Application development with Docker, Kubernetes, and OpenShift

Steve Pousty

Slides Used in Class

Both the slides and these workshop notes are released under Creative Commons License allowing free use as long you provide attribution.

Installing the Vagrant Box

Let's go ahead and install the Vagrant box file and get everything up and running.

Hardware Requirements

1. 10 gigs of disk to start but may need up to 30
2. At least 8 gigs of RAM by default. Our VM will use 4 by default so we want to leave 4 gigs for the OS and others

Software Requirements

WARNING	You must do this **BEFORE** the class - we will not have time to do this in class.

1. Install VirtualBox. Vagrant doesn't work with the 5.1 versions
2. Install Vagrant. There is a nasty SSH bug in the latest so we need to go back one

VirtualBox is a requirement. I know some of you have parallels and vmware and other virtualization tech but I only built the image for VirtualBox. My reasoning for doing this is that it is the only FOSS VM software out there. Sorry but that's the breaks.

WARNING	If you are using the new Docker for Windows, it requires HyperV which prevents VirtualBox from seeing the VT-X extension. It is a real hassle to switch back and forth. Everytime you want to go to VirtualBox you basically need to "repair it" from applications. So turning on HyperV or VirtualBox will require at least one reboot. I found instructions on how do avoid the whole repair step in this post by Scott Hanselman and while I did not try this myself I trust him.

Account requirements

WARNING	You must do this **BEFORE** the class - we will not have time to do this in class.

1. GitHub account, if you don't have one

Getting the Vagrant box on your local machine

You can use any directory you want but for the instructions I will make one titled *os_workshop*. ---- ==== Using the box on Atlas

If you are doing this at home (which I hope you did before the class) do the following commands

```
C:\_os_workshop> vagrant init thesteve0/imagemakers
C:\_os_workshop> vagrant up --provider=virtualbox
```

There is a 2.6 Gig file that needs to be downloaded so it may take a while. After that is download Vagrant will bring everything up.

Go ahead and download the OpenShift client tools for your laptop OS. There will be a single binary in the archive - for convenience you should put it somewhere in your path. But it is also fine to place it somewhere else, you will just need to prefix you calls with the path to binary.

https://github.com/openshift/origin/releases/tag/v1.3.0-alpha.3

Manually copying the box off of a USB drive

If you are doing this in the class we are going to manually bring the box to your machine.

WARNING	You need to have access to your USB drive to choose this method to bring up your Vagrant box

Please try to come 15 minutes early to class so we can try to do the copying and installing before the class starts.

Copy the *Vagrantfile* and *uberconf-openshift-origin.box* file from the USB stick into the *vagrant* directory created above.

Then in that directory do the following commands:

```
C:\_os_workshop> vagrant box add --name imagemakers <sd card>\imagemakers.box
C:\_os_workshop> vagrant init imagemakers
C:\_os_workshop> vagrant up --provider=virtualbox
```

Final step

When "vagrant up" is done you should see a message that ends with:

```
==> default: Successfully started and provisioned VM with 2 cores and 4 G of memory.
==> default: To modify the number of cores and/or available memory modify your local
Vagrantfile
==> default:
==> default: You can now access the OpenShift console on:
https://10.2.2.2:8443/console
==> default:
==> default: Configured users are (<username>/<password>):
==> default: admin/admin
==> default: But, you can also use any username and password combination you would
like to create
==> default: a new user.
==> default:
==> default: You can find links to the client libraries here:
https://www.openshift.org/vm
==> default: If you have the oc client library on your host, you can also login from
your host.
==> default:
==> default: To use OpenShift CLI, run:
==> default: $ vagrant ssh
==> default: $ oc login https://10.2.2.2:8443
```

If this doesn't happen raise your hand or, if you are a good student and did this at home, email me. If you see this then you have installed everything properly.

I know some of you are going to be curious and are going to start playing with this VM. That's OK!

Right before class (or if you break it) you just need to do the following steps:

```
C:\_os_workshop\vagrant>vagrant destroy --force
C:\_os_workshop\vagrant>rm -rf .vagrant #basically delete the .vagrant dir - I do it
bc I am superstitious
C:\_os_workshop\vagrant>vagrant up --provider=virtualbox
```

You don't even need to be online for this command to work! Welcome to the future.

Brief Introduction to OpenShift

OpenShift is Red Hat's Container Application Platform. What this means in plain English is OpenShift helps your run Docker containers in an orchestrated and efficient way on a fleet of machines. OpenShift will handle all the networking, scheduling, services, and all the pieces that make a up a modern application.

