

Red Hat OpenShift

Openshift 3 Introduction *Workshop for developers*

Mark Roberts / Ben Holmes / Anthony Kesterton

Version 1.0.0, October 09, 2019

Table of Contents

Introduction	1
Attendee details	1
What is Openshift	1
Links	1
Web console	1
CLI (oc)	2
Option 1: Installing the CLI on localhost	2
Option 2: Use a pre-configured docker image on OpenShift	3
Using your terminal (Both options)	5
Web Console overview	7
Sandbox project	7
Build options	8
The catalog	9
The oc login token	10
Simple application lifecycle	11
Quick Introduction to the Build process	11
Deploying an application using S2I	11
The running application	13
Application Services	15
Application Pods	15
Application Scaling	16
Application Route	18
Application from CLI	19
Health Checks	21
Application Deployment Strategies	22
Storage	24
Config Maps	28
Secrets	30
Clean up	32
Software Defined Networking	33
User Role based access control	38
CI/CD Pipeline	40
Wrap-up	43

Introduction

Attendee details

Name:	
User ID (userX):	

This workshop is designed to give developers an introduction to Openshift.

This workshop aims to introduce the basics of devex within OCP without diving too deep into the development languages - by design this intend to focus on the advantages and efficiencies of using Openshift as a complete dev platform as opposed to in-depth technical examples of languages

What is Openshift

Red Hat® OpenShift® is a hybrid cloud, enterprise Kubernetes application platform.

OpenShift is a family of containerization software developed by Red Hat. Its flagship product is the OpenShift Container Platform—an on-premises platform as a service built around Docker containers orchestrated and managed by Kubernetes on a foundation of Red Hat Enterprise Linux. The family's other products provide this platform through different environments: OKD serves as the community-driven upstream (akin to CentOS), OpenShift Online is the platform offered as software as a service, and Openshift Dedicated is the platform offered as a managed service

Links

- <https://www.openshift.com/learn/what-is-openshift>
- <https://en.wikipedia.org/wiki/OpenShift>
- <https://www.openshift.com/products/container-platform> == Setup

Web console

You will be assigned an ID by the facilitator. At any time that the content refer to *userX*, (or *anythingX*) you need to replace X with your ID. Your userId to log onto the web console will be *userX* and the password will be *openshift*

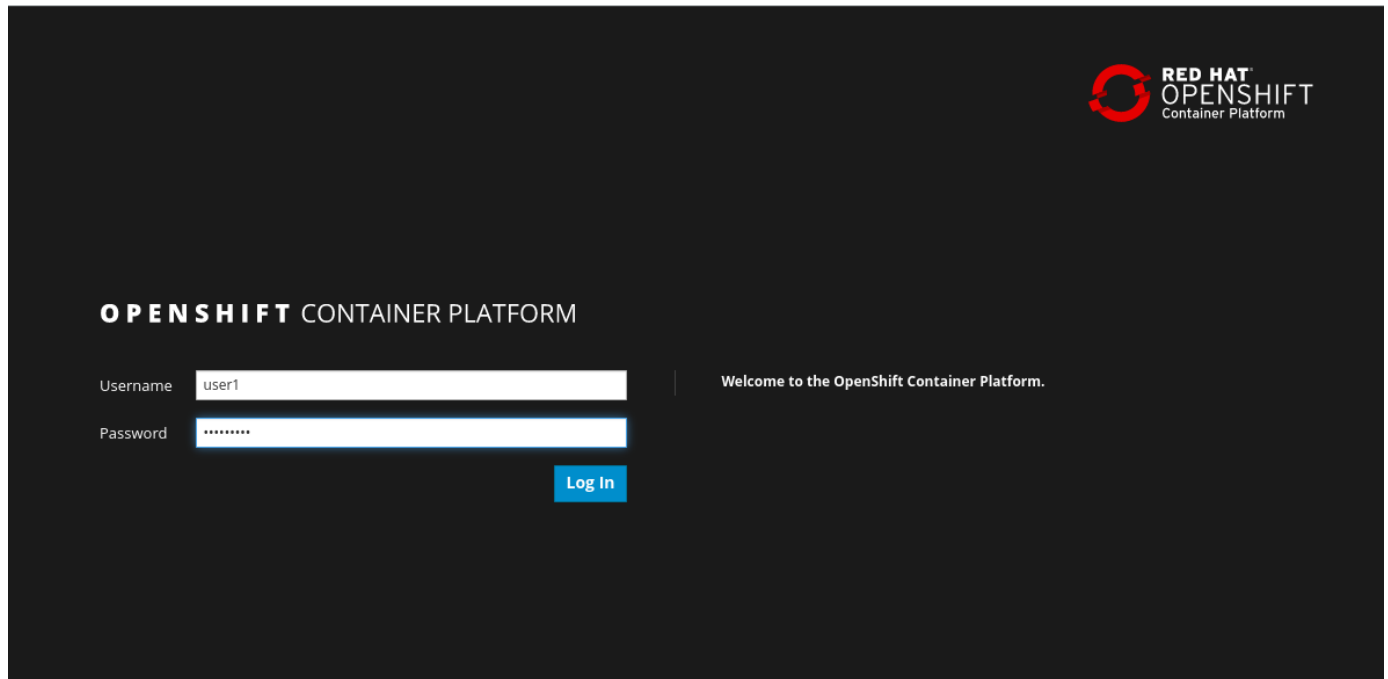
You can access the web console using your browser:

<https://master.meetup1-d243.open.redhat.com/console>

Example: If your ID is 1:

- **Username:** user1
- **Password:** openshift

Click [**Log In**]



CLI (oc)

With the OpenShift Enterprise command line interface (CLI), you can create applications and manage OpenShift projects from a terminal. The CLI is ideal in situations where you are:

- Working directly with project source code.
- Scripting OpenShift Enterprise operations.
- Restricted by bandwidth resources and cannot use the web console.

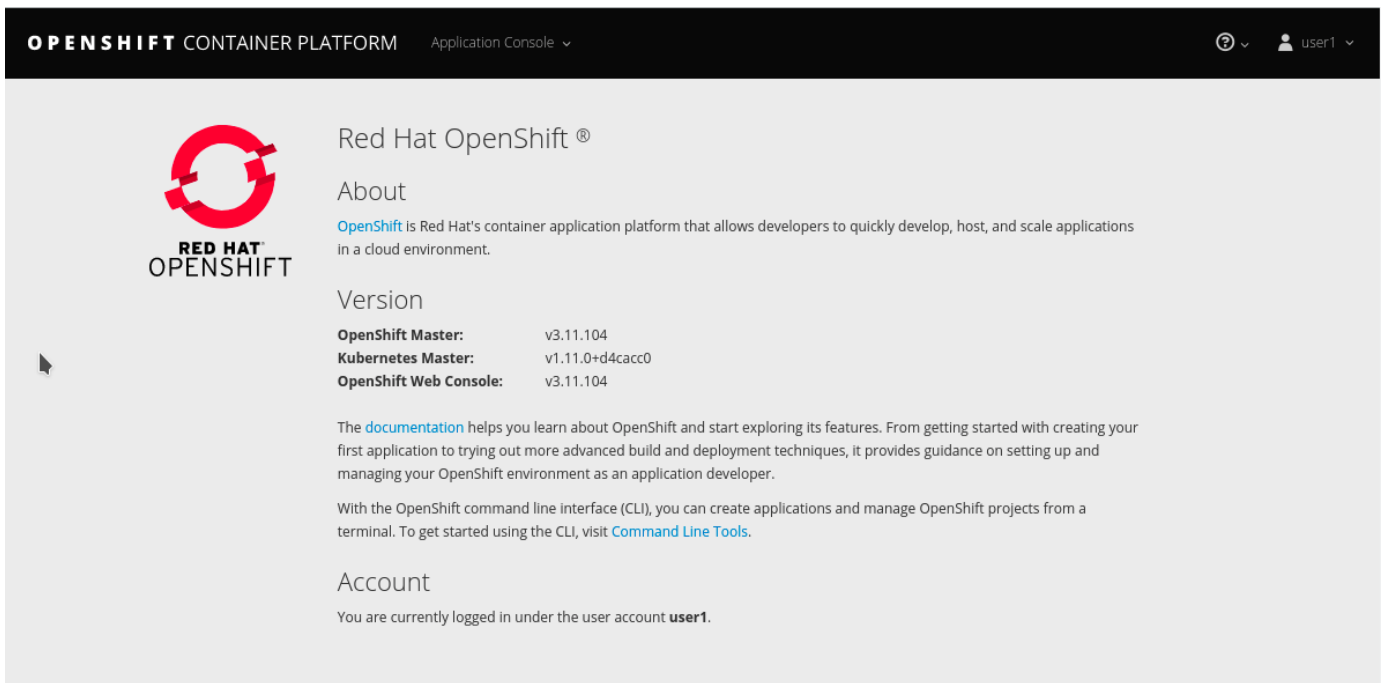
The CLI is available using the oc command:

```
oc {command}
```

You have two options to get a working CLI. Either install **oc** on your localhost or use a docker image in OpenShift:

Option 1: Installing the CLI on localhost

The easiest way to download the CLI is by accessing the About page on the web console [(?) → About]



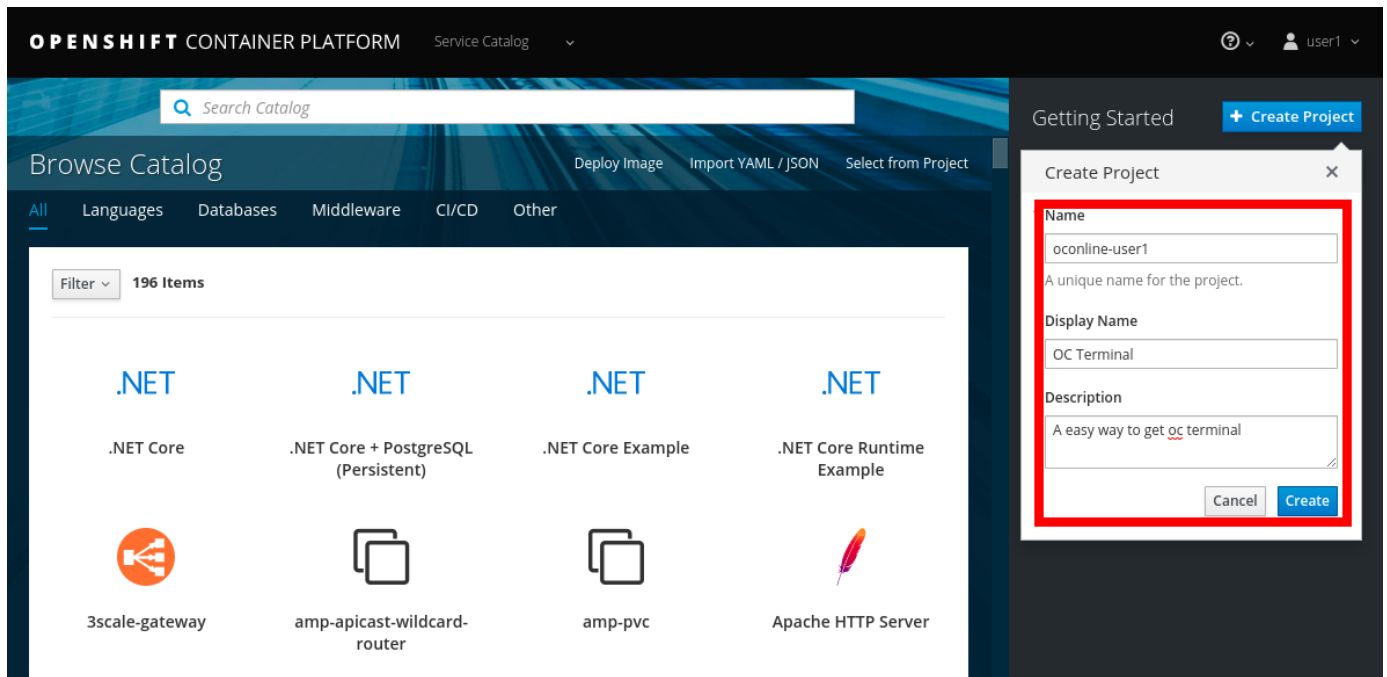
or you can follow the instructions from the [official documentation](#)

