OpenShift

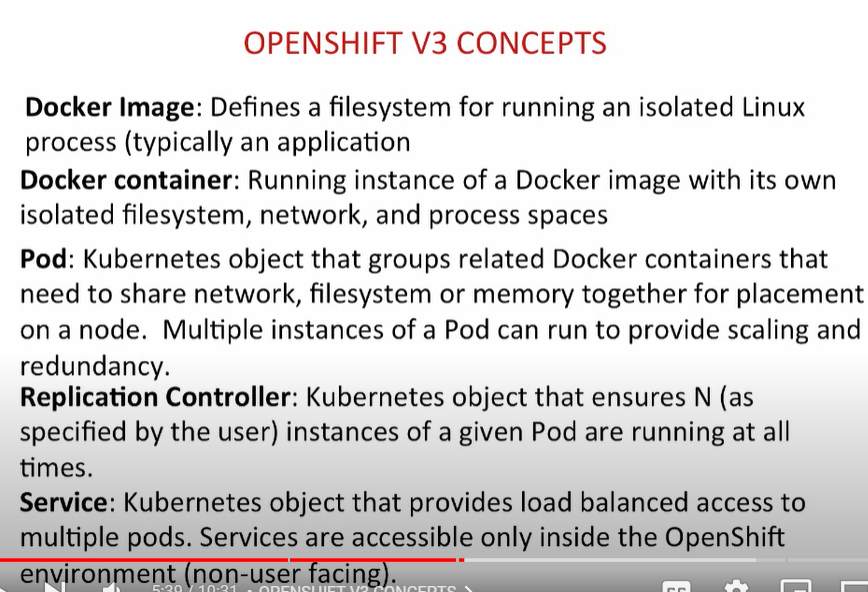
# Open shift Stack

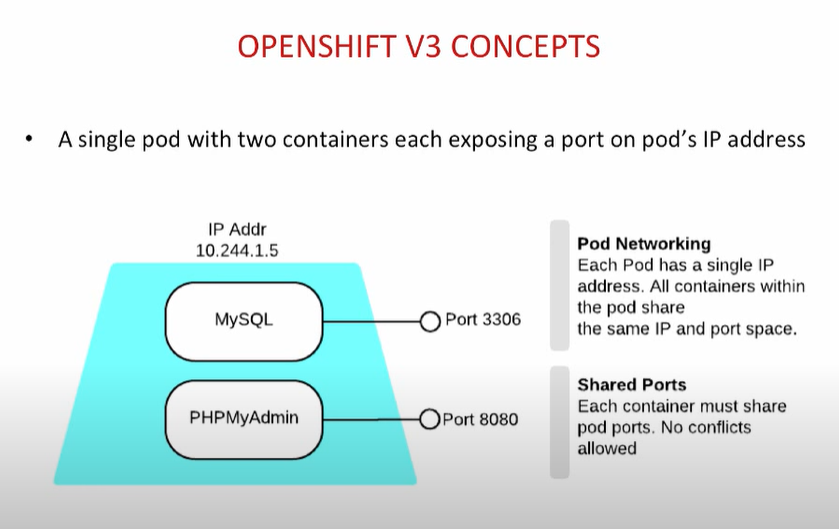
Kubernetes is an orchestration tools which mean it manage individual docker containers.

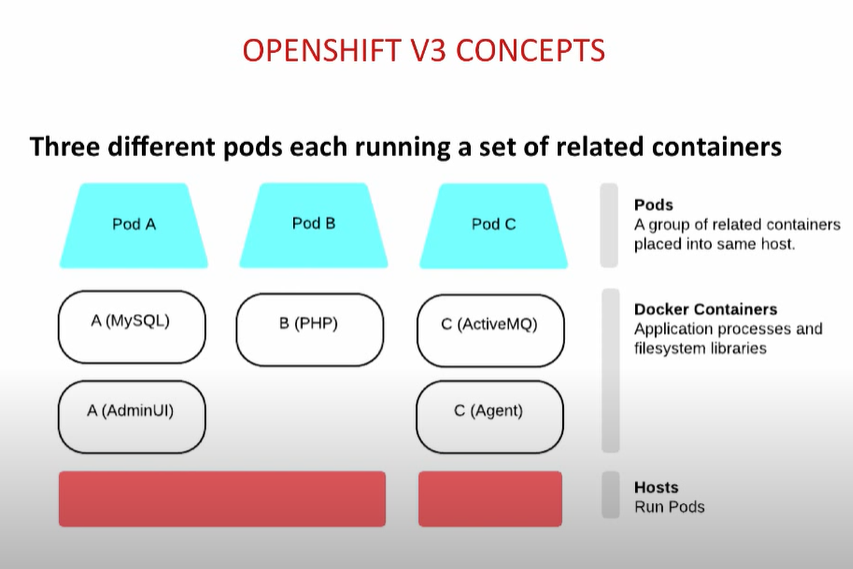
OpenShift is sitting at top as user experience layer which also provide containerised services like Jenkins, app server images and marketplace