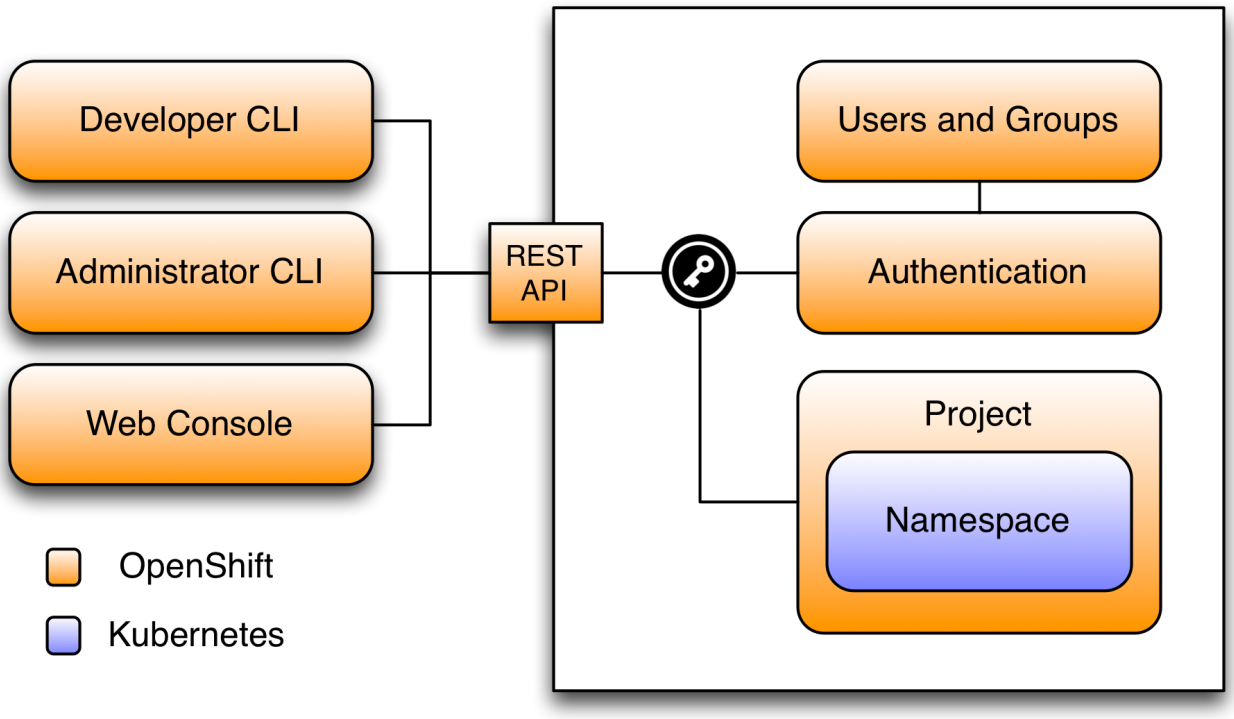
To accomplish this there are required pieces to the platform. We use the Docker container engine to handle just the running of the containers. Above that there is Kubernetes, the Open Source project for orchestrating, scheduling, and other tasks for actually building real applications out of Docker containers. Finally, we layer on top of this the engineering work of OpenShift to build a developer and system adminstrator experience that allows for ease of use when developer and administering containerized applications.

For both Docker and Kubernetes, OpenShift does not do any forking or proprietary extensions. With an OpenShift cluster you can use the OpenShift interfaces OR you can always use the Docker and Kubernetes interfaces. Granted, those interfaces will only support operations possible in that part of the stack.

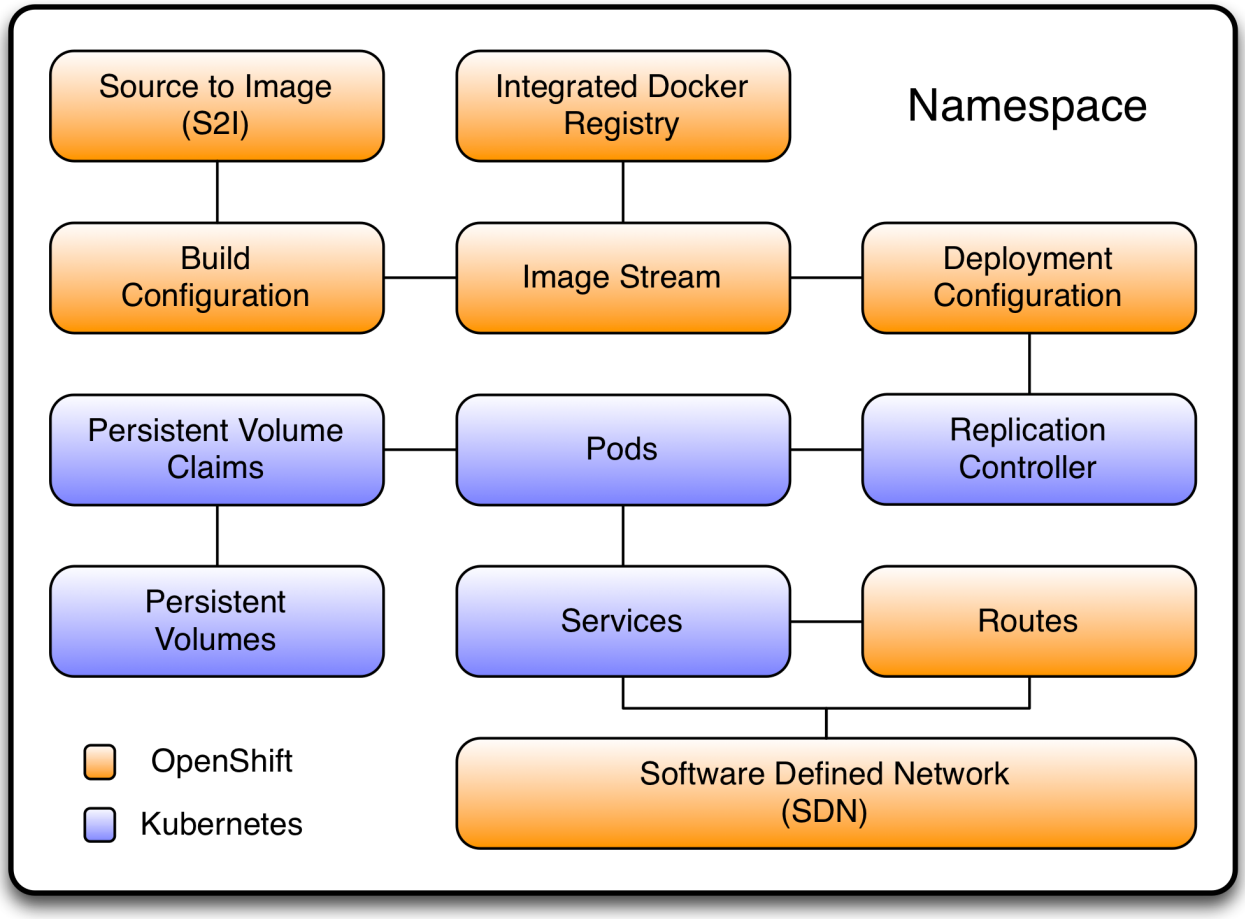
I am going to put two diagrams here that we will be discussing in class. You can refer back to these diagrams while you do the exercices. The key points for these diagrams are:

1. You can look and see how all the pieces fit together
2. You can see which pieces are core Kubernetes (blue) and which pieces are OpenShift (gold).