Option 2: Use a pre-configured docker image on OpenShift

This option install and run a docker image inside OpenShift that already has oc installed and configured. To install this image, do the following:

- Login to the OCP system through the UI. (userX / openshift)
- Create a new project:
 - **Name:** oonline-userX
 - **Display Name:** OC Terminal

Click [**Create**]

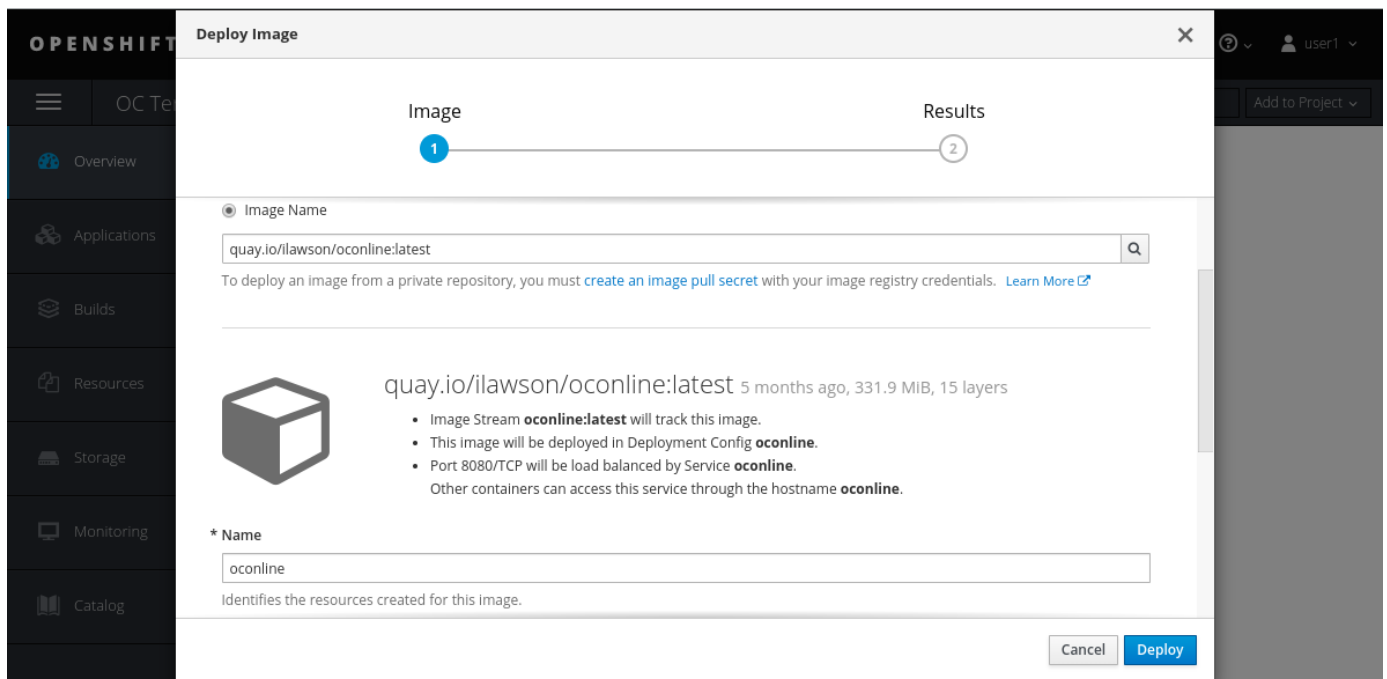


Select the newly created **OC Terminal** project and then select **Deploy Image**

Use the Image Name option and add the following name:

`quay.io/ilawson/oconline:latest`

Click [**Search**] and then leave the default values in the metadata form and click [**Deploy**]



Once the image has deployed and a Pod will appear

- Click on the Pod ring (blue)

- Select **Terminal** from the options (This will be *your terminal* you can use to do **oc** commands)

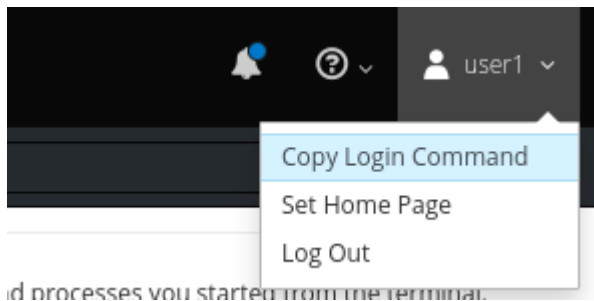


It might be easier to keep this project with the terminal view open in a separate browser tab, so that you can switch back and forth to it easy

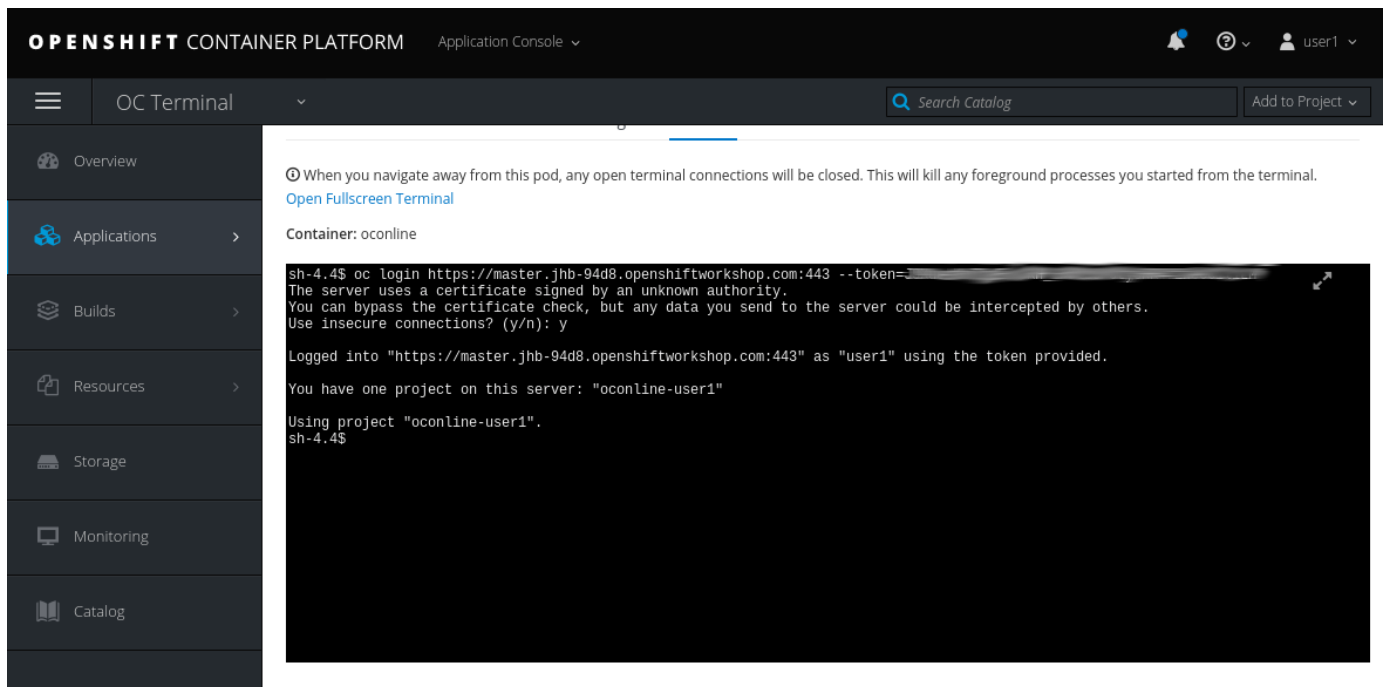
Using your terminal (Both options)

If this document refer to *your terminal* it will either be on your localhost or the docker install terminal depending your option above.

- In the Web Console, select the top right pulldown, choose [**Copy Login Command**]



- Paste that command into *your terminal*, hit return - hit 'y' for insecure access



Make sure **oc** is working, type:

```
oc whoami
oc version
```

```
sh-4.4$ oc whoami
user1
sh-4.4$ oc version
oc v3.11.0+0cbc58b
kubernetes v1.11.0+d4cacc0
features: Basic-Auth GSSAPI Kerberos SPNEGO

Server https://master.jhb-94d8.openshiftworkshop.com:443
openshift v3.11.104
kubernetes v1.11.0+d4cacc0
sh-4.4$
```



Also see the **Command-Line Walkthrough**: https://docs.openshift.com/container-platform/3.11/getting_started/developers_cli.html

Web Console overview

Sandbox project

So you have already created your first project using the web console (**oonline-userX**).

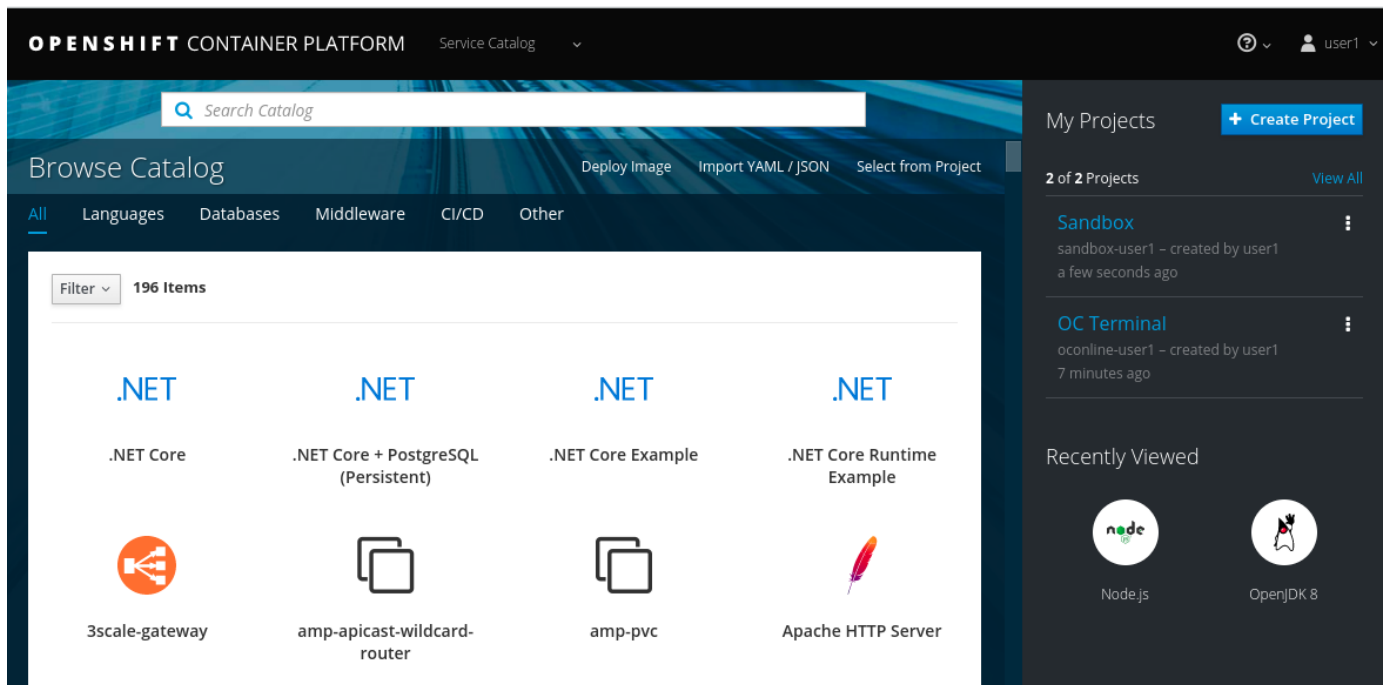
Let's create another project where we can play in:

Create a new project:

- **Name:** sandbox-userX
- **Display Name:** Sandbox

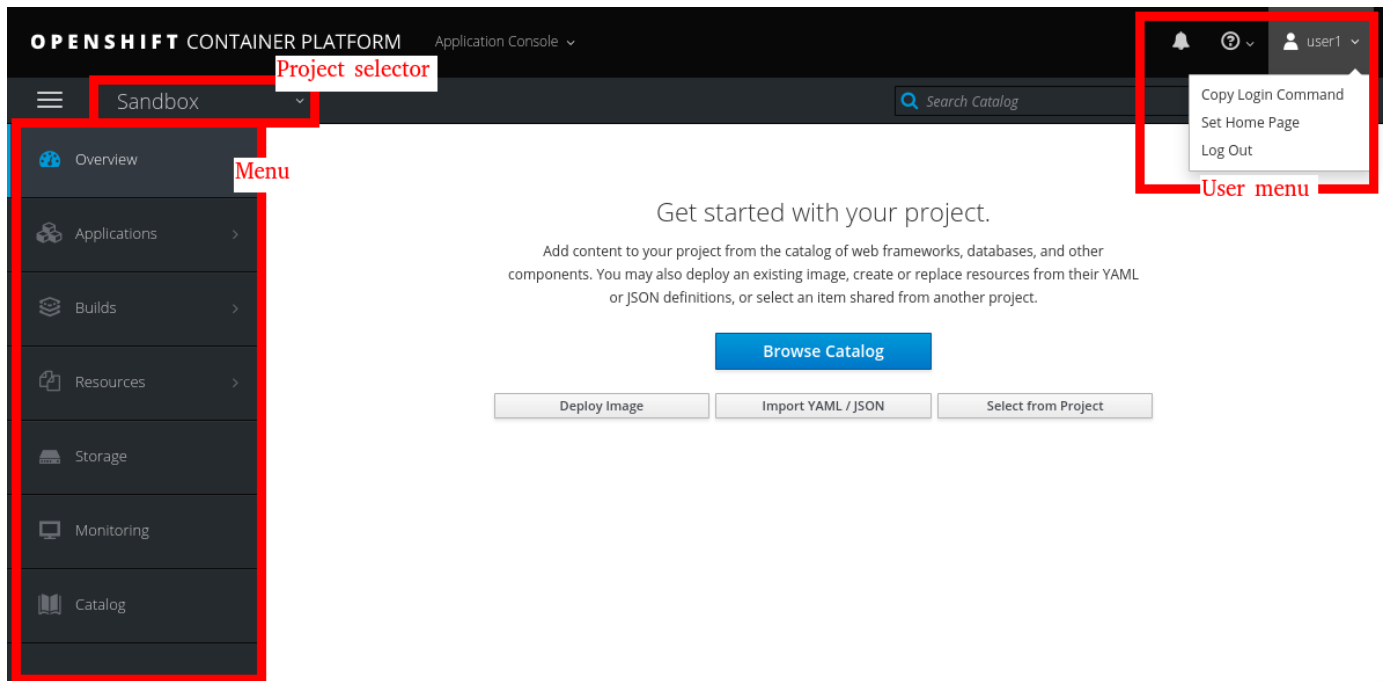
Click [**Create**]

You should now have 2 projects like this:



Click the **Sandbox** project to go to the *Application Console* for that project, you will see:

- **Project selector** (top left): Here you can switch between your projects
- **Menu** (left): This is all the sections available.
- **Context area** (center): This will display content based on the menu item selected.
- **User menu** (top right): Here you can get the cli login command.



Build options

A build is the process of transforming input parameters into a resulting object. Most often, the process is used to transform input parameters or source code into a runnable image. A BuildConfig object is the definition of the entire build process. OpenShift Container Platform leverages Kubernetes by creating Docker-formatted containers from build images and pushing them to a container image registry. Build objects share common characteristics: inputs for a build, the need to complete a build process, logging the build process, publishing resources from successful builds, and publishing the final status of the build. Builds take advantage of resource restrictions, specifying limitations on resources such as CPU usage, memory usage, and build or pod execution time.

The OpenShift Container Platform build system provides extensible support for build strategies that are based on selectable types specified in the build API. There are three primary build strategies available:

- Docker build
- Source-to-Image (S2I) build
- Custom build

By default, Docker builds and S2I builds are supported. The resulting object of a build depends on the builder used to create it. For Docker and S2I builds, the resulting objects are runnable images. For Custom builds, the resulting objects are whatever the builder image author has specified.

Additionally, the Pipeline build strategy can be used to implement sophisticated workflows:

- continuous integration
- continuous deployment



Also see the **Builds and Image Streams**: https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/builds_and_image_streams.html#builds

We have already done a docker build with **online**.

The catalog

Click [**Browse Catalog**].



OpenShift Container Platform provides out of the box a set of languages and databases for developers with corresponding implementations and tutorials that allow you to kickstart your application development. Language support centers around the Quickstart templates, which in turn leverage builder images.

Openshift is effectively **Kubernetes**, and as such it works with Objects that are part of the Kubernetes/Openshift Object model. When interacting with this model through the interfaces, be it CLI via *oc* or the web interface, a User is creating, deleting and changing Objects in this model.

The **Catalog** is designed as a quickstart way to interact with the Openshift instance by pre-creating the objects you need for a specific Application. It does this by providing Templates, which are pre-created sets of objects with parameterised components that have to be provided to create the application - a good example of this is the node.js Template, which requires only three initial parameters (with a lot of advanced and optional parameters for deeper configuration) to create a full Application within Openshift.

These parameters, in the case of the node.js Template, are the base image to build upon, the name to assign to all the Objects created as part of the Template and the git repo containing the node.js source to build into the Application image. When you choose this Template from the catalog you are provided with a Wizard that prompts for those three essential parameters (plus provides advanced inputs for the additional ones).

There is another type of Template supported by Openshift called a *Quickstart*. This is identical in theory to the Template but instead has all the required Parameters pre-defined so a User can simply create an instance of the Quickstart without having to provide any information other than the name of the Application to create.

In Openshift 3 the Catalog also supports **Services**. This is an implementation of the Service Broker concept and allows external and pre-created Services, where Service in this case refers to an external Service providing functionality as opposed to the services internally which are Application endpoints within Openshift. Services provided via the Service Broker are external interfaces with a provided Service definition that defines how to call them and what they contain. A User can add a Service to his namespace/project, and when he does that Service is callable via the defined interface from his Applications in the namespace/project. This functionality is being wound down as part of Openshift 4 with the functionality being instead provided by **Operators**.

Operators are a fantastic new concept that simplifies the administration and on-going maintenance of Kubernetes applications. How they work is very elegant - an Operator maintains the state of a set of Objects within the Openshift/Kubernetes object model **but** can also extend it. An Operator is itself a running Pod, and this Pod handles installation of Objects, updates of Objects and monitoring and response to change events in the Object state and behaviour.

An easier explanation is this - imagine you are writing an Application that needs to be deployed in Openshift. This Application is complex, consisting of multiple Pods, configuration options, and you want to be able to update it in real time as fast as you can. By producing an Operator, which installs the Object model and then can act on changes and events concerning it (which it does by extending the Openshift/Kubernetes Object model with Custom Resource Definitions, which can then be triggered to be updated by simply providing updated YAML definitions of the type defined), the Application can be deployed simply by passing the appropriate YAML for the Custom Resource, which is specific and contextual *only* to the Application, to the Operator.

Red Hat has been instrumental in creating the operatorhub.io site, where people can publish Operators than can be easily consumed by Openshift 4. In fact the functionality for Operators has been back ported into Openshift 3.11 because it is so powerful and essential. As of Openshift 4 the Catalog has direct linkage to the operatorhub and allows Users to request Operators to consume.

Interestingly these are handled in a much more secure way than with vanilla Kubernetes. The Operator, when installed, needs to alter the core Object model of the Cluster, which is very dangerous. Openshift prohibits the installation of Operators (you need Cluster Admin anyway) by implementing an Operator itself, called the Operator Lifecycle Manager.

This Operator (shortened to **OLM**) allows Admins to install Operators as *Subscriptions* which can then be used by named Users, Projects or globally depending on the security required. The OLM handles the update of the Operators, and the Users can consume them simply by creating Objects of the appropriate Custom Resource. This way Users do not need to install the Operators themselves - installation and Object model update is controlled centrally by the OLM and Administrators.

The oc login token



It is good practise to use the *Copy Login Command* from the UI to generate the login statement for accessing the system via oc. This provided token is expired when the UI login expires or the User logs out.

Openshift provides a rich and full RESTful API for interacting, based on the security constraints of the user, with the Openshift object model. The UI will generate a valid OAUTH token based on a successful login which can and should be used to attach the oc client to a valid session.



Also see the **Web Console Walkthrough**: https://docs.openshift.com/container-platform/3.11/getting_started/developers_console.html

Simple application lifecycle

Quick Introduction to the Build process

One of the most exciting features of OpenShift from a developer's perspective is the concept of the S2I, or **Source-2-Image** which provides a standardised way of taking a base image, containing, for example, the framework for running a node.js application, a source code repository, containing the code of the application that matches the framework provided in the base image, and constructing and delivering a composite Application image to the Registry.

The standardised nature of the S2I approach means that, in the case where Red Hat do not provide out-of-the-box facilities for a framework, it is simple to create the S2I scripts and use them in exactly the same fashion as the out-of-the-box scripts.

The S2I works in effectively three phases - a compilation phase, which is OpenShift agnostic and simply, in the case of requiring it, compiles the source material to the binary components, or interim bytecode formats, required for the framework. In the case of Java, for example, this would entail executing a Maven build on the source provided in the repository. The second phase is to construct the composite image for the Application by injecting the binary components into the appropriate filesystem in the base image. The third phase is to setup the execution context for the Application, i.e. how it is started, for example in the case of a Java web-app running on a Tomcat base this would be executing the bin/startup.sh script.

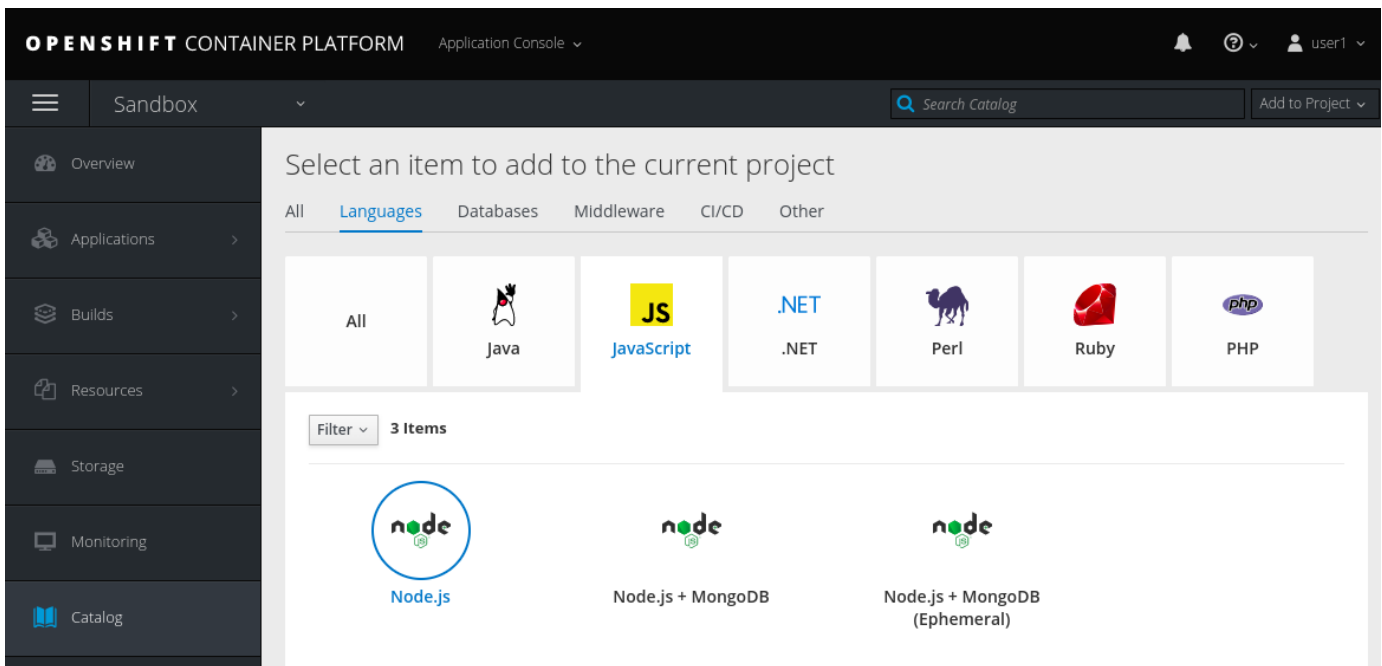
A nice feature of this approach is the capability of the S2I scripts to also be executed in what is called **Binary** mode. This allows the script to ignore the compilation phase and be provided with the binary components required to build the Application. This is a usecase for developers who wish to either bring pre-compiled components for their Apps or in the situation where an organisation has an existing build farm for applications where the binary artefacts are tested and config controlled outside of OpenShift. This mode is called *binary build*.

In the following example we will take advantage of the node.js builder S2I script in OpenShift.

Deploying an application using S2I

In the Catalog of your sandbox project:

- Filter (top tab bar) **Languages**
- Select **JavaScript**
- Choose **Node.js**



You will now go through a wizard to gather all data needed for this S2I build config:

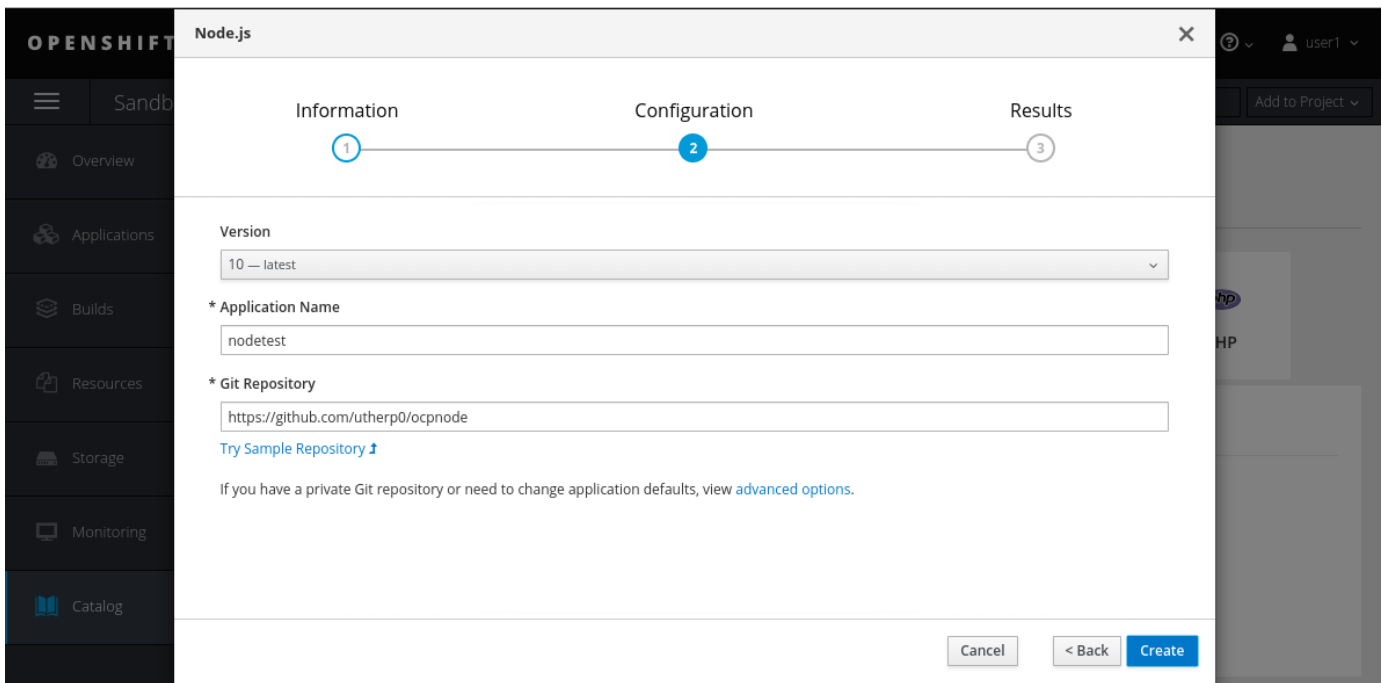
The UI provides Wizards based on the templates installed (which are the selections chosen from in the Catalog). These Wizards explain the nature of the chosen template, and then prompt the User for specific information which is required to perform the full lifecycle build and deployment of the target application.

- Click **[Next]** on first page of Wizard.

Each template has a set of parameters that are either optional or mandatory. The Wizard provides a way for the User to enter the information - in the case of the node.js there are only three pieces of mandatory information required to create, build and deploy the node.js application. The Wizards also provide additional *advanced* options which are the optional parameters for specialising the build, for example if you are using a code repository that has secure access you need to provide what is known as a **Pull Secret** which will allow OpenShift, which is running the Build as a standalone Pod, to clone the source material from your repository. This is one of the optional parameters.

- Select node image version **10**
- Enter name as **nodetest**
- Enter the following url as the github repo <https://github.com/utherp0/ocpnode>

Click **[Create]**, then **[Close]** to close the wizard



Now go back to the **Overview** page.

What has happened is that the OpenShift system has processed the template, in this case one for a node.js Application based on an S2I build, and produced **all** of the Objects needed within the User's project to realise this. If you are quick enough to watch the Overview you'll see first a Build appear. This is a Build Pod that has been spun off within the Project using the configuration generated from the Templates and implemented in a **BuildConfig** Object. The small window in the bottom right corner shows the log of the build as it happens - the source will be cloned into the Build Pod, the code compiled appropriately (using npm in this case) and then the composite Application Image is constructed (you will see layers being written).

When the Image is completed it is delivered from the BuildPod into the Integrated Registry within OpenShift with the appropriate labels and tags that identify it as the nodetest Application in this Project. As part of the template realisation a DeploymentConfig would have been created as well - this is sitting watching the Registry for the creation or update of the named Image, and when the build is complete the addition of that Image to the Registry will automatically start a deployment.

The Template defines this deployment to be a single Pod of the Application with a service (hooking into the 8080 port of the Pod), and a **Route** for external connectivity. All of these are created automatically and once the deployment starts you can watch the state of the deployment by looking at the colour of the ring around the Pod - when gray it is **pulling** the Image from the Integrated Registry and delivering it to the Node that has been targetted (internally within OpenShift) to host this instance. When the ring turns to light blue the Pod is running successfully and is ready to service requests.

The running application

When in the Overview page, you will see all running applications. Expand the **nodetest** application we just deployed. You will see an overview of the running application:

- Information on the running container
- Number of pods and the status (a.k.a **Pod ring**) of the pods
- Networking information including internal port mapping and external routes
- Build history and information

The screenshot shows the OpenShift Container Platform Application Console. The left sidebar contains navigation links: Overview, Applications, Builds, Resources, Storage, Monitoring, and Catalog. The main panel displays the 'nodetest' application details. At the top, it shows the application name 'nodetest' and its URL: <http://nodetest-sandbox-user1.apps.jhb-94d8.openshiftworkshop.com>. Below this, the 'DEPLOYMENT CONFIG' section shows 'nodetest, #1'. The 'CONTAINERS' section displays a circular 'Pod ring' with '1 pod'. The 'NETWORKING' section shows internal traffic (Service - Internal Traffic) and external traffic (Routes - External Traffic). The 'BUILDS' section shows 'nodetest' with a status of 'Build #1 is complete'.

To see the application in action, click on the link in the external route. This will open the basic node.js application:



Test page for nodejs-ex
Served from res.render (listening on /).

Reveal

JSON TEST

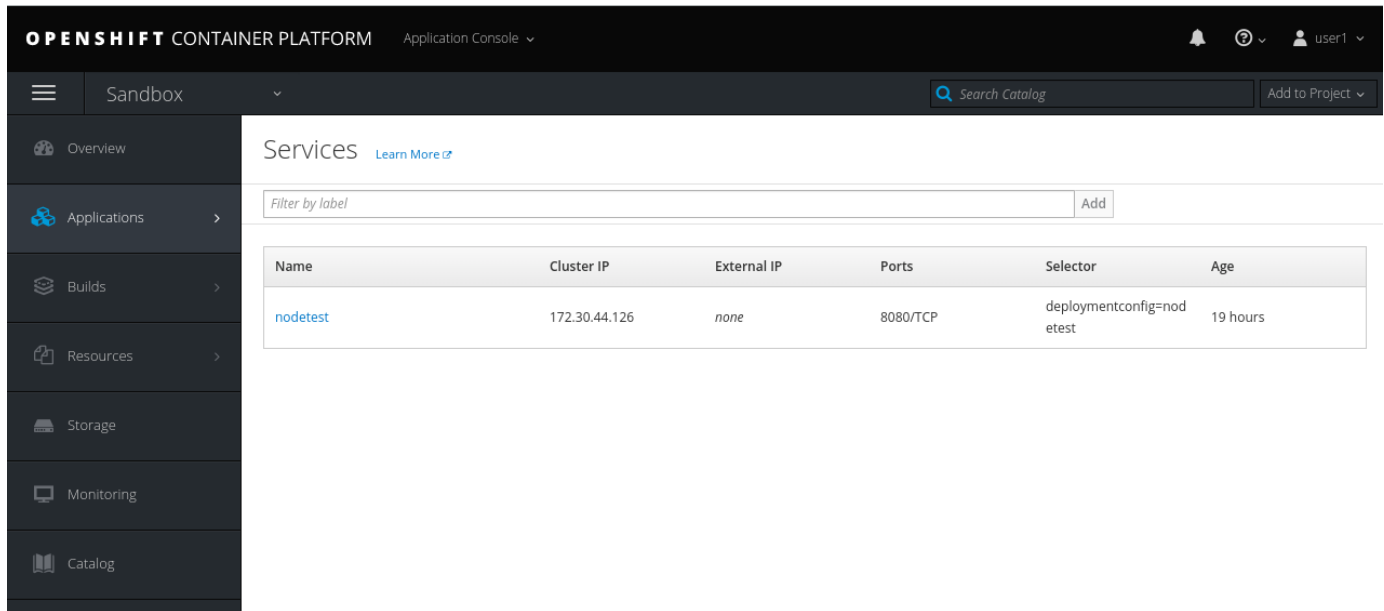


On July 16 the Moon celebrated the 50th anniversary of the launch of Apollo 11 with a lunar eclipse visible from much of planet Earth. In this view part of the lunar disk is immersed in Earth's dark, reddened umbral shadow. Near the maximum eclipse phase, it just touches down along a mountain ridge. The rugged Tyrolean nightscape was recorded after moonrise south of Innsbruck, Austria with a dramatically lit communication tower along the ridgeline. Of course eclipses rarely travel alone. This partial lunar eclipse was at the Full Moon following July 2nd's New Moon and total eclipse of the Sun.

This is a simple webpage rendered from the node.js application at the root endpoint.

Application Services

Using the menu on the left go to the **Applications** › **Services** page.



The screenshot shows the OpenShift Container Platform Application Console. The top navigation bar includes the OpenShift logo, 'Application Console', a search bar, and a user profile. The left sidebar contains a menu with options: Overview, Applications (selected), Builds, Resources, Storage, Monitoring, and Catalog. The main content area is titled 'Services' and includes a 'Filter by label' input field and an 'Add' button. Below this is a table with the following data:

Name	Cluster IP	External IP	Ports	Selector	Age
nodetest	172.30.44.126	none	8080/TCP	deploymentconfig=nodetest	19 hours

Another thing that makes Kubernetes and OpenShift so powerful is the abstracted nature of the service endpoints. Each running Application has a single service endpoint, which is the load-balancing point for the Application within the **Software-Defined-Network** that OpenShift uses for internal communication. From the outside consumers of the Application use a FQDN (fully qualified domain name) which refers to the **Route**. This is translated at the edge of OpenShift into the singular service cluster IP. This is then a load-balancer across **all** the replicas of the Application. This means that you can change the number of replicas and even the target services of the Route without having to rebuild or redeploy the applications. This disconnect is very powerful in usecases such as Canary Deployments and zero-downtime upgrades/downgrades of Applications.



More info here: https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/pods_and_services.html#services

Go back to the **Overview** page.

Application Pods

Click on the **Pod ring**, or alternatively use the menu **Applications** › **Pods** › **nodetest-***

The screenshot shows the OpenShift Container Platform Application Console. The top navigation bar includes the OpenShift logo, 'Application Console', and user information. The left sidebar contains navigation links for Overview, Applications, Builds, Resources, Storage, Monitoring, and Catalog. The main content area displays the details for a pod named 'nodetest-1-2g2dz', created 19 hours ago. The pod is in a 'Running' state. The 'Status' section shows the pod is running on node 'node1.jhb-94d8.internal (192.168.0.17)' with IP '10.1.2.67'. The 'Template' section shows the pod is using the 'nodetest' image and build, with ports 8080/TCP and a mount for 'default-token-5992n'. The 'Containers' section shows the pod is running on the 'nodetest' container. The 'Volumes' section shows the pod is using the 'default-token-5992n' volume.

Status	
Status:	Running
Deployment:	nodetest, #1
IP:	10.1.2.67
Node:	node1.jhb-94d8.internal (192.168.0.17)
Restart Policy:	Always

Container nodetest	
State:	Running since Jul 17, 2019 3:28:38 PM
Ready:	true
Restart Count:	0

Template	
Containers	
nodetest	
Image:	sandbox-user1/nodetest 631a90e 681.5 MiB
Build:	nodetest, #1
Source:	Added hide and reveal of the ENV details f35527c authored by Ian 'Uther' Lawson
Ports:	8080/TCP
Mount:	default-token-5992n → /var/run/secrets/kubernetes.io/serviceaccount read-only
CPU:	50 millicores to 500 millicores
Memory:	256 MiB to 1536 MiB

Volumes	
default-token-5992n	



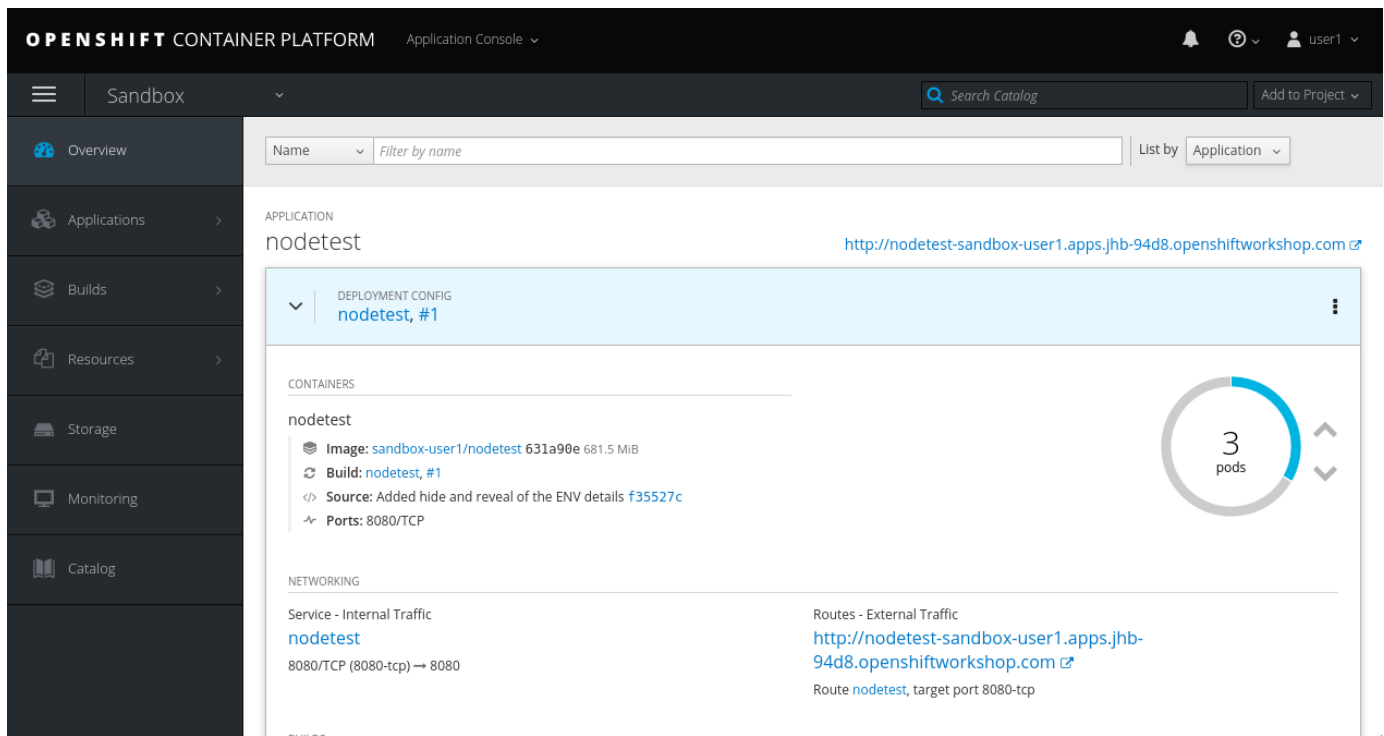
See the differing IP address for the Pod compared to the cluster IP

Go back to the **Overview** page.

Application Scaling

Let's pretend that this app is suddenly getting many requests from many users (so there is a load increase on the app). So we need to scale the application to 3 instances.

Click the **Up arrow** (^) until there are 3 replicas.



A quick note on the colour schemes of the Pod UI - the system uses the colour to indicate the state of the Pod(s) in realtime. It can be one of five different colours - transparent indicates the Pod is being physically created; it has been allocated to a Node, that node is allocating the file space but has yet to copy the Image file-layers to it.

Gray indicates the Image is being loaded from the Registry and pushed to the Node.

Light blue indicates the Pod is operational and reporting healthy.

Dark blue indicates the Pod has received an event to terminate and is gracefully removing itself.

Red indicates the Pod is in what is called a CrashLoopBackoff state. This is due to the way in which Pods report healthy and the way in which OpenShift deals with unhealthy Pods. A Pod has two health probes - **liveness** and **readiness**. Liveness indicates the Pod file-layers have been initialised, any additional file mounts have been completed **but** the Application is not ready (internally) to process requests. **Liveness** indicates the Application within the Pod is ready to service requests. OpenShift maintains x replicas, as defined by the **deploymentconfig** of an Application and the replica count applies to Pods that are reporting **live**. If a Pod fails to report **live** it is sent a closedown event, OpenShift decides where to deploy the Pod and the Pod is rescheduled. A CrashLoopBackoff occurs when the Pod continually reports unhealthy and is continually recreated, a state, because of the nature of the system, which is attempting to self-recover. We'll talk later about ways to debug these kind of *Red ring* Pods.

Click on the **Pod ring**, or alternatively use the menu **Applications > Deployments > nodetest > #1 (latest)**.

Scroll down to where the Pods are listed:

OPENSIFT CONTAINER PLATFORM Application Console

Sandbox

Search Catalog Add to Project

Overview Applications Builds Resources Storage Monitoring Catalog

Template

Container nodetest does not have health checks to ensure your application is running correctly. [Add Health Checks](#)

Containers

nodetest

Image: sandbox-user1/nodetest 631a90e 681.5 MiB

Build: nodetest, #1

Source: Added hide and reveal of the ENV details f35527c authored by Ian 'Uther' Lawson

Ports: 8080/TCP

Volumes

[Add Storage](#) | [Add Config Files](#)

Pods

Name	Status	Containers Ready	Container Restarts	Age
nodetest-1-558hp	Running	1/1	0	8 minutes
nodetest-1-9qt62	Running	1/1	0	8 minutes
nodetest-1-2e2dz	Running	1/1	0	20 hours

master.jhb-94d8.openshiftworkshop.com/console/.../overview



See the difference in age between the initial pod and the 2 recent scaled pods.

Select on of the recent (younger) pods.



Note the IP difference compared to the initial pod.

Application Route

Using the menu on the left go to the **Applications** > **Routes** page.

OPENSIFT CONTAINER PLATFORM Application Console

Sandbox

Search Catalog Add to Project

Overview Applications Builds Resources Storage Monitoring Catalog

Routes [Learn More](#) [Create Route](#)

Filter by label Add

Name	Hostname	Service	Target Port	TLS Termination
nodetest	http://nodetest-sandbox-user1.apps.jhb-94d8.openshiftworkshop.com	nodetest	8080-tcp	



Note the mapping of the fully qualified domain name to the cluster IP via the service name

Select the nodetest link in the service column.

The screenshot shows the OpenShift Container Platform Application Console. The left sidebar contains navigation links: Overview, Applications, Builds, Resources, Storage, Monitoring, and Catalog. The main content area displays the details for the 'nodetest' service, created 20 hours ago. It includes tabs for 'app' and 'nodetest', and sub-tabs for 'Details' and 'Events'. The 'Details' tab shows the following information:

- Selectors:** deploymentconfig=nodetest
- Type:** ClusterIP
- IP:** 172.30.44.126
- Hostname:** nodetest.sandbox-user1.svc.cluster.local
- Session affinity:** None

Below this information is a 'Traffic' section with a table showing the route configuration:

Route	Service Port	Target Port	Hostname	TLS Termination
nodetest	8080/TCP (8080-tcp)	8080	http://nodetest-sandbox-user1.apps.jhb-94d8.openshiftworkshop.com	

Below the traffic table is a 'Pods' section with a table showing the pod status:

Pod	Status	Containers Ready	Container Restarts	Age	Receiving Traffic
-----	--------	------------------	--------------------	-----	-------------------



Note that the route maps to the cluster IP

Application from CLI

Now let's go to the console (either using `localhost` or `online` as explained in the [CLI \(oc\)](#) section)

Make sure you are still logged in:

```
oc whoami
```

(if not, log in again as explained in the [Using your terminal \(Both options\)](#) section)

Make sure we are using our sandbox project:

```
oc project sandbox-userX
```

This will print:

```
Now using project "sandbox-userX" on server "https://master.meetup1-  
d243.open.redhat.com/console:443".
```

You can find all **objects** that you can interact with in this namespace/project:

```
oc get all
```

Get all **pods**:

```
oc get pods -o wide
```

This will output something similar to this:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
NOMINATED NODE						
nodetest-1-2g2dz	1/1	Running	0	23h	10.1.2.67	node1.jhb-94d8.internal
nodetest-1-54fw7	1/1	Running	0	3h	10.1.2.74	node1.jhb-94d8.internal
nodetest-1-6xw6g	1/1	Running	0	3h	10.1.2.75	node1.jhb-94d8.internal
nodetest-1-build	0/1	Completed	0	23h	10.1.2.65	node1.jhb-94d8.internal



Note the pod used to build the project is there, just inactive.
Also note the differing IPs for the individual Pods and the NODE information.

In the Web Console, make sure you are on the **[Overview]** page, then do the following in CLI while watching the page:

```
oc delete pod nodetest-****
```

(Replace ** with once of the running pods)

DEPLOYMENT CONFIG
nodetest, #1

CONTAINERS
nodetest
Image: [sandbox-user1/nodetest](#) 631a90e 681.5 MiB
Build: [nodetest, #1](#)
Source: Added hide and reveal of the ENV details [f35527c](#)
Ports: 8080/TCP

NETWORKING
Service - Internal Traffic
nodetest
8080/TCP (8080-tcp) → 8080
Routes - External Traffic
<http://nodetest-sandbox-user1.apps.jhb-94d8.openshiftworkshop.com>
Route **nodetest**, target port 8080-tcp

```

sh-4.4$ oc get pods -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP            NODE                NOMINATED NODE
nodetest-1-2g2dz     1/1     Running   0           23h   10.1.2.67     node1.jhb-94d8.internal   <none>
nodetest-1-54fw7     1/1     Running   0           3h    10.1.2.74     node1.jhb-94d8.internal   <none>
nodetest-1-6xw6g     1/1     Running   0           3h    10.1.2.75     node1.jhb-94d8.internal   <none>
nodetest-1-build     0/1     Completed 0           23h   10.1.2.65     node1.jhb-94d8.internal   <none>
sh-4.4$ oc delete pod nodetest-1-2g2dz
pod "nodetest-1-2g2dz" deleted

```

Health Checks

In the Web Console, go to **Applications > Deployments > nodetest > Configuration**.

Under Template, click **Add Health Checks**:

OPENSIFT CONTAINER PLATFORM

Application Console

Sandbox

Search Catalog

Add to Project

Overview

Applications

Builds

Resources

Storage

Monitoring

Catalog

Deployments > nodetest > Edit Health Checks

Health Checks: nodetest

Container health is periodically checked using readiness and liveness probes.
[Learn More](#)

Readiness Probe

A readiness probe checks if the container is ready to handle requests. A failed readiness probe means that a container should not receive any traffic from a proxy, even if it's running.

[Add Readiness Probe](#)

Liveness Probe

A liveness probe checks if the container is still running. If the liveness probe fails, the container is killed.

[Add Liveness Probe](#)

Save

Cancel

TIP: Click on the **Learn More** link or here: https://docs.openshift.com/container-platform/3.11/dev_guide/application_health.html to read more about Health probes

Application Deployment Strategies

From the menu: **Applications > Deployment > nodetest > Configuration**

The deployments can be done in one of two strategic ways. These strategies were designed to cater for two prominent real-world usecases. The first of which is the case where you want to perform a zero-downtime deployment. This is the case where, for example, you have an Application that requires cosmetic changes. In this situation you would set the strategy of the deployment to *Rolling*. This works by ensuring that during the transition, which entails creating a new copy of the Application Pod, then sending one of the previous Pods a *shutdown* event. This means at **all** time the count of active Pods does **not** fall below the required replica count.

The second usecase involves needing to remove **all** active copies of the older versions of the Pod before starting the new version. This would cater for situation where, for instances, there was a serious security flaw in the Container Image that needed to be fixed instantly. Setting the strategy to *recreate* forces OpenShift to not only send a shutdown event to every active Pod in the Application but also to wait for all the Pods to complete terminating before the new versions of the Application are started.

In the top right corner, click the [**Actions > Edit**] button.

Change the [**Strategy Type**] to **Recreate** and click [**Save**]

The screenshot shows the OpenShift Container Platform Application Console. The left sidebar contains navigation links: Overview, Applications, Builds, Resources, Storage, Monitoring, and Catalog. The main content area is titled 'Edit Deployment Config nodetest'. Under 'Deployment Strategy', the 'Strategy Type' is set to 'Recreate'. A note explains that the recreate strategy has basic rollout behavior and supports lifecycle hooks. The 'Timeout' is set to 600 seconds. Below this, there is a section for 'Images' where 'Container nodetest' is listed. A checkbox 'Deploy images from an image stream tag' is checked. The 'Image Stream Tag' is configured as 'sandbox-user1 / nodetest : latest'.

Now go to **Applications > Deployments > notetest**



Note that Deployment \#1 is active.

Click the [**Deploy**] button (top right) and the quickly go back to the **Overview** page.

OPENSIFT CONTAINER PLATFORM Application Console

Sandbox

Search Catalog Add to Project

Overview

Filter by name

List by Application

APPLICATION

notetest <http://notetest-sandbox-user1.apps.jhb-94d8.openshiftworkshop.com>

DEPLOYMENT CONFIG

notetest, #2

Recreate deployment is running... View Events Cancel

CONTAINERS

notetest

Image: sandbox-user1/notetest 631a90e 681.5 MiB

Build: notetest, #1

Source: Added hide and reveal of the ENV details f35527c

Ports: 8080/TCP

0 pods

scaling to 1...

NETWORKING

Service - Internal Traffic

notetest

8080/TCP (8080-tcp) → 8080

Routes - External Traffic

<http://notetest-sandbox-user1.apps.jhb-94d8.openshiftworkshop.com>

Route notetest, target port 8080-tcp



Note that all instances is being recreate and there is zero instances available above.

Go back to **Applications > Deployments > notetest**



Note that Deployment \#2 is active.

Change back to Rolling Strategy: [**Actions > Edit**] then change the [**Strategy Type**] to **Rolling** and click [**Save**]

Now again click the [**Deploy**] and quickly go back to the **Overview** page.



Note that the number of available pods never drops beneath the required number of replicas

Read more about deployment strategies here: https://docs.openshift.com/container-platform/3.11/dev_guide/deployments/deployment_strategies.html

Storage

Go to **Storage >]** page and select **[Create Storage :**

- **Name:** test
- **Access Mode:** RWO
- **Size:** 1 GiB

Click **[Create]**

Persistent Volumes and **Persistent Volume Claims** are a fantastic feature of OpenShift that allow file systems to be mounted into Containers that retain file storage when the Container goes away or is restarted. The concept of **Persisted Volume** is a chunk of storage that is ring-fenced for a Project. This means when you create a PV that space is assigned to the Project only, but **not** mapped into the Containers. To map the storage the concept of **Persisted Volume Claim** is used which locks the storage into a mount point in the Container.

The PVs can be created with different retention strategies. How this works is when there are no longer any PVCs that refer to the PV it is either **retained**, meaning the PV is kept ring-fenced and the contents of the PV are left as is so when a new PVC is created it can continue, or **deleted** meaning the contents

of the PV are wiped and the storage is released for other projects to consume.

Now we will assign this storage to our application. Go to **Applications > Deployments** and select **nodetest** and select the **Configuration** tab. Under the Volumes section, click **Add Storage**

Select the **test** storage option (This is the one we just created)

In the **Mount Path** make sure that the path is unique to you, so make it **/usrX** (Where X is your assigned ID). Click **[Add]**

Deployments > nodetest > Add Storage

Add Storage to nodetest

Add an existing persistent volume claim to the template of deployment config nodetest.

*** Storage**

☒ **test** 10 GiB (Read-Write-Once) Bound to volume **vol411**

Select storage to use or [create storage](#).

Volume

Specify details about how volumes are going to be mounted inside containers.

Mount Path

Mount path for the volume inside the container. If not specified, the volume will not be mounted automatically.

Subpath

Optional path within the volume from which it will be mounted into the container. Defaults to the volume's root.

Volume Name

Unique name used to identify this volume. If not specified, a volume name is generated.

☐ **Read only**
Mount the volume as read-only.

☐ **Pause rollouts for this deployment config**
Pausing lets you make changes without triggering a rollout. You can resume rollouts at any time. If unchecked, a new rollout will start on save.

Add **Cancel**

Go back to the **Overview** page.



Note the redeployment, this is because above is a config change and a new image needs to be build to make this mount point available

Click on the **Pod ring** and select the first (top) pod in the **Pods** section. Select the **Terminal** tab and then type the following:

```
id
```

This will print the unique id for this pod, example:

```
uid=1000360000 gid=0(root) groups=0(root),1000360000
```

Now type the following in the terminal:

```
df -h
```

This will report information on the disk space for this pod, example:

Filesystem	Size	Used	Avail	Use%	Mounted
overlay	50G	7.3G	43G	15%	/
tmpfs	32G	0	32G	0%	/dev
tmpfs	32G	0	32G	0%	
/sys/fs/cgroup					
support1.fourways-3631.internal:/srv/nfs/user-vols/vol411	197G	498M	187G	1%	/usr1
/dev/xvda2	50G	7.3G	43G	15%	
/etc/hosts					
shm	64M	0	64M	0%	/dev/shm
tmpfs	32G	16K	32G	1%	
/run/secrets/kubernetes.io/serviceaccount					
tmpfs	32G	0	32G	0%	
/proc/acpi					
tmpfs	32G	0	32G	0%	
/proc/scsi					
tmpfs	32G	0	32G	0%	
/sys/firmware					

You will see the volume create earlier mounted under `/usrX`.

Type the following:

```
ps -ef
```

At their core Containers are simply file systems with delusions of grandeur. The Image is written to the Node and the Container Runtime then executes it as a process. As far as the Container is concerned it is an Operating System. OpenShift cleverly locks the aspects of the Node specifically to the individual Containers - each Container is locked down using SELINUX constraints meaning not only can it not see the Host components that don't have the appropriate SELINUX labels but also not see other Containers *from the same Application* as they have sub-labelled SELINUX constraints of their own. This means that when you remote-shell into the Application and examine the user, the processes and the filesystems you **only** see the ones specific to the Container - OpenShift **contains** Containers.

Now let's go to the volume mount point and create a file in the root:

```
cd /usrX
touch test.txt
```

List the contents of the folder:

```
ls -a1Z
```

You will see a list of directories and files, example:

```
drwxrwxrwx. root      root  system_u:object_r:nfs_t:s0      .
drwxr-xr-x. root      root  system_u:object_r:container_file_t:s0:c9,c19  ..
-rw-r--r--. 1000360000 65534 system_u:object_r:nfs_t:s0      test1.txt
```

Note the SELINUX constraints on the file systems. The s0:c9:c19 label is specific to the *container*.

Now, in the CLI (NOT the terminal we have been using just now), do the following:

```
oc get pods -o wide
```

You will see a list of all pods, example:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
NOMINATED NODE						
nodetest-1-build-94d8.internal	0/1	Completed	0	1h	10.1.4.8	node1.jhb-
nodetest-2-5lcdq-94d8.internal	1/1	Running	0	15m	10.1.4.12	node1.jhb-
nodetest-2-7dnjv-94d8.internal	1/1	Running	0	12m	10.1.4.14	node1.jhb-
nodetest-2-nfnlf-94d8.internal	1/1	Running	0	12m	10.1.4.13	node1.jhb-

Choose two Pods that have landed on different physical Nodes. Make a note of the two Pod names - for information the structure of the Pod name is as follows - *(ApplicationName)-(deployment number)-(random five character identifier)*

Go back to the **Overview** page.

Click on the **Pod ring** and select the first (top) pod in the **Pods** section. Select the **Terminal** tab and then type the following:

```
cd /usrX
vi test.txt
```

Once in **vi**, press **i** to enter **insert** mode.

Now type something, example: **Hello World**.

Then press **Esc** (to exit the insert mode) and then **:wq** to write and quit vi. You can do a **cat** to make sure the contents is saved in the file:

```
cat test.txt
Hello world
```

Now go back to the **Overview** page.

Click on the **Pod ring** and select any pod except the first (top) one in the **Pods** section. Select the **Terminal** tab and then type the following:

```
cd /usrX
cat test.txt
Hello world
```

As you can see the file is available on all pods.

Note that because we chose the **RWO** (ReadWriteOnce) this sets up a Persisted Volume that exists **once** across the entire Cluster. This means that if you have multiple replicas of an Application in Pods on different Nodes they are all mounted to the **same** piece of filesystem. If one Pod changes the contents of a file that change is visible to all Pods of that Application.

Config Maps

Navigate to **Resources > Config Maps** and then click [**Create Config Map**]

In this scenario we are going to express the contents of a configmap within the Container as environment variables. You can already define environment variables to be expressed into an Application as part of the **deploymentconfig** but these are Application specific in that they are controlled and injected as part of the deployment. With a configmap that is within the Project but external to the Application these environment variables can be shared across multiple Applications and centrally changed.

Enter the following in the fields: * **Name:** configmapenv * **Key:** CONFIGENV * **Value:** somevaluefortheenv

Then click [**Create**]

Config Maps > Create Config Map

Create Config Map

Config maps hold key-value pairs that can be used in pods to read application configuration.

* Name

A unique name for the config-map within the project.

* Key

A unique key for this config-map entry.

Value

Enter a value for the config-map entry or use the contents of a file. [Browse...](#)

[Clear Value](#)

1	somevaluefortheenv
---	--------------------

[Remove Item](#) | [Add Item](#)

[Create](#) [Cancel](#)

Navigate to **Applications > Deployments** select `nodetest` and then the **Environment** Tab.

In the **Environment From** section, select the `configmapenv` we just created.

Click the **Add ALL Values from Config Map or Secret** link and then **[Save]**.

Now go back to the **Overview** page, and watch the deployment finish.

Click on the **Pod ring** and select the first (top) pod in the **Pods** section. Select the **Terminal** tab and then type the following:

```
env | grep CONFIGENV
```

You will see the key/value we just created.

Now let's create another config map. Navigate back to **Resources > Config Maps** and then click **[Create Config Map]**

This approach entails expressing the contents of a configmap into the Container as a physical file. This is extremely powerful in that you can physically overwrite files that are defined in the Image itself - a good example of this is having an Image for a technology that is configured at runtime by reading values from a configuration file. Using a configmap you can express different versions of the file into the Container *without having to rebuild the Image*.

Enter the following in the fields: * **Name:** configmapfile * **Key:** myapp.conf * **Value:** hello!

Then click **[Create]**

Navigate to **Applications > Deployments** select **nodetest** and then the **Configuration** Tab.

In the **Volumes** section, select **Add Config Files**:

- **Source:** configmapfile
- **Mount Path:** /config/app

Click [**Add**]

Now go back to the **Overview** page, and watch the deployment finish.

Click on the **Pod ring** and select the first (top) pod in the **Pods** section. Select the **Terminal** tab and then type the following:

```
cd /config/app  
cat myapp.conf
```

You will see the value we just created.

How this actually works is the configmap, and also **secrets** which work in a similar fashion but are encrypted at rest and on the nodes, are copied to the filesystem of the Node and then linked into the running Container using a symbolic link locked down with the appropriate SELINUX constraints. The file doesn't appear as a mounted filesystem, like the **Persisted Volumes** and is separate physically from the Container, again meaning you can change the contents of the configmap, a deployment will automatically occur (as you've changed the configuration of the deployment) and the new Pod will have the new configmap file visible.

Secrets

Navigate to **Resources > Secrets** and then click [**Create Secrets**]. Enter the following:

- **Secret Type:** Generic Secret
- **Secret Name:** nodetestsecret
- **Key:** mypassword
- **Value:** mydodgypassword

Click [**Create**]

Secrets > Create Secret

Create Secret

Secrets allow you to authenticate to a private Git repository or a private image registry.

Secret Type

Generic Secret

*** Secret Name**

nodetestsecret

Unique name of the new secret.

*** Key**

mypassword

A unique key for this secret entry.

Value

Browse...

Enter a value for the secret entry or use the contents of a file.

[Clear Value](#)

1	mydoggypassword
---	-----------------

[Remove Item](#) | [Add Item](#)

[Create](#) [Cancel](#)

Now select the newly created secret `nodetestsecret` and then click **[Add to Application]**.

Select the `nodetest` application and click **[Save]**.

Now go back to the **Overview** page, and watch the deployment finish.

Click on the `Pod ring` and select the first (top) pod in the `Pods` section. Select the `Terminal` tab and then type the following:

```
env | grep password
```

Secrets are a specially handled Object within OpenShift in that they are encrypted at creation, so before they are linked to any deployment they are unreadable, and when they are written to the Nodes and expressed into the Container they are encrypted on the filesystem of the Node, meaning a sysadmin with access to the Node would not be able to read them. They are unencrypted within the running Container.

Now, in the CLI (NOT the terminal we have been using just now), do the following:

```
oc describe secret nodetestsecret
```

This will show the secret, example:

```
Name:      nodetestsecret
Namespace:  sandbox-user1
Labels:     <none>
Annotations: <none>
```

```
Type: Opaque
```

```
Data
```

```
====
```

```
mypassword: 15 bytes
```

Now look at the secret in the object:

```
oc edit secret nodetestsecret
```

Here you can see the secret is encrypted:

```
apiVersion: v1
data:
  mypassword: bX1kb2RneXBhc3N3b3Jk
kind: Secret
metadata:
  creationTimestamp: "2019-07-31T05:09:01Z"
  name: nodetestsecret
  namespace: sandbox-user1
  resourceVersion: "167161"
  selfLink: /api/v1/namespaces/sandbox-user1/secrets/nodetestsecret
  uid: 4f06e44d-b351-11e9-b116-16c647cb1fdc
type: Opaque
```

Clean up

Now let's clean up everything we did in the [Simple application lifecycle](#) section:

```
oc describe bc nodetest
oc delete all -l "app=nodetest"
```

Note - when we created the Application the template prompted us for the name of the Application. This was then auto-appended to **all** Objects created as part of the Application as a label set to "app=name", in this example "app=nodetest". This allows us to issue commands dealing with all the Objects by appending the **-l "app=nodetest"** which makes the command apply **only** to all Objects with that label.

Software Defined Networking

OpenShift maintains an isolated internal Network for traffic between the Services that are orchestrated within it. This **SDN** (Software Defined Network) maintains an abstracted internal Network allowing the Services to interact or be isolated from each other. The other important feature of this SDN is that all Services have an abstracted interface such that each Service has a unique Cluster address, which provides the linkage to the outside world via Routes that are attached to the Service, and the actual instances of the Application Pods. This singular Cluster IP address works as a load-balancer across all the actual instances, which have separate IP addresses. This nice feature means that the Service endpoint, as opposed to the Pod endpoint, is singular within the SDN and additional configuration can be applied to define the behaviour of internal load-balancing (for instance round-robin, least-connection or label driven routing).

If you have **Cluster Admin** rights which provide visibility of **all** Objects in the Cluster (you shouldn't at this point), you can get the cluster network:

```
oc get clusternetwork
```

NAME	CLUSTER NETWORKS	SERVICE NETWORK	PLUGIN NAME
default	10.1.0.0/16:9	172.30.0.0/16	redhat/openshift-ovs-subnet

OpenShift 3.x provides three types of **SDN** plugin which offer different levels of visibility for the Objects within them. By default an installation will implement the **Subnet** plugin, meaning the SDN is flat and all projects share the same network ID, meaning Project A can directly call Project B across the Service network.

The next available type is **Multitenant** which enforces a different Network ID **per** Project. This means that the Projects are network isolated by default, although Cluster Admin rights holders can *join* Project networks, which makes individual Projects share Network IDs, so that Services can call each other (but only in Projects that have been joined).

The highest level of fine-grain network isolation and control is imposed by installing the **Network Policy** SDN plugin. This allows individual Network Policies to be implemented for *Objects* within the Projects. This allows for a complex and controlled definition of ingress and egress for the traffic, for instance you could define a Policy allowing a single Service within a Project to be visible outside of the Project and all other Services to be hidden.

Let's create a new test project using the CLI:

```
oc new-project sdntest-userX --description="SDN Test" --display-name="SDN Test"
```

You can make sure you are using the newly created project:

```
oc get project sdntest-user1
```

Now let's add Applications to our new project:

```
oc new-app registry.access.redhat.com/rhsc1/nodejs-8-  
rhel7~https://github.com/uthep0/nodejs-ex --name="nodetest1"  
oc new-app registry.access.redhat.com/rhsc1/nodejs-8-  
rhel7~https://github.com/uthep0/nodejs-ex --name="nodetest2"
```

Note that unlike the use of a Template creating an Application with the *oc new-app* command does not automatically expose a Route. The Applications will be deployed as per normal with defined Service endpoints but no external connectivity.

Back in the Web Console. Make sure that you select the **SDN Test** Project in the project dropdown.

In the **Overview** page you will see the two nodetest application we just created.

Click on the **Pod ring** for nodetest1, under the **Status** section you will see the IP address for this Pod.

Navigate to **Applications > Services**. Note the Cluster IP's for nodetest1 and nodetest2.

Now go back to **Overview** and select the **Pod ring** for nodetest1.

Select the **Terminal** tab and type the following:

```
curl http://localhost:8080
```

This will return the basic HTML for the node.js application, example:

```
<html>  
<head>  
  <title>Test Page for nodejs-ex app</title>  
</head>  
  
<body>  
<b>Test page for nodejs-ex</b> - served from res.render.  
  
This is a simple webpage rendered from the node.js application at the root endpoint.<p/>  
  
</body>  
</html>
```

Again note that when within the Container itself it thinks it is an OS. The networking exposed to the

Container allows the Container to refer to its own network space using the *localhost* name. Also note that, unlike Routes which auto-redirect the traffic to the appropriate Port by definition, we need to add the :8080 when using an internal curl.

Now do the same, but using the pod name:

```
curl http://nodetest1:8080
```

You will see the same response.

Note that we can refer *directly* to the Service name itself as a resolvable network address. The internal network space provided to the Container injects a DNS resolvable name for **all** Services in the Project (unless over-written by Network Policies particular to that Project).

Let's see if we can see the other Pod from here:

```
curl http://nodetest2:8080
```

As you can see the name is resolved to the correct IP for a Pod.

Note that the most important thing about this example is that we can directly refer and connect to **other** Services within the Project using the shortname of the Service.

Let's now do a full name resolution:

```
getent hosts nodetest1
```

You will see the Cluster IP and the full URL for nodetest1, example:

```
172.30.179.70    nodetest1.sdntest-user1.svc.cluster.local
```

In addition to exposing the shortnames of the Services as resolvable DNS entries into the networking for the Container OpenShift also, depending on the type of SDN installed, exposes **all** Services in the Cluster with a **FQDN** (Fully Qualified Domain Name). This name takes the format of **(service_shortname).(project_shortname).svc.cluster.local**. Again note that this is dependent on the SDN installed - in the case of multitenant for example every Project will, by default, be defined in its own NetworkID and the FQDNs for Services external to the Project will not be resolvable.

Now curl the full URL:

```
curl http://nodetest1.sdntest-userX.svc.cluster.local:8080
```

You will again get the test page as a response.

A quick re-mention here of the mechanisms by which the end IP is determined - the FQDN resolves to the singular Service IP address which acts as a load-balancer (using HAProxy) across all the current replicas for the Application. This IP translates, based on the rules chosen (by default RoundRobin) to the IP of the *Pod*.

Let's try the person next to you:

```
curl http://nodetest1.sdntest-userX+1.svc.cluster.local:8080
```

Now let's create a route, navigate to **Applications > Route** and click **[Create Route]**:

- **Name:** canaryroute
- **Service:** nodetest1
- **Alternate Services:** select *Split Traffic Across Multiple Services*
 - **Service:** nodetest2
 - **Service Weights:** 80% nodetest1, 20% nodetest2

Click **[Create]**

The screenshot shows the 'Create Route' form in a Kubernetes dashboard. The form is divided into several sections:

- * Name:** A text input field containing 'canaryroute'. Below it is a note: 'A unique name for the route within the project.'
- Hostname:** A text input field containing 'www.example.com'. Below it are two notes: 'Public hostname for the route. If not specified, a hostname is generated.' and 'The hostname can't be changed after the route is created.'
- Path:** A text input field containing '/'. Below it is a note: 'Path that the router watches to route traffic to the service.'
- * Service:** A dropdown menu showing 'nodetest1'. Below it is a note: 'Service to route to.'
- Target Port:** A dropdown menu showing '8080 → 8080 (TCP)'. Below it is a note: 'Target port for traffic.'
- Alternate Services:** A section with a checkbox labeled 'Split traffic across multiple services' which is checked. Below it is a note: 'Routes can direct traffic to multiple services for A/B testing. Each service has a weight controlling how much traffic it gets.'
- * Service:** A dropdown menu showing 'nodetest2'. Below it is a note: 'Alternate service for route traffic.'
- Remove Service:** A blue link.
- Service Weights:** A section showing 'nodetest1 80%' and '20% nodetest2' with a slider bar between them.

Now select the newly created **canaryroute**.

What we have done here by creating a Route but splitting the traffic, on a percentage basis, across two

Service endpoints is to introduce a further level of initial load-balanacing for the traffic coming into this Route. Now, when a consumer initially makes a call to the Route, the target **Service** is chosen based on the percentage of traffic. This Endpoint is then given to the Route for that consumer, which then goes through the load-balancing rules for the *Service* itself to determine which end **Pod** is resolved by the Route.

If you go back to the **Overview** page, note that both applications has got the same external route.

Click on the **Pod ring** of nodetest1, select the **Terminal** tab and then type the following:

```
getent hosts canaryroute-sdntest-userX.(domain)
```

This will return the IP of the OCP router, example:

```
3.219.175.39    canaryroute-sdntest-user1.apps.jhb-94d8.openshiftworkshop.com
```

Note that the IP returned to the consumer for **all** Routes will be one of the IPs for the **Routers** running in the OpenShift Cluster. This is because the resolution of the endpoint is done internally within OpenShift and the Routers are the **only** way to access the Services. You can associate other IP addresses to refer to Projects but this functionality is outside of the scope of this workshop.

User Role based access control

OpenShift maintains a comprehensive and configurable **RBAC** (Role Based Access Control) mechanism across the Objects within a Project.

In order to discuss and understand this we need to understand the nature of **Users** and **Roles** within the OpenShift system. There are actually two types of Users within a Project, the *Users* which are physical Users, and *Service Accounts* which are effectively internal *virtual* Users that are used by OpenShift to actually execute tasks.

Think of *Service Accounts* as proxy-users within the Project that are individually controlled and constrained on the *actions* they can do within the Project. For example by default a Service Account is created within the Project with the following name - `system:serviceaccount:*(PROJECT_NAME).default` - **this user is used to do the builds and deployments within the Project and has access to the Objects within the Project** *and the rights to perform a build, push to the registry in the name of this Project and deploy Containers.

Users themselves are bound by a visibility role that gives them different levels of visibility and accessibility to the Objects within the Project. There are four distinct tiers of access provided:

Admin - a user with Admin rights, which are applied to the creating User of a Project, can create and delete Objects within the Project and give and remove access rights to other Users.

Edit - a user with Edit rights can change the **value** of existing Objects within the Project but cannot create or delete Objects, or assign rights to other Users.

Read - a user with Read rights can see the values of existing Objects within the Project but cannot create or delete objects and cannot change the values of existing Objects either.

None - without being assigned rights to a Project a user has **no** visibility of the Project or the Objects within it.

Using the project dropdown, go back to the **Sandbox** project we create earlier.

Go to the **Resources > Membership** page. (using the left side menu)

You will see 3 tabs:

- **Users:** This should list your user (userX).
- **Groups:** - TODO: explain the difference between the full name `system:serviceaccount:sandbox-userX` and the *all* groups `system:serviceaccounts:projectName`
- **Service Accounts** - TODO: explain the concept of a *virtual* user within the namespace.

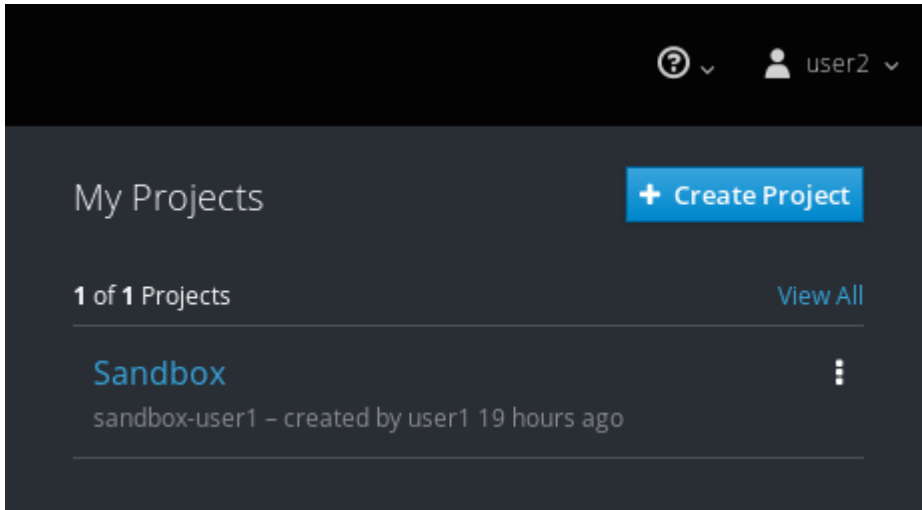
In the **Users** Tab, click [**Edit Membership**] and add the attendee next to you as a user:

- **Name:** UserX+1

- **Roles:** admin

Click [**Add**]

You will now see the project assigned to you in your Project list, example **User2** can see **sandbox-user-1** project:



(click on the OPENSIFT CONTAINER PLATFORM logo)

Now do a build of the application in that project.

Next, let's remove the access from that user using CLI.

First make sure you are using the correct project:

```
oc project sandbox-userX
```

Next remove the access:

```
oc adm policy remove-role-from-user admin userX+1
```

Now go back to your project list and note that your access has been removed.

CI/CD Pipeline

OpenShift provides an integrated facility for the execution of Pipelines as part of the Build process. It was decided, at OpenShift 3.x, to integrate to Jenkins - this is facilitated by running a Jenkins Master in a Pod as an Application and then executing an externally held Pipeline definition, written in the Jenkins DSL with OpenShift DSL extensions (in Groovy) which is contained as a Build within OpenShift.

This allows for some sophisticated steps as part of a Build (the Pipeline strategy is one of the available Build types with OpenShift) including automated testing using tools such as Sonatype, manual stage-gates and multiple Build steps. A successful Build is indicated by Jenkins completing the Pipeline script with no failures.

As of OpenShift4 this functionality is replaced with the Kubernetes Tekton project, which introduces CICD Pipelines as **fully** integrated and controlled Objects within the Object model. This is a far nicer approach - handing off responsibility and ownership of the entire Pipeline to an embedded Jenkins Pod removes control from the orchestration platform.

This exercise is provided for use within the Workshop as an example of the integration of CICD within OpenShift but is highly advised that any Users wishing to use Pipelines going forward look at Tekton.

This exercise requires a little manual intervention in the scripts which is described below - this uses a script to install, pre-configure and define the Pipeline to be used. Normally this would be an automated process but for the sake of the workshop it has been simplified.

In your chosen CLI, let's get the workshop workshop_material:

```
mkdir ocp_workshop
cd ocp_workshop
git clone https://github.com/utherp0/workshop_material
cd workshop_material/attendee
ls
```

We will be using two scripts from this repo for this example, **groovy_pipeline.sh** which sets up the Application we will be using and injects the Groovy Pipeline DSL, which is defined embedded in a **BuildConfig** within **pipeline_complex_bc.yaml**. Note that in actual real world usage the Groovy components would be injected from a repo (which is the other option for definition as part of a BuildConfig as opposed to embedding the actual script into YAML).

```
cat groovy_pipeline.sh
```

We now have to perform a manual task to replace the **PROJECT.NAME** placeholder in both scripts. This is because a Project name needs to be unique across an OpenShift cluster so each user has to use a unique Project name. We will use the Linux *sed* command to run a replace as shown in the following instructions.

```
sed -i 's/PROJECT.NAME/pipeline-userX/g' groovy_pipeline.sh
cat groovy_pipeline.sh
```

You should now see the PROJECT.NAME being replaced by your unique name.

```
sed -i 's/PROJECT.NAME/pipeline-userX/g' pipeline_complex_bc.yaml
```

Before we run the script, make sure you are still connected to the OCP Cluster:

```
oc whoami
```

Then run the script

```
./groovy_pipeline.sh
```

What the script is doing is defining a new Application, called *nodetest*, setting up the Build and Deployment that we will then automate (automatically using the script) using the Pipeline technology. The script creates the Application, waits for the Pod to appear, then install and starts the Pipeline process.

The Pipeline will Build, Deploy and then **Scale** the Application automatically.

Note that we had to remove the automatic trigger on the Deployment - the reason for this is by default an Application will automatically deploy when a Build has finished. If we are running the Build and then the Deploy as part of the Pipeline we would see **two** deployments, the first being as a result of the Build, executed by the Pipeline, and the second manually by the Pipeline. The script changes the configuration of the Deployment to have no automatic triggers.

Once the process has started, go to the Web Console and select the *pipeline-userX* project.

In the **Overview** page you will see the Jenkins Ephemeral Pod being started up.

Note this step may take a while. The Jenkins Pod is a large and complex Pod that takes a while to start up, and if you are running this on a demo installation of OpenShift the shared resources may make starting a large number of these Pods take a while. Once the Jenkins Pod has started correctly (indicated by the Pod ring going Blue) you can click on the Route and see the logs within Jenkins as it kicks off the Build Pod, completes, then executes the Deployment.

Back in the Web Console, go to **Builds > Pipelines**. Here you can see the pipeline running.

Note that all the stages of the Pipeline are rendered separately. Each step covers the appropriate code in the Groovy script including start, build, deploy and scale.

If for any reason the Build or the Deployment fails check the script to ensure the Project name is set correctly.

```
oc delete bc {old_name}  
oc create -f pipeline_complex_bc.yaml
```

[UI] Overview - show the six pods running, point out the deployment number, Discuss the removal of the triggers on the deployment and why (with automation in place the creation of an image would force a deployment which would interfere with the pipeline, we disable the deployment on image and config triggers so the pipeline can control it all) Discuss complexities of pipelines, the addition of Jenkins commands into the Groovy etc [UI] Go back to the Jenkins Route - select Open Blue Ocean. Choose the Pipeline, click on the build, show the prettier Jenkins UI [UI] Builds/Pipelines - click on complexpipeline - Actions/Edit [UI] Replace the line 'dc.scale("--replicas=6")' with 'dc.scale("--replicas=4")' [UI] Start Pipeline, then switch to Overview Explain what is happening, show the Pod count changing

Wrap-up

- Q&A
- Also see <https://launch.openshift.io/>
- Operators on OCP4



Mark Roberts / Ben Holmes / Anthony Kesterton

Senior Solution Architect

marrober@redhat.com

Original workshop content done by Ian *Uther* Lawson