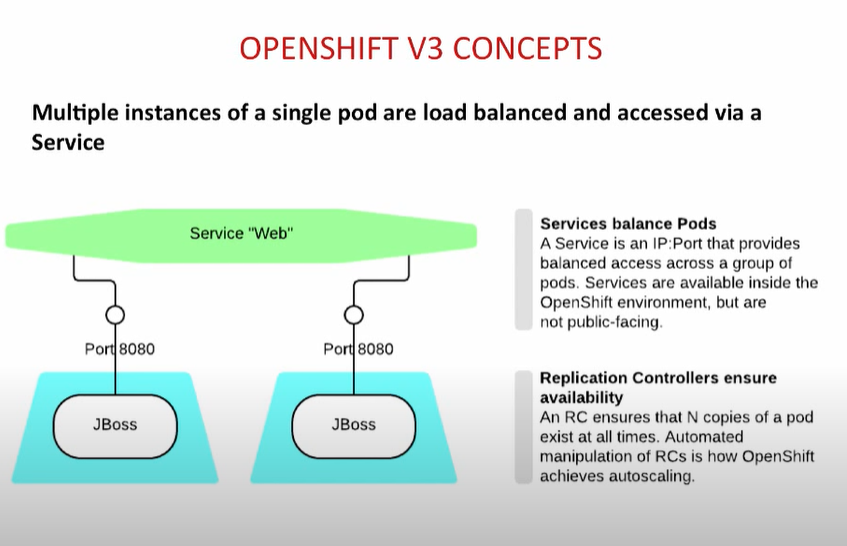


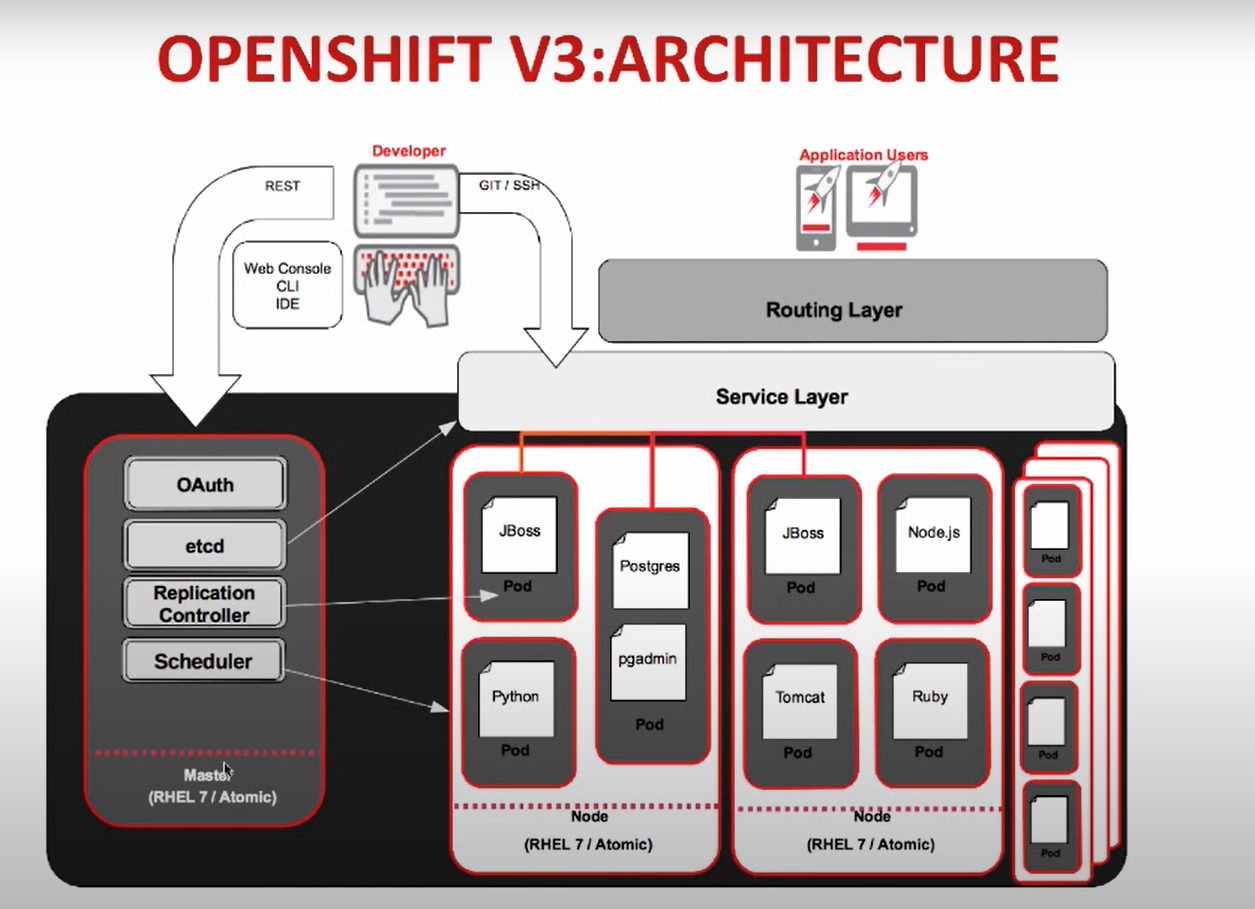
Openshift Concepits

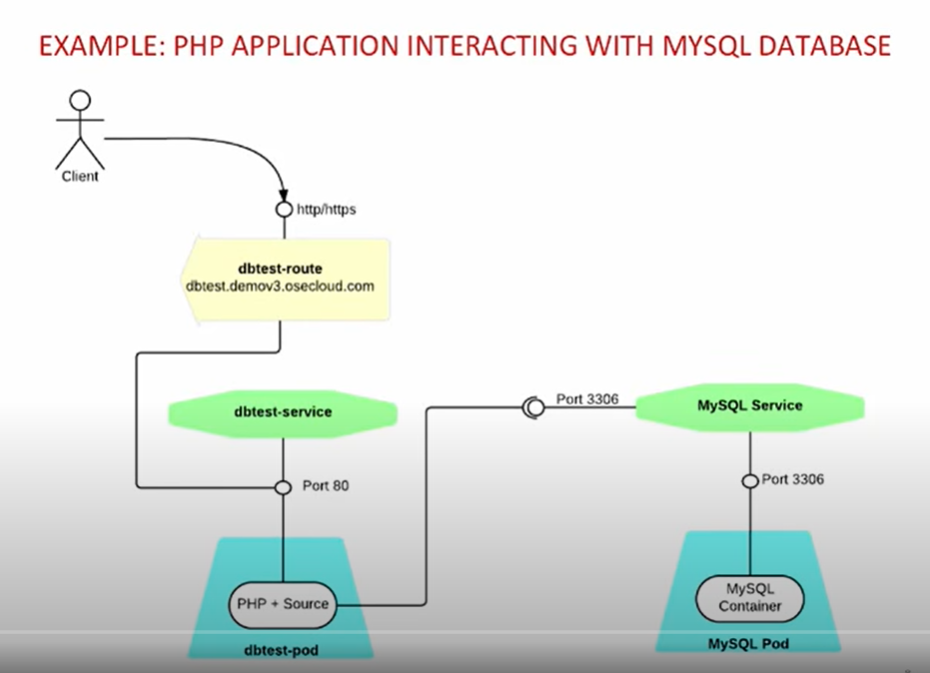












# Build in OpenShift

There are 3 different types of build in openshift

1. **Docker image Build** ---*docker build -t getting-started .*
2. **SoucetoImage** -----*s2i build https://github.com/IBM-Cloud/get-started-node nodeshift/centos7-s2i-nodejs:latest us.icr.io/$MYNAMESPACE/webapp*
3. Custom Build

**Note – In S2I- base image is passed along with source code to build the final image as shown in above example**

Docker Image Build – {here docker file contains the base image}In the above example, first we will use mysql image from docker hub and will use it to create the mysql containerized app.

Then we will create the image from our code outside the OpenShift and will push this image to dockerhub and then will use this image to launch our app container which will talk to mysql through service layers as shown above

Now to create an Application, we can not deploy the above images directly. A Json file (say template ) is required to create the application. This Json contain the code to deploy the above images.

Json template not only just contain the reference of an image but also the additional thing …like once the container will be created using below image, template will set some env variable to access that mysql.