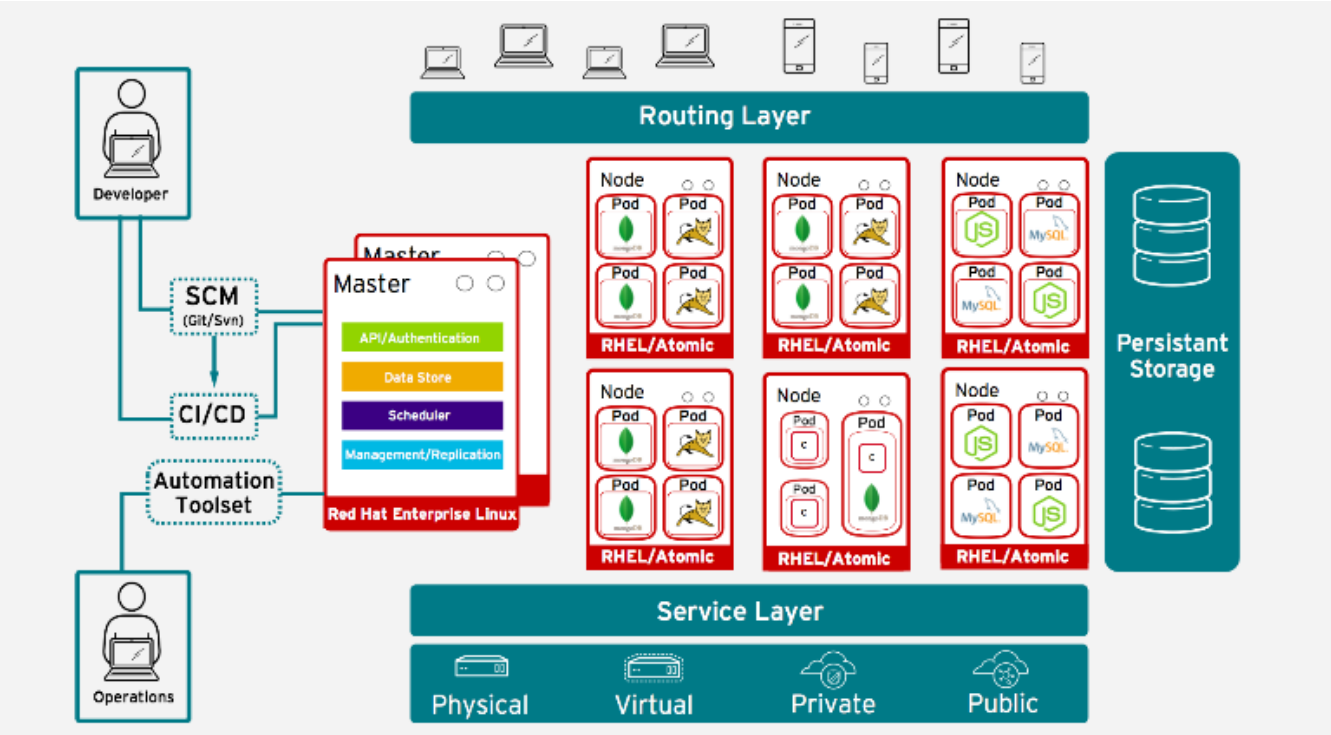
Here is the top level, showing the pieces in the interfaces and authentication:



And here are the lower level pieces that make up the core objects.



Finally, here is a diagram showing how an OpenShift/Kubernetes cluster is put together from different data center "pieces"



Now that we have done enough talking - the rest of the class will be action!

Introduction to the OpenShift All-In-One VM

Before we get started, it is assumed that you have already installed the OpenShift All-In-One VM and command line tools.

The all-in-one VM (A1VM) is intended for a developer or sysadmin to have a quick and easy way to use OpenShift (which also is a great way to use Kubernetes). The focus of the A1VM is:

1. Works right out of the box
2. Relaxes security restrictions so you can run Docker images that run as root
3. Give developers a means to carry out rapid development without needing an external server
4. Give the OpenShift evangelists an easy way to run workshops - like this one

WARNING

We wanted to allow developers to use any Docker image they want, which required us turning off some security in OpenShift. By default, OpenShift will not allow a container to run as root or even a non-random container assigned userid. Most Docker images in Dockerhub do not follow this best practice and instead run as root. Further multiplying this error, a large majority of Dockerhub images are not patched for well known vulnerabilities. Therefore, please use images from Dockerhub with caution. We think some of the risk is mitigated because you are running OpenShift in a VM, but still - be careful which Docker images you run.

TIP

If you want to run the VM with same permissions as in the Enterprise or Online products you can do the following command inside the Vagrant machine:

```
$ oc login #username admin #password any password you want
$ oc adm policy remove-scc-from-group anyuid system:authenticated
```

This commands removes the ability to for any OpenShift UID from running as system:authenticated and resets it to the default which is only allow that permission for random UID.

Your VM now behaves like Online or Enterprise in terms of not letting Docker images run as root user.

The command line

And with that we begin the journey. Our first step will be to login to the VM both from the command line (cli) and the web console (console). Open a terminal and type in:

```
> oc login https://10.2.2.2:8443

# on windows or if you get a cert. error do the following

> oc login https://10.2.2.2:8443 --insecure-skip-tls-verify=true
```

You should be prompted for a username and passsword. For ease of use, in the workshop, we will use username *user* password *password*. You can actually use any password you want with the admin or user account. You can also use any username and password you want. The username will "matter" in that any project created in that username can only be seen by that username - but these are not secure accounts. Remember - the focus of this VM is ease of use.

After logging in you should see the following message

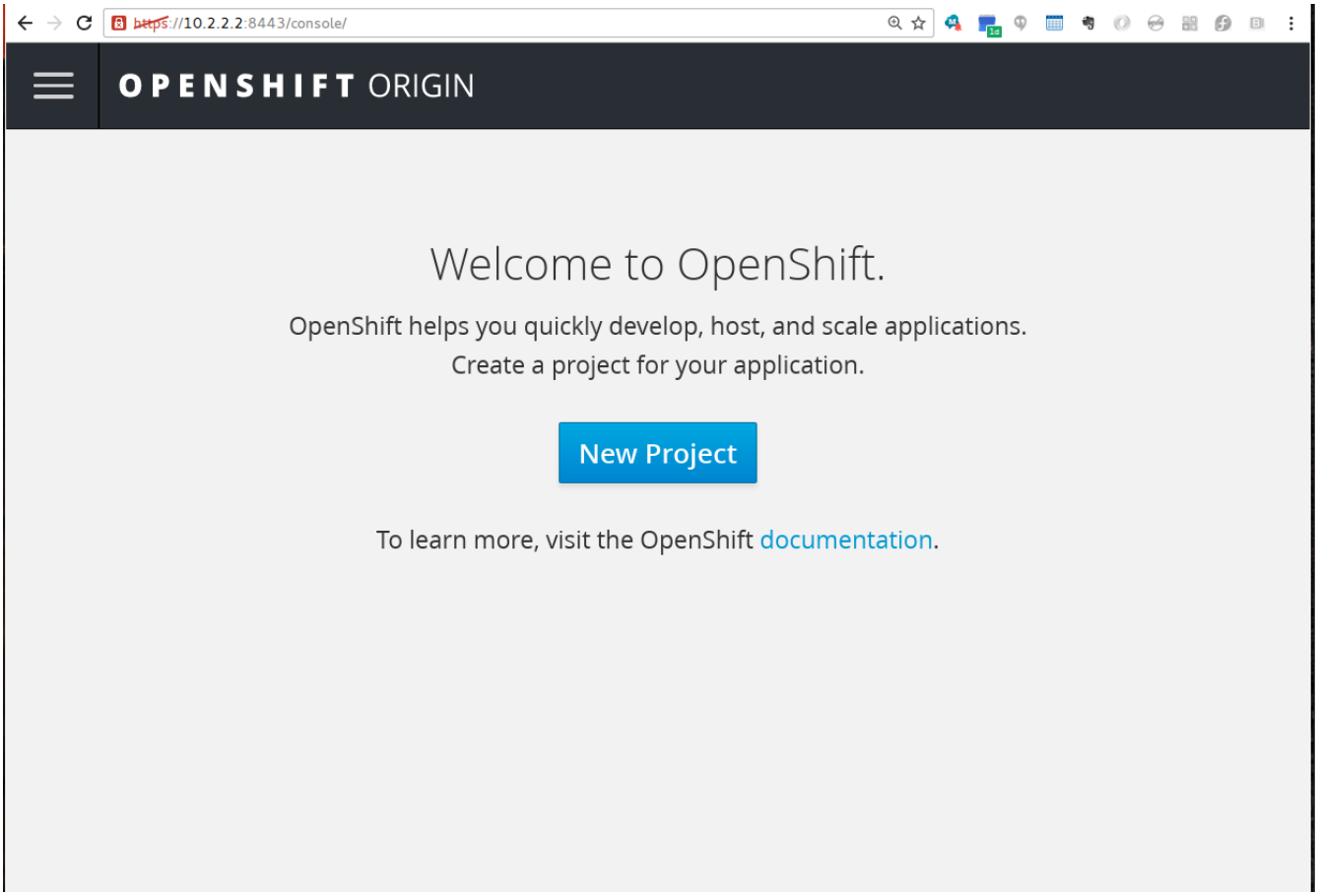
```
Login successful.

You don't have any projects. You can try to create a new project, by running

    oc new-project <projectname>
```

Web Console

Now in your browser go to https://10.2.2.2:8443 . You will get a warning about the certificate and this is to be expected since we are using self-signed certificates throughout the installation - so we will need to work around this. You will then get to the login screen. Again use the *user* and *password* username password combination and you should see something that looks like this (except for the red box):



Now go ahead and login as the account we will use for most of the remaining exercises. In the username and password prompt go ahead and type *user* and *user* in both fields (again the password can really be anything you want).

And with that we are ready to get down to business. Let's start by just using a plain Docker container image from Docker hub!

Running a Docker Container Image

Let's start by just running a plain ole' Apache HTTPD server in OpenShift. By way of doing this I will also introduce you to many of the major pieces in an OpenShift project.

We are going to move back and forth between the command line and the web UI so please have them both ready to go and logged in as the user.

Create a Project

In order to get going we need to create a *project* to hold all of the pieces of our application(s). A *project* is a materialized *namespace* in Kubernetes to provide permissions and access control between different users resources.

The first thing we need to do as a user is create the *project*

```
> oc new-project webpages
Now using project "webpages" on server "https://10.2.2.2:8443".
...
```

We created a *project* named webpages and all subsequent commands will be executed against this *project*.

There is a special *project* that admins have access to in the cluster named "openshift". Any templates (explained later) or other objects placed in this project are available to all users of the cluster with read permissions.

To see all the projects that you have permissions to see go ahead and:

```
> oc get projects
```

Bringing in Docker Container Image from DockerHub

Typically you will not bring in images directly from DockerHub because of the numerous flaws and vulnerabilities in DockerHub images. One big probem for many images on DockerHub is that they run as *root* user when it is not even required. By default, an OpenShift cluster will not let you run images as *root*. In the case of the A1VM we turn off this security to allow for ease of use.

Let's go ahead and start with an NGINX image put out by the CentOS group. Bringing this into our OpenShift environment is as simple as

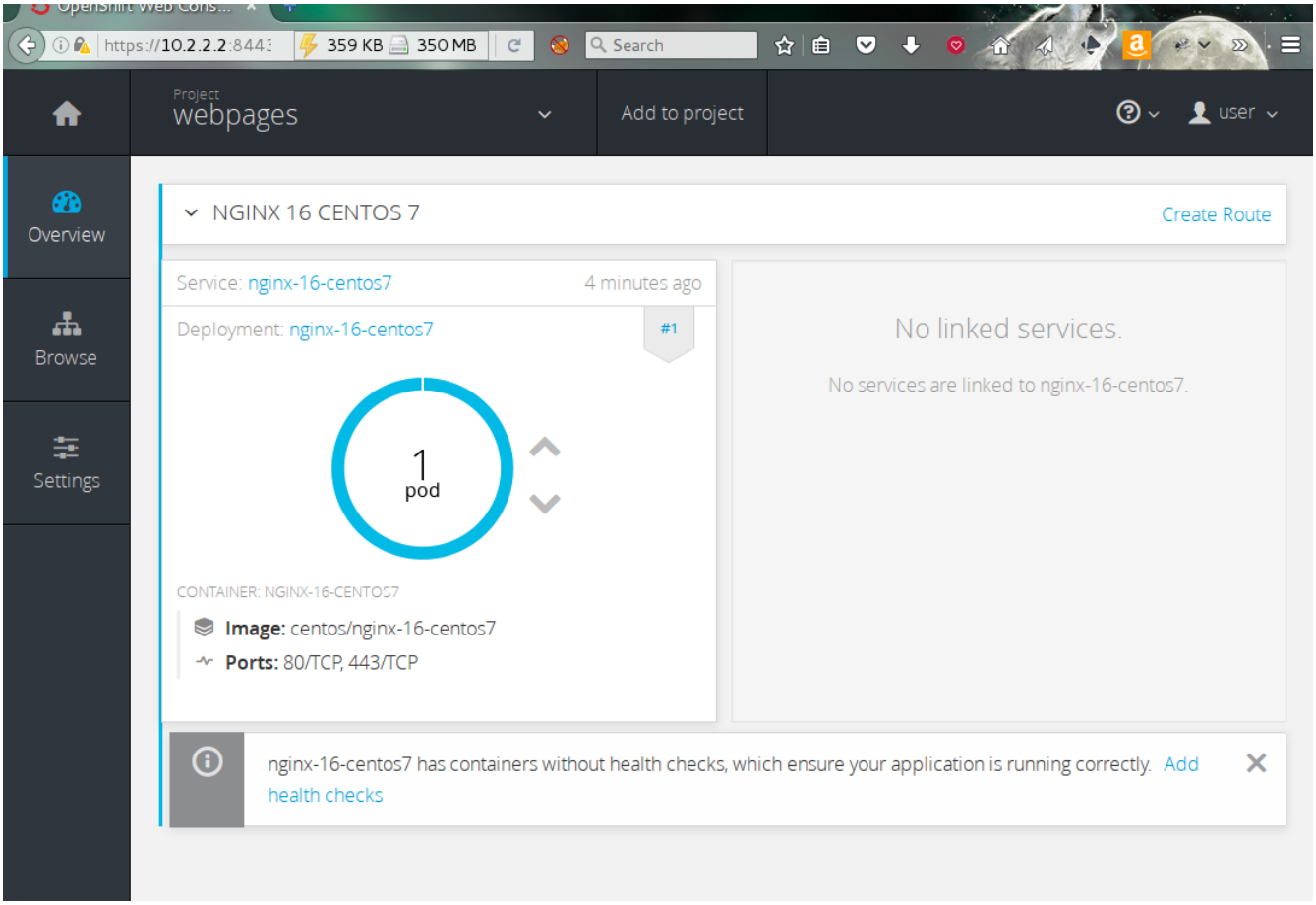
```
> oc new-app centos/nginx-16-centos7
--> Found Docker image 32eebae (13 days old) from Docker Hub for "centos/nginx-16-
centos7"

    * An image stream will be created as "nginx-16-centos7:latest" that will track
this image
    * This image will be deployed in deployment config "nginx-16-centos7"
    * Ports 443/tcp, 80/tcp will be load balanced by service "nginx-16-centos7"
      * Other containers can access this service through the hostname "nginx-16-
centos7"
    * This image declares volumes and will default to use non-persistent, host-local
storage.
      You can add persistent volumes later by running 'volume dc/nginx-16-centos7
--add ...'
    * WARNING: Image "centos/nginx-16-centos7" runs as the 'root' user which may not
be permitted by your cluster administrator

--> Creating resources with label app=nginx-16-centos7 ...
    imagestream "nginx-16-centos7" created
    deploymentconfig "nginx-16-centos7" created
    service "nginx-16-centos7" created
--> Success
    Run 'oc status' to view your app.
```

The command 'oc new-app' actually does a lot of things for you. The idea behind new-app is to "do the right thing" when given a non-OpenShift artifact, I like to call it 'oc translate'. So in this case we are telling OpenShift, "here is a docker image, do you best to make all the pieces needed to make this run as a user would expect in OpenShift.

If you go back to the Web UI and look in the webpages project you should see something like this (may take a while depending on the time it takes to download the Docker image):



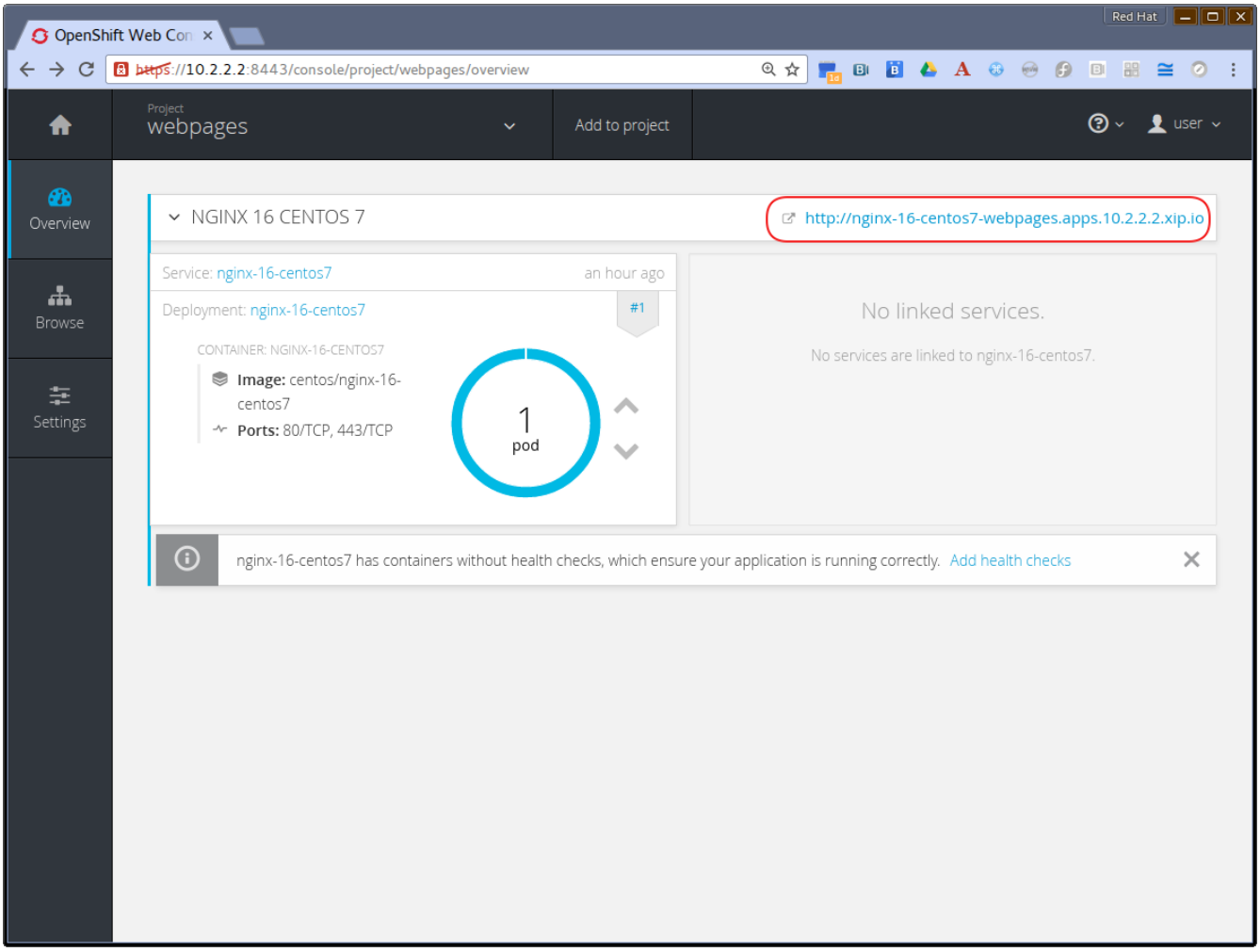
Now is a good time to reflect back on the the architecture diagram I explained earlier and think about the objects created above. Start from the pod and work our way up to *Services* and *Deployment Configurations*

Making a Route to our Service

It's all well and fine that we have this pod running and fronted by a service to do load balancing and proxying, but how do we actually expose this to the outside world? This is where the OpenShift Route can be used to expose our service with a URL.

In the Web UI, there is a link in the top right of NGINX 16 CENTOS7 titled "Create Route". Go ahead and click it and accepts all the defaults.

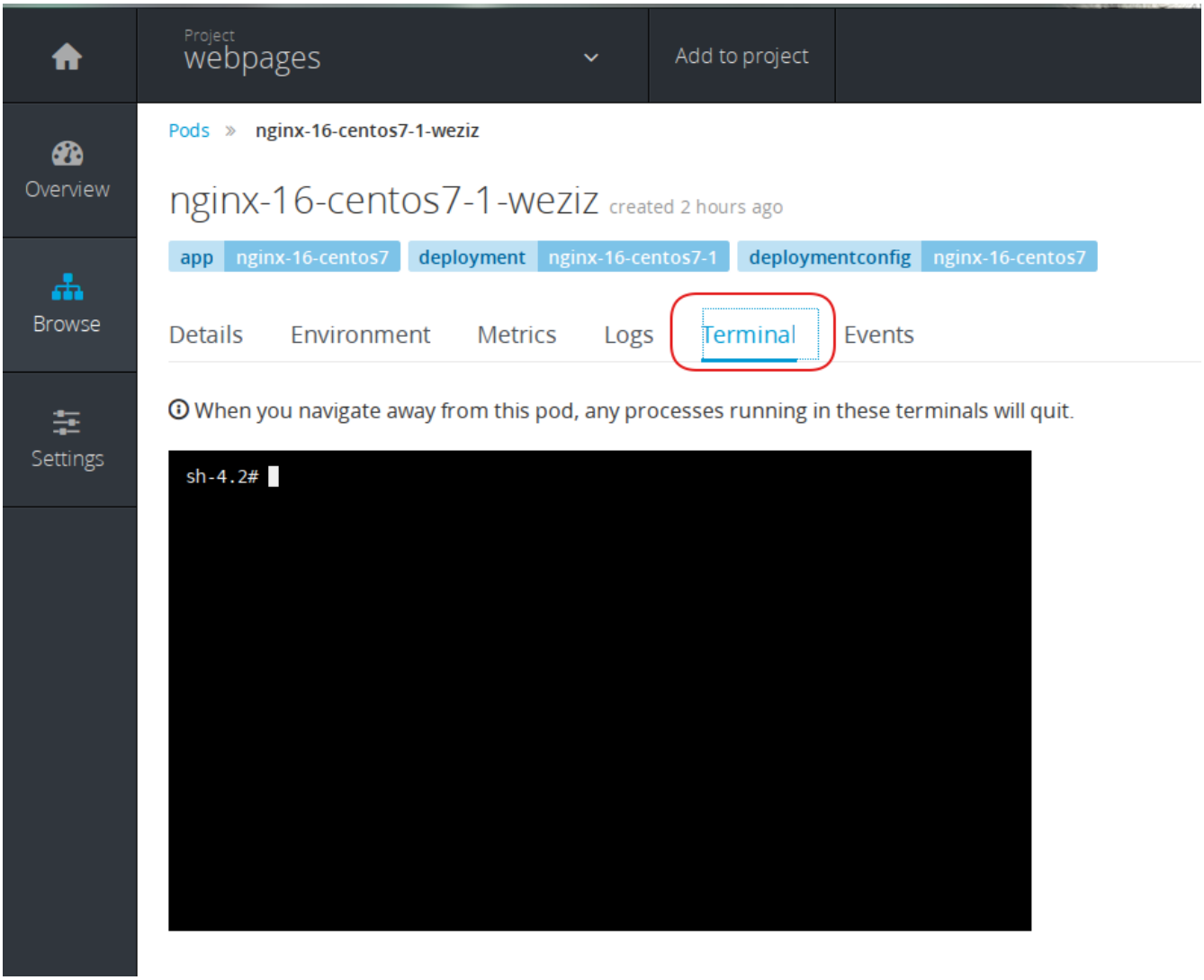
When you get back to the overview screen you will now have a URL up in the top of the box, go ahead and click (but be ready for sadness).



The resulting page gives you a 403 error message because there is no html content in the mapped volume. This docker container specified a mapped volume to hold its HTML content and we never populated it with files. For this exercise we will go in manually to the image and create some content.

Opening a Terminal Into the Pod

In the web UI click on the circle that says 1 pod, then click on the only pod in the list. This will bring you to the detailed description page for the pod. On that page please click on the we Terminal tab.



What you have in front of you is a shell inside the running container. For some reason you have to click twice inside the terminal to get a prompt where you can type. To create a HTML page that will be served up by NGINX just do the following command (you can use tab completion to fill in the elements on the path):

```
# echo  "hello world" > /opt/rh/nginx16/root/usr/share/nginx/html/index.html
```

Now if you go back and reload the URL you will see "hello world" in your browser. But since Docker containers are immutable and the path we mounted for storage was a host path, this page will go away if we deploy a new image or the image is destroyed.

You may have noticed on this page that there was a warning about "No health checks defined". In the next section we will discuss what these warnings mean and how to fix it.

Health Checks for Your Containers

Thanks to the health check mechanisms built into Kubernetes, OpenShift just inherits a powerful way to make sure you app is running and responding. These check are over and above a container dying, which just automatically gets redeployed by OpenShift. Health checks are for the case where the pod is up and running but it is not actually "working" the way you intended.

There are two basic types of health checks in OpenShift

1. A liveness check - is the pod running and responding. If it is not then the cluster will kill and respawn the pod
2. A readiness check - is the server in the pod ready to respond to requests. If it is not then the cluster will not route requests to it and not continue with the deployment

The readiness check is helpful because it can prevent requests being sent to the pod until the pod is actually ready to serve content. This can be a problem with servers that take longer to become ready to serve content, such as some of the larger JavaEE servers or with a large database cluster.

These checks can use HTTP responses, Socket Connections, or executing a command in the running container. With the HTTP check, the cluster expects a response code of 200 or 399 for a live or ready pod. For a socket connection, the cluster just needs to be able to make a connection on the given port.

You can read more about these checks in the OpenShift documentation.

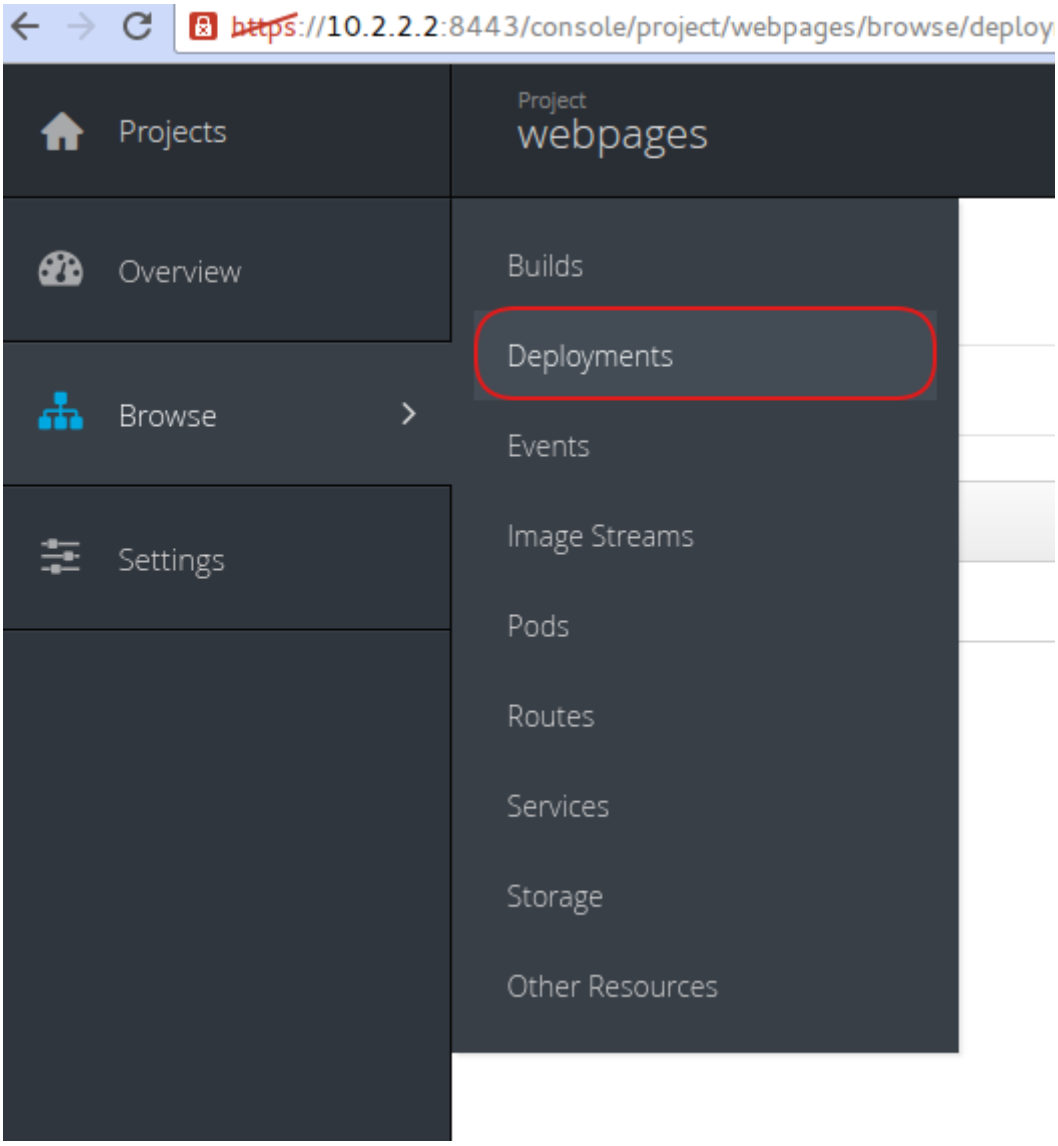
Let's go ahead and define each of these checks for our NGINX pod.

Liveness Check

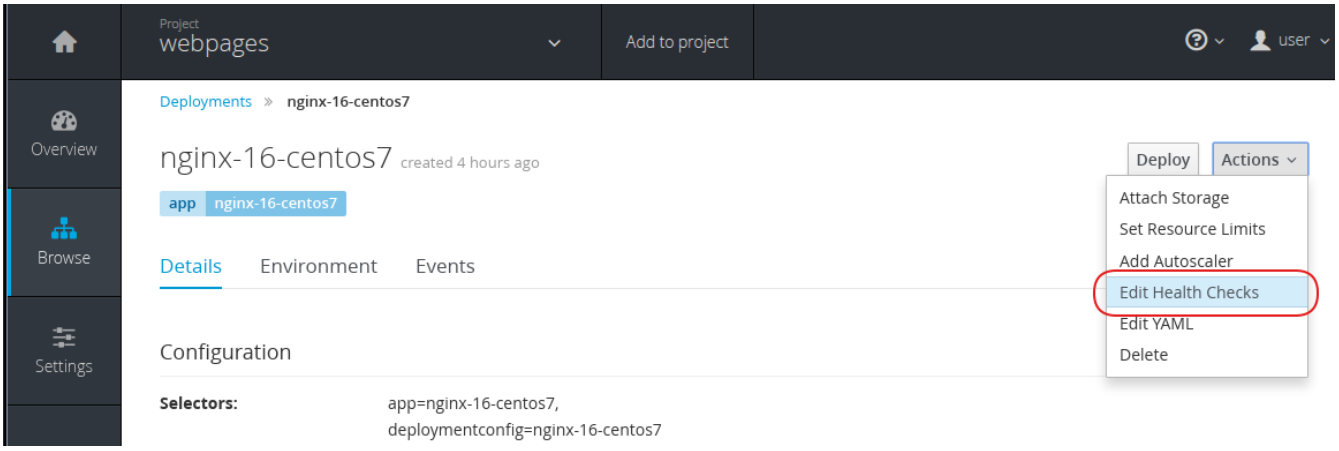
If there is already a good check built into your docker container, then defining the check can be quite easy. In our example we could consider if we get a 200 from the root URL then it is alive.

Let's try that first.

Go to the web UI, on the left side pick Browse, and then choose Deployments:

