
OPENSHIFT
ENTERPRISE
by Red Hat®

Contents

Introduction to OpenShift 3
About the Lab 3
Deployment Architecture 4
Lab 1: Installing the OpenShift CLI 5
Lab 2: Smoke Test and Quick Tour 7
Lab 3: Deploy a Docker Image 9
Exercise: Deploying your first Image 9
Lab 4: Creating Routes by Exposing Services 15
Exercise: Creating a Route 15
Lab 5: Remote Operations 16
Exercise: Remote Shell Session to a Container 16
Exercise: Execute a Command in a Container 17
Lab 6: Scaling and Self-Healing 17
Exercise: Scaling up 18
Application "Self-Healing" 20
Lab 7: Deploying Java Code on JBoss 20
Exercise: Creating a JBOSS EAP application 21
Lab 8: Adding a Database 25
Lab 9: Using Templates 31

Introduction to OpenShift

OpenShift is a computer software product from Red Hat for container-based software deployment and management. In concrete terms it is a supported distribution of Kubernetes using Docker containers and DevOps tools for accelerated application development.

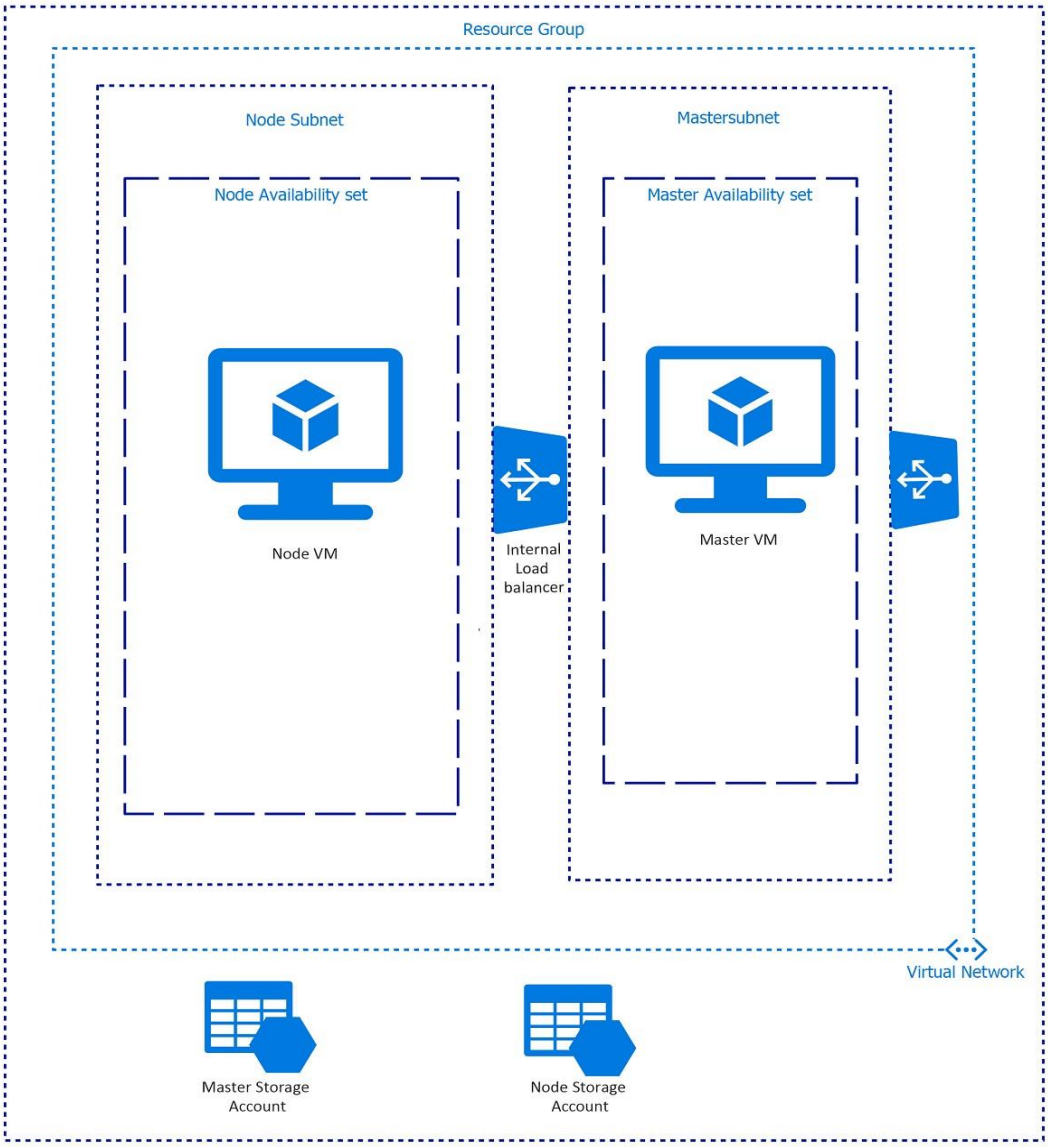
OpenShift is designed to provide one thing for Developers: Ease of Use without Worries. OpenShift's mission is to make your job easier by taking care of all the messy IT aspects of app development and allowing you to focus on your job: Coding your Application and satisfying your customers.

OpenShift Container Platform (formerly known as **OpenShift Enterprise**) is Red Hat's on-premise private platform as a service product, built around a core of application containers powered by Docker, with orchestration and management provided by Kubernetes, on a foundation of Red Hat Enterprise Linux.

About the Lab

This lab will help you experience the capabilities of OpenShift Container Platform. 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 sample application that we will be deploying as part of this exercise is called national parks. This application is a Java EE-based application that performs 2D geo-spatial queries against a MongoDB database to locate and map all National Parks in the world. That was just a fancy way of saying that we are going to deploy a map of National Parks. Are you ready to take the Driver seat and experience a Red Hat OpenShift test drive……?

Deployment Architecture



Lab 1: 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 OpenShift origin client tools v3.9 for Windows
- Mac OS X OpenShift origin client tools v3.9 for Mac
- Linux 64 OpenShift origin client tools v3.9 for Linux 64

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: `C:\OpenShift`

OS X: `~/OpenShift`

Linux: `~/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:

```
$ tar zxvf oc-macosx.tar.gz
```

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:

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

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.

OS X:

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

Linux:

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

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
```

You should see the following (or something similar):

```
oc v3.9.0+191fece

kubernetes v1.9.1+a0ce1bc657

features: Basic-Auth
```

If you get an error message, you have not updated your path correctly.

END OF LAB

Lab 2: Smoke Test and Quick Tour

COMMAND LINE

Once the OpenShift environment has been provisioned, the first thing we want to do is ensure that we can login to the OpenShift environment. In order to login, we will use the oc command and then specify the server that we want to authenticate to. Issue the following command:

```
$ oc login masterdnsyxoson5ssgwcq.westeurope.cloudapp.azure.com
```

Note: *After entering in the above command, you may be prompted to accept the security certificate.*

You may see the following output:

```
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 test drive, 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 (e.g.: Thawte, Verisign, StartSSL, etc.) or signed by a corporate-standard CA that you already have installed on your system.

Once you issue the oc login command, you will be prompted for the username and password combination for your user account.

Username: `ocpadmin`
Password: `password`

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

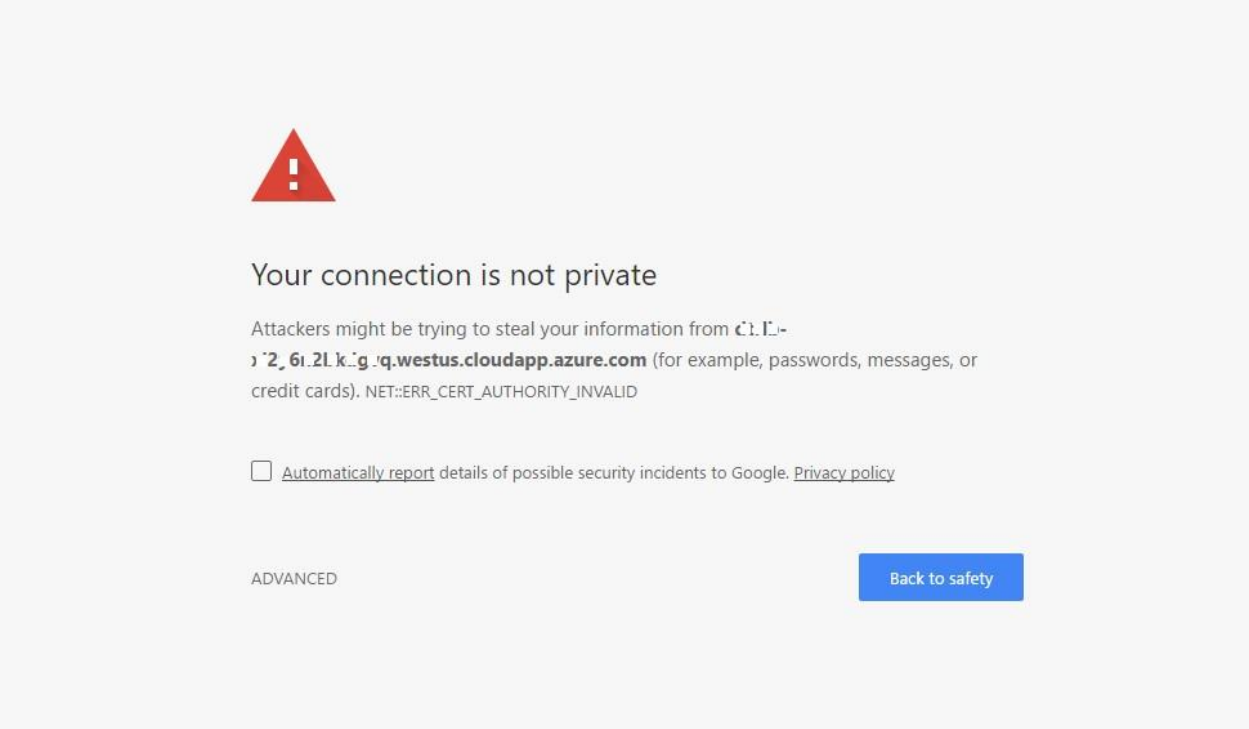
```
Login successful.

You have access to the following projects and can switch between them with
'oc project <projectname>':

… <list of available projects> …
```

OPENSHIFT 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://masterdnsyxoson5ssgwcq.westeurope.cloudapp.azure.com/console

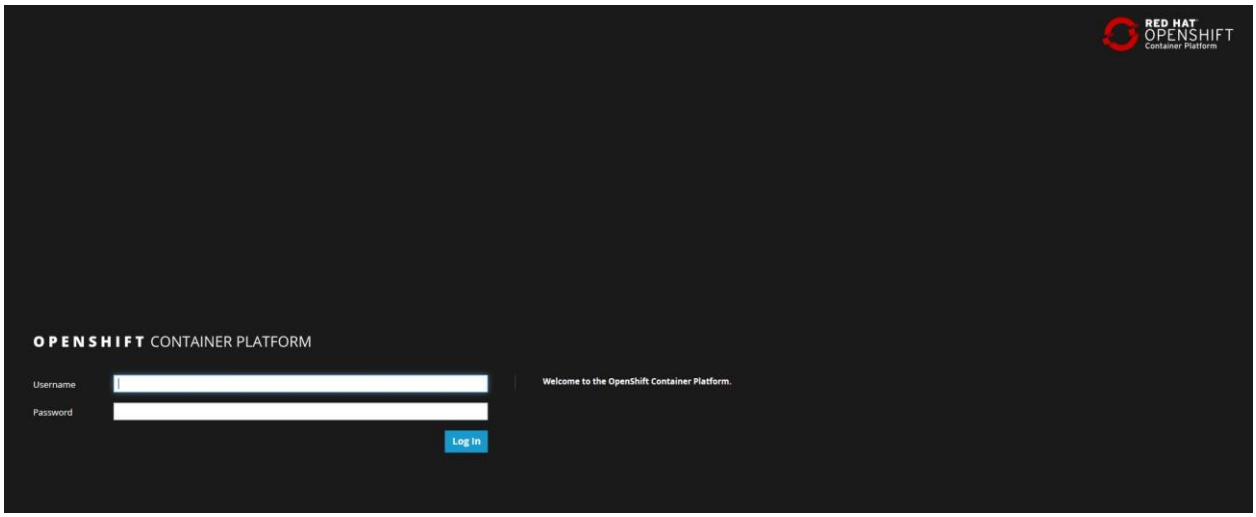
When we enter the URL in the browser you see a unsecure site issue as shown below. It is safe to click on **ADVANCED** and click on Proceed to the site.



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

Username: `ocpadmin`
Password: `password`

OpenShift Login Screen



After you have authenticated to the web console, you will be presented with a list of projects that your user has permission to work with.

END OF LAB

Lab 3: Deploy a Docker Image

Background: Containers and Pods

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 doc:

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 analyser, and a file service to help manage uploaded files.

Exercise: Deploying your first Image

Let's start by doing the simplest thing possible - get a plain old Docker image to run inside of OpenShift. This is incredibly simple to do. We are going to use the Kubernetes Guestbook application (https://registry.hub.Docker.com/u/kubernetes/guestbook/) for this example.

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.

The first thing we want to do is create a new Project called guestbook. Remember that Projects group resources together.

```
$ oc new-project usertestdrive-guestbook-<initials>
```

where <initials> are your initials in lowercase (to make sure that you don't use the same project name as someone else running the lab at the same time). The new-project command will automatically switch you to use that Project. You will see something like the following:

```
Now using project "usertestdrive-guestbook-ta" on server "https://
masterdnsyxoson5ssgwcq.westeurope.cloudapp.azure.com:443".

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

    oc new-app centos/ruby-22-centos7~https://github.com/openshift/ruby-ex.git

to build a new example application in Ruby.
```

To see all the Projects, you have access to, you can simply use oc get.

```
$ oc get projects
```

You should see a list like the following:

```
NAME                        DISPLAY NAME   STATUS
default                                    Active
kube-public                                Active
kube-system                                Active
logging                                    Active
management-infra                           Active
openshift                                  Active
openshift-infra                            Active
openshift-node                             Active
openshift-web-console                      Active
usertestdrive-guestbook                    Active
```

With the new Project created, in order to tell OpenShift to define and run the Docker image, you can simply execute the following command:

```
$ oc new-app kubernetes/guestbook
```

You will see output similar to the following:

```
--> Found Docker image 4305190 (3 years old) from Docker Hub for
"kubernetes/guestbook"

* An image stream will be created as "guestbook:latest" that will track this image
* This image will be deployed in deployment config "guestbook"
* Port 3000/tcp will be load balanced by service "guestbook"
* Other containers can access this service through the hostname "guestbook"     *
  WARNING: Image "guestbook" runs as the 'root' user which may not be permitted by
  your cluster administrator

--> Creating resources with label app=guestbook ...
    imagestream "guestbook" created
    deploymentconfig "guestbook" created
    service "guestbook" created
--> Success
    Application is not exposed. You can expose services to the outside world by
    executing one or more of the commands below:
    'oc expose svc/guestbook'
    Run 'oc status' to view your app.
```

Pretty easy, huh?

