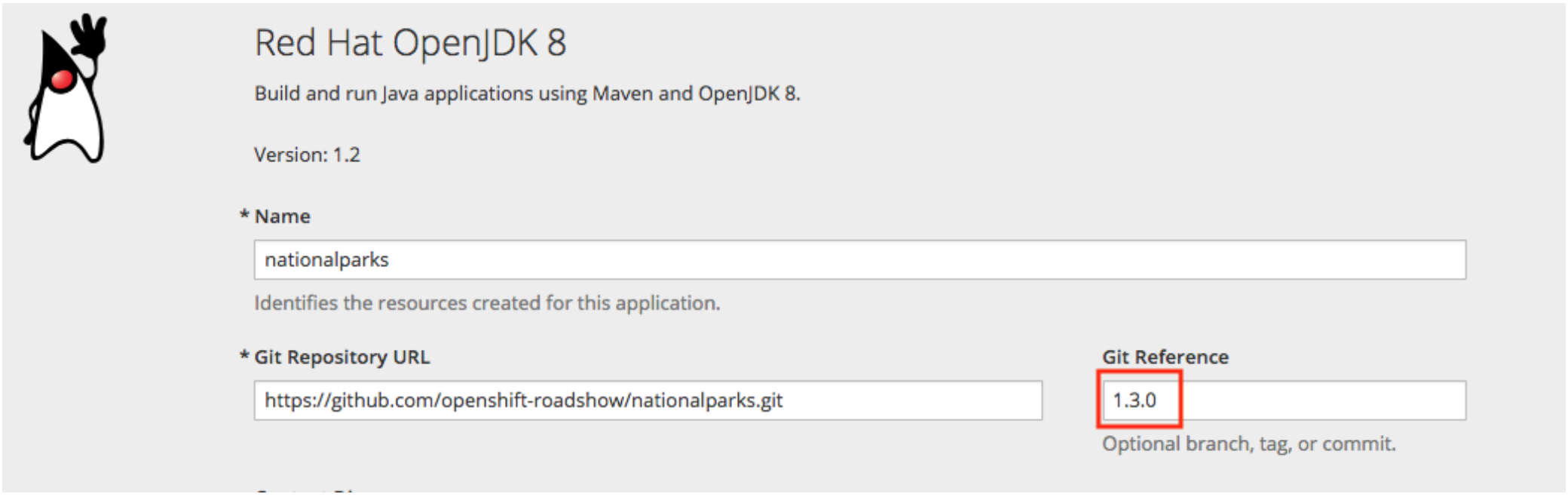


Click on the "Next" button and then enter a name and a Git repository URL. For the name, enter `nationalparks`, and for the Git repository URL, enter:

```
http://<gitlab>.apps.10.2.2.2.xip.io/userXY/nationalparks.git
```

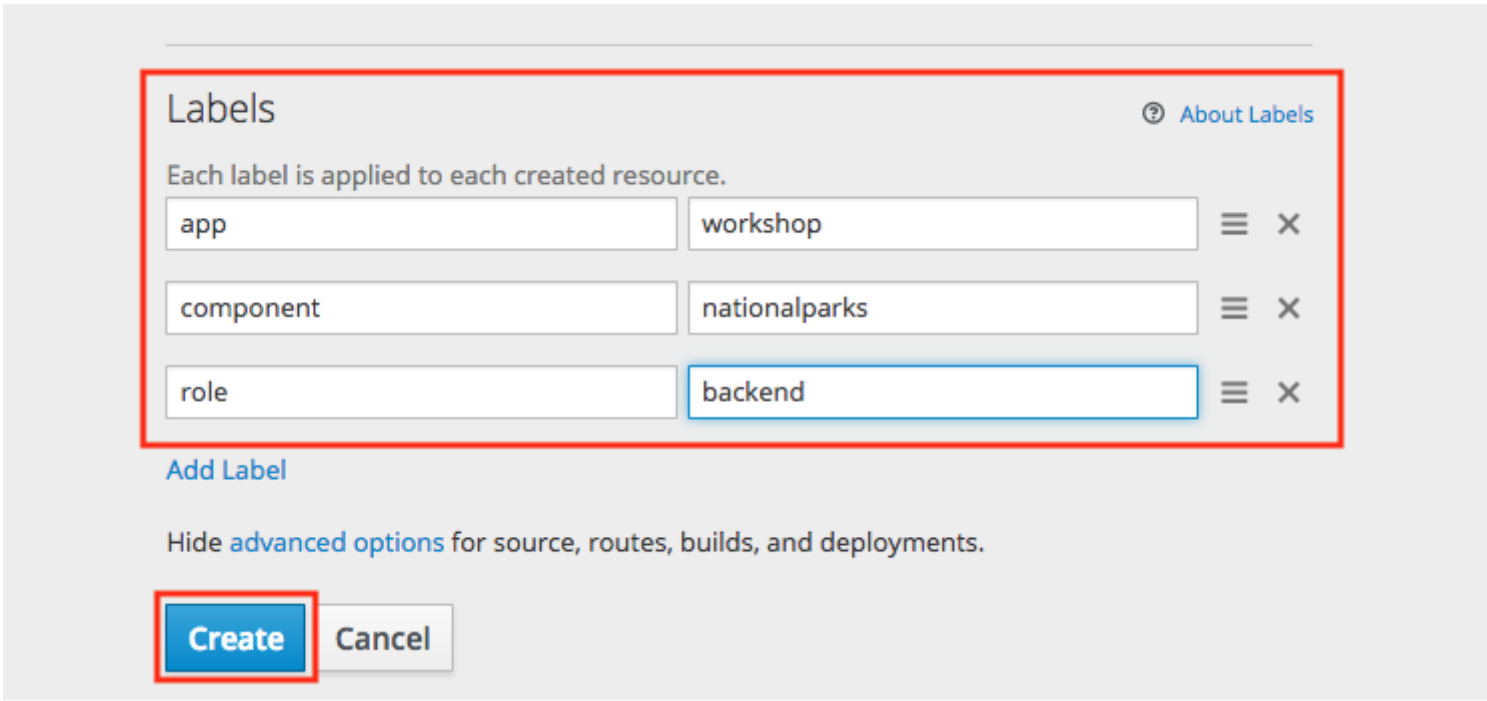
All of these runtimes shown are made available via **Templates** and **ImageStreams**, which will be discussed in a later lab.

These labs were written against specific points in time for these applications. With Git as our version control system (VCS), we are using the concept of **Branches/Tags**. Click on **Advanced Options**. In the **Git Reference** field enter "**<version>**". This will cause the S2I process to grab that specific tag in the code repository.

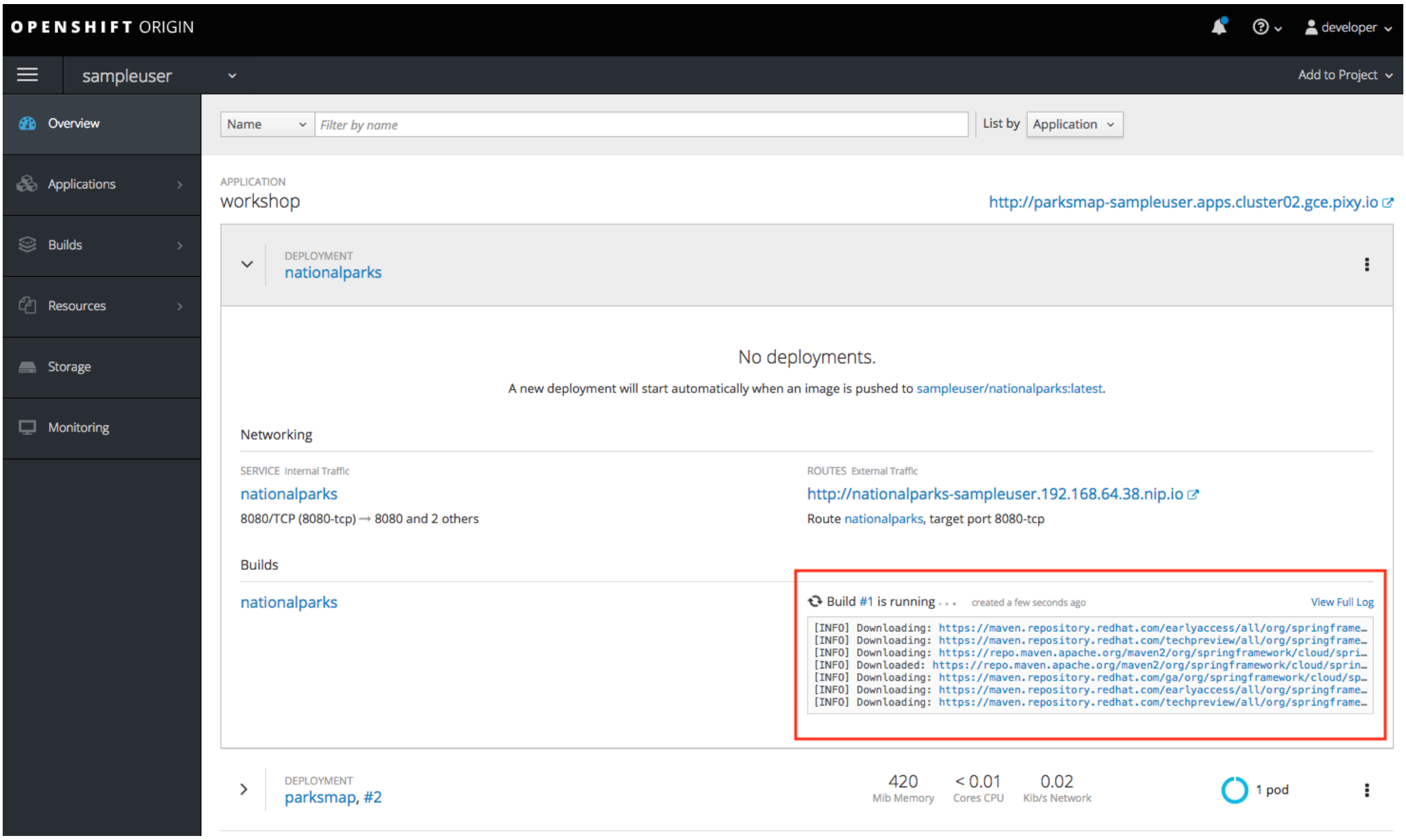


We will again set 3 labels to the deployment.

- *app*=**workshop** (the name we will be giving to the app)
- *component*=**nationalparks** (the name of this deployment)
- *role*=**backend** (the role this component plays in the overall application)



You can then hit the button labeled "**Create**". Then click **Continue to overview**. You will see the build log output directly there.



This is a Java-based application that uses Maven as the build and dependency system. For this reason, the initial build will take a few minutes as Maven downloads all of the dependencies needed for the application. You can see all of this happening in real time!

From the command line, you can also see the **Builds**:

```bash
$ oc get builds
```

You'll see output like:

```bash
NAME              TYPE      FROM          STATUS     STARTED              DURATION
nationalparks-1   Source    Git@b052ae6   Running    About a minute ago   1m2s
```

You can also view the build logs with the following command:

```bash
$ oc logs -f builds/nationalparks-1
```

After the build has completed and successfully:

- The S2I process will push the resulting Docker-formatted image to the internal OpenShift registry
- The **DeploymentConfiguration** (DC) will detect that the image has changed, and this will cause a new deployment to happen.
- A **ReplicationController** (RC) will be spawned for this new deployment.
- The RC will detect no **Pods** are running and will cause one to be deployed, as our default replica count is just 1.

In the end, when issuing the `oc get pods` command, you will see that the build Pod has finished (exited) and that an application **Pod** is in a ready and running state:

```bash
NAME                     READY     STATUS      RESTARTS   AGE
nationalparks-1-tkid3    1/1       Running     3          2m
nationalparks-1-build    0/1       Completed   0          3m
parksmap-1-4hbtk         1/1       Running     0          2h
```

If you look again at the web console, you will notice that, when you create the application this way, OpenShift also creates a **Route** for you. You can see the URL in the web console, or via the command line:

```bash
$ oc get routes
```

Where you should see something like the following:

```
BASH
NAME            HOST/PORT                                                PATH      SERVICES        PORT       TERMINATION
nationalparks   nationalparks-project-userXY.apps.10.2.2.2.xip.io           nationalparks   8080-tcp
parksmap        parksmap-project-userXY.apps.10.2.2.2.xip.io               parksmap        8080-tcp
```

In the above example, the URL is:

```
BASH
http://nationalparks-project-userXY.apps.10.2.2.2.xip.io
```

Since this is a back-end application, it doesn't actually have a web interface. However, it can still be used with a browser. All back ends that work with the parks map front end are required to implement a `/ws/info/` endpoint. To test, the complete URL to enter in your browser is:

```
BASH
http://nationalparks-project-userXY.apps.10.2.2.2.xip.io/ws/info/
```

The trailing slash is **required**.

You will see a simple JSON string:

```
JSON
{"id":"nationalparks","displayName":"National Parks","center":{"latitude":"47.039304","longitude":"14.505178"},"zoom":4}
```

Earlier we said:

```
BASH
This is a Java Spring Boot application that performs 2D geo-spatial queries
against a MongoDB database
```

But we don't have a database. Yet.

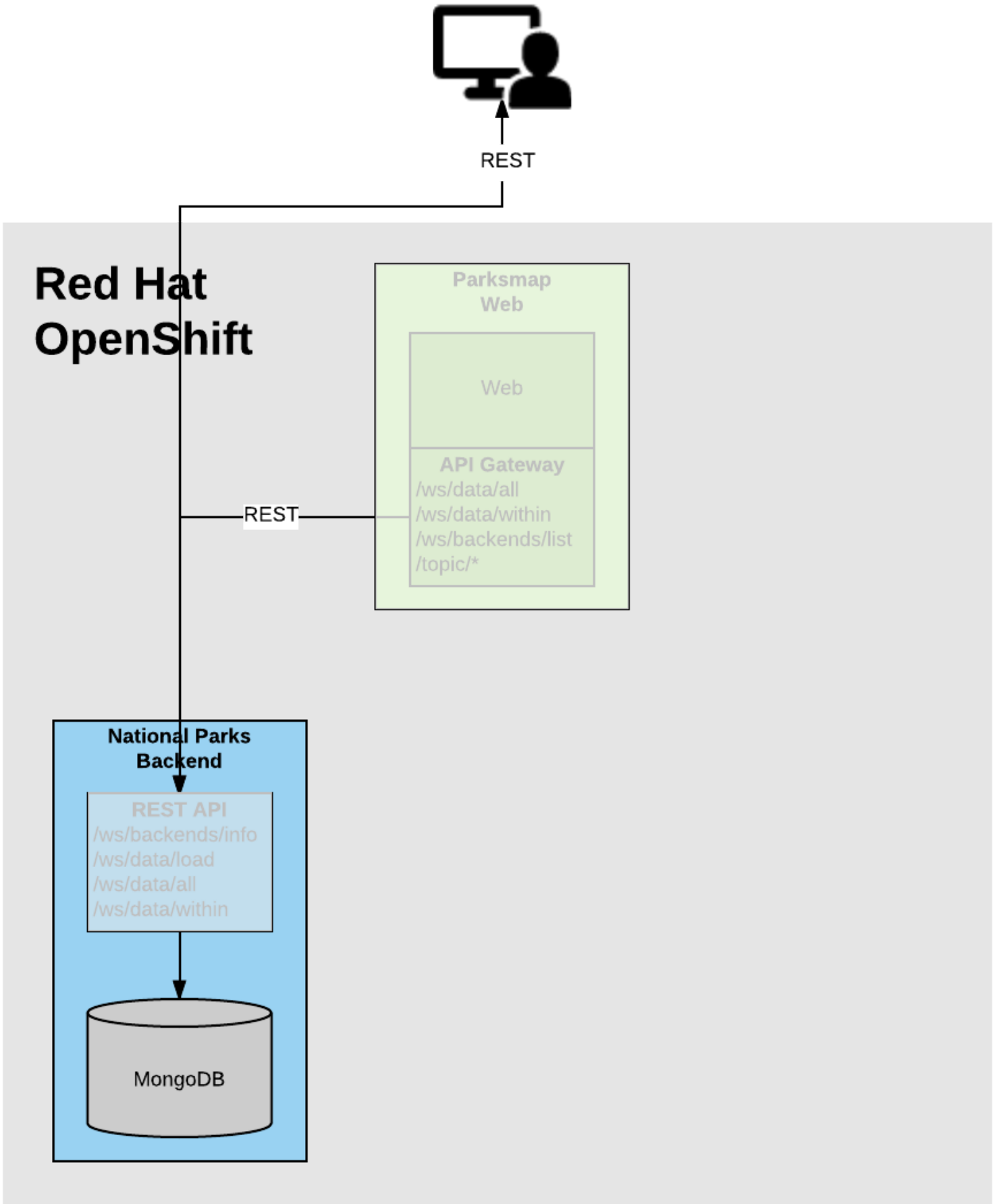
Adding a Database (MongoDB)

Lab: Adding a Database

Application Description

In this lab, we're going to deploy a Mongo database that will be used to store the data for the `nationalparks` application. We will also connect the `nationalparks` service with the newly deployed Mongo database, so that the `nationalparks` service can load and query the database for the corresponding information.

Finally, we will mark the `nationalparks` application as a backend for the map visualization tool, so that it can be dynamically discovered by the `parksmap` component using the OpenShift discovery mechanism and the map will be displayed automatically.



Background: Storage

Most useful applications are "stateful" or "dynamic" in some way, and this is usually achieved with a database or other data storage. In this lab we are going to add MongoDB to our `nationalparks` application and then rewire it to talk to the database using environment variables via a secret.

We are going to use the MongoDB image that is included with OpenShift.

By default, this will use **EmptyDir** for data storage, which means if the **Pod** disappears the data does as well. In a real application you would use OpenShift's persistent storage mechanism to attach real-world storage (NFS, Gluster, EBS, etc) to the **Pods** to give them a persistent place to store their data.

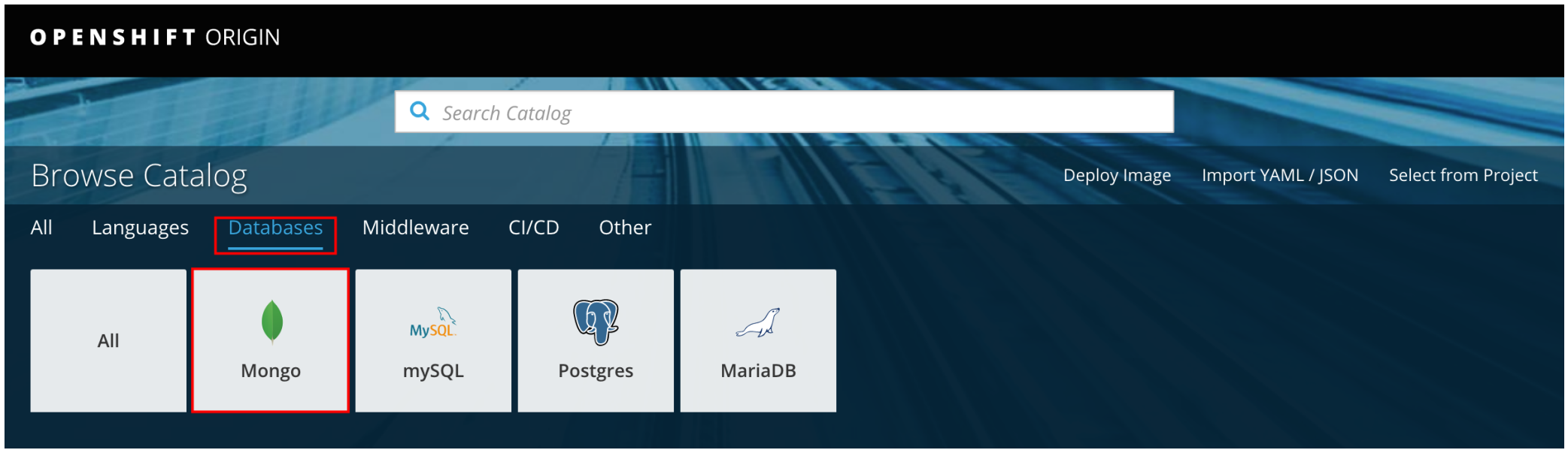
Exercise: Deploy MongoDB

As you've seen so far, the web console makes it very easy to deploy things onto OpenShift. When we deploy the database, we pass in some values for configuration. These values are used to set the username, password, and name of the database.

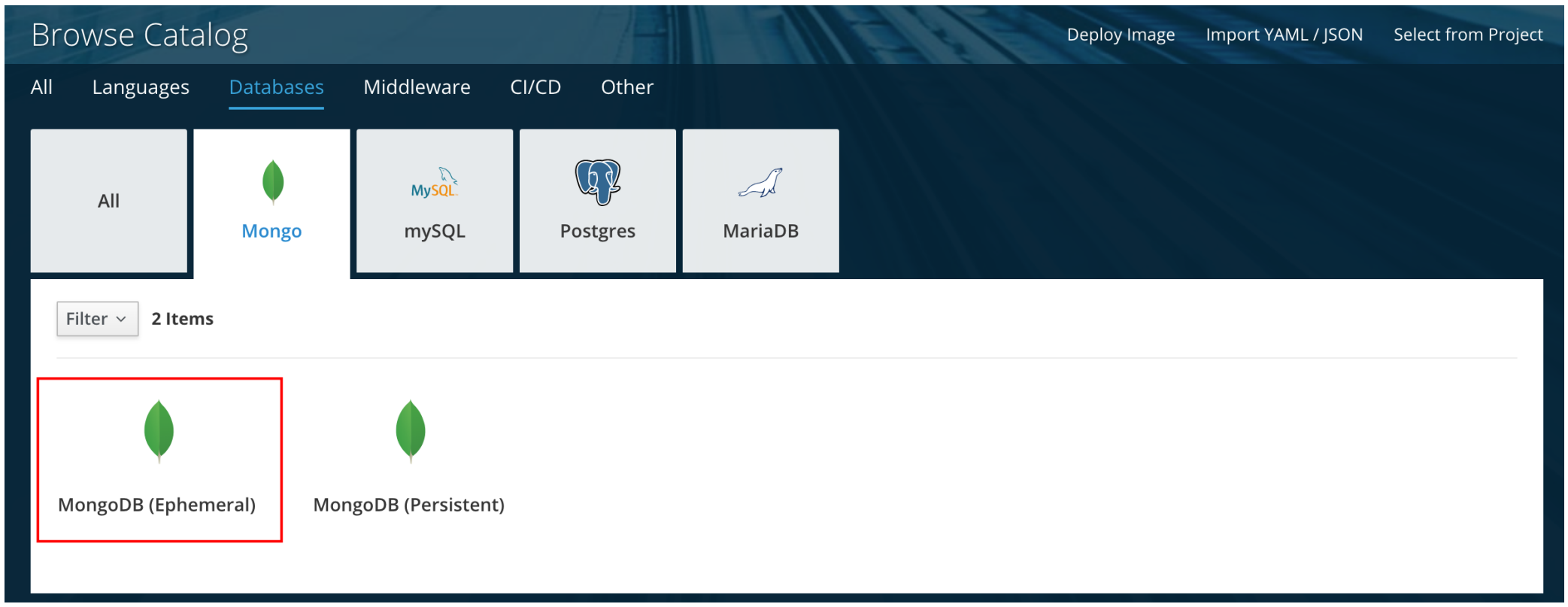
The database image is built in a way that it will automatically configure itself using the supplied information (assuming there is no data already present in the persistent storage!). The image will ensure that:

- A database exists with the specified name
- A user exists with the specified name
- The user can access the specified database with the specified password

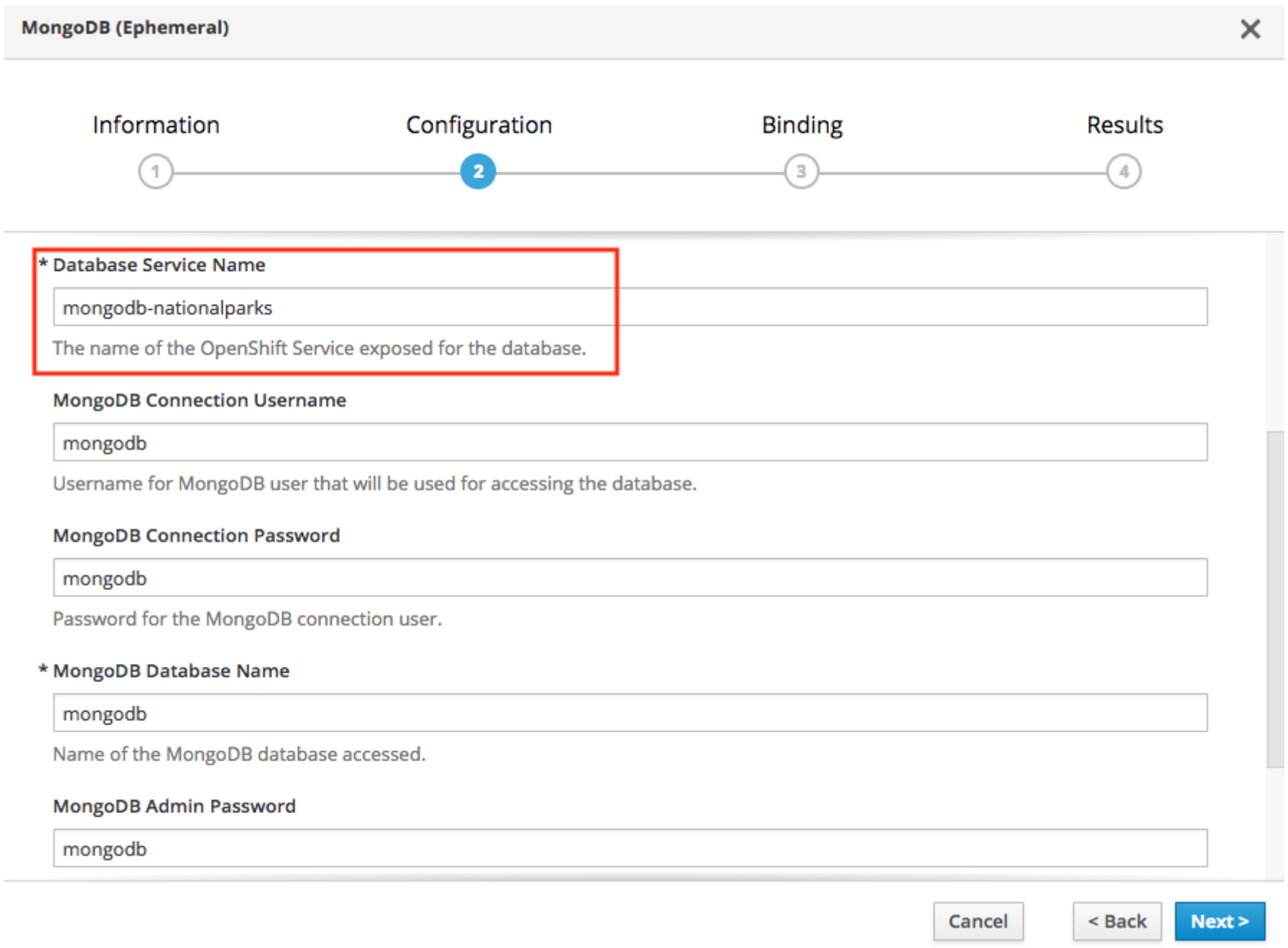
In the web console in your `project-userXY` project, click the **"Add to Project"** button and then **"Browse Catalog"**. Select the **"Databases"** category and then **"Mongo"**.



Alternatively, you could type `mongodb` in the search box. Once you have drilled down to see MongoDB, scroll down to find the **MongoDB (Ephemeral)** template, and select it. You will notice that there are several MongoDB templates available. We do not need a database with persistent storage, so the ephemeral Mongo template is what you should choose. Go ahead and select the ephemeral template and click the next button.



When we performed the application build, there was no template. Rather, we selected the builder image directly and OpenShift presented only the standard build workflow. Now we are using a template - a preconfigured set of resources that includes parameters that can be customized. In our case, the parameters we are concerned with are — user, password, database, and admin password.

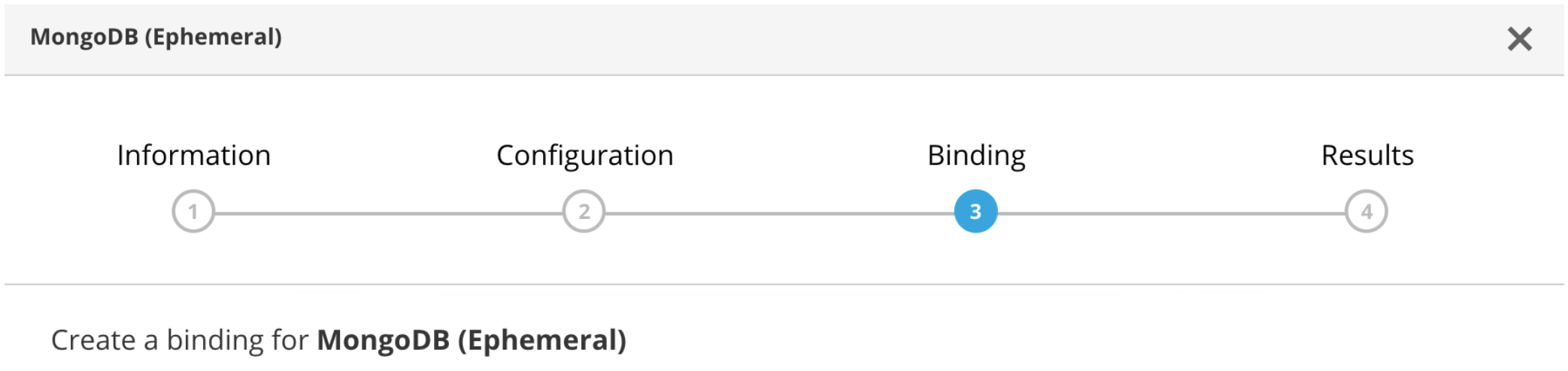


Make sure you name your database service name **mongodb-nationalparks**

You can see that some of the fields say **"generated if empty"**. This is a feature of **Templates** in OpenShift. For now, be sure to use the following values in their respective fields:

- `MONGODB_USER` : `mongodb`
- `MONGODB_PASSWORD` : `mongodb`
- `MONGODB_DATABASE` : `mongodb`
- `MONGODB_ADMIN_PASSWORD` : `mongodb`

Once you have entered in the above information, click on "Next" to go to the next step which will allow us to add a binding.



This creates a **"secret"** in our project that we can use in other components, such as the national parks backend, to authenticate to the database.

While the database deploys, we will fix the labels assigned to the deployment. Currenly we can not set a label when using templates from the Service Catalog, so in this case we will fix this manually.

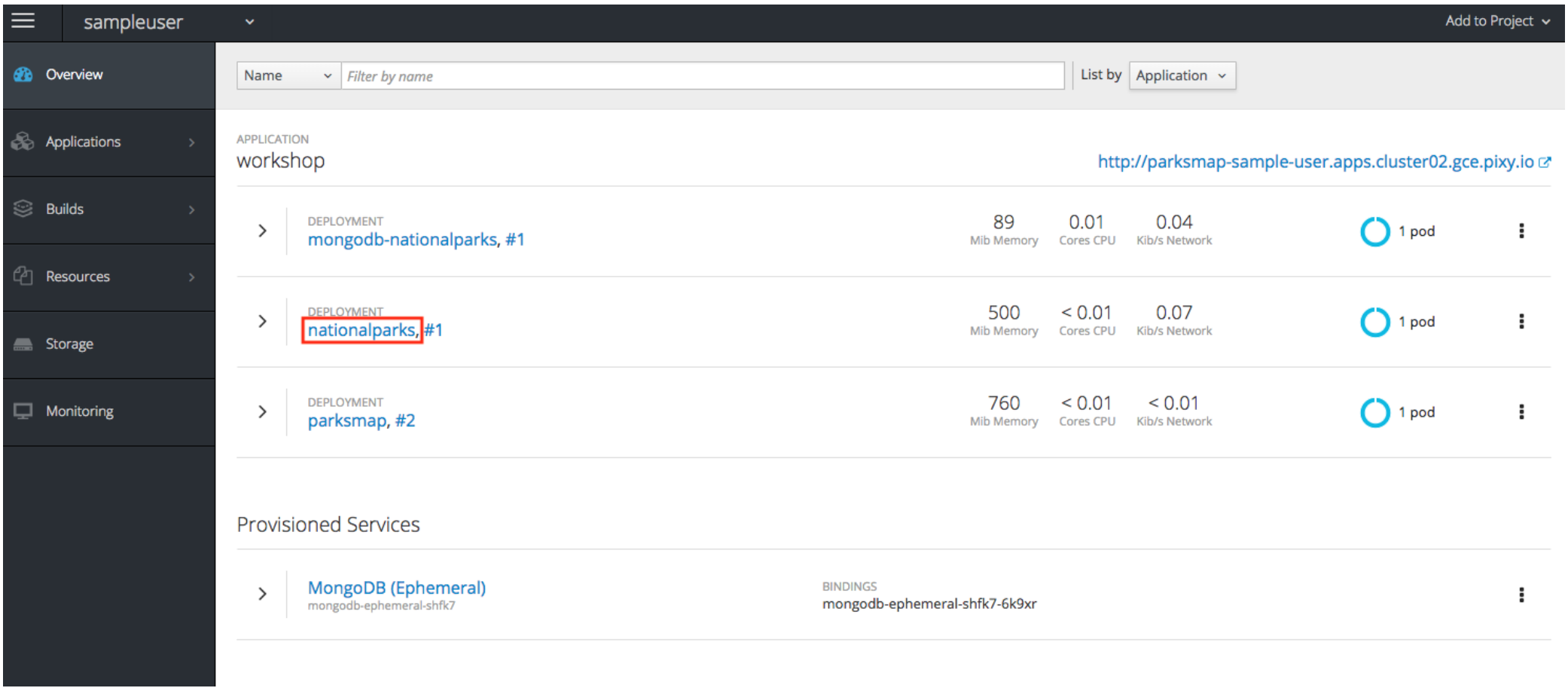
Like before, we will add 3 labels:

- ***app***=**workshop** (the name we will be giving to the app)
- ***component***=**nationalparks** (the name of this deployment)
- ***role***=**backend** (the role this component plays in the overall application)

Execute the following command:

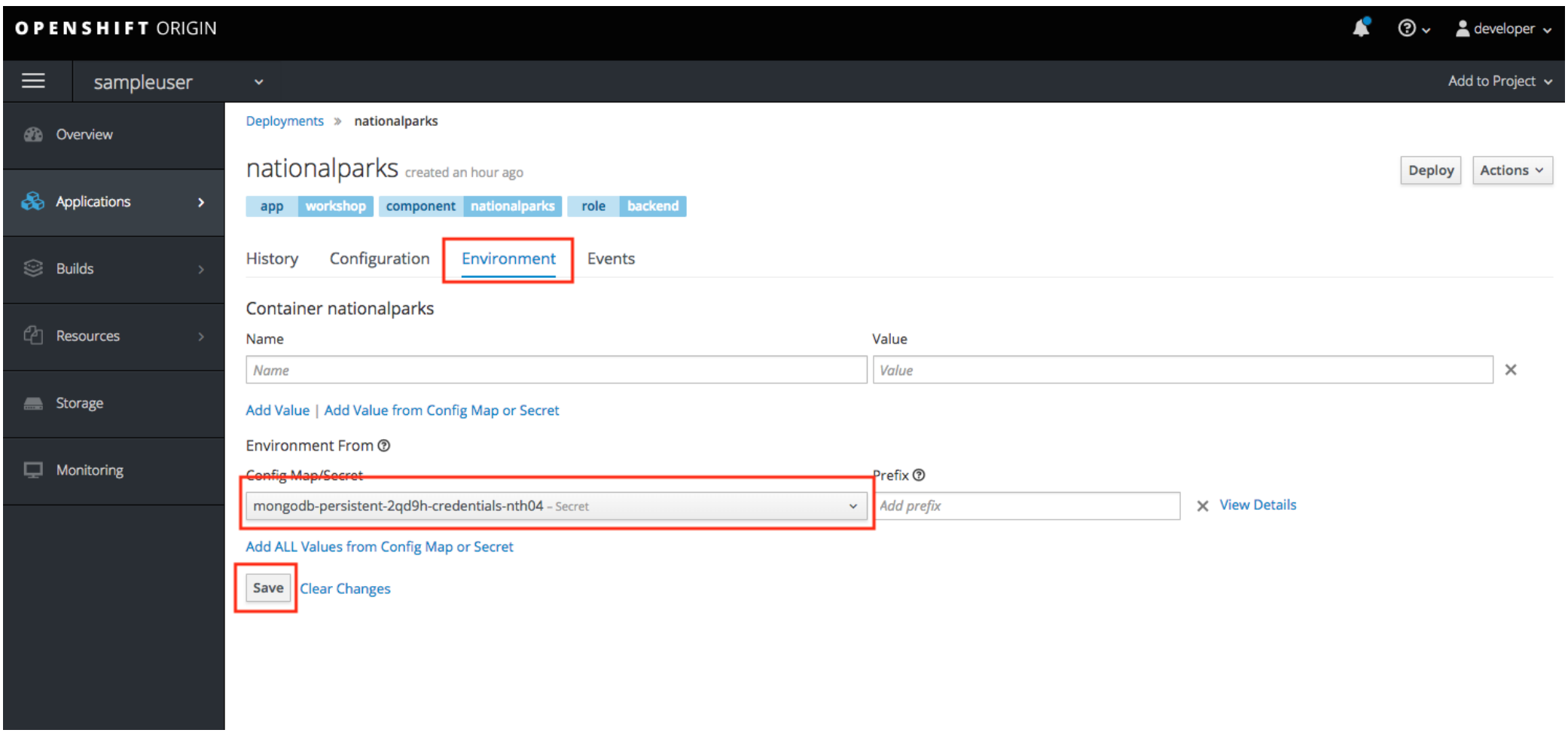
```
$ oc label dc/mongodb-nationalparks svc/mongodb-nationalparks app=workshop component=nationalparks role=database --overwrite
```
BASH

Now that the connection and authentication information is bound to our project, we need to add it to the national parks backend. Go to the project overview screen and click on the national parks deployment:



This will bring up the configuration for the deployment of the national parks backend. If you think way back to the beginning of the labs, you will recall that a **DeploymentConfiguration** tells OpenShift how to deploy something.

In order to make the authentication information available to the java code, we need to add the secret as part of the deployment by modifying the environment information. To do this, click on **"Environment"** and then select the monogo-ephemeral-podid-credentials and click on **"Add ALL Values from ConfigMap or Secret"**. Finally click the **"Save"** Button.



Exercise: Exploring OpenShift Magic

As soon as we changed the **DeploymentConfiguration**, some magic happened. OpenShift decided that this was a significant enough change to warrant updating the internal version number of the **DeploymentConfiguration**. You can verify this by looking at the output of `oc get dc`:

```
BASH
NAME                    REVISION   DESIRED   CURRENT   TRIGGERED BY
mongodb-nationalparks   1          1         1         config,image(mongodb:3.2)
nationalparks           2          1         1         config,image(nationalparks:<version>)
parksmap                2          1         1         config,image(parksmap:<version>))
```

Something that increments the version of a **DeploymentConfiguration**, by default, causes a new deployment. You can verify this by looking at the output of `oc get rc`:

```
BASH
NAME                      DESIRED   CURRENT   READY     AGE
mongodb-nationalparks-1   1         1         1         24m
nationalparks-1           0         0         0         3h
nationalparks-2           1         1         1         8m
parksmap-1                0         0         0         6h
parksmap-2                1         1         1         5h
```

We see that the desired and current number of instances for the "-1" deployment is 0. The desired and current number of instances for the "-2" deployment is 1. This means that OpenShift has gracefully torn down our "old" application and stood up a "new" instance.

Exercise: Data, Data, Everywhere

Now that we have a database deployed, we can again visit the `nationalparks` web service to query for data:

```
BASH
http://nationalparks-project-userXY.apps.10.2.2.2.xip.io/ws/data/all
```

And the result?

```
BASH
[]
```

Where's the data? Think about the process you went through. You deployed the application and then deployed the database. Nothing actually loaded anything **INTO** the database, though.

The application provides an endpoint to do just that:

```
BASH
http://nationalparks-project-userXY.apps.10.2.2.2.xip.io/ws/data/load
```

And the result?

```
BASH
Items inserted in database: 2740
```

If you then go back to `/ws/data/all` you will see tons of JSON data now. That's great. Our parks map should finally work!

There's some errors reported with browsers like firefox 54 that don't properly parse the resulting JSON. It's a browser problem, and the application is working properly.

```
BASH
http://parksmap-project-userXY.apps.10.2.2.2.xip.io
```

Hmm... There's just one thing. The main map **STILL** isn't displaying the parks. That's because the front end parks map only tries to talk to services that have the right **Label**.

You are probably wondering how the database connection magically started working? When deploying applications to OpenShift, it is always best to use environment variables, secrets, or configMaps to define connections to dependent systems. This allows for application portability across different environments. The source file that performs the connection as well as creates the database schema can be viewed here:

```
BASH
http://www.github.com/openshift-
roadshow/nationalparks/blob/<version>/src/main/java/com/openshift/evg/roadshow/parks/db/MongoDBConnection.java#L44-l48
```

In short summary: By referring to bindings to connect to services (like databases), it can be trivial to promote applications throughout different lifecycle environments on OpenShift without having to modify application code.

Exercise: Working With Labels

We explored how a **Label** is just a key=value pair earlier when looking at **Services** and **Routes** and **Selectors**. In general, a **Label** is simply an arbitrary key=value pair. It could be anything.

- `pizza=pepperoni`
- `wicked=googly`
- `openshift=awesome`

In the case of the parks map, the application is actually querying the OpenShift API and asking about the **Routes** and **Services** in the project. If any of them have a **Label** that is `type=parksmap-backend`, the application knows to interrogate the endpoints to look for map data.

You can see the code that does this here (https://github.com/openshift-roadshow/parksmap-web/blob/<version>/src/main/java/com/openshift/evg/roadshow/rest/RouteWatcher.java#L20) .

Fortunately, the command line provides a convenient way for us to manipulate labels. `describe` the `nationalparks` service:

```
$ oc describe route nationalparks

Name:                   nationalparks
Namespace:              project-userXY
Created:                2 hours ago
Labels:                 app=workshop
                        component=nationalparks
                        role=backend
Requested Host:         nationalparks-project-userXY.apps.10.2.2.2.xip.io
                        exposed on router router 2 hours ago
Path:                   <none>
TLS Termination:        <none>
Insecure Policy:        <none>
Endpoint Port:          8080-tcp

Service:                nationalparks
Weight:                 100 (100%)
Endpoints:              10.1.9.8:8080
```
BASH

You see that it already has some labels. Now, use `oc label`:

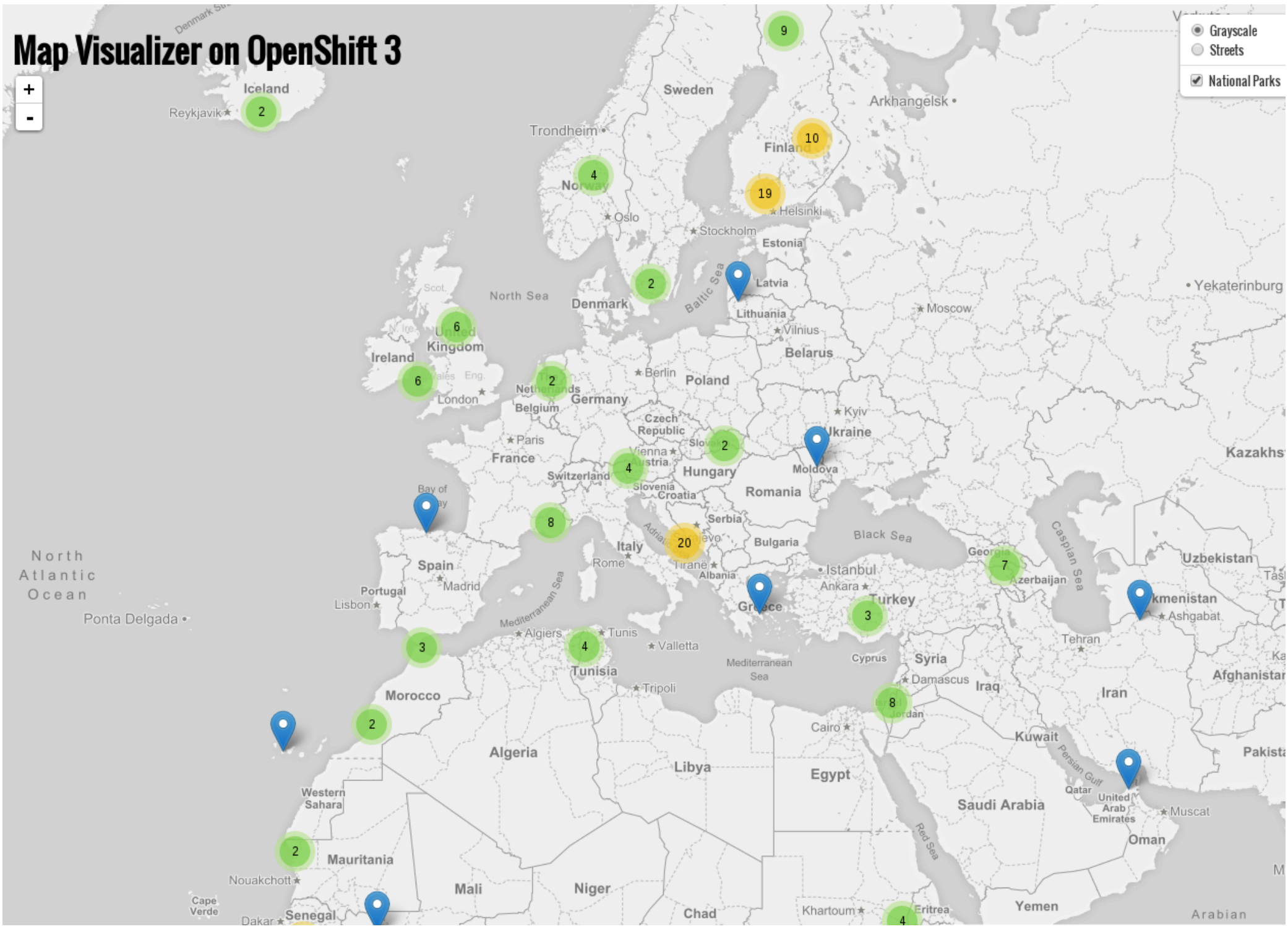
```
$ oc label route nationalparks type=parksmap-backend
```
BASH

You will see something like:

```
route "nationalparks" labeled
```
BASH

If you check your browser now:

```
http://parksmap-project-userXY.apps.10.2.2.2.xip.io/
```
BASH



You'll notice that the parks suddenly are showing up. That's really cool!

Application Health

Lab: Application Health

Background: Readiness and Liveness Probes

As we have seen before in the UI via warnings, there is a concept of application health checks in OpenShift. These come in two flavors:

- Readiness probe
- Liveness probe

From the Application Health (https://<url>/dev_guide/application_health.html) section of the documentation, we see the definitions:

Liveness Probe

A liveness probe checks if the container in which it is configured is still running. If the liveness probe fails, the kubelet kills the container, which will be subjected to its restart policy. Set a liveness check by configuring the `template.spec.containers.livenessprobe` stanza of a pod configuration.

Readiness Probe

A readiness probe determines if a container is ready to service requests. If the readiness probe fails a container, the endpoints controller ensures the container has its IP address removed from the endpoints of all services. A readiness probe can be used to signal to the endpoints controller that even though a container is running, it should not receive any traffic from a proxy. Set a readiness check by configuring the `template.spec.containers.readinessprobe` stanza of a pod configuration.

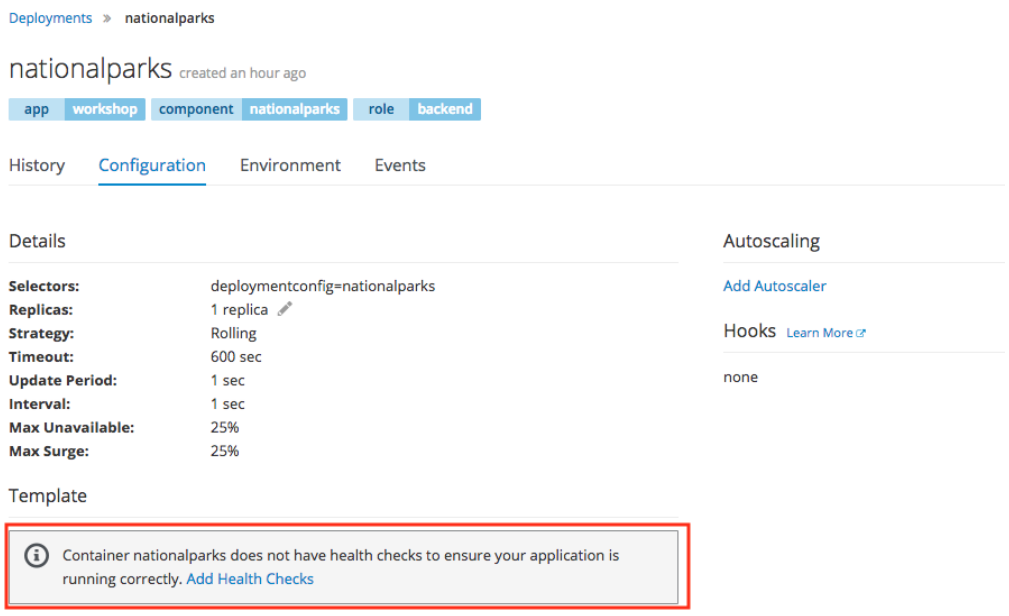
It sounds complicated, but it really isn't. We will use the web console to add these probes to our `nationalparks` application.

Exercise: Add Health Checks

As we are going to be implementing a realistic CI/CD pipeline, we will be doing some testing of the "development" version of the application. However, in order to test the app, it must be ready. This is where OpenShift's application health features come in very handy.

We are going to add both a readiness and liveness probe to the existing `nationalparks` deployment. This will ensure that OpenShift does not add any instances to the service until they pass the readiness checks, and will ensure that unhealthy instances are restarted (if they fail the liveness checks).

Click **Applications** → **Deployments** on the left-side bar. Click `nationalparks`. Click on `Configuration`. You will see the warning about health checks, with a link to click in order to add them.



Click **Add health checks** now.

You will want to click both **Add Readiness Probe** and **Add Liveness Probe** and then fill them out as follows:

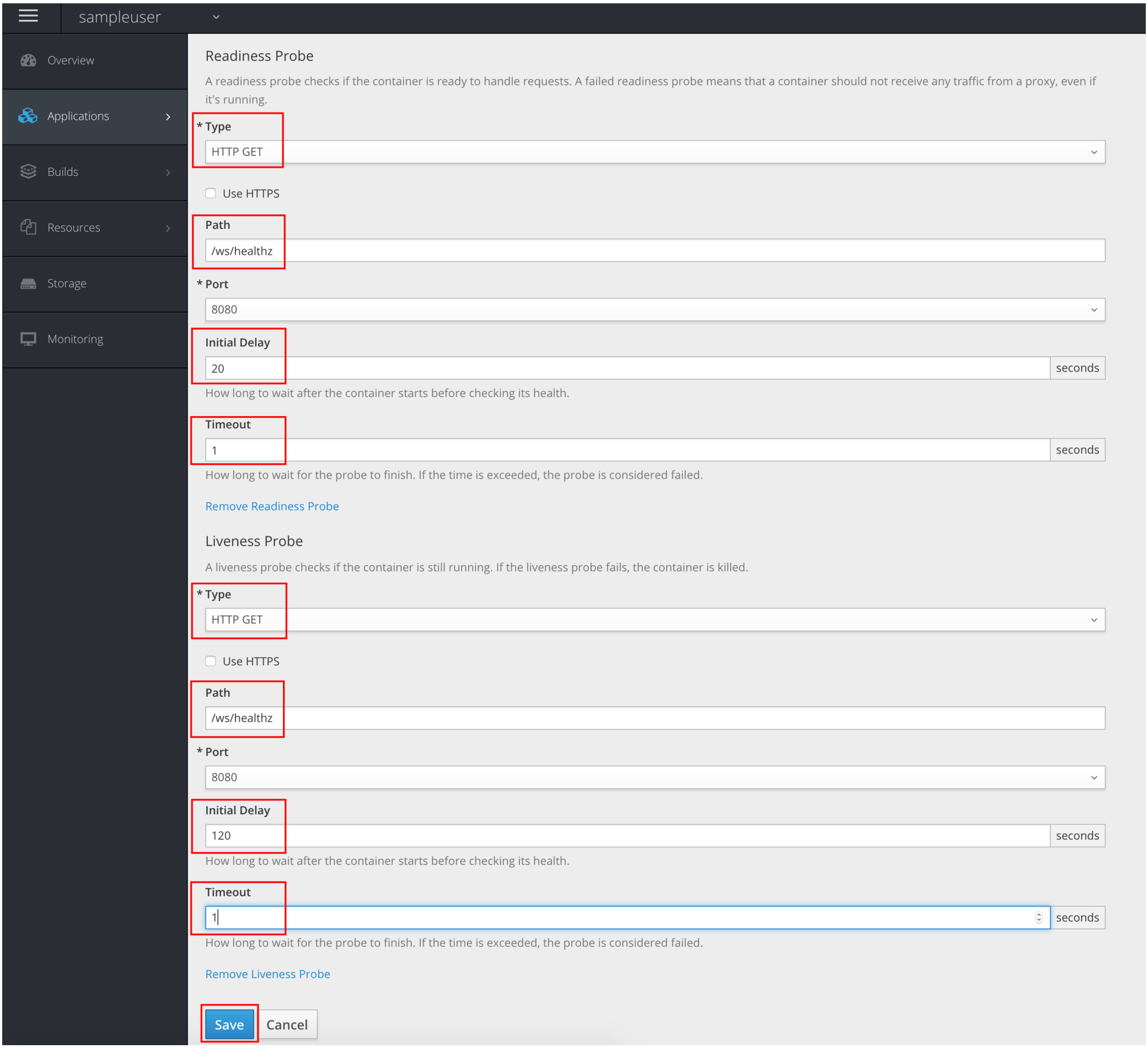
Readiness Probe

- Path: `/ws/healthz/`
- Initial Delay: `20`
- Timeout: `1`

Liveness Probe

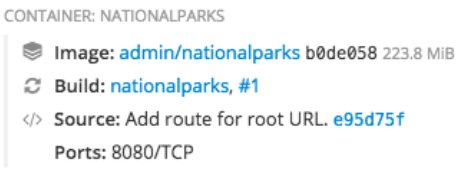
- Path: `/ws/healthz/`
- Initial Delay: `120`
- Timeout: `1`

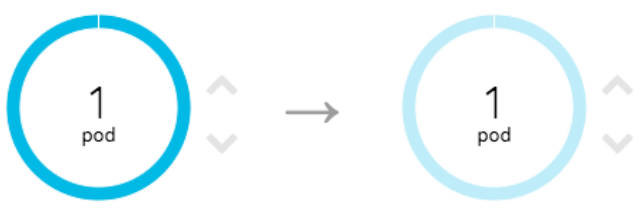
This is shown in the following image:



Once you have entered in the values show above, click **Save** and then click the **Overview** button in the left navigation. You will notice that these changes caused a new deployment — they counted as a configuration change.

You will also notice that the circle around the new deployment stays light blue for a while. This is a sign that the pod(s) have not yet passed their readiness checks — it's working!





Using Source 2 Image for Code Changes

Lab: Making Code Changes and using webhooks

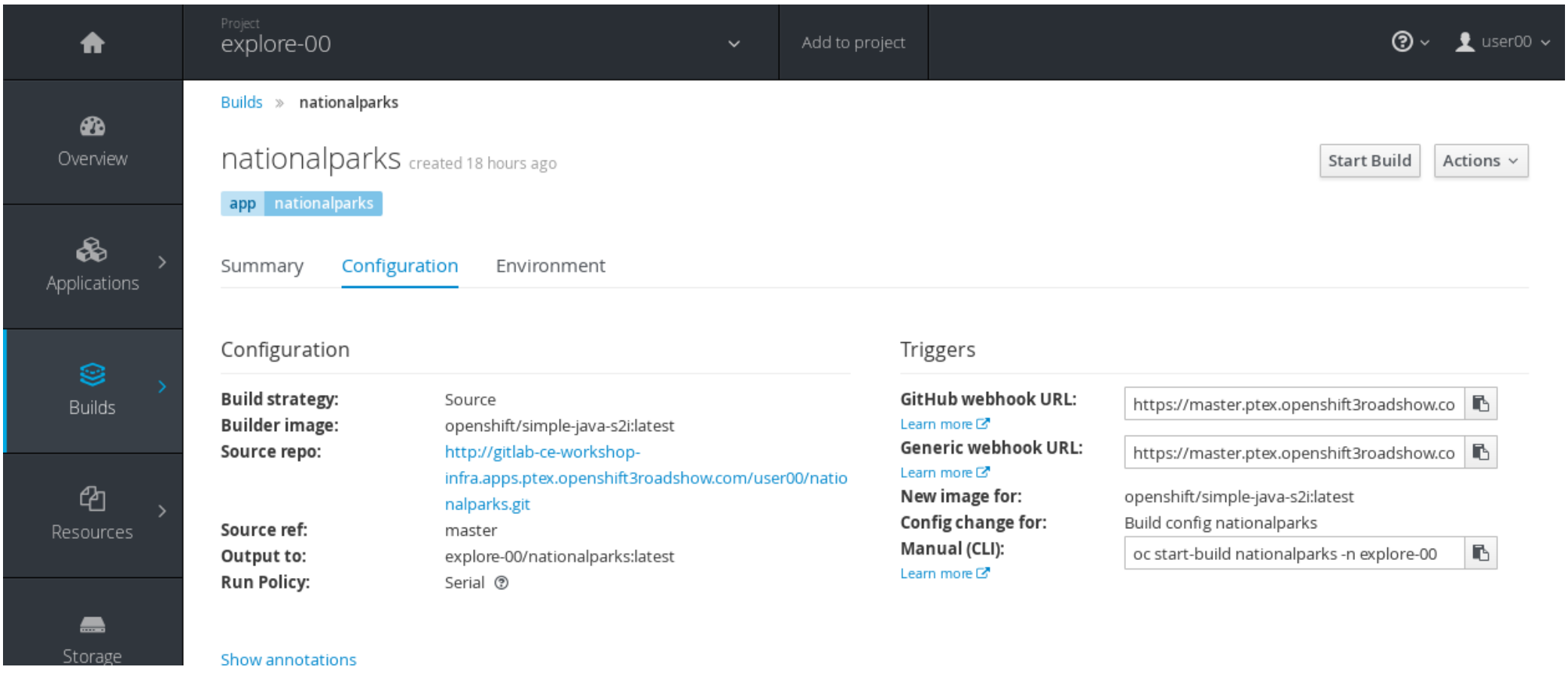
Background: Web Hooks

Most Git repository servers support the concept of web hooks — calling to an external source via HTTP(S) when a change in the code repository happens. OpenShift provides an API endpoint that supports receiving hooks from remote systems in order to trigger builds. By pointing the code repository's hook at the OpenShift API, automated code/build/deploy pipelines can be achieved.

Exercise: Configuring Gogs Web Hooks

In the OpenShift web console, navigate to your `project-userXY` **Project**, and then mouse-over **Browse** and then **Builds**. Click the `nationalparks` build.

On this screen you will see the option to copy the **generic** webhook URL as shown in the following image:



Once you have the URL copied to your clipboard, navigate to the code repository that you have on your local Gogs:

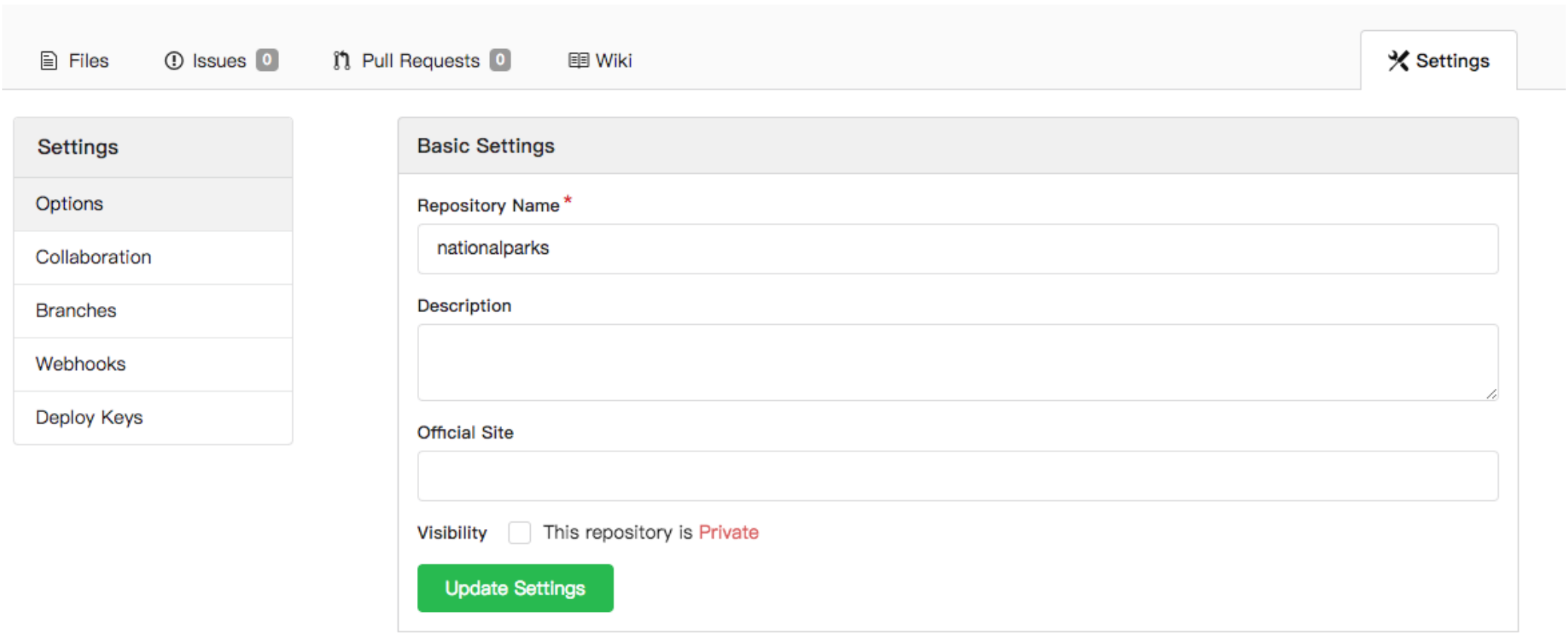
```
http://<gitlab>.apps.10.2.2.2.xip.io/userXY/nationalparks
```
BASH

The credentials for this Gogs instance are:

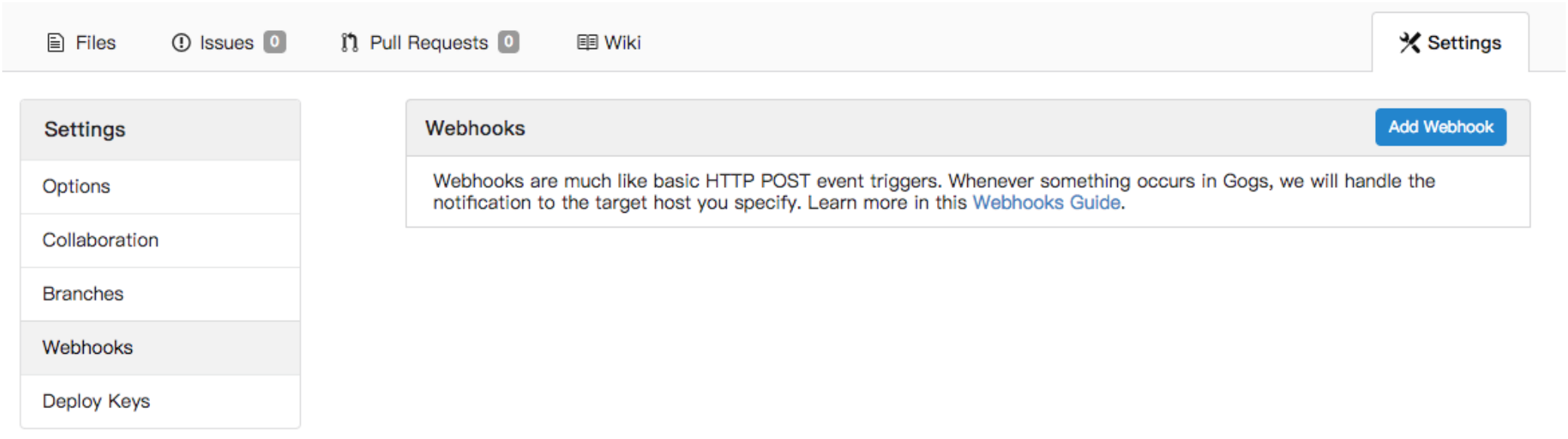
Username: userXY

Password: <password>

Click the Settings link on the top right of the screen:



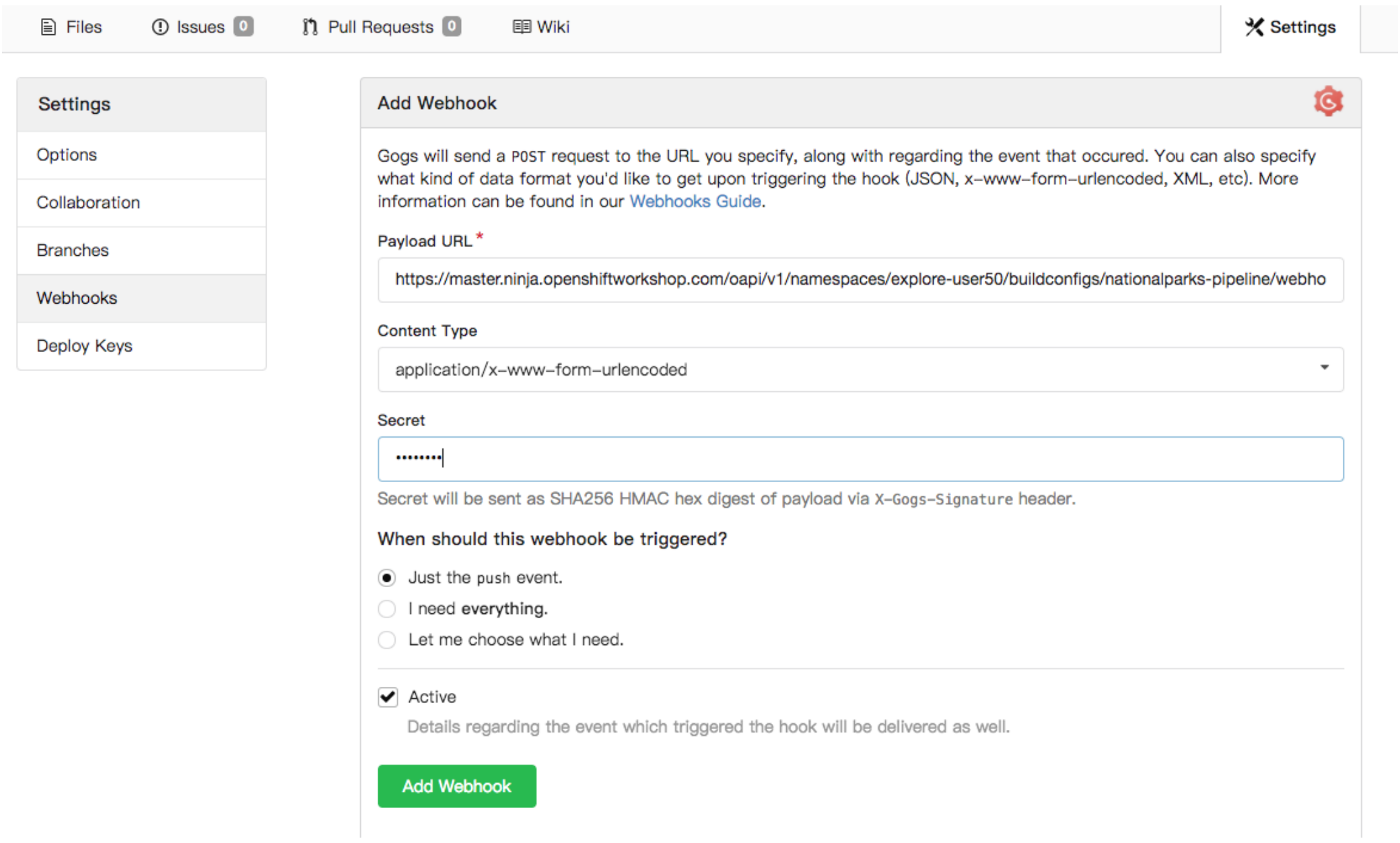
Click on webhooks, and the on `Add Webhook` button.



In the next screen, paste your link into the "URL" field. You can leave the secret token field blank — the secret is already in the URL and does not need to be in the payload, or copy it from the URL.

Change the `Content Type` to `application/x-www-form-urlencoded`.

Finally, click on "Add webhook".



Boom! From now on, every time you commit new source code to your Gogs repository, a new build and deploy will occur inside of OpenShift. Let's try this out.

Exercise: Using Gogs Web Hooks

Click "Project" at the top of the Gogs page, and then "Files" towards the middle of the page. This is Gogs's repository view. Make sure that the drop-down menu at the upper right is set for the `<version>` branch. Navigate to the following path:

```bash
src/main/java/com/openshift/evg/roadshow/parks/rest/
```

Then click on the `BackendController.java` file.

Once you have the file on the screen, click the edit button in the top right hand corner as shown here:

Files | Issues 0 | Pull Requests 0 | Wiki | Settings

Branch: 1.2.0 — nationalparks / src / main / java / com / openshift / evg / roadshow / parks / rest / BackendController.java

BackendController.java 767 B — Permalink | History | Raw

```java
package com.openshift.evg.roadshow.parks.rest;

import com.openshift.evg.roadshow.rest.gateway.model.Backend;
import com.openshift.evg.roadshow.rest.gateway.model.Coordinates;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

/**
 * Provides information about this backend
 *
 * Created by jmorales on 26/09/16.
 */
@RequestMapping("/ws/info")
@RestController
public class BackendController{

    @RequestMapping(method = RequestMethod.GET, value = "/", produces = "application/json")
    public Backend get() {
        return new Backend("nationalparks","National Parks", new Coordinates("47.039304", "14.505178"), 4);
    }
}

```

Change line number 20:

```java
return new Backend("nationalparks","National Parks", new Coordinates("47.039304", "14.505178"), 4);
```

To

```java
return new Backend("nationalparks","OpenShift National Parks", new Coordinates("47.039304", "14.505178"), 4);
```

Click on Commit changes at the bottom of the screen. Feel free to enter a commit message.

Once you have committed your changes, a **Build** should almost instantaneously be triggered in OpenShift. Look at the **Builds** page in the web console, or run the following command to verify:

```bash
$ oc get builds
```

You should see that a new build is running:

```bash
NAME              TYPE      FROM          STATUS     STARTED          DURATION
nationalparks-1   Source    Git@b052ae6   Complete   18 hours ago     36s
nationalparks-2   Source    Git@3b26e1a   Running    43 seconds ago
```

Once the build and deploy has finished, verify your new Docker image was automatically deployed by viewing the application in your browser:

```bash
http://nationalparks-project-userXY.apps.10.2.2.2.xip.io/ws/info/
```

You should now see the new name you have set in the JSON string returned.

To see this in the map's legend itself, you will need to scale down your parksmap to 0, then back up to 1 to force the app to refresh its cache.

Exercise: Rollback

OpenShift allows you to move between different versions of an application without the need to rebuild each time. Every version (past builds) of the application exists as a Docker-formatted image in the OpenShift registry. Using the `oc rollback` and `oc deploy` commands you can move back- or forward between various versions of applications.

In order to perform a rollback, you need to know the name of the **Deployment Config** which has deployed the application:

```bash
$ oc get dc
```

The output will be similar to the following:

```
BASH
NAME                REVISION   DESIRED   CURRENT   TRIGGERED BY
mongodb             1          1         1         config,image(mongodb:3.2)
parksmap            2          1         1         config,image(parksmap:<version>)
nationalparks       9          1         1         config,image(nationalparks:latest)
```

Now run the following command to rollback the latest code change:

```
BASH
$ oc rollback nationalparks
```

You will see output like the following:

```
BASH
#5 rolled back to nationalparks-3
Warning: the following images triggers were disabled: nationalparks:live
  You can re-enable them with: oc set triggers dc/nationalparks --auto
```

Once the deploy is complete, verify that the page header is reverted to the original header by viewing the application in your browser.

```
BASH
http://nationalparks-project-userXY.apps.10.2.2.2.xip.io/ws/info/
```

Automatic deployment of new images is disabled as part of the rollback to prevent unwanted deployments soon after the rollback is complete. To re-enable the automatic deployments run this:

```
BASH
$ oc set triggers dc/nationalparks --auto
```

Exercise: Rollforward

Just like you performed a rollback, you can also perform a roll-forward using the same command. You'll notice above that when you requested a **rollback**, it caused a new deployment (#3). In essence, we always move forwards in OpenShift, even if we are going "back".

So, if we want to return to the "new code" version, that is deployment #4.

```
BASH
$ oc rollback nationalparks-4
```

And you will see the following:

```
BASH
#6 rolled back to nationalparks-4
Warning: the following images triggers were disabled: nationalparks:live
  You can re-enable them with: oc set triggers dc/nationalparks --auto
```

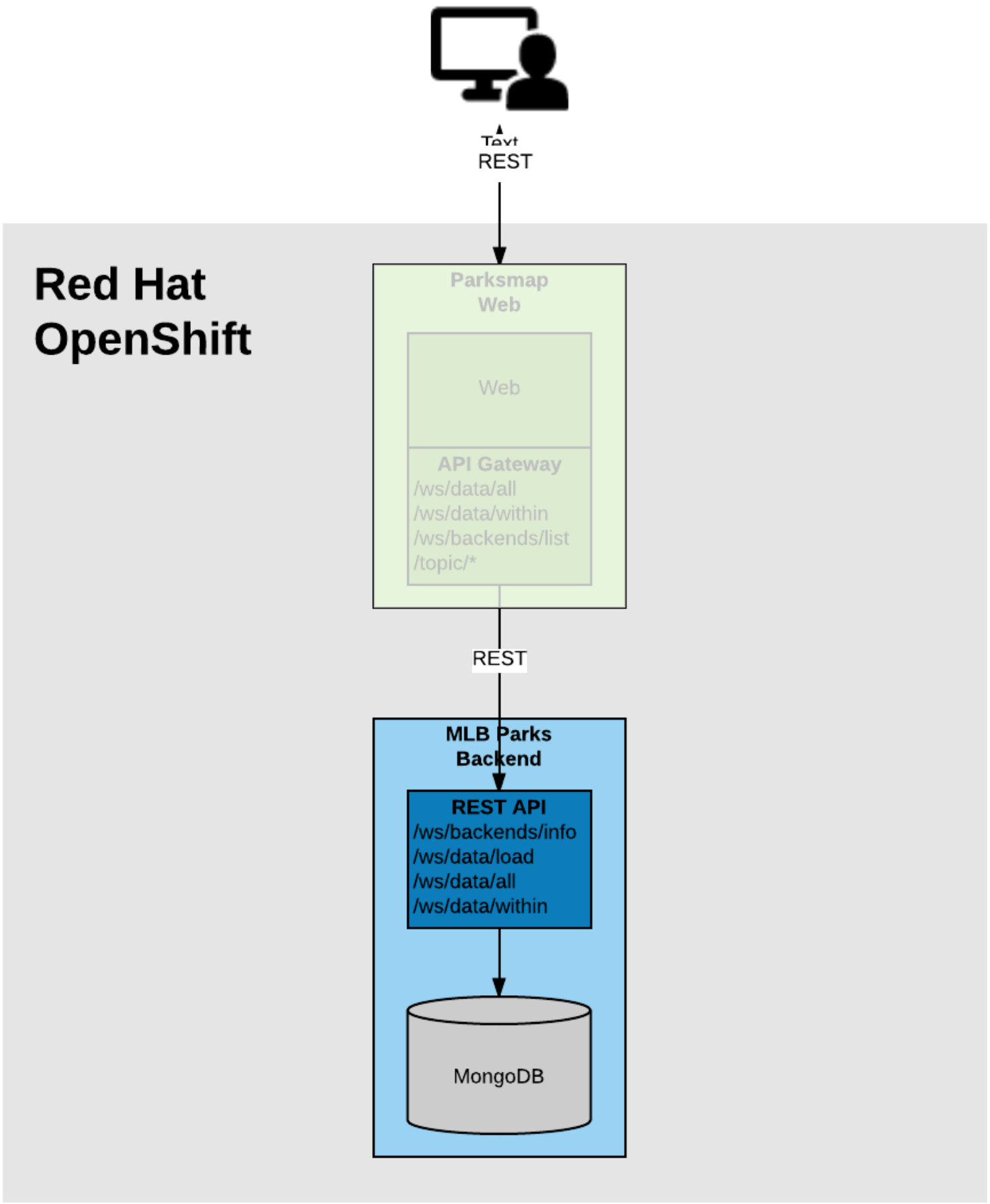
Cool! Once the **rollback** is complete, verify you again see "OpenShift National Parks".

Using Application Templates

Lab: Using Templates

Application description

In this lab, we're going to deploy a complete backend application, consisting of an REST API backend and a mongo database. The complete application will already be wired together and described as a backend for the map visualization tool, so that once the application is built and deployed, you will be able to see the new map.



Background: Templates

Running all these individual commands can be tedious and error prone. Fortunately for you, all of this configuration can be put together into a single **Template** which can then be processed to create a full set of resources. As you saw with MongoDB, a **Template** may define parameters for certain values, such as DB username or password, and they can be automatically generated by OpenShift at processing time.

Administrators can load **Templates** into OpenShift and make them available to all users, even via the web console. Users can create **Templates** and load them into their own **Projects** for other users (with access) to share and use.

The great thing about **Templates** is that they can speed up the deployment workflow for application development by providing a "recipe" of sorts that can be deployed with a single command. Not only that, they can be loaded into OpenShift from an external URL, which will allow you to keep your templates in a version control system.

Let's combine all of the exercises we have performed in the last several labs by using a **Template** that we can instantiate with a single command. While we could have used templates to deploy everything in the workshop today, remember that it is important for you to understand how to create, deploy, and wire resources together.

Exercise: Instantiate a Template

The front end application we've been working with this whole time will display as many back end services' data as are created. Adding more stuff with the right **Label** will make more stuff show up on the map.

Now you will deploy a map of Major League Baseball stadiums in the US by using a template. It is pre-configured to build the back end Java application, and deploy the Mongo database. It also uses a **Hook** to call the `/ws/data/load` endpoint to cause the data to be loaded into the database from a JSON file in the source code repository. Execute the following command:

```
$ oc create -f https://raw.githubusercontent.com/openshift-roadshow/mlbparks/<version>/ose3/application-template-wildfly.json
```
BASH

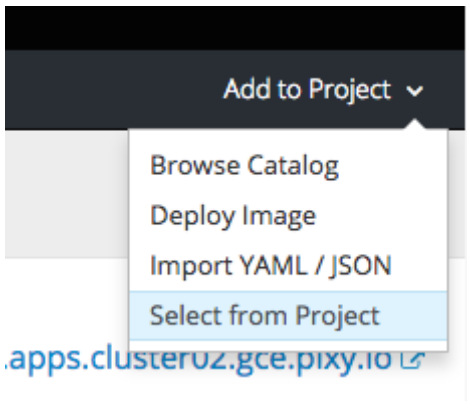
What just happened? What did you just `create`? The item that we passed to the `create` command is a **Template**. `create` simply makes the template available in your **Project**. You can see this with the following command:

```
$ oc get template
```
BASH

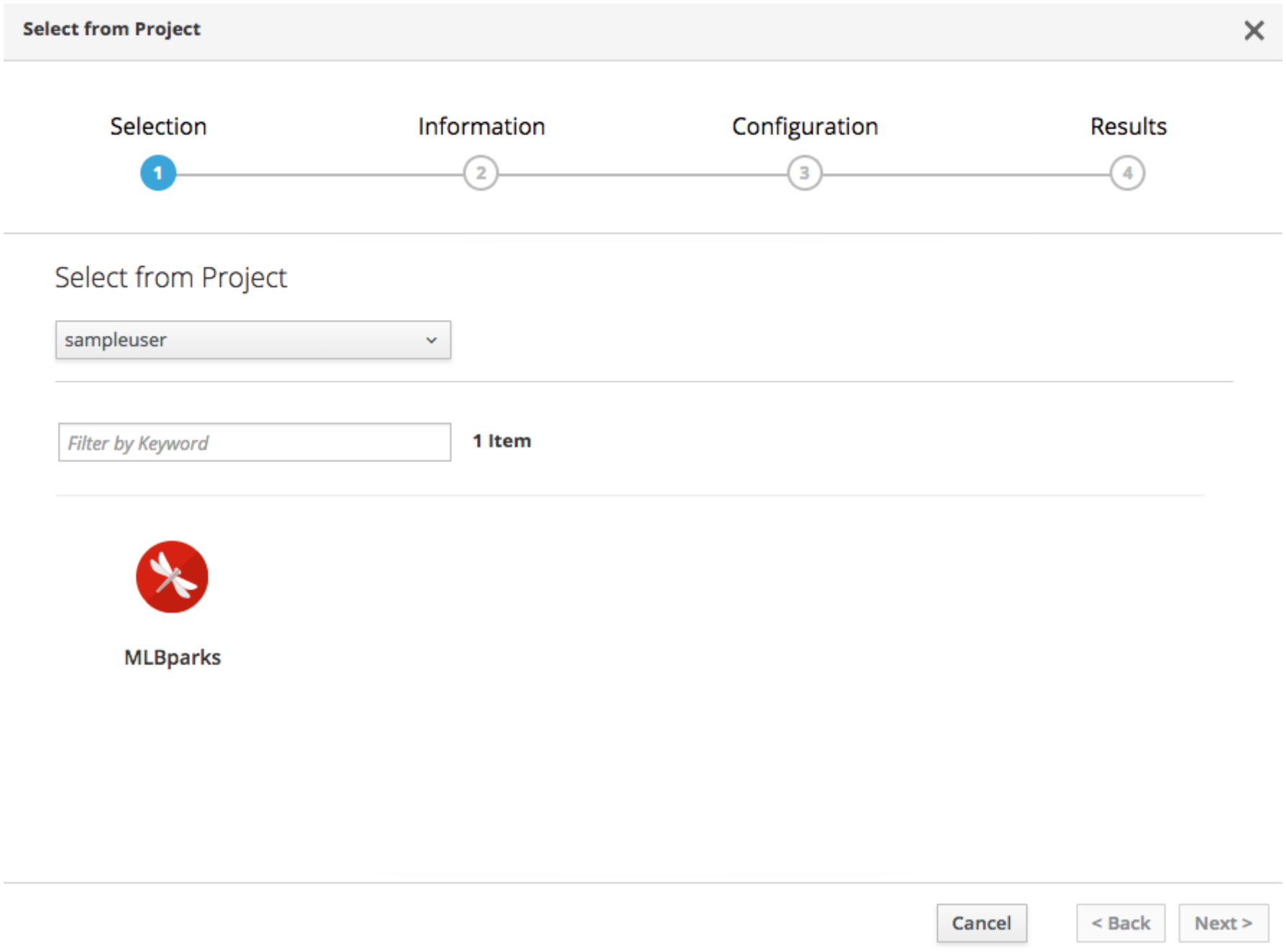
You will see output like the following:

```
mlbparks      Application template MLBParks backend running on WildFly and using MongoDB   15 (5 blank)   8
```
BASH

In the web UI, you can see the templates you have in your project, by clicking on the **"Add to project -> Select from project"** dropdown menu.



You'll be presented with a list of templates local to your own project. You will see the template we just loaded.



Are you ready for the magic command?

Here it is!

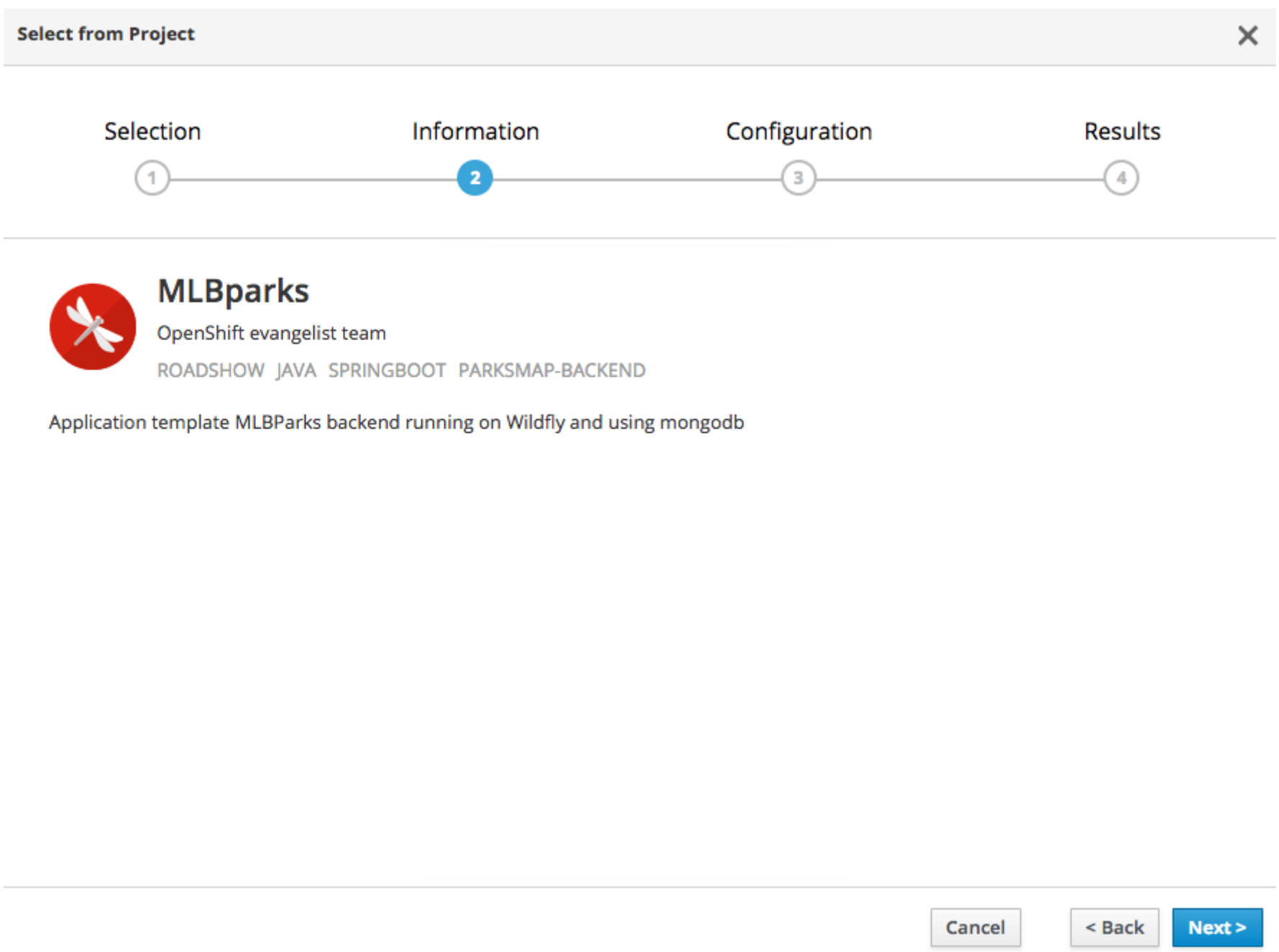
If you want to use command line:

```
oc new-app mlbparks -p APPLICATION_NAME=mlbparks -p GIT_REF=<version>
```
BASH

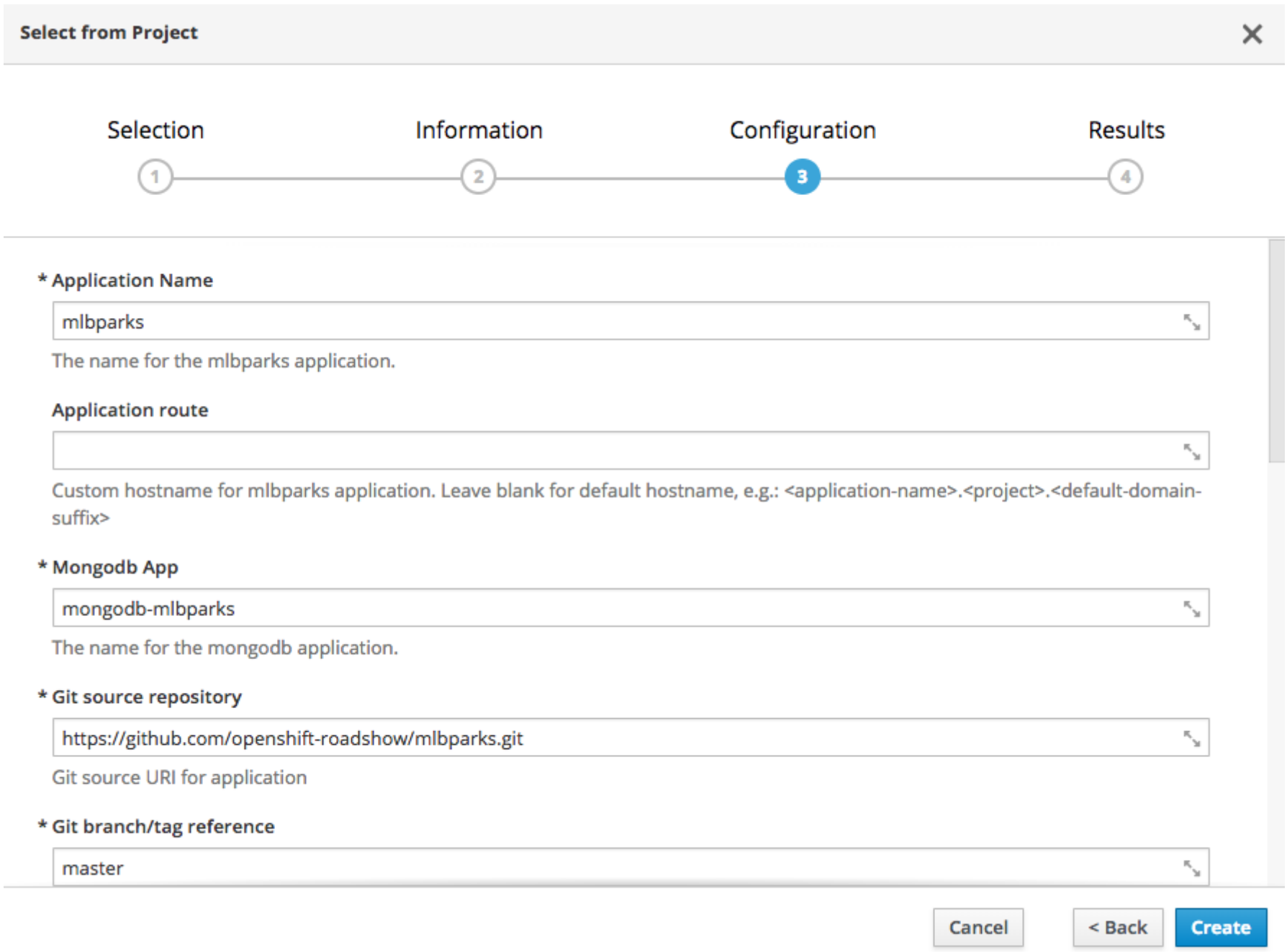
You will see the following output:

```
--> Deploying template mlbparks

     mlbparks
     ---------
     Application template MLBParks backend running on WildFly and using MongoDB

     * With parameters:
        * Application Name=mlbparks
        * Application route=
        * Mongodb App=mongodb-mlbparks
        * Git source repository=https://github.com/openshift-roadshow/mlbparks
        * Git branch/tag reference=<version>
        * Database name=mongodb
        * MONGODB_NOPREALLOC=
        * MONGODB_SMALLFILES=
        * MONGODB_QUIET=
        * Database user name=userGhR # generated
        * Database user password=KhnHKCQI # generated
        * Database admin password=UyUV6ReU # generated
        * GitHub Trigger=dAOuD7s4 # generated
        * Generic Trigger=tWSkmNLn # generated

--> Creating resources ...
    configmap "mlbparks" created
    service "mongodb-mlbparks" created
    deploymentconfig "mongodb-mlbparks" created
    imagestream "mlbparks" created
    buildconfig "mlbparks" created
    deploymentconfig "mlbparks" created
    service "mlbparks" created
    route "mlbparks" created
--> Success
    Build scheduled, use 'oc logs -f bc/mlbparks' to track its progress.
    Run 'oc status' to view your app.
```

Or if you prefer using the web UI, just click on the template we just saw:



Then, you'll see a guided wizard showing you some information and allowing you to provide the desired configuration to instantiate the template



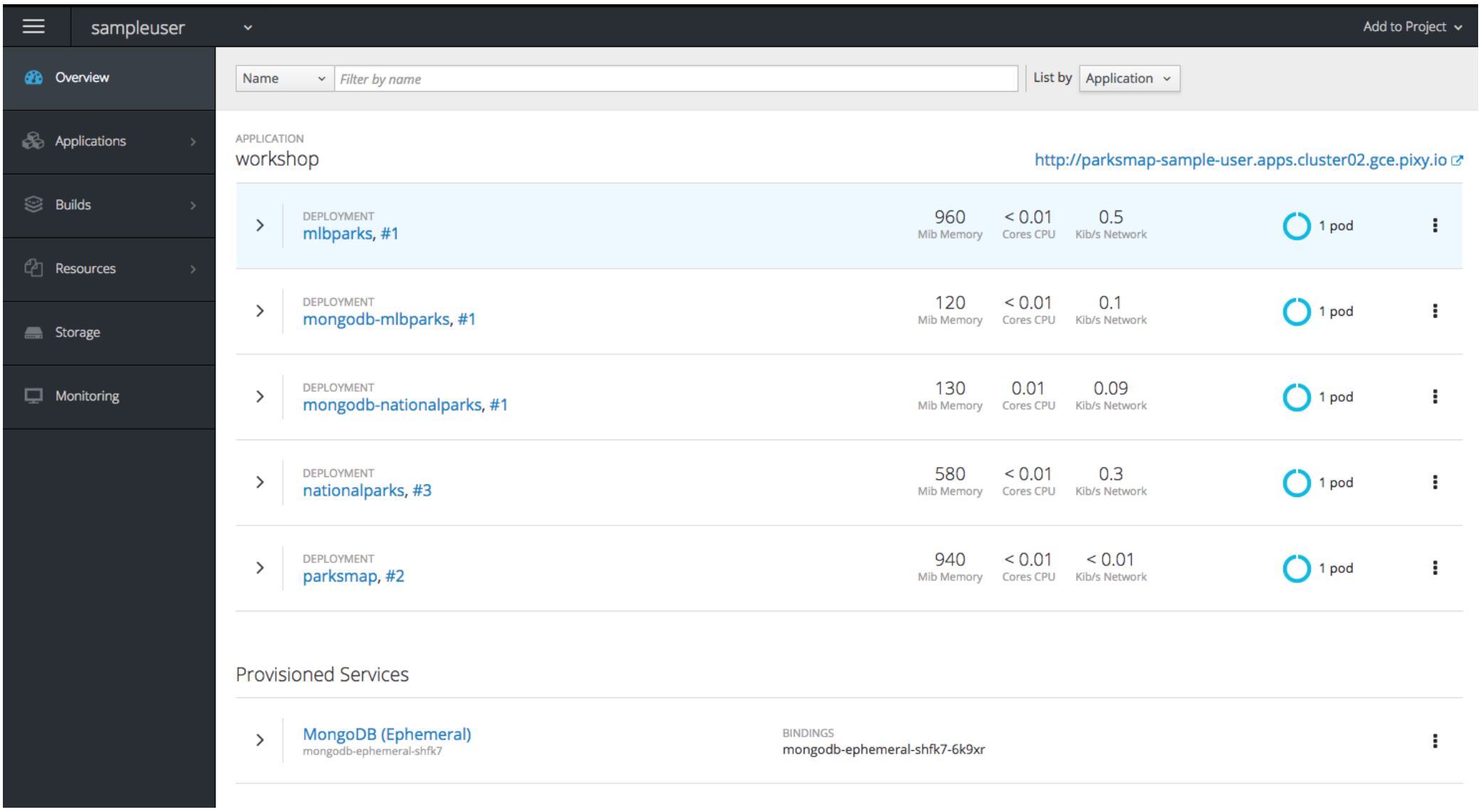
Go ahead and click **"Create"**

OpenShift will now:

- Configure and start a build
  - From the supplied source code repository
- Configure and deploy MongoDB
  - Using auto-generated user, password, and database name
- Configure environment variables for the app to connect to the DB
- Create the correct services
- Label the app service with `type=parksmap-backend`

All with one command!

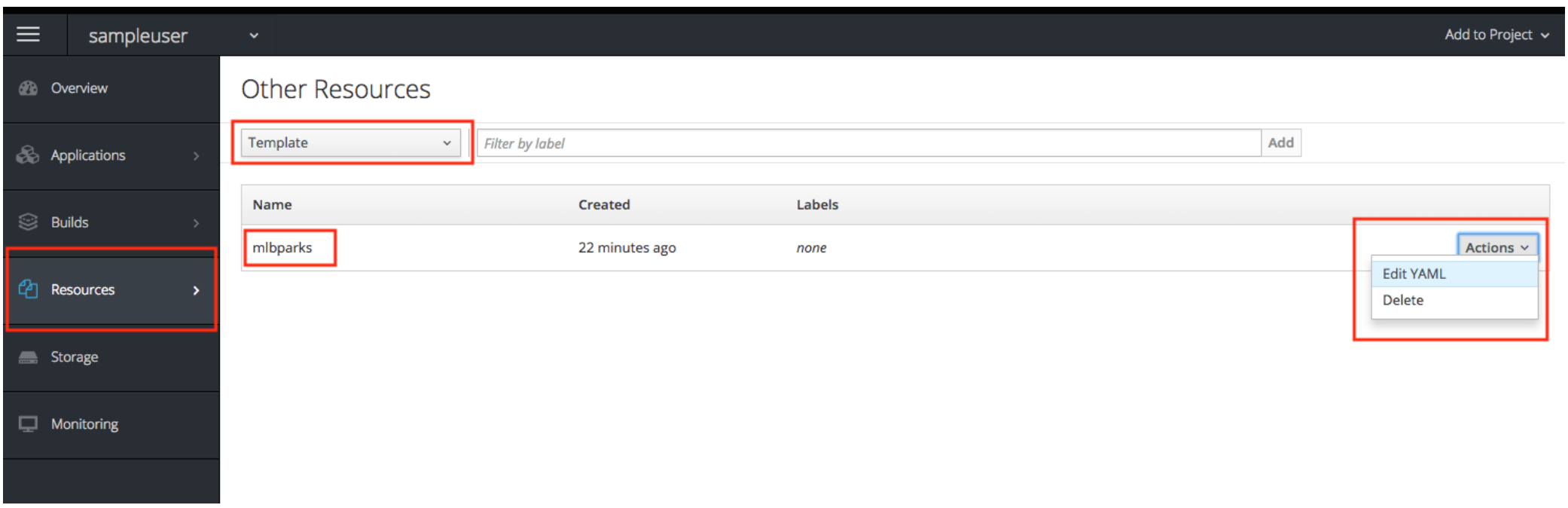
When the build is complete, visit the parks map. Does it work? Think about how this could be used in your environment. For example, a template could define a large set of resources that make up a "reference application", complete with several app servers, databases, and more. You could deploy the entire set of resources with one command, and then hack on them to develop new features, microservices, fix bugs, and more.



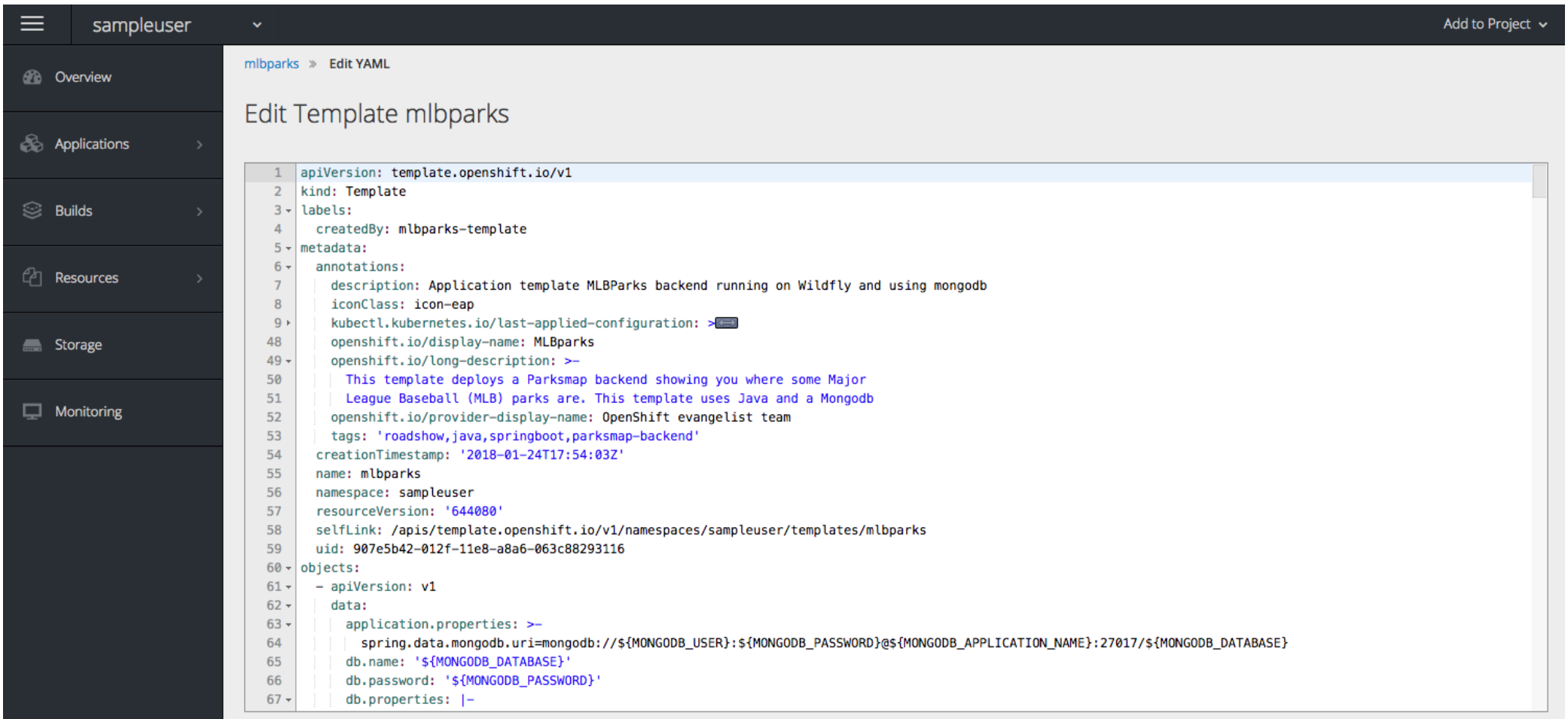
As a final exercise, look at the template that was used to create the resources for our **mlbparks** application.

```
$ oc get template mlbparks -o yaml
```
BASH

But as always, you can use the OpenShift console to do the same. Under **"Resources menu"**, click on **"Other resources"**, then select **"Templates"** from teh dropdown, and select the **"Edit YAML"** action, on the **"Actions"** dropdown.



You'll be able to see/edit the YAML as well from here.



Binary Builds for Day to Day Development

Lab: Binary Builds for Day to Day Development

Moving on From S2I

As you saw before S2I is a great way to get from source code to a container, but the process is a bit too slow for daily fast iteration development. For example, if you want to change some CSS or change one method in a class you don't want to go through a full git commit and build cycle. In this lab we are going to show you a more efficient method for quick iteration. While we are at it we will also throw in showing you how to debug your code as well.

Now that we built the MLB parks service let's go ahead and make some quick changes

Fast Iteration Code Change Using Binary Deploy

The OpenShift command line has the ability to do a deployment from your local machine. In this case we are going to use S2I, but we are going to tell OpenShift to just take the war file from our local machine and bake it in the image.

Doing this pattern of development let's us do quick builds on our local machine (benefitting from our local cache and all the horsepower on our machine) and then just quickly send up the war file.

You could also use this pattern to actually send up your working directory to the S2I builder to have it do the Maven build on OpenShift. Using the local directory would relieve you from having Maven or any of the Java toolchain on your local machine AND would also not require git commit with a push. Read more in the official documentation (https://<url>/dev_guide/dev_tutorials/binary_builds.html)

Exercise: Using binary deployment

Clone source

The first step is to clone the MLB source code from GitHub to your local machine:

```bash
git clone https://github.com/openshift-roadshow/mlbparks.git
```

We are using Intellij here in the guide for screenshots but this should work regardless of your tool chain. JBoss Developer Studio and JBoss Developer Tools have built in functionality that makes this close to seamless right from the IDE.

Setup the Build of the war file

Maven

If you have Maven all set up on your machine, then you can just do a

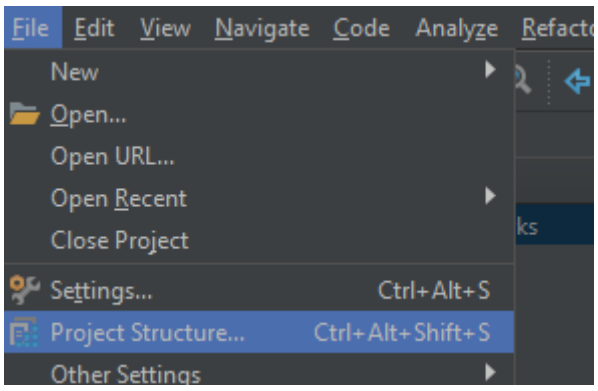
```bash
mvn package
```

Pay attention to the output location for the ROOT.war, we will need that directory later.

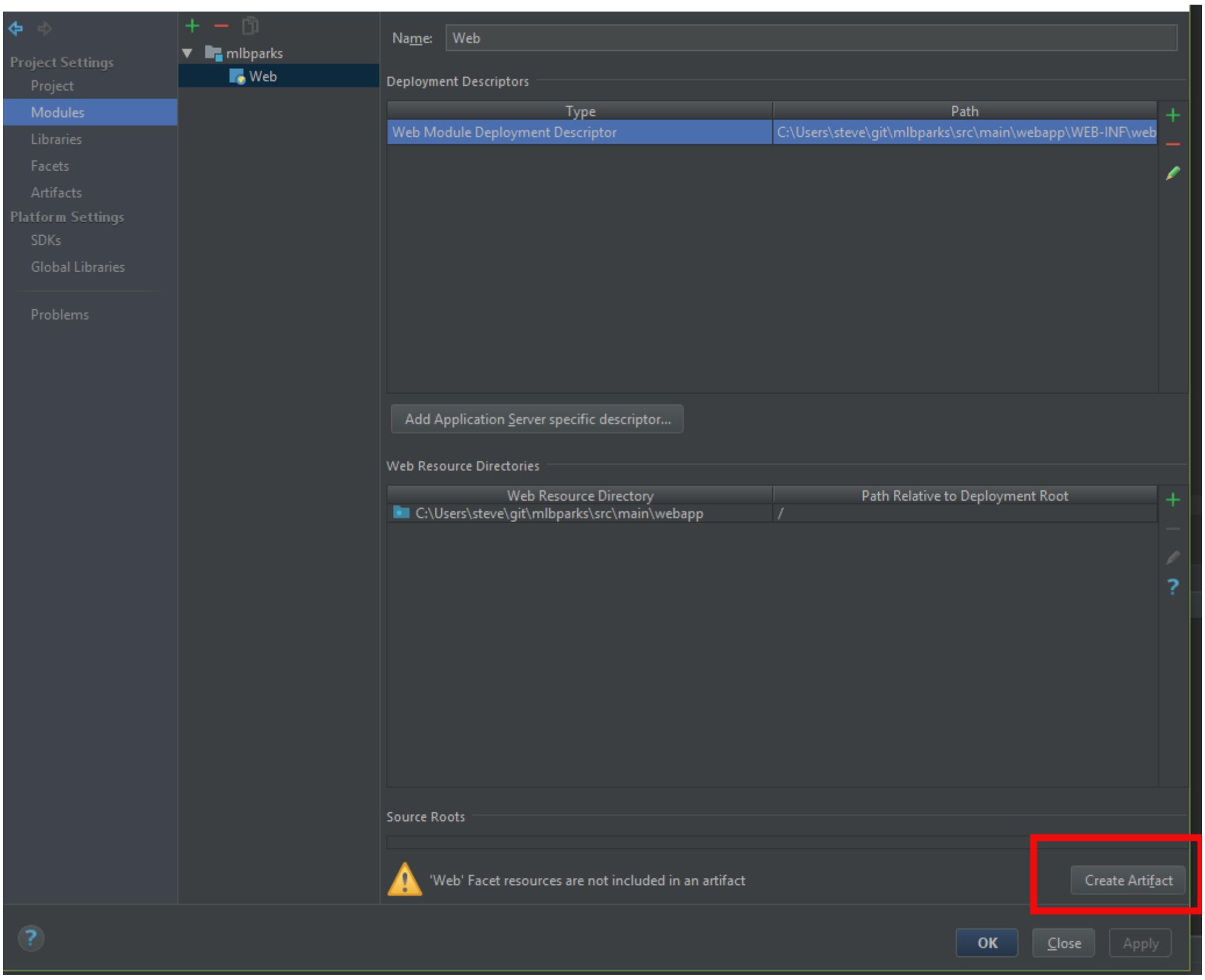
Intellij

In Intellij, once the project is imported, we need to create a configuration to build an artifact

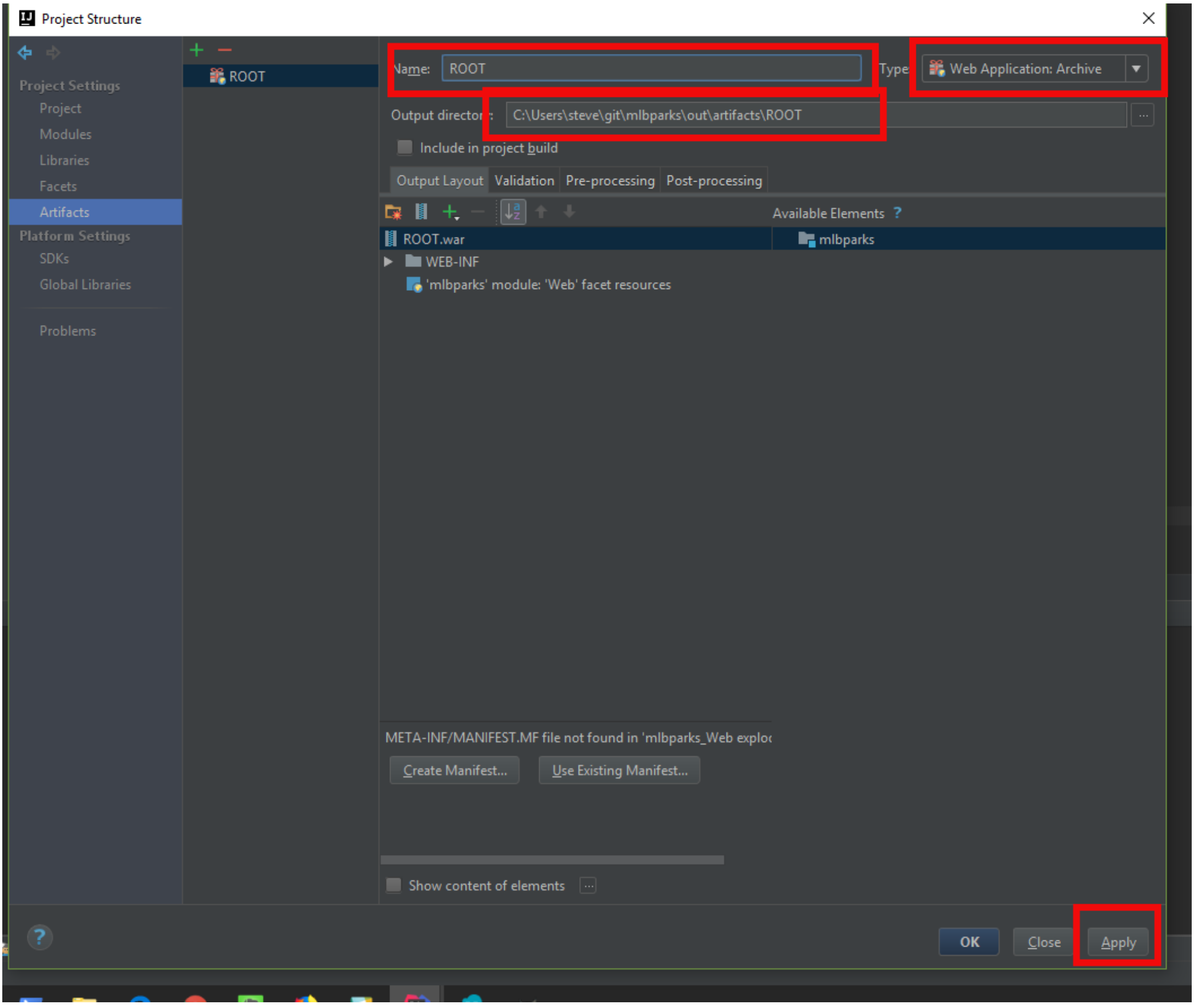
Go to Project Structure



Then got to Modules, where you will see a notice that web facet resources is not included in the artifact. Click the "create artifact" button, which will bring you to the next page in the process



Now on this page, Intellij will try to create an exploded archive, we don't want that. So on the top right change the drop down to *Web Application: Archive*. Then change the name to *ROOT* which should also change the output layout to ROOT.war. Finally, either change the output directory to the location you want for the war file or just note the location autogenerated.



Finally make sure to click Apply.

Now you can generate artifacts and it will make the ROOT.war file. Build artifacts is located under the build menu (I have mapped it to ctrl-shift-f10).

Code Change

Time for a source code change! Go to src/main/java/com/openshift/evg/roadshow/rest/BackendController.java. This is the REST endpoint that gives basic info on the service and can be reached at

```bash
http://mlbparks-project-userXY.apps.10.2.2.2.xip.io/ws/info/
```

Please change line 23 to add a *AMAZING* in front of "MLB Parks" look like this:

```java
return new Backend("mlbparks", "AMAZING MLB Parks", new Coordinates("39.82", "-98.57"), 5);
```

Don't forget to save the file.

Doing the Binary Build

Alright we have our war file built, time to kick off a build using it.

If you built your war with Maven:

```bash
oc start-build bc/mlbparks --from-file=target/ROOT.war --follow
```

The –follow is optional if you want to follow the build output in your terminal.

For Intellij:

```
oc start-build bc/mlbparks --from-file=location/you/put/in/output/ROOT.war --follow
```
BASH

Change the file path above to the location you set in Intellij

Using labels and a recreate deployment strategy, as soon as we get the new deployment finishes the map name will be updated. Under a recreate deployment strategy we first tear down the pod before doing our new deployment. When the pod is torn down, the parksmap service removes the MLB Parks map from the layers. When it comes back up, the layer automatically get's added back with our beautiful new title. This would not have happened with a rolling deployment because rolling spins up the new version of the pod before it takes down the old one. Rolling strategy enables a zero-downtime deployment.

When the deployment is finished and the container is deployed you will see the new content at:

```
http://mlbparks-project-userXY.apps.10.2.2.2.xip.io/ws/info/
```
BASH

So now you have seen how we can speed up the build and deploy process.

Using Port-Forwarding and Remote Debugging

Lab: Setting up Debugging

Background: Port Forwarding and Debugging

Just as we did before with remote shelling into our pods, we can also set up a port-forward between our local machine and our pod. The is useful for operations like connecting to a database running in a pod, viewing an administrative web interface we don't want to expose to the public, or, in our case, attach a debugger to the JVM running our application server.

You can read more about port-forwarding from the developer documentation (https://<url>/dev_guide/port_forwarding.html).

By port forwarding the debugging port for the application server, we can attach the debugger from our IDE and actually step through the code in the pod as it is running in real time. By default EAP is not in debug mode, therefore we first need to turn on the debug ports

Exercise: Enabling Debugging in EAP on OpenShift

It is very simple to turn on debugging. The EAP S2I container we are using is looking for an environment variable to control whether or not to enable the debug port. All we need to do is set an environment variable for the deployment.

```
$ oc set env dc/mlbparks DEBUG=true
```
BASH

This will force a redeploy of our MLB park pod, this time with the JDWT transport enabled and serving on port 8787.

Exercise: Port-Forwarding from the pod to our local machine

It is quite simple to do port-forwarding.

First get the pods:

```
oc get pods
NAME                        READY     STATUS      RESTARTS   AGE
mlbparks-1-build            0/1       Completed   0          9m
mlbparks-2-7c6zb            1/1       Running     0          4m
mongodb-mlbparks-1-887pq    1/1       Running     0          8m
parksmap-2-wms5l            1/1       Running     0          13m
```
BASH

Now we are all set to set up the port-forward:

```
$ oc port-forward mlbparks-2-7c6zb 8787:8787
```
BASH

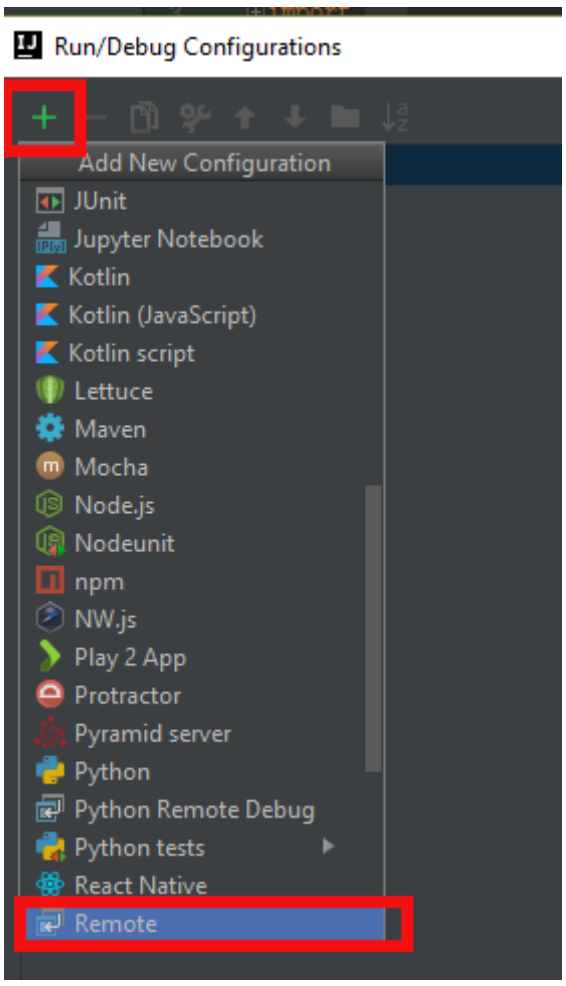
We said to port-forward from port 8787 on the pod to 8787 on the local machine. Now we can attach a remote debugger.

To stop port-forwarding just hit ctrl-c in the terminal window where you did the port forward command

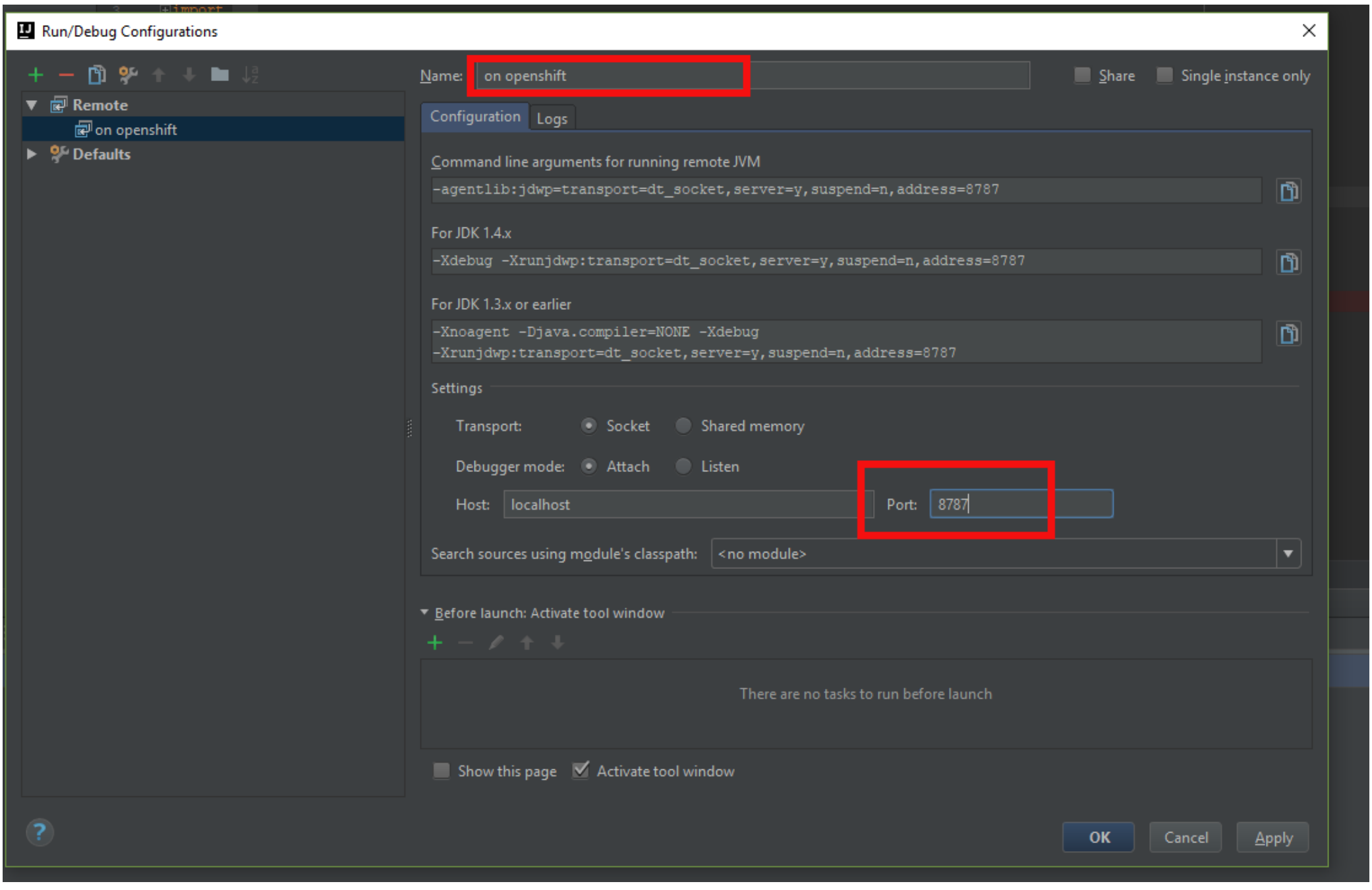
Setting up Remote Debug in Intellij

Setting up remote debugging is quite easy in Intellij. First edit the run/debug configurations. Under the Run menu (alt-u), choose edit configurations. This will bring up the new configuration dialog.

Click the plus in the top left corner, scroll down, and choose remote.



On the resulting dialog page, change the name at the top to "on OpenShift" or whatever is informative to you. Then towards the bottom right, change the port number to 8787. When you have done that click "OK".



Now when you click the Debug icon in Intellij, it will open the debugger and attach to JVM in the pod on OpenShift. Go ahead and set a break point in any class you want and it will do normal debugging - just like you know and love!

Clustering Stateful Java EE Applications

Lab: Clustering Stateful Java EE Applications

JBoss EAP Clustering

Clustering in JBoss EAP is achieved using the Kubernetes discovery mechanism for finding other JBoss EAP containers and forming a cluster. This is done by configuring the JGroups protocol stack in `standalone-openshift.xml` with `<openshift.KUBE_PING/>` elements.

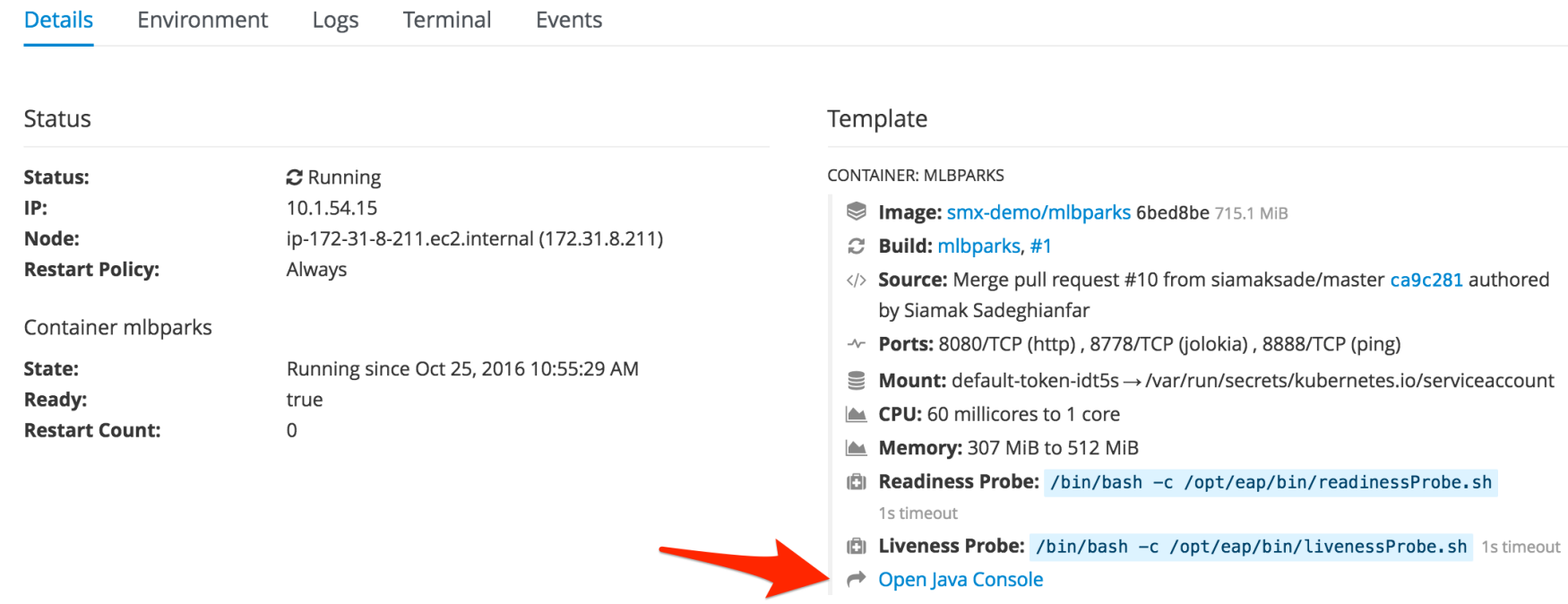
For `KUBE_PING` to work, the following steps must be taken:

1. The `OPENSHIFT_KUBE_PING_NAMESPACE` environment variable must be set. If not set, the server will act as if it is a single-node cluster (a "cluster of one").
2. The `OPENSHIFT_KUBE_PING_LABELS` environment variables should be set. If not set, pods outside of your application (albeit in your namespace) will try to join.
3. Authorization must be granted to the service account the pod is running under to be allowed to access Kubernetes' REST api.

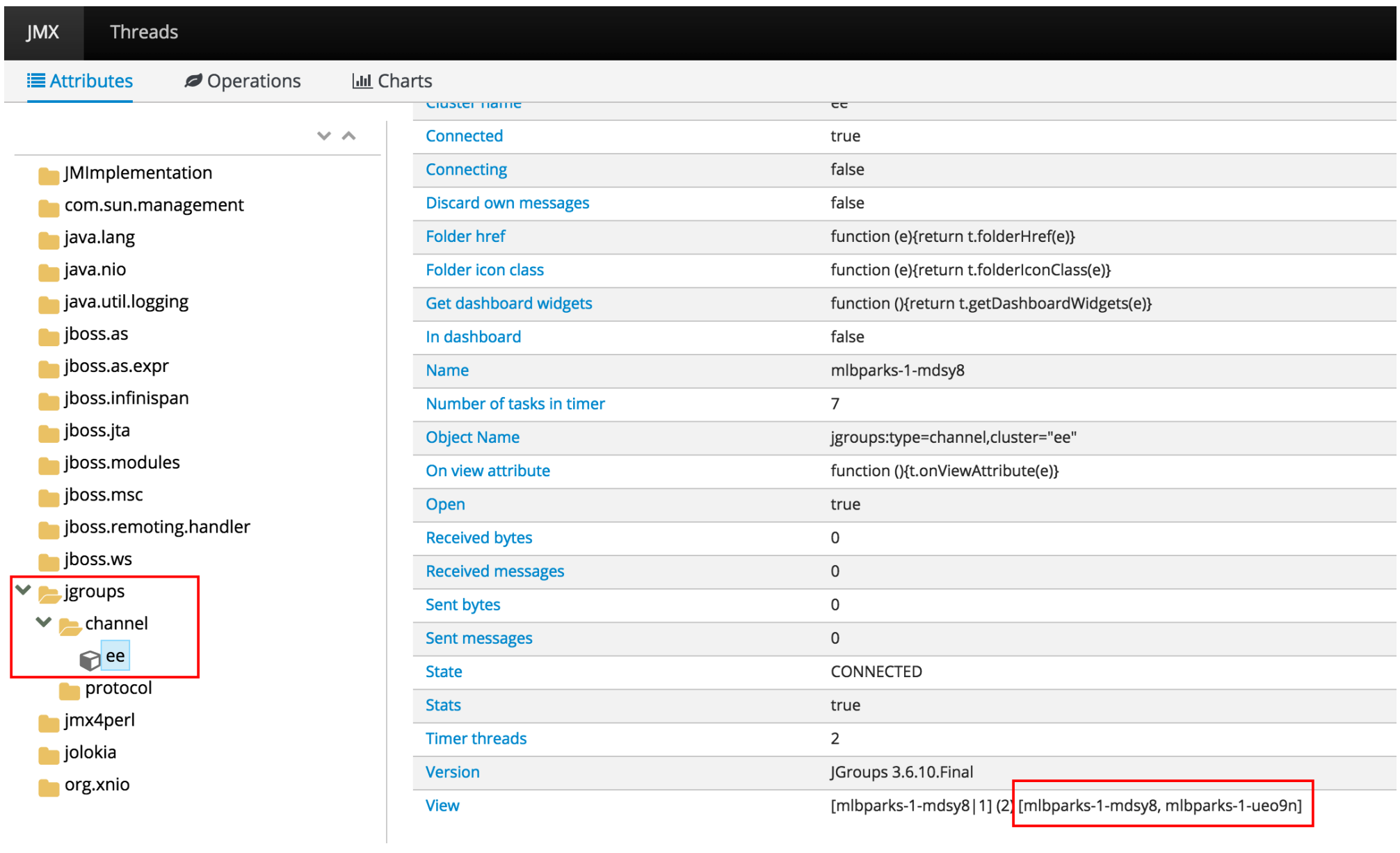
You have already configured the default service account to have access to Kubernetes' REST api during previous labs. Now you can scale up the **mlbparks** pod to two by clicking on the upper arrows on the console. Alternatively, you can use the command line to scale up the pods and form a two-member cluster.

```
$ oc scale dc/mlbparks --replicas=2
```

For pods based on Java images, the web console also exposes access to a hawt.io-based JVM console for viewing and managing any relevant Java components. A link is displayed in the pod's details on the **Applications** → **Pods** page, provided the container has a port named jolokia. On the console, click on mlbparks pods, then on any of the two pods deployed. On the **Details** tab, click on **Open Java Console**.



In the Java Console, use the **JMX** browser and click on **jgroups → channel → ee**. The right pane shows the list of clustering JMX attributes including **view** which is the current state of the cluster. This attribute shows the name of two pods which are currently members of the cluster. When **mlbparks** pod gets scaled up or down, JBoss EAP gets notified by calling Kubernetes' REST API and updates the cluster status based on the number of pods available.



Further resources

Further Resources

The exercices in this workshop has shown you how OpenShift can be used not only for deploying stateless web applications, but also applications which require persistent file system storage.

This flexibility makes OpenShift an ideal platform for deploying both web applications and databases.

If you have finished this workshop early and want to experiment some more, we have additional exercises you can try out using our online interactive learning environment.

- OpenShift Interactive Learning Portal (https://learn.openshift.com/) - An online interactive learning environment where you can run through various scenarios related to using OpenShift.

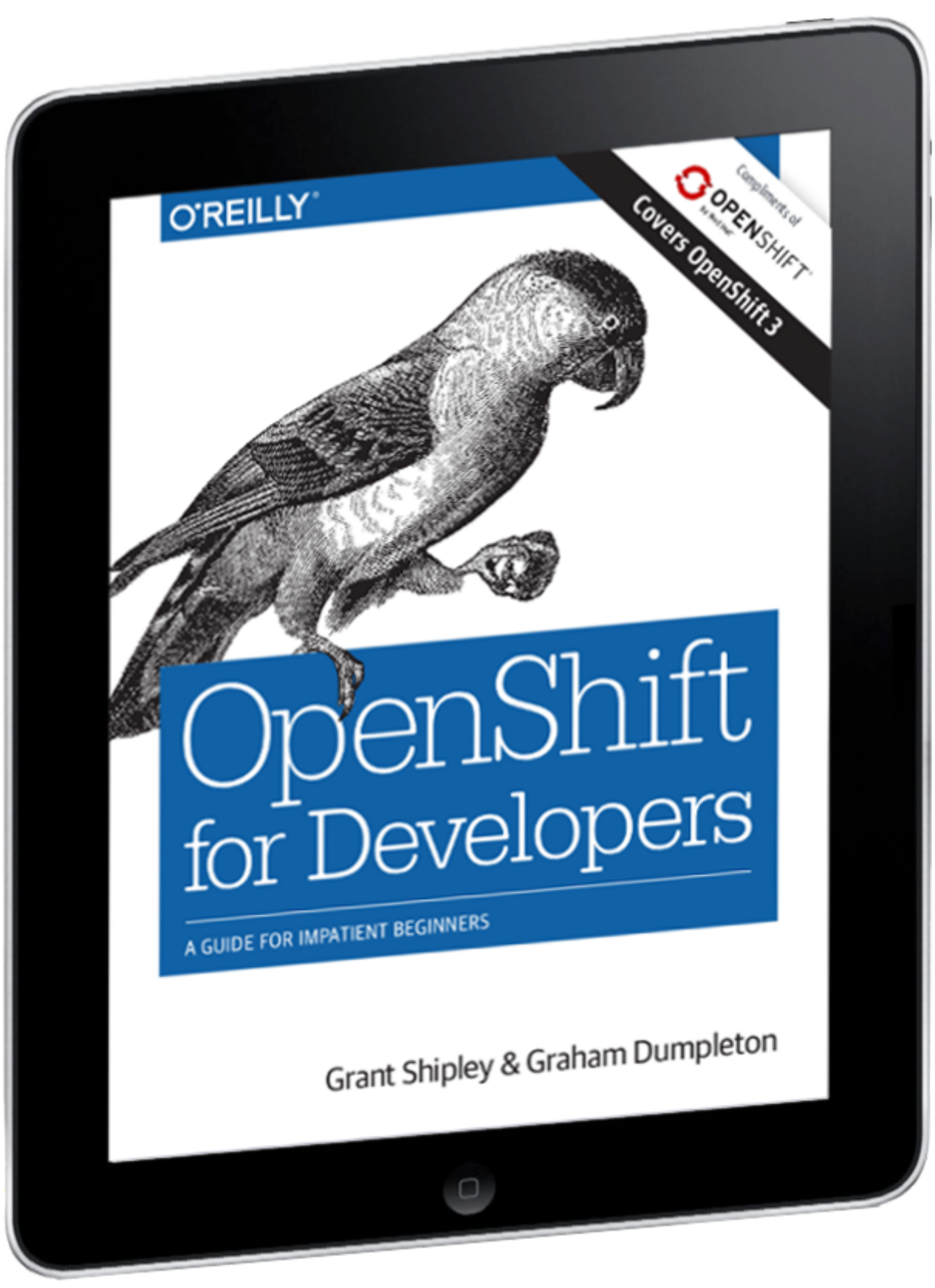
The online interactive learning environment is always available so you can continue to work on those exercises even after the workshop is over.

Below you will find further resources for learning about OpenShift and running OpenShift on your own computer, as well as details about OpenShift Online or other OpenShift related products and services.

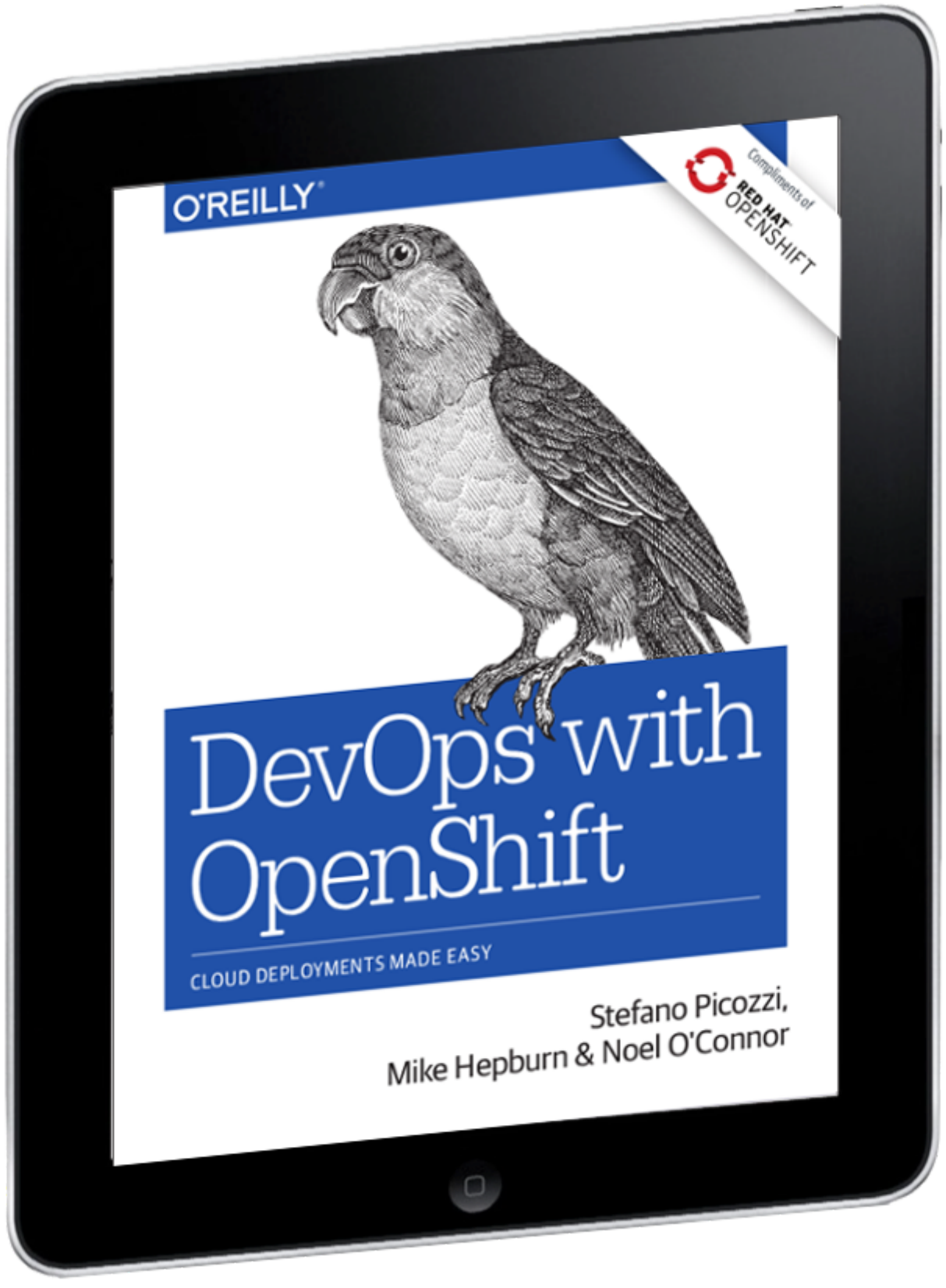
- OpenShift Origin (https://www.openshift.org/) - The upstream Open Source community project that powers OpenShift.
- Minishift (https://www.openshift.org/minishift/) - A tool which can be used to install a local OpenShift cluster on your own computer, running in a virtual machine.
- OpenShift Online (https://manage.openshift.com/) - A shared public hosting environment for running your applications using OpenShift.
- OpenShift.io (https://openshift.io/) - An online development environment for planning, creating and deploying hybrid cloud services using OpenShift.
- OpenShift Dedicated (https://www.openshift.com/dedicated) - A dedicated hosting environment for running your applications, managed and supported for you by Red Hat.
- OpenShift Container Platform (https://www.openshift.com/) - The Red Hat supported OpenShift product for installation on premise or in hosted cloud environments.

The following free online eBooks are also available for download related to OpenShift.

- OpenShift for Developers (https://www.openshift.com/promotions/for-developers.html)



- DevOps with OpenShift (https://www.openshift.com/promotions/devops-with-openshift.html)


O'REILLY
Compliments of
RED HAT
OPENSHIFT
DevOps with OpenShift
CLOUD DEPLOYMENTS MADE EASY
Stefano Picozzi,
Mike Hepburn & Noel O'Connor