And not only additional setting as described above. It will also create few additional thing like service layer as below

For example – Below json code will create a container with mysql using the image from dockerhub and will also set the env values like below



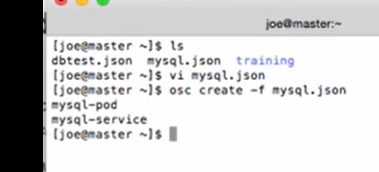
This Template will also create the service as below.

**Important Note** - How would a service know about its pods/containers. How would it know where it needs to direct the traffic.

This is where selector comes into role. As below example….Under Service. Selector name is “**mysql-label”** and pod/container was also created using the same label…so Service layer would know to which pod or container it has to send the traffic.



Command to create application from Jason

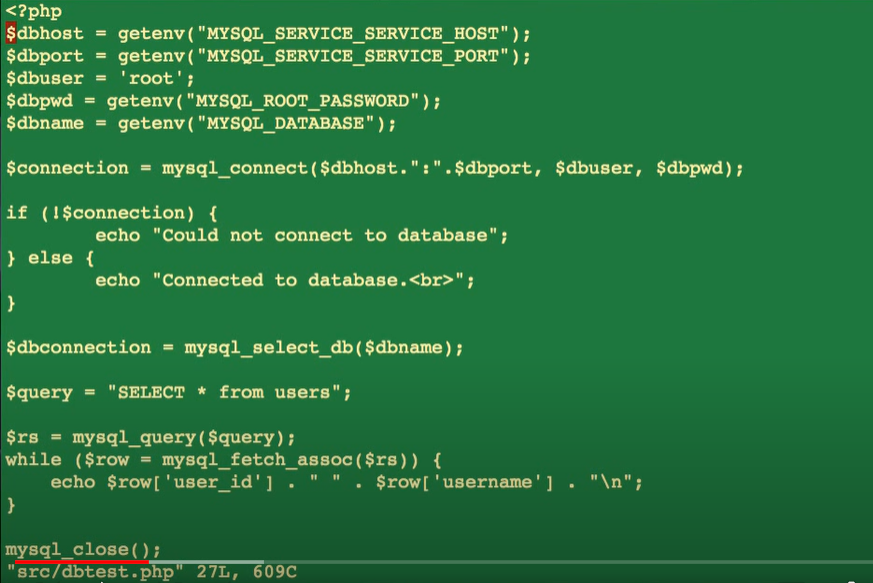


# Creation of client application from source code

We have a code in our local and we want to first create the image out of openshift which we will push to docker hub. Using this image, application will be create using Jason file.

As per the architecture diagram, client application will talk to DB service layer. All the env variable are already set in code, the only thing remaining is DB service host ip and port (which I believe was passed as switch -H and -P) during the creation of application through Jason.

DBTest.php code below



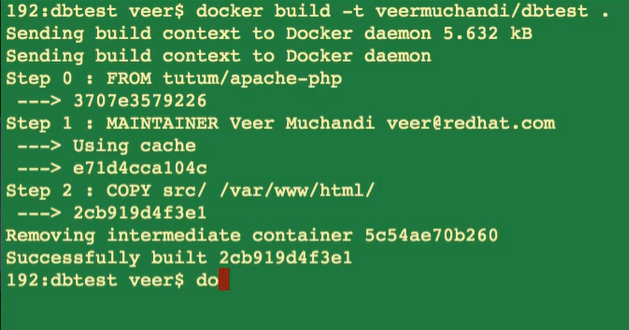
**To create an DB image for our code, we need a docker file.**

**With out the docker file, image cannot be created.**

**This docker file simply tells about the app server on which my code will run and location where the code will be copied.**



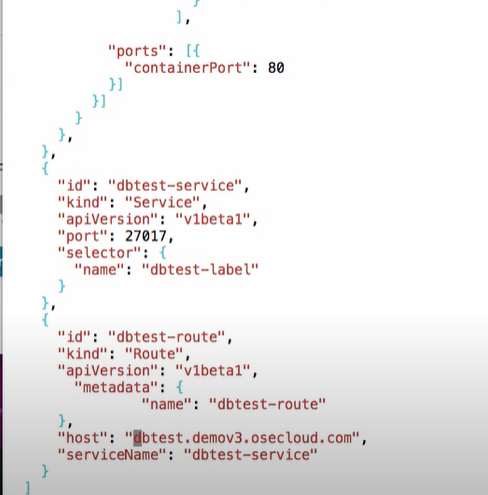
## Command to build docker Image for code



**Now we need to push this image to docker hub to use it further to create an application.**

**Now to launch application, as did before we need a Jason file like below**





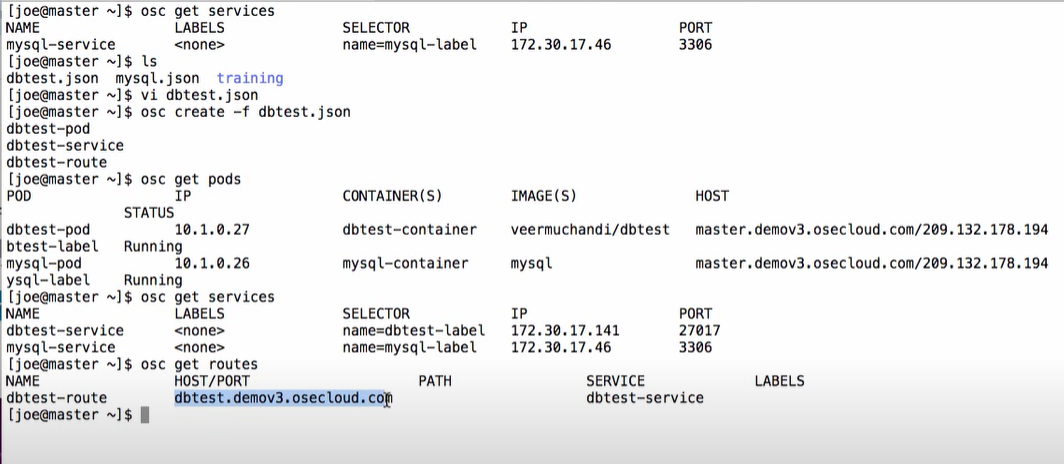
This file simply tells what pod will be created using which image and env variables values. (These values will be passed to code env variables)

It will also create service layer with port and route.

**See the dependency between route, service , pods and containers. In the above json Route will tell which service it will send the traffic to, Service is telling to which pods/container (through selector ) it will send the traffic .**

**In Json ID tells the name, Kind tells which service ( pod/container/service/dc/route etc), port tell where it is serving, label tells how to group them under one umbrella ( like service and pods relation)**

# Command to launch code application using Jason



**2.** SourceToImage (S2I)

Here we directly make the image from the source code in github with the help of image builder.Automated builds are triggered though webhooks

# Replication Controller (Scaling and HA)

RC make sure that your application has n number of pods at any given point of time

As in below example, dbtest only running on 1 pod, we can resize it to work on 2 pods always as shown below

A picture containing text

Description automatically generated

**Some Important Points**

**Builds** -> It works to builds the source code to create the Image. Building means creating the image

**Deployment** - > It means deploying the image (created in build) to pods

Graphical user interface, application

Description automatically generated

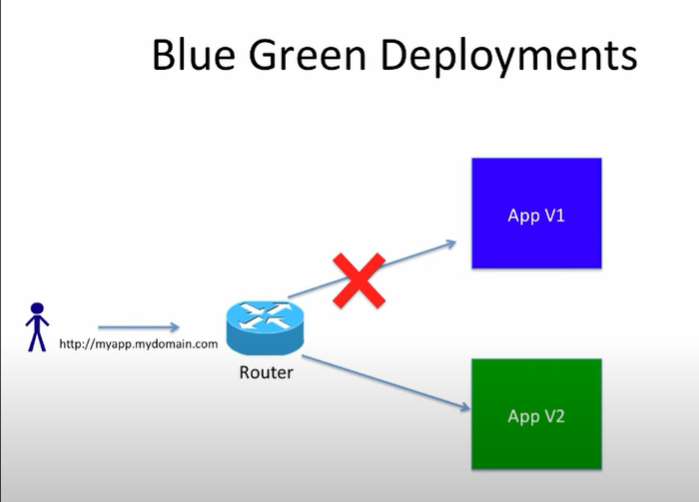
# Deployments

**Rolling Deployment** (by default)- If the replication controller are sized to more than 1 and we try to deploy the new code in github, then by default openshift do the rolling deployment…means it will deploy the code on pods one by one so there wont be any downtime.

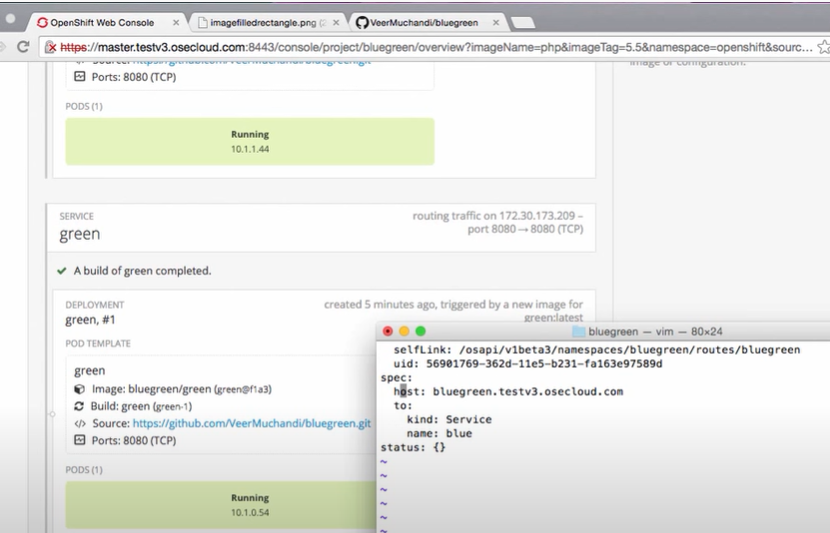
**BlueGreen – In this we edit the router to direct the traffic to service layer**

**AB – In this we edit the service layer and DC to direct the traffic to newly created pods ( DC)**

# BlueGreen Deployment



Editing the route to swich from one deployed service to other.



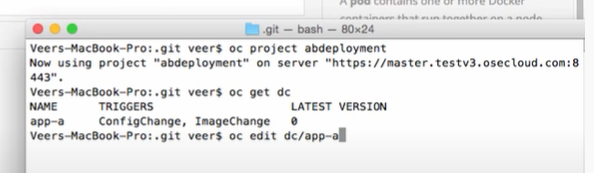
DC - Deployment Config

DC is used to create the pods so all the setting of DC are applied to pods like labels, selector etc.

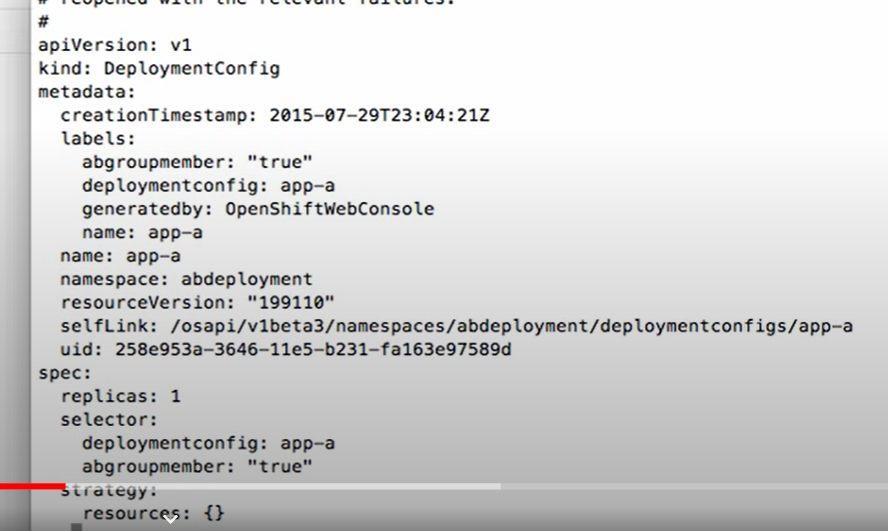
# AB Deployment

Under Ab deployment, dc is edited to make sure it can direct traffic to the pods which have got updated version of deployment

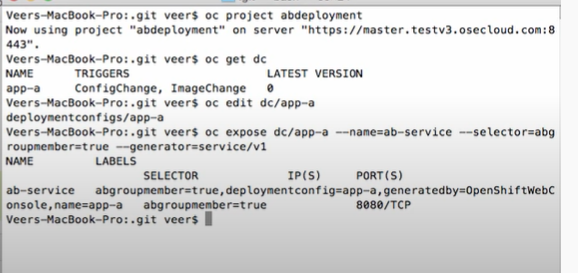
To achive this , edit the DC config and make change in the label and selector. This will ensure that whenever new pods will be created , it will have the lebel and selector which will be picked by service layer.



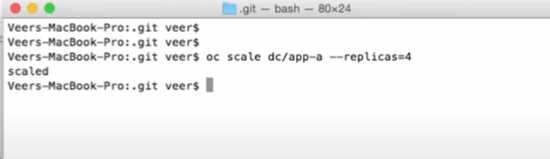
**Add label to the selector also**



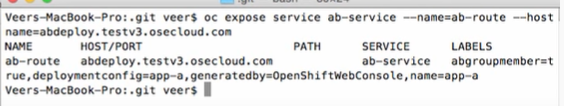
**Now add a new service layer on the top of this dc..create it using the same selector, this will make sure and new pod created using this dc , service will recongnize that pod with the help of selector and add under it**



**Now we can scale up the pods using the same dc like below**



**Now we need to expose this service to a new route.**



Now make change in code and create a separate build ….In openshift it will be considered new app.

But since our new pod has the same label which was given in the selector of service…..the service layer will pick it and start sending traffic to this updated pod

# Persistence Volume PV

Administrator create the persistent volume on host machine and make it available for the user to use. Users create the pods and claim the persistence volume. Multiple pods can do the same PVC.

# SSL with OpenShift

## Edge Termination

In Edge termination, Traffic from browser till router is encrypted and router decrypt the encryption.

A picture containing timeline

Description automatically generated

Below router json/yaml can be edited to terminate the SSL through default certificate

Text, letter, email

Description automatically generated

If you have your own certificate, the we can generate the keystore file. It will give you certificate key and private key

Now copy these keys in the route json

Text

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with low confidence

## Pass Through Termination

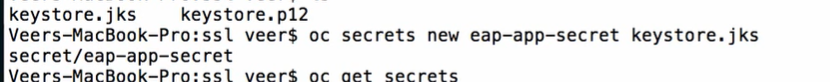
In this Request remain encrypted till pods and at pods certificate is decripted.

To do it we use openshift Secrets. Secrets are used to store the certificate keys.

A picture containing diagram

Description automatically generated

Keys are added to secret as below.



Now while creating the application/pods we can attach this secret to any application or pod.

## Re-Encrypt Termination

Here router terminates the certificate and re-encript it is using same or different certificate. ( we can say it Is a combination of both the above certificates)

Diagram

Description automatically generated with medium confidence

# Build Promotions

Build Promotion Scripts

<https://github.com/VeerMuchandi/jenkinsose3>

Diagram

Description automatically generated

To follow the build promotion, first we need to create a new Application in development project with below command

oc login -u dev1

oc project development

oc new-app --template=eap6-basic-sti -p APPLICATION\_NAME=myapp,APPLICATION\_HOSTNAME=myapp-dev.apps.demov3.osecloud.com,EAP\_RELEASE=6.4,GIT\_URI=https://github.com/VeerMuchandi/kitchensink.git,\

GIT\_REF=,GIT\_CONTEXT\_DIR= -l name=myapp

Once Application is created, checkout the image stream with below commands

oc get is

oc describe is

Now we need to tag the image using the *PullSpec*  from below screenshot.

Tag the Image as below

oc tag PullSpec development/myapp:promote

Graphical user interface, text, application, email

Description automatically generated

Now again if you do the oc get is

Then you will see 2 tagged images i.e: latest and promote

Graphical user interface, text, application

Description automatically generated

Now Image is ready to test…we need to login to QA project in OC and create the application using this tagged image like below

OC new-app development/myapp:promote

A picture containing text

Description automatically generated

Since, in QA we created the application using the image not with **template** so we need to create the route manually by exposing the service( please note, exposing service automatically creates the route)

Oc expose service myapp

Text

Description automatically generated

Now QA deployment is completed for Version 1.

To create a automated build process, lets make change in the development project git code and

Create the new image using the OC start-build myapp.

This command will create a new image like below. Please note creating a new image will automatically deploy the image in container ( mean oc new-app myimage not necessary)

Becaause it trigger on image change.

Now we can tag the new image like before as shown in below pic. The testing environment will also automatically deployed as I described above ( because image with promote tag got changed)

Graphical user interface, text, application

Description automatically generated

If you images are externally hosted , then we have to do the same above build promotion using them (ex dockerhub or docker registry)

**Most Important – During the Build process oc start-build myapp, Image is created using docker file. This image is used to create the application. So oc new-app image directly create the containerised application without creating any image.**

# Build Promotion using Jenkins

The same workflow which we described above can be put between below 2 jobs i.e: Deploy to Dev and Deploy to QA

Graphical user interface, application

Description automatically generated

Above Jenkins Build promotion scripts can be found at below github location

Build Promotion Scripts

<https://github.com/VeerMuchandi/jenkinsose3>

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# Templates

Template is packaging full running openshift application in a single object and present in json or yaml format.

# To upload a template to your current project’s template library, pass the JSON or YAML file with the following command:

$ oc create -f <filename>

You can upload a template to a different project using the -n option with the name of the project:

$ oc create -f <filename> -n <project>

To export and create our own .json or .yaml template file from a running Openshift application.

oc get dc/jenkins -o yaml --export > amit.yaml

This command will package the full application in amit.yaml file and save in the current directory.

This template file can be used to deploy this application in any openshift environment with the help of below command.

First create a library template with below command

oc create -f C:\Users\Amit\amit.yaml

Now create full application with this template

oc new-app jenkins --name="hellojenkins"

**To check the templates available in master openshift project**

oc get templates --namespace openshift

Now we can use any template to create an application like below

oc new-app openjdk18-web-basic-s2i --name="testapp"

ImageStreams

To get a list of the images available, use the command oc get imagestreams. The

--namespace openshift option should again be supplied to list those in the open

shift project

**Below command will list down all the templates and imagestreams together**

oc new-app -L

This produces a result similar to what would be available from the Browse Catalog page, combining application templates and builder images for both the current project and the open

shift project in the output.

Application deployment in OpenShift

Applications can be deployed to OpenShift in a number of different ways, using the

web console and the command-line oc client.

The main methods for deploying an application are:

• From an existing container image hosted on an image registry located outside the

OpenShift cluster.

• From an existing container image that has been imported into the image registry

running inside the OpenShift cluster.

• From application source code in a Git repository hosting service. The application

source code would be built into an image inside OpenShift, using an S2I builder.

• From image source code in a Git repository hosting service. The image source

code would be built into an image inside OpenShift using instructions provided

in a Dockerfile.

• From application source code pushed into OpenShift from a local filesystem

using the command-line oc client. The application source code would be built

into an image inside OpenShift using an S2I builder.

• From image source code pushed into OpenShift from a local filesystem using the

command-line oc client. The image source code would be built into an image

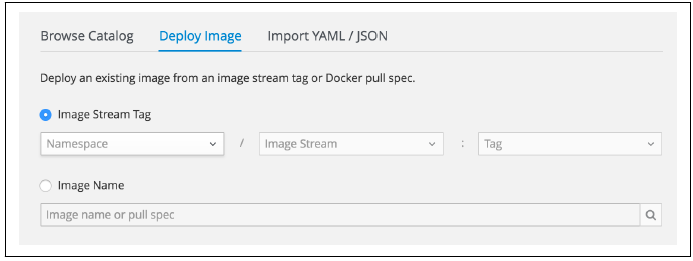
inside OpenShift using instructions provided in a Dockerfile.

**The openshift project acts as a global repository for builder images and templates. If**

**an administrator wants to make available a builder image or application template to**

**the whole OpenShift cluster, this is where they should add them.**

# Deploying an image



To use an image that resides in the OpenShift cluster, select Image Stream Tag and

then select the project that the image is owned by, the image, and the tag. You will be

able to see only projects that you are the owner of, other projects that you have been

explicitly granted access to, and the openshift project.

To use an image that is hosted on an image registry outside the OpenShift cluster,

select Image Name and enter the name of the image, including the hostname of the

image registry if using an image registry other than Docker Hub.

**There are two types of images we can deploy in openshift. 1st is third party images and second is Openshift images (imagestreams)**

**OC new-app host\_name/project/application –name testname ( here hostname could be docker hub or optum docker registry)**

Oc new-app *openshiftkatacoda/blog-django-py –name testname*

The full name of the image used is *docker.io/openshiftkatacoda/*

*blog-django-py*. When you leave off the hostname for the image

registry, OpenShift will default to first looking for the image on any

global image registries that a cluster admin has specified in the

cluster configuration. It is typical to have the Docker Hub image

registry included in that list. A company image registry or the Red

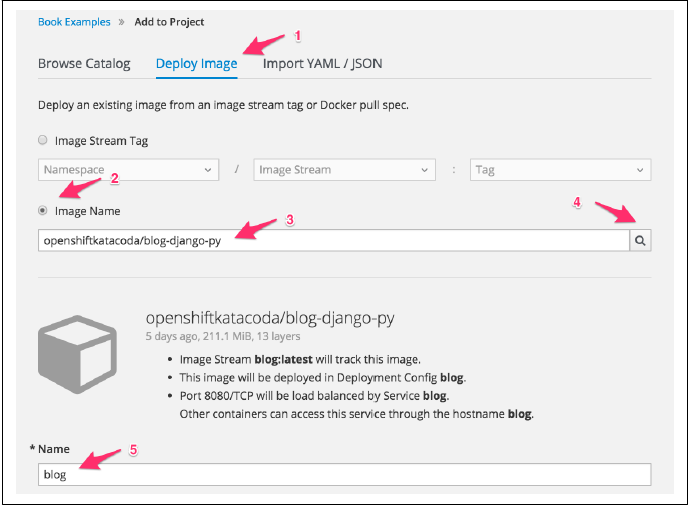
Hat Container Registry might also be included.

**Scaling up the application**

**$ oc scale --replicas=3 dc/blog**

deploymentconfig "blog" scaled

# Deploying third party images using web console



# ImageStreams

When you deploy an application from an existing container image hosted on an

external image registry, a copy of the image is downloaded and stored into an image

registry internal to OpenShift. The image is then copied from there to each node in a

cluster where the application is run.

In order to track the image that has been downloaded, an image stream definition is

created. To see the list of the image stream definitions, run oc get is:

**$ oc get is**

NAME DOCKER REPO TAGS UPDATED

blog 172.30.118.67:5000/book/blog latest About a minute ago

The *<IP>:<PORT>* shown under DOCKER REPO is the address of the internal image registry.

Because the image is being used as part of an application deployment, it is labeled

with the app label for the application. If you delete the application using the label, this

will also delete the image stream and image.

If you need to deploy multiple separate applications from one image, you should

import the image into OpenShift first using oc import-image:

**$ oc import-image openshiftkatacoda/blog-django-py --confirm**

The import completed successfully.

Name: blog-django-py

...

You can then deploy the applications from the imported image:

$ oc new-app blog-django-py --name blog

# OpenShift (Image)Build Strategies

OpenShift provides four different build strategies:

*Source*

This uses Source-to-Image to produce ready-to-run images by injecting application

source (or other assets) into a builder image.

*Docker*

This uses docker build to take a Dockerfile and associated source files and create

a runnable image.

*Pipeline*

This uses Jenkins and a workflow defined by a *Jenkinsfile* to create a pipeline for

building a runnable image.

*Custom*

This uses your own custom image to control the build process for creating the

runnable image.

**Deploying from Source to Image**

OpenShift provides S2I builders for common programming languages including Java,

NodeJS, Perl, PHP, Python, and Ruby. The builders will take your application source

code, compile it if necessary, and integrate it with the application server stack provided

with the builder image. When the container is run, the server will be started

and your application code run.

**Below Sign ~ denotes that S2I is used to deploy the application**

To deploy from application source code, example is below

oc new-app python:3.5~https://github.com/openshift-katacoda/blog-django-py --name sourcetoimage

Here above python:3.5 is the S2I builder

**Difference between new-app and new-build**

When the Source build strategy is invoked by oc new-app, it sets up two steps. The

first step is to run the build using S2I, combining the source files with the builder

image to create the runnable image. The second step is to deploy the runnable image

and start up the web application

You can perform the build step separately by running the oc new-build command

instead of the oc new-app command:

**$ oc new-build --name blog \**

**python:3.5~https://github.com/openshift-katacoda/blog-django-py**

**The** below above command will create the imagestream and build config only like below

--> Creating resources with label build=blog ...

imagestream "blog" created

buildconfig "blog" created

--> Success

So if we want to deploy the same app , then we just need to run OC new-app blog

**Triggering a new Build**

In the event that the source files used as input to the Source build strategy have

changed, a new build can be triggered using the oc start-build command:

**$ oc get bc**

NAME TYPE FROM LATEST

blog Source Git 1

**$ oc start-build bc/blog**

build "blog-2" started

Even though you created the build separately from setting up the deployment, when

the build has completed and the imagestream updated, a redeployment will be automatically

triggered. This occurs because oc new-app automatically sets up an image

change trigger in the deployment configuration.

# Deploying from docker file

Having a Dockerfile like the below format is must.

FROM openshift/python:3.5

USER root

RUN yum install -y wget

USER 1001

Below command is used to create the image from docker file

*docker build -t myimage .*

or we can directly refer the git repo location where Dockerfile is there

*docker build -t testimage* [*https://github.com/amit-gitgub/blog-django-py.git*](https://github.com/amit-gitgub/blog-django-py.git)

Once we have the docker image from above command, we can then login to any docker registry repo and push this image to there

Now from Openshift, we need to run the below command to run that image and create application in Openshift

*oc new-app dockeramit/private\_dtr\_amit --name tester*

# Configuration, Configmaps and Secrets

To set the environment variables all in one command.

**$ oc set env dc/blog BLOG\_BANNER\_COLOR=blue BLOG\_SITE\_NAME="My Blog"**

It is recommended to set multiple env variable in one command as it triggers the deployment. Setting it in multiple command will trigger un-necessary deployments.

# Configuration Files (Config maps)

Environment variables are the easiest mechanism to use for injecting configuration

information into a container. Configuration passed using environment variables is,

though, restricted to being in the form of simple key/value pairs. This method is not

well suited for passing more complex structured data to applications, such as JSON,

YAML, XML, or INI-formatted configuration files.

For working with more complex data, OpenShift provides the configmap resource

type. This also provides the ability to store keyed data, but the data values can be

more complex.

To create a config map you can use oc create configmap, or oc create with a

JSON/YAML resource definition for the config map.

If you only need to store simple key/value pairs, you can create the config map by

running oc create configmap and passing the --from-literal option along with

the names and values for the settings:

oc create configmap blog-settings \

--from-literal BLOG\_BANNER\_COLOR=blue \

--from-literal BLOG\_SITE\_NAME="My Blog"

When a config map is created, it is not associated with any application. To pass the

settings in this config map as environment variables in a deployment configuration,

you need to run the extra step of:

**$ oc set env dc/blog --from configmap/blog-settings**

To create the config map, instead of using --from-literal, use --from-file:

**$ oc create configmap blog-settings-file --from-file blog.json**

# Secrets

Secrets are also like config maps. The only deference is that they provide more security hence passwords, certs are saved in secrets.

To create a secret using Username and password

**$ oc create secret generic blog-secrets \**

**--from-literal DATABASE\_USERNAME=user145c30ca \**

**--from-literal DATABASE\_PASSWORD=EbAYDR1sJsvW**

Secrets uses base64 encoding. Hence once stored, they cannot be read.

To pass/attach this secret to some deployment config

**$ oc set env dc/blog --from secret/blog-secrets**

To Create Secret using a file

**$ oc create secret generic blog-webdav-users**

**--from-file .htdigest=webdav.htdigest**

To mount the secret, use oc set volume, using the --secret-name option to identify

the secret to use:

**$ oc set volume dc/blog --add --secret-name blog-webdav-users \**

**--mount-path=/opt/app-root/secrets/webdav**

# Delete Configuration and Secrets

Since Secrets and Configurations are created standalone and not with the deployment dc, hence we need to somehow attach them with some label.

**$ oc label secrets/blog-secrets app=blog**

Once they are associated with some labels, we can further delete them like below

**$ oc delete all,configmap,secret --selector app=blog**

# PODs and Containers

PODS can have multiple containers and all the containers will share the same IP address of POD.

In Openshift, each POD will have distinct IP. From any application running in the same project,

you can connect to another pod using its IP address on the port the application that is

running in that pod is using.

But Since IP addresses of PODS are not permanent (as it changes as pod restarted), Openshift provide the Service layer.

To see a list of the pod IP addresses associated with a service, you can use the oc get

endpoints command:

PS C:\Users\Amit> oc get endpoints blog

NAME ENDPOINTS AGE

blog 10.131.22.159:8080,10.131.23.143:8080 1d

This means 2 pods(as there are 2 IPs) are associated with this deployment and service layer is sending traffic to them.

Note - Always use selector with labels to find any information on some resource like below

**$ oc get pods,services --selector app=blog**

# Persistent Storage

OpenShift supports a number of underlying storage technologies including NFS,

GlusterFS, Ceph RBD, OpenStack Cinder, AWS Elastic Block Storage, GCE Persistent

Storage, Azure Disk, Azure File, iSCSI, Fibre Channel, and VMware vSphere

**Types of Persistent Storage**

Access modes for persistent storage are:

ReadWriteOnce *(*RWO*)*

The volume can be mounted as read/write by a single node. **This is by default Access mode in Openshift**– (*you cannot*

*use it with a scaled application and you will not be able to use rolling deployments.*

*This is because a persistent volume supporting only that access mode can be mounted*

*against only one node in the OpenShift cluster at any one time. When you scale an*

*application, you are not guaranteed that all instances will run on the same node. In*

*the case of a rolling deployment, even if the replica count is set to one instance, a new*

*instance of the application will be started before the existing one is shut down. This*

*presents the same problem as when an application is scaled up.*)

ReadOnlyMany *(*ROX*)*

The volume can be mounted as read-only by many nodes.

ReadWriteMany *(*RWX*)*

The volume can be mounted as read/write by many nodes.

It is possible that persistent storage available in an OpenShift cluster will not support

all access modes. If the only supported access mode is ReadWriteOnce, this will limit

how you can use persistent storage.

To Claim PV below is the command

**$ oc set volume dc/blog --add \**

**--type=pvc --claim-size=1Gi --claim-mode=ReadWriteOnce \**

**--claim-name blog-data --name data --mount-path** /opt/app-root/src/media

persistentvolumeclaims/blog-data

deploymentconfig "blog" updated

When a persistent volume is added to a deployment configuration using

oc set volume, the application will be automatically redeployed.

To list down the PV added to deployment like below

**$ oc set volume dc/blog**

**Unmounting a PV**

To stop using a persistent volume with an application, you can use the oc set

volume --remove command. You must supply the name used to identify the volume

mount in the deployment configuration:

**$ oc set volume dc/blog --remove --name data**

If the access mode of a persistent volume is ReadWriteMany or ReadOnlyMany, you can

safely mount that persistent volume against multiple applications at the same time.

This will allow you to use a single persistent volume to share data between the applications.

Delete a Persistent Volume

**$ oc delete pvc/blog-data**

In case of accidental deletion of PV. Check with cluster admin as if they have set the reclaim policy to Retain at cluster level , we can retrieve the data otherwise not.

**Copying data to Volume**

If you have your application running and a persistent volume mounted, you can copy

a directory from your local system into the persistent volume using oc rsync.

First determine the name of the pod for your application that mounts the persistent

volume:

**$ oc get pods --selector app=blog**

NAME READY STATUS RESTARTS AGE

blog-1-5m3q6 1/1 Running 0 2m

You can then run oc rsync to copy the directory.

$ **oc rsync /tmp/images blog-1-5m3q6:/opt/app-root/src/media --no-perms**

The --no-perms option tells oc rsync not to attempt to preserve permissions on

directories and files. This is necessary, when copying files to the local container filesystem,

and the directory into which files are being copied is not owned by the user

ID the container is running as, but rather by the user that the S2I builder was run as.

Without this option, oc rsync would fail when it attempts to change the permissions

on the directory.

The oc rsync command can also be used to copy directories or files from a running

container back to the local system. Copying in either direction can be run as a one-off

event, or you can have oc rsync continually monitor for changes and copy files each

time they are changed.

# Readiness and Liveness Probes

The readiness probe is used to determine whether your application is in a state where

it is okay for other applications or external users to communicate with it. The liveness

probe is used to determine whether your application is still running correctly

When the readiness probe succeeds for the new pod, the

IP address for the pod will be added to the list of endpoints associated with the service.

If no readiness probe is provided, the pod will be assumed to always be ready, with

the IP address being added to the list of endpoints associated with the service as soon

as the pod is started and only removed if the pod is shut down.

It is recommended that a readiness probe always be used if using the Rolling deployment

strategy. This is because it ensures that a new pod will have requests directed to

it only when it’s ready, ensuring that you have zero downtime when deploying a new

version of your application.

When you provide a liveness probe, it will be used to periodically check whether the

instance of your application running in a pod is still running and whether it is also

working correctly.

If the probe keeps failing, the pod will be shut down, with a new pod started up to

replace it.

**Using HTTP method to set probes**

**$ oc set probe dc/blog --readiness --get-url=http://:8080/healthz/ready**

When specifying the URL, leave out the hostname part. The hostname will be automatically

filled in with the IP address of the pod that the probe is being used to check.

The port number must be specified and should be the port the application running in

the pod uses to accept connections for HTTP requests.

Although you could use an existing URL that a web application handles, it is recommended

that you create dedicated handlers for each type of probe. This way you can

tailor each handler to implement checks specific to the type of probe.

To remove the probes, you can use the --remove option to oc set probe:

**$ oc set probe dc/blog --readiness --liveness –remove**

Setting of an initial delay would be required in cases where during startup an application,

although it may accept new connections for requests, might not be ready to start

handling live requests immediately and will return an HTTP error response until it is

ready.

For example, to set the delay, you can use the --initial-delay-seconds option:

**$ oc set probe dc/blog --readiness --get-url=http://:8080/healthz/ready \**

**--initial-delay-seconds 10**

# Useful commands

To list down or to delete all the resources in an application (just the name)

PS C:\Users\Amit> oc get all --selector app=insults -o name

pod/insults-9-k962c

replicationcontroller/insults-4

replicationcontroller/insults-5

replicationcontroller/insults-6

replicationcontroller/insults-7

replicationcontroller/insults-8

replicationcontroller/insults-9

service/insults

deploymentconfig.apps.openshift.io/insults

buildconfig.build.openshift.io/insults

build.build.openshift.io/insults-4

build.build.openshift.io/insults-5

build.build.openshift.io/insults-6

# Deployment rollback

oc rollback insults --to-version=10

#12 rolled back to insults-10

Warning: the following images triggers were disabled: insults:latest

You can re-enable them with: oc deploy insults --enable-triggers -n insultapp

# With the help of describe command, you can see more details about resource

PS C:\Users\Amit> oc describe pod/insults-9-k962c

Name: insults-9-k962c

Namespace: devproject

Priority: 0

PriorityClassName: <none>

Node: ip-172-31-53-117.us-west-2.compute.internal/172.31.53.117

Start Time: Mon, 25 Feb 2019 22:12:19 +0530

Labels: app=insults

deployment=insults-9

deploymentconfig=insults

# Now above labels can be further used like below

PS C:\Users\Amit> oc get pod --selector app=insults

NAME READY STATUS RESTARTS AGE

insults-9-k962c 1/1 Running 0 1h

PS C:\Users\Amit> oc get pod --selector deployment=insults-9

NAME READY STATUS RESTARTS AGE

insults-9-k962c 1/1 Running 0 1h

PS C:\Users\Amit> oc get pod --selector deploymentconfig=insults

NAME READY STATUS RESTARTS AGE

insults-9-k962c 1/1 Running 0 1h

PS C:\Users\Amit>

Container Logs

When an error occurs within your application, generally the first place you want to

look is in the log files. In OpenShift, there are two types of log files: the build logs and

the runtime application logs.

Environment Variables

To see a list of what environment variables will be exported to a container when run

by OpenShift you can use the oc set env command with the --list option to query

the deployment configuration for your application.

$ oc set env dc/helloworld --list

# deploymentconfigs helloworld, container helloworld

PATH=/opt/app-root/src/bin:/opt/app-root/bin:/usr/local/sbin:/usr/local/bin:...

STI\_SCRIPTS\_URL=image:///usr/libexec/s2i

STI\_SCRIPTS\_PATH=/usr/libexec/s2i

HOME=/opt/app-root/src

BASH\_ENV=/opt/app-root/etc/scl\_enable

# To set an environment variable

oc set env dc/helloworld MYSQL\_DATABASE=mysql

If you want to update the value of more than one environment variable at the same

time, each can be listed on the same command line separated by a space.

If you want to change more than one environment variable, then they should all be

updated with the one command. This is because the configuration change will by

default trigger a redeployment of your application with the new environment variables.

Setting the environment variables one at a time would thus cause successive

redeployments.

You set required environment variables by using the --env option when running the

oc new-app command:

**$ oc new-app openshiftkatacoda/blog-django-py --name blog \**

**--env BLOG\_BANNER\_COLOR=green**

Optional environment variables can be set later by running the oc set env command

against the deployment configuration:

**$ oc set env dc/blog BLOG\_BANNER\_COLOR=green**

deploymentconfig "blog" updated

When the environment variables are updated using oc set env, the application will

be redeployed automatically with the new configuration. If you want to see what environment

variables will be set in the container, you can use oc set env with the --

list option:

**$ oc set env dc/blog --list**

**#** deploymentconfigs blog, container blog

BLOG\_BANNER\_COLOR=green

# OC debug command

oc debug command will start up an instance of the image for your application, but instead of running the normal application command, it will run an interactive shell instead.

$ oc debug dc/helloworld

Debugging with pod/debug-helloworld, original command: <image entrypoint>

Waiting for pod to start ...

Hit enter for command prompt

# If application is failed to start

When your application fails to start up and exits, OpenShift will attempt to restart the

application. If this keeps occurring, then the pod will be moved into a state called

*CrashLoopBackOff*. You will see that a pod has entered this state when you use oc get

pods to get a listing of the pods for your application:

helloworld-5-cqlo4 0/1 CrashLoopBackOff 2 1m

If you run oc logs on this pod, you will usually see a less than useful message.

$ oc logs helloworld-5-cqlo4

Error from server: Internal error occurred: Pod "helloworld-5-cqlo4" in

namespace "wfproject": container "helloworld" is in waiting state.

This is a final error message from OpenShift resulting from the successive failures to

start your application. This is not the log messages from your application.

In this case, to see the logs from the last attempt to start your application, you should

use the --previous option with the oc logs command.

# Token

Token is used to login in openshift ( to make sure session gets time out for security reasons)

If you are already logged in, then find token using below command.

oc whoami -t