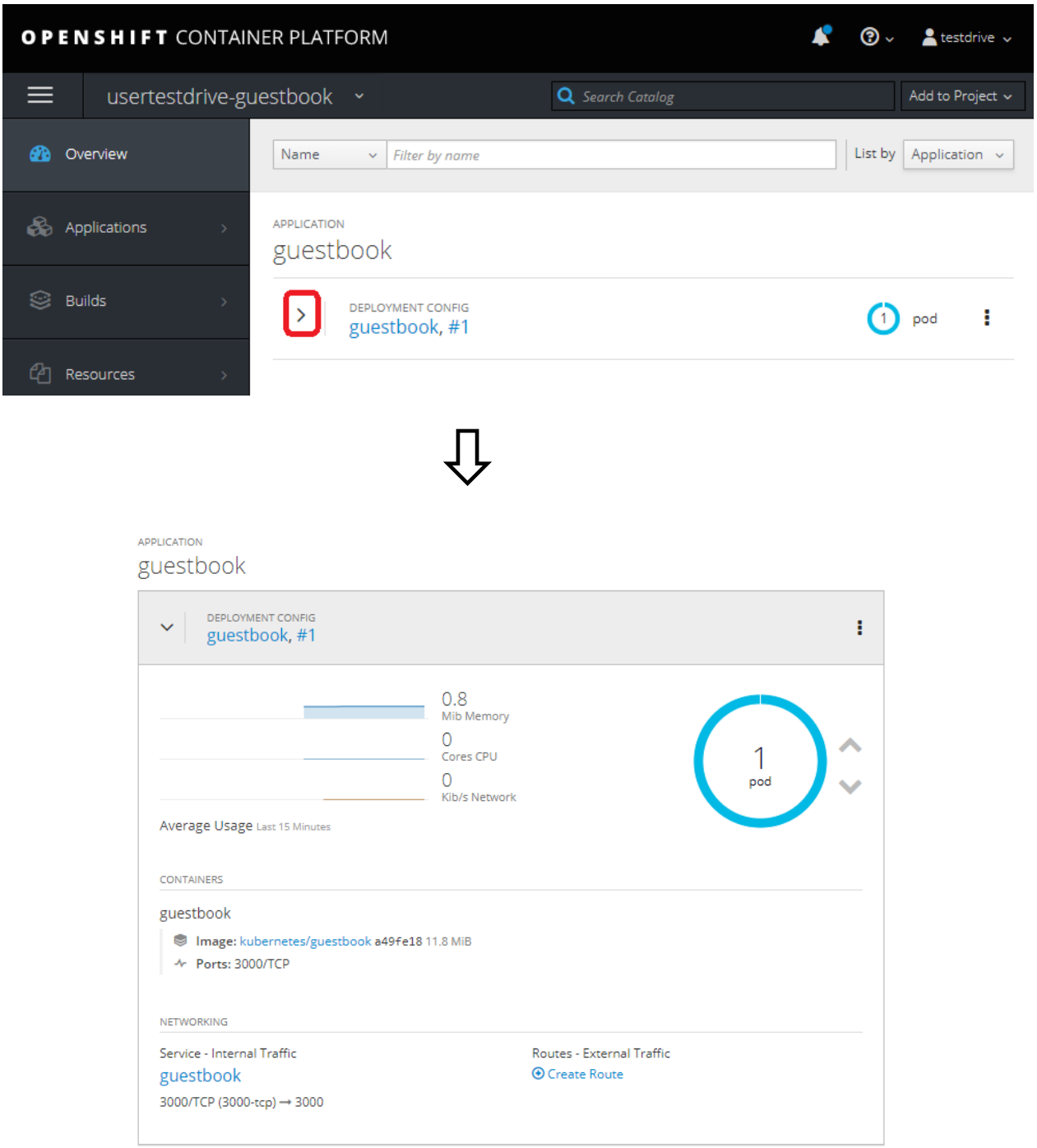
This may take a while to complete. Each OpenShift node has to pull (download) the Docker image for kubernetes/guestbook from the Docker hub if it does not already have it locally. You can check on the status of the image download and deployment by:

1.Going into the web console

2.Select Project usertestdrive-guestbook-<*initials*>

3.Expand the section of the guestbook deployment config (see figure below)

4.Select Pods

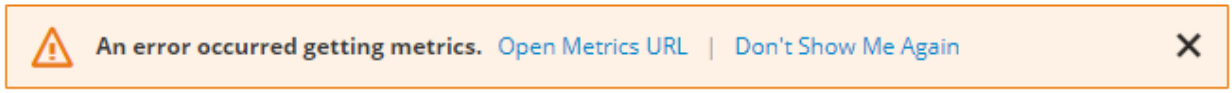


You can also use the oc command line tool to watch for changes in pods:

```
$ oc get pods
```

To exit, hit Control+C(^c).

NOTE: If you see the following message on your browser,



Then click on "Open Metrics URL" and accept the certificate that you see for the hawkular metrics (similar to what you did for Lab #2).

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 behaviour. They are documented in specifying an image as well as image pull policy.

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

Note: *It is important to understand that, for security reasons, OpenShift does not allow the deployment of Docker images that run as root by default. 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 for OpenShift, or you can look at the section on enabling images to run with a USER in the Docker file.*

Background: Services

You may be wondering how you can access this application. There was a Service that was created, but Services are only used inside OpenShift - they are not exposed to the outside world by default. Don't worry though, we will cover that later in this lab.

You can see that when we ran the new-app command, OpenShift actually created several resources behind the scenes in order to handle deploying this Docker image. new-app created a Service, which maps to a set of Pods (via Labels and Selectors). Services are assigned an IP address and port pair that, when accessed, balance across the appropriate back end (Pods).

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 Guestbook 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.

There is a lot more information about 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 kubernetes/guestbook image that we just deployed. In order to view the Services defined in your Project, enter in the following command:

```
$ oc get services
```

You should see output similar to the following:

```
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
guestbook   ClusterIP   172.30.69.222   <none>        3000/TCP   12m
```

In the above output, we can see that we have a Service named guestbook with an IP/Port combination of 172.30.69.222/3000. Your IP address may be different, as each Service receives a unique IP address upon creation. Service IPs never change for the life of the Service.

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

```
$ oc get service guestbook -o json
```

You should see output similar to the following:


```
{
    "apiVersion": "v1",
    "kind": "Service",
    "metadata": {
        "annotations": {
            "openshift.io/generated-by": "OpenShiftNewApp"
        },
        "creationTimestamp": "2018-04-15T19:20:54Z",
        "labels": {
            "app": "guestbook"
        },
        "name": "guestbook",
        "namespace": "usertestdrive-guestbook",
        "resourceVersion": "9998",
        "selfLink": "/api/v1/namespaces/usertestdrive-guestbook/services/guestbook",
        "uid": "1dccb664-40e2-11e8-9499-000d3a946527"
    },
    "spec": {
        "clusterIP": "172.30.69.222",
        "ports": [
            {
                "name": "3000-tcp",
                "port": 3000,
                "protocol": "TCP",
                "targetPort": 3000
            }
```

```
        ],
        "selector": {
            "app": "guestbook",
            "deploymentconfig": "guestbook"
        },
        "sessionAffinity": "None",
        "type": "ClusterIP"
    },
    "status": {
        "loadBalancer": {}
    }
}
```

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

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

$ oc get pods

You should see output similar to the following:

```
NAME                READY     STATUS    RESTARTS   AGE
guestbook-1-bdg5t   1/1       Running   0          15m
```

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

`$ oc get pod guestbook-1-bdg5t -o json`

Under the metadata section you should see the following:

```
"labels": {
   "app": "guestbook",
   "deployment": "guestbook-1",
   "deploymentconfig": "guestbook"
},
```

- The Service has selector stanza that refers to app=guestbook, deploymentconfig=guestbook.
- The Pod has multiple Labels:
    - deploymentconfig=guestbook
    - app=guestbook

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:

`$ oc describe service guestbook`

You should see the following output:

```
Name:              guestbook
Namespace:         usertestdrive-guestbook
Labels:            app=guestbook
Annotations:       openshift.io/generated-by=OpenShiftNewApp
Selector:          app=guestbook,deploymentconfig=guestbook
Type:              ClusterIP
IP:                172.30.69.222
Port:              3000-tcp  3000/TCP
TargetPort:        3000/TCP
Endpoints:         10.130.0.5:3000
Session Affinity:  None
Events:            <none>
```

You may be wondering why only one end point is listed. That is because there is only one guestbook Pod 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 guestbook Service.

END OF LAB

Lab 4: Creating Routes by Exposing Services

Background: Routes

By default, the new-app command does not expose the Service it creates to the outside world. If you want to expose a Service as an HTTP endpoint you can easily do this with a Route. The OpenShift router uses the HTTP header of the incoming request to determine where to proxy the incoming request. 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. 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        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
guestbook   ClusterIP   172.30.69.222   <none>        3000/TCP   12m
```

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

```
$ oc expose service guestbook
```

```
route "guestbook" exposed
```

Verify the Route was created with the following command:

```
$ oc get route
```

```
NAME        HOST/PORT                                          PATH   SERVICES    PORT       TERMINATION   WILDCARD
guestbook   guestbook-usertestdrive-guestbook.52.173.141.139.nip.io          guestbook   3000-tcp                 None
```

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

Route http://guestbook-usertestdrive-guestbook.52.173.141.139.nip.io/

Pretty nifty, huh? This application is now available at the above URL.

END OF LAB

Lab 5: 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

OpenShift allows establishing remote shell sessions to a container without the need to run an SSH service inside each container. In order to SSH into 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
guestbook-1-bdg5t    1/1       Running   0          1h
```

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

```
$ oc rsh guestbook-1-bdg5t
```

You would see the following output:

```
$ oc rsh guestbook-1-bdg5t

BusyBox v1.21.1 (Ubuntu 1:1.21.0-1ubuntu1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/app #
```

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 static files for the guestbook application within the container:

```
$ ls public/
index.html          script.js       style.css
```

Exercise: Execute a Command in a Container

In addition to remote shell, it is also possible to run a command remotely on an already running container using the *oc exec* command.

In order the get the list of files in the *public* directory of the container, run the following:

```
$ oc exec guestbook-1-bdg5t ls public
```

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

```
$ oc rsh guestbook-1-bdg5t ls public
index.html          script.js       style.css
```

END OF LAB

Lab 6: 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, Replication Controllers (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 Replication Controller is needed. Without an RC, any Pods that are killed or somehow die are not automatically restarted, either. Replication Controllers are how OpenShift "self-heals".

A Deployment Configuration (DC) defines how something in OpenShift should be deployed. From the deployments documentation:

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, Replication Controller and Deployment Configuration 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: Scaling up

Now that we know what a Replication Controller and Deployment Config are, we can start to explore scaling in OpenShift. Take a look at the Deployment Config (DC) that was created for you when you told OpenShift to stand up the guestbook image:

```
$ oc get dc
```

```
NAME        REVISION   DESIRED   CURRENT   TRIGGERED BY
guestbook   1          1         1         config,image(guestbook:latest)
```

To get more details, we can look into the Replication Controller (RC).

Take a look at the Replication Controller (RC) that was created for you when you told OpenShift to stand up the guestbook image:

```
$ oc get rc
```

```
NAME          DESIRED   CURRENT   READY     AGE
guestbook-1   1         1         1         1h
```

Once you know the name of RC, you can use the following command:

```
$ oc get rc guestbook-1 -o json
```

If you look at the output, you will be able to notice the following lines:

```
  "spec": {
     "replicas": 1,
```

and

```
  "status": {
…
     "replicas": 1,
```

This lets us know that, right now, we expect one Pod to be deployed (spec), and we have one Pod actually deployed (status). By changing the spec, we can tell OpenShift that we desire a different number of Pods.

Ultimately, OpenShift's auto scaling capability will involve monitoring the status of an "application" and then manipulating the RCs accordingly.

Let's scale our guestbook "application" up to 3 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.

```
$ oc scale --replicas=3 dc/guestbook
```

If you switch to the OpenShift web console, you will notice that three pods have been requested, which will eventually be set to a status of active (dark blue).

To verify that we changed the number of replicas by modifying the RC object, issue the following command:

```
$ oc get rc
```

```
NAME        DESIRED   CURRENT   READY     AGE
guestbook-1   3         3         3         1h
```

You can see that we now have 3 replicas. Let's verify that with the oc get pods command:

```
$ oc get pods
```

```
NAME                 READY     STATUS    RESTARTS   AGE
guestbook-1-8flq2    1/1       Running   0          2m
guestbook-1-bdg5t    1/1       Running   0          1h
guestbook-1-sbnfl    1/1       Running   0          2m
```

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

```
$ oc describe svc/guestbook
```

You will see something like the following output:

```
Name:              guestbook
Namespace:         usertestdrive-guestbook
Labels:            app=guestbook
Annotations:       openshift.io/generated-by=OpenShiftNewApp
Selector:          app=guestbook,deploymentconfig=guestbook
Type:              ClusterIP
IP:                172.30.69.222
Port:              3000-tcp  3000/TCP
TargetPort:        3000/TCP
Endpoints:         10.130.0.5:3000,10.130.0.6:3000,10.130.0.7:3000
Session Affinity:  None
Events:            <none>
```

That's how simple it is to scale up Pods in a Service. Application scaling can happen extremely quickly because OpenShift is just launching new instances of an existing Docker image that 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 three 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:

```
$ oc delete pod guestbook-1-8flq2
```

Then, as fast as you can, switch to the Openshift Web Console to see what happens.

Now run again:

```
$ oc get pods
```

Did you notice anything different? The names of the Pods are slightly changed. That's because OpenShift almost immediately detected that the current state (2 Pods) didn't match the desired state (3 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 OpenShift decided that our guestbook application instance wasn't alive, it would kill the instance and then start another one, always ensuring that the desired number of replicas was in place.

END OF LAB

Lab 7: Deploying Java Code on JBoss

Background: Source-to-Image (S2I)

In lab three we learned how to deploy a pre-existing Docker image from a Docker registry. Now we will expand on that a bit by learning how OpenShift builds using source code from an existing repository.

Source-to-Image (S2I) is another open source project sponsored by Red Hat. Its 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 Docker files, 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 test drive, but you can find more information about it either in the OpenShift S2I documentation or on GitHub. The only key concept you need to remember about S2I is that it's magic.

Exercise: Creating a JBOSS EAP application

The sample application that we will be deploying as part of this exercise is called national parks. This application is a Java EE-based application that performs 2D geo-spatial queries against a MongoDB database to locate and map all National Parks in the world. That was just a fancy way of saying that we are going to deploy a map of National Parks.

Create Project

The first thing you need to do is create a new project called nationalparks:

```
$ oc new-project usertestdrive-nationalparks-<initials>
```

where <initials> are your initials.

Using application code on embedded GitLab

OpenShift can work with Git repositories on GitHub, GitLab,... You can even register web hooks to initiate OpenShift builds triggered by any update to the application code on your Git hosting solution.

The repository that we are going to use is located at the following URL:

https://github.com/anagna/nationalparks

If you are familiar with Java EE applications, you will notice that there is nothing special about our application - it is a standard, plain-old JEE application.

Combine the code with the Docker image on OpenShift

While the new-app command makes it very easy to get OpenShift to build code from a GitHub/GitLab repository into a Docker image, we can also use the web console to do the same thing -- it's not all command line and green screen where we're going! Let's use the provided gitlab repository with OpenShift's JBoss EAP S2I image.

In the **OpenShift web console**, find your usertestdrive-nationalparks project, and then click the "Browse Catalog" button. You will see a number of runtimes that you can choose from, but you will want to select the one titled “JBoss EAP 7.0 (no https)”. As you might guess, this is going to use an S2I builder image that contains JBoss EAP.

After you select the runtime, click “Next” to go to the Configuration screen.

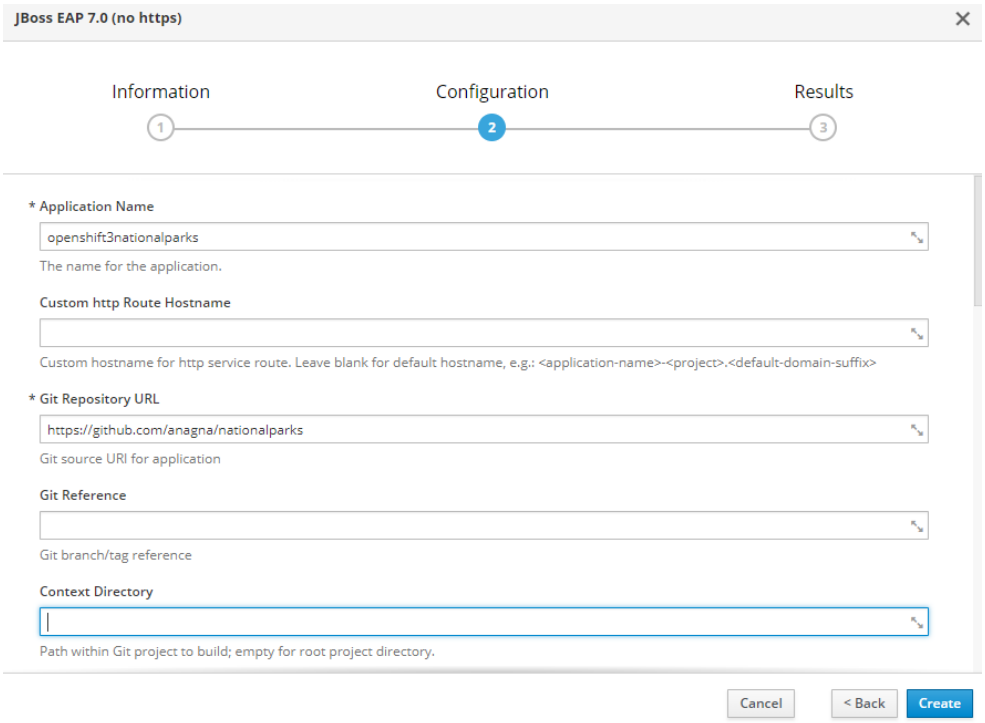
For the Application Name, enter `openshift3nationalparks`, and for the Git repository URL, enter:

https://github.com/anagna/nationalparks

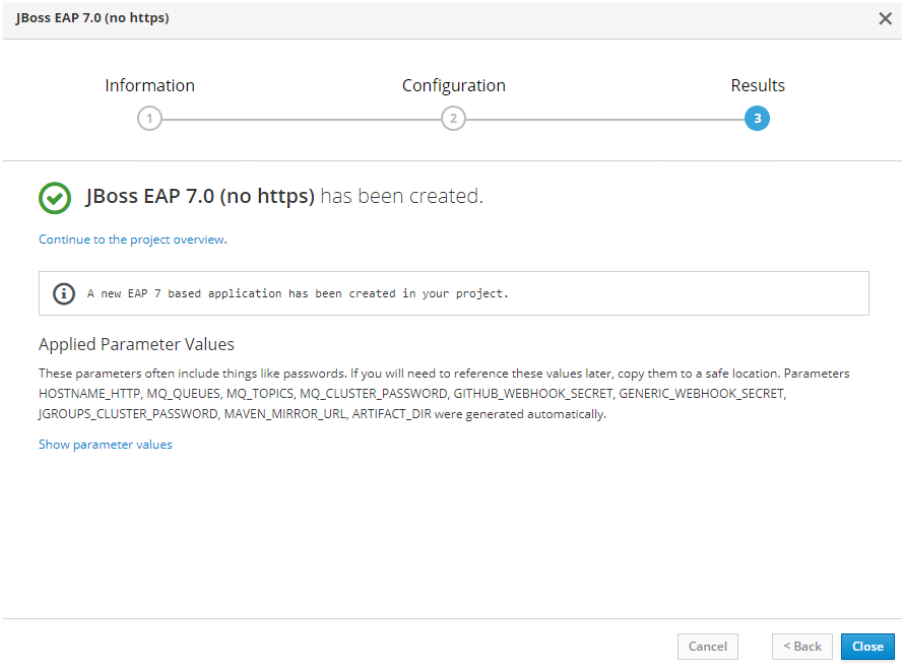
Then delete the values both in `Git Reference` and in `Context Directory`, so that these fields are blank.

You can leave all other fields to their default values.

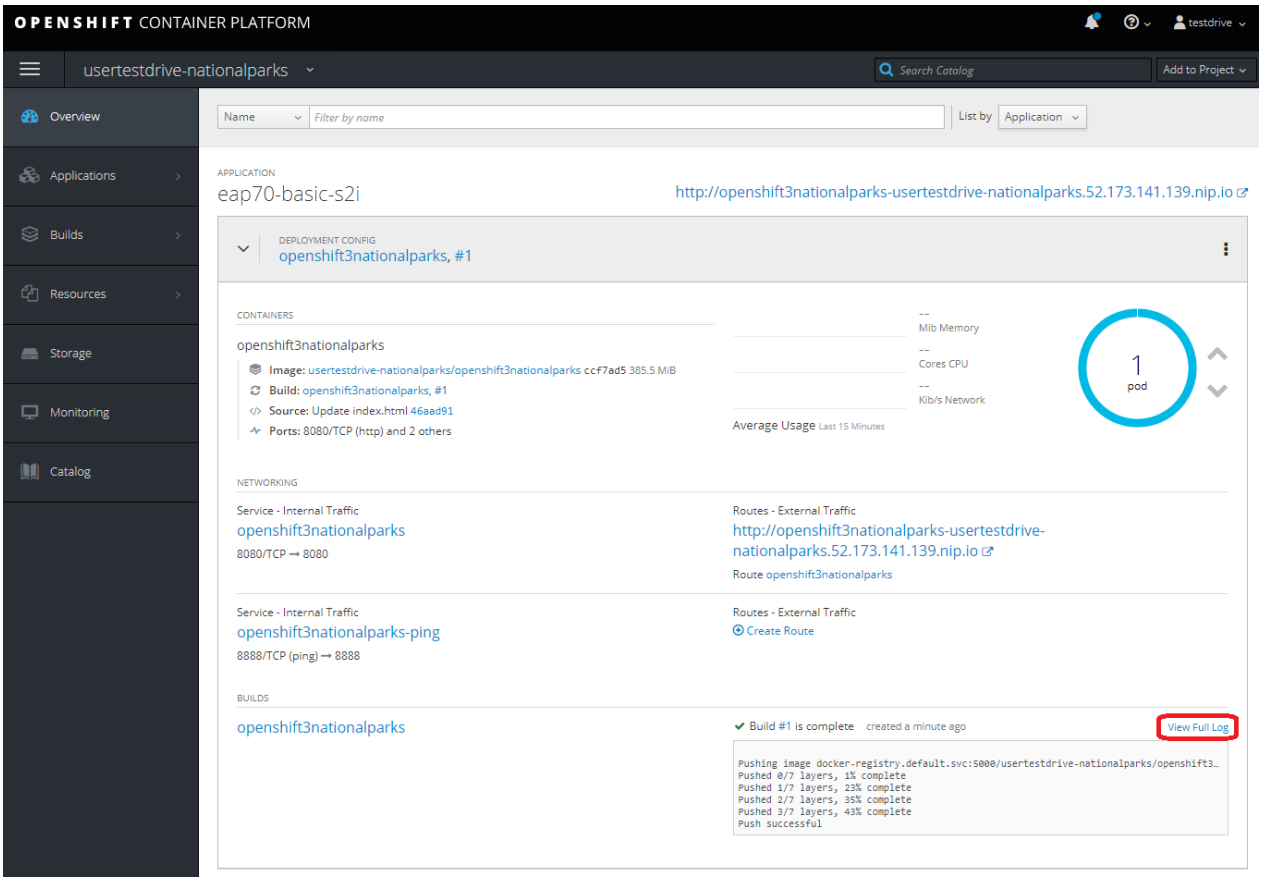
Note: *All of these runtimes shown are made available via Templates, which will be discussed in a later lab.*



You can then hit the button labelled "Create".



Then click Close and select the Overview tab from the menu on the left. You will eventually see this in the web console:



View Log

Go ahead and click "View Full Log" at the bottom right of the screen. This is a new Java-based project 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:

```
$ oc get builds
```

You'll see output like:

```
NAME                          TYPE      FROM          STATUS     STARTED          DURATION
openshift3nationalparks-1     Source    Git@46aad91   Complete   4 minutes ago    44s
```

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

```
$ oc logs -f openshift3nationalparks-1-build
```

After the build has completed and successfully:

- The S2I process will push the resulting Docker 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.

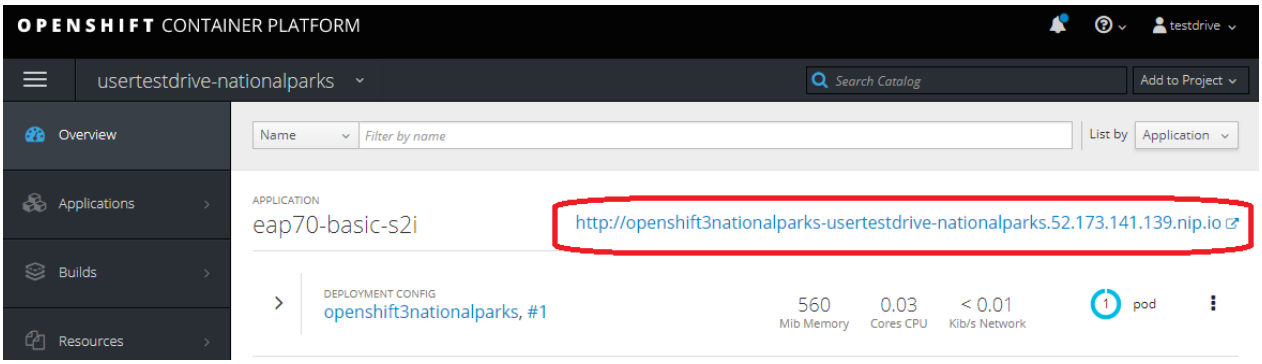
In the end, when issuing the `oc get pods` command, you will see that the build Pod has finished (exited) and that an application Pod is in a ready and running state:

```
NAME                                READY     STATUS      RESTARTS   AGE
openshift3nationalparks-1-build     0/1       Completed   0          5m
openshift3nationalparks-1-mr5gp     1/1       Running     0          4m
```

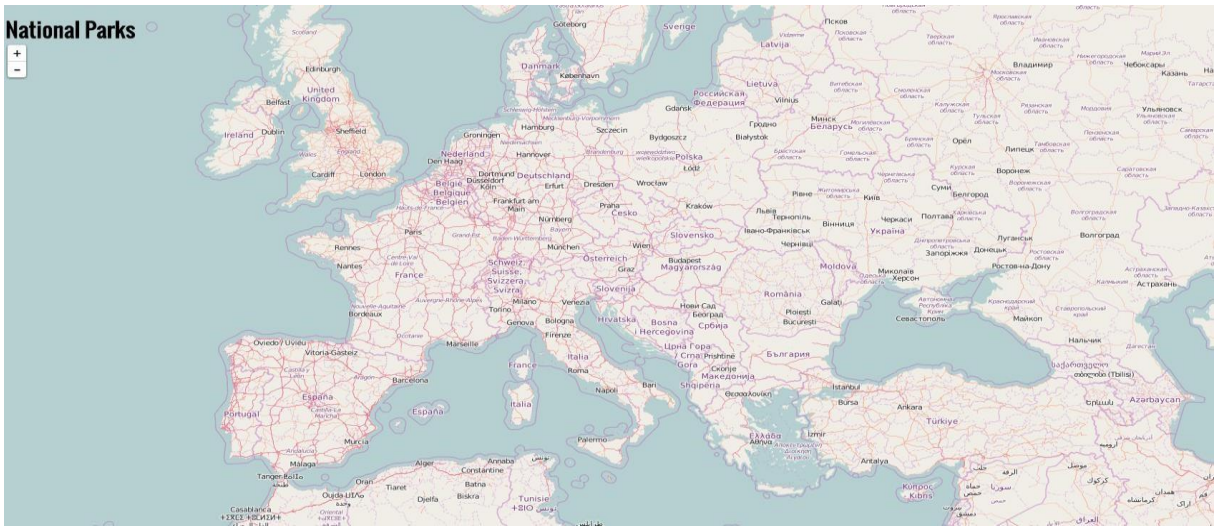
If you look again at the web console, you will notice that, when you create the application this way, OpenShift also creates a Route for you. You can see the URL in the web console, or via the command line:

```
$ oc get routes
```

Now verify your application is working by going back to the project overview screen and clicking on the route URL.



You should see the following:



Wait a second! Why are the national parks not showing up? Well, that is because we haven't actually added a database to the application yet. We will do that in the next lab. Congratulations on deploying your first application using S2I on the OpenShift Platform!

END OF LAB

Lab 8: Adding a Database

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

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 with the database Pods to give them a persistent place to store their data.

Environment Variables

As you saw in the last lab, the web console makes it pretty easy to deploy application components as well. When we deploy the database, we need to pass in some environment variables to be used inside the container. These environment variables are required to set the username, password, and name of the database. You can change the values of these environment variables to anything you would like. The variables we are going to be setting are as follows:

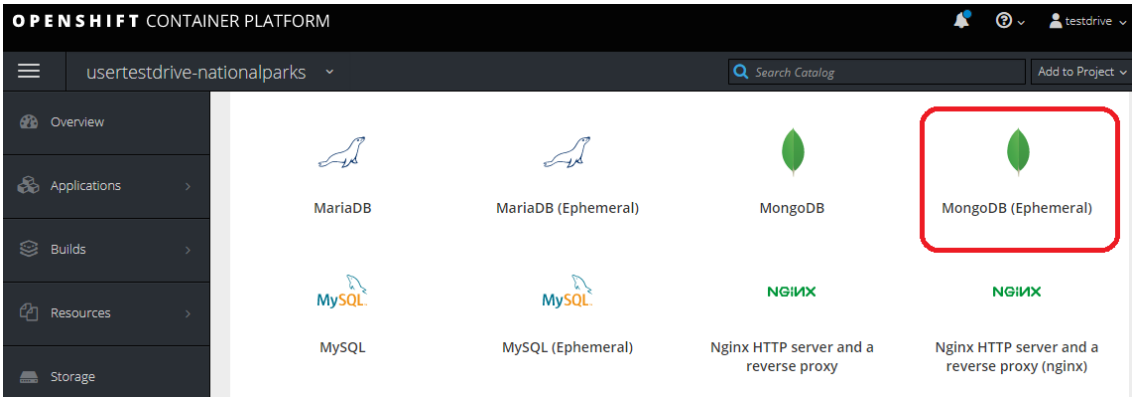
- MONGODB_USER
- MONGODB_PASSWORD
- MONGODB_DATABASE

- MONGODB_ADMIN_PASSWORD

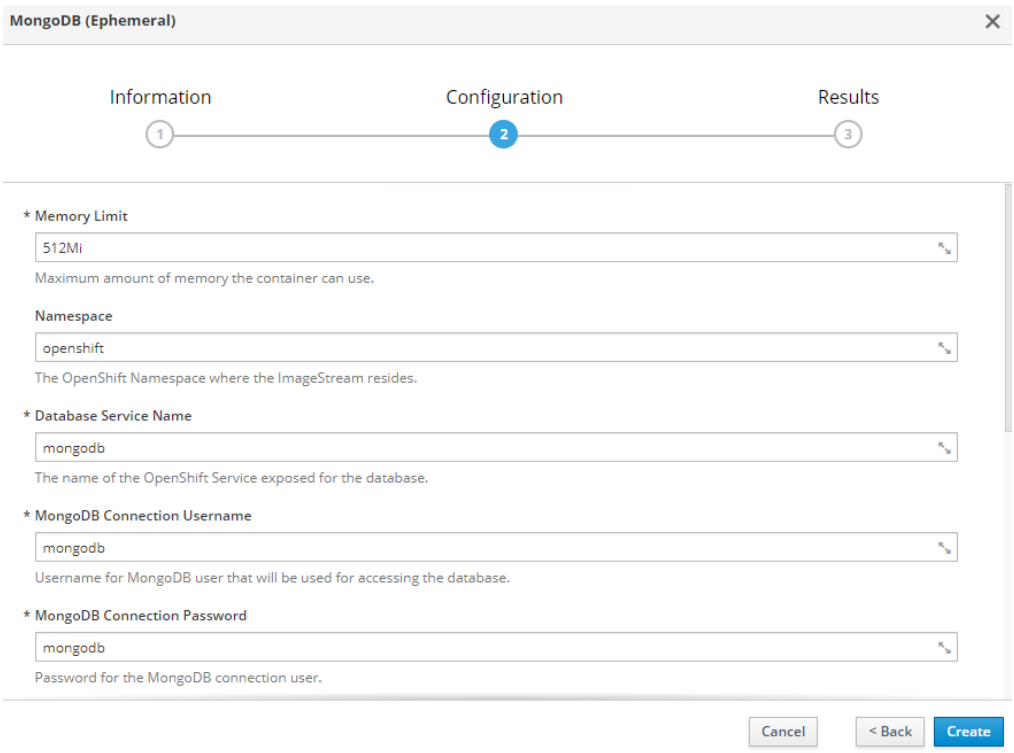
By setting these variables when creating the Mongo database, the image will ensure that:

- 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 nationalparks project, click the "Add to Project" button, and then browse the catalog to find the MongoDB (Ephemeral) template, and click it.



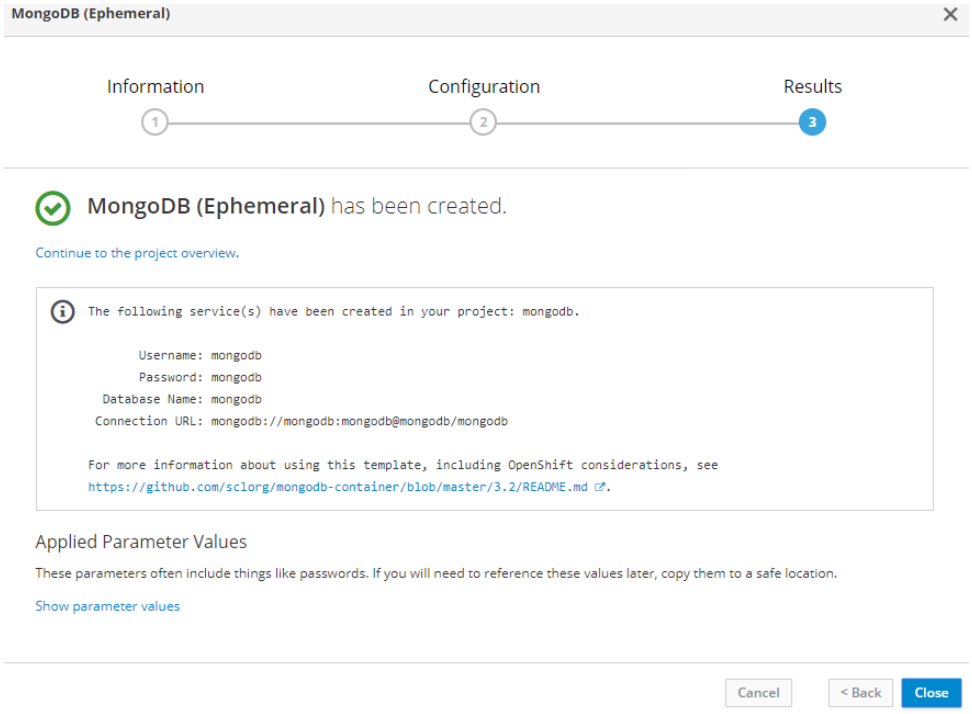
Your view on the next page is slightly different than before. Since this template requires several environment variables, they are predominantly displayed:



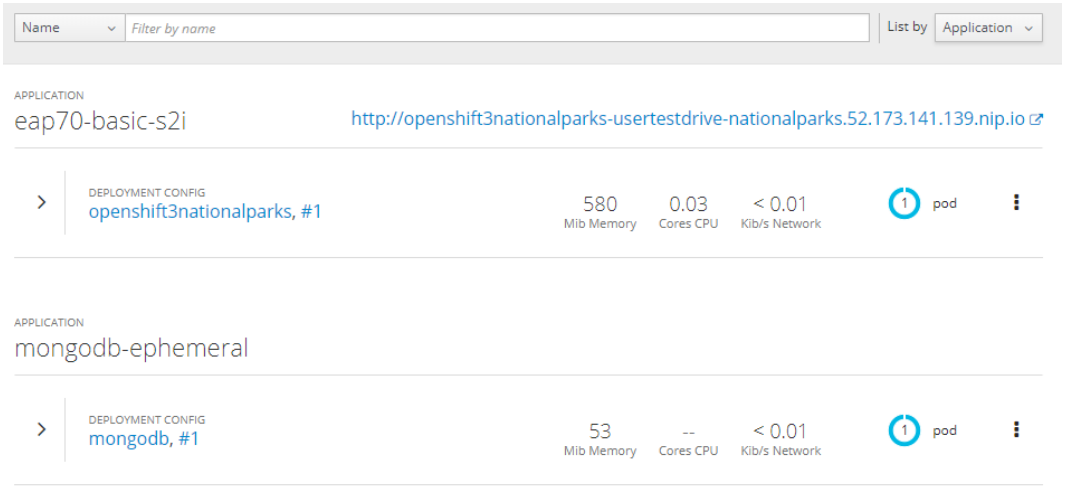
You can see that some of the fields say "generated if empty". This is a feature of Templates in OpenShift that will be covered in the next lab. For now, let's use the following values:

- **MongoDB Connection Username**: mongodb
- **MongoDB Connection Password**: mongodb
- **MongoDB Database Name**: mongodb
- **MongoDB Admin Password**: mongodb

You can leave the rest of the values as their defaults, and then click "Create".



Then click Close and select the Overview tab from the menu on the left. The MongoDB instance should quickly be deployed.



Wiring the JBoss EAP 7.0 pod(s) to communicate with our MongoDB database

When we initially created our JBoss application, we provided no environment variables. The application is looking for a database, but can't find one, and it fails gracefully (you don't see an error).

In order for our JBoss Pod(s) to be able to connect to and use the MongoDB Pod that we just created, we need to wire them together by providing values for the environment variables to the EAP Pod(s). In order to do this, we simply need to modify the DeploymentConfiguration.

First, find the name of the DC:

```
$ oc get dc
```

Then, use the `oc env` command to set environment variables directly on the DC:

```
$ oc env dc openshift3nationalparks -e MONGODB_USER=mongodb -e
MONGODB_PASSWORD=mongodb -e MONGODB_DATABASE=mongodb
```

After you have modified the DeploymentConfig object, you can verify the environment variables have been added by viewing the JSON document of the configuration:

```
$ oc get dc openshift3nationalparks -o json
```

You should see the following section:

```
"env": [
   {
      "name": "MONGODB_USER",
      "value": "mongodb"
   },
   {
      "name": "MONGODB_PASSWORD",
      "value": "mongodb"
   },
   {
      "name": "MONGODB_DATABASE",
      "value": "mongodb"
   }
],
```

OpenShift Magic

As soon as we set the environment variables on 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`

```
NAME                      REVISION   DESIRED   CURRENT   TRIGGERED BY
mongodb                   1          1         1         config,image(mongodb:3.2)
openshift3nationalparks   2          1         1         config,image(openshift3nationalparks:latest)
```

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`

```
NAME                          DESIRED   CURRENT   READY     AGE
mongodb-1                     1         1         1         5m
openshift3nationalparks-1     0         0         0         19m
openshift3nationalparks-2     1         1         1         2m
```

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.

If you refresh your application, you'll notice that the parks suddenly are showing up. That's really cool!

You are probably wondering how this magically started working? When deploying applications to OpenShift, it is always best to use environment variables 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 in DBConnection.java

In short summary: By referring to environment variables 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.

Using the Mongo command line shell in the container

To interact with our database, we will use the oc exec command, which allows us to run arbitrary commands in our Pods. If you are familiar with Docker exec, the oc command essentially is proxying Docker exec through the OpenShift API -- very slick! In this example we are going to use the bash shell that already exists in the MongoDB Docker image, and then invoke the mongo command while passing in the credentials needed to authenticate to the database. First, find the name of your MongoDB Pod:

```
$ oc get pods
```

```
NAME                              READY     STATUS      RESTARTS   AGE
mongodb-1-rkgqm                   1/1       Running     0          6m
openshift3nationalparks-1-build   0/1       Completed   0          21m
openshift3nationalparks-2-ptf5n   1/1       Running     0          2m
```

```
$ oc exec -ti mongodb-1-rkgqm -- bash -c "mongo -u mongodb -p mongodb mongodb"
```

Note: *If you used different credentials when you created your MongoDB Pod, ensure that you substitute them for the values above.*

Note: *You will need to substitute the correct name for your MongoDB Pod.*

Once you are connected to the database, run the following command to count the number of MLB teams added to the database:

```
> db.parks.count();
```

You can also view the json documents with the following command:

```
> db.parks.find();
```

OpenShift's Web Console Terminal

If you go back to the web console in your nationalparks Project and then mouse-over "Browse" and then select Pods, you'll be taken to the list of your pods. Click the MongoDB pod, and then click the tab labeled Terminal.

OpenShift's web console gives you the ability to execute shell commands inside any of the Pods in your Project.

In the terminal for your Mongo Pod, run the same mongo command from before:

```
sh-4.2$ mongo -u mongodb -p mongodb mongodb
```

Then you can issue the same db.parks.count(); command from before, without having to use the CLI! This is seriously cool.

Note: *Don't forget to use the right user and password and database information.*

Note: *You currently can't copy/paste into the terminal.*

END OF LAB

Lab 9: Using 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. 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 three labs into a single Template that we can then instantiate with a single command. I bet you are probably hating us now for having you go through all of that work when you could have issued a single command! Just remember that it is important for you to understand how to create, deploy, and wire resources together. In order for the magic to happen, first create a new project and add the template to the project:

```
$ oc new-project nationalparks-template-<initials>
```

```
$ oc create -f https://raw.githubusercontent.com/anagna/openshift3nationalparks
/master/nationalparks-template-eap.json
```

Important: Please note that when you copy-paste the above command, you should remove the space between **openshift3nationalparks** and **/master …**

Now we have access to the application template in our project. As a side note, administrators have the capability to add templates to the general openshift project which will in turn provide an application template to any user on the system.

Are you ready for the magic command? Here it is:

```
$ oc new-app --template=nationalparks-eap  --name=nationalparks
```

You will see the following output:

```
--> Deploying template "nationalparks-template/nationalparks-eap" to project nationalparks-
template

     nationalparks-eap
     ---------
     Application template for MLB Parks application on EAP 6 & MongoDB built using STI

     * With parameters:
        * APPLICATION_NAME=nationalparks
        * APPLICATION_HOSTNAME=
        * GIT_URI=https://github.com/anagna/openshift3nationalparks.git
```

```
        * GIT_REF=master
        * Maven mirror url=
        * MONGODB_DATABASE=root
        * MONGODB_NOPREALLOC=
        * MONGODB_SMALLFILES=
        * MONGODB_QUIET=
        * MONGODB_USER=userphI # generated
        * MONGODB_PASSWORD=kDvdYl2s # generated
        * MONGODB_ADMIN_PASSWORD=LmAseVts # generated
        * GITHUB_TRIGGER_SECRET=xiLWTeKy # generated
        * GENERIC_TRIGGER_SECRET=iGBfC0Cy # generated

--> Creating resources ...
    buildconfig "nationalparks" created
    imagestream "nationalparks" created
    deploymentconfig "nationalparks-mongodb" created
    deploymentconfig "nationalparks" created
    route "nationalparks" created
    service "mongodb" created
    service "nationalparks" created
    service "ping" created
--> Success
    Build scheduled, use 'oc logs -f bc/nationalparks' to track its progress.
    Access your application via route 'nationalparks-nationalparks-
template.52.173.141.139.nip.io'
    Run 'oc status' to view your app.
```

OpenShift will automatically start a build for you. When it is complete, visit your app. 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, micro services, fix bugs, and more.

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

https://raw.githubusercontent.com/anagna/openshift3nationalparks/master/nationalparks-template-eap.json

END OF LAB

Thank You for following the lab.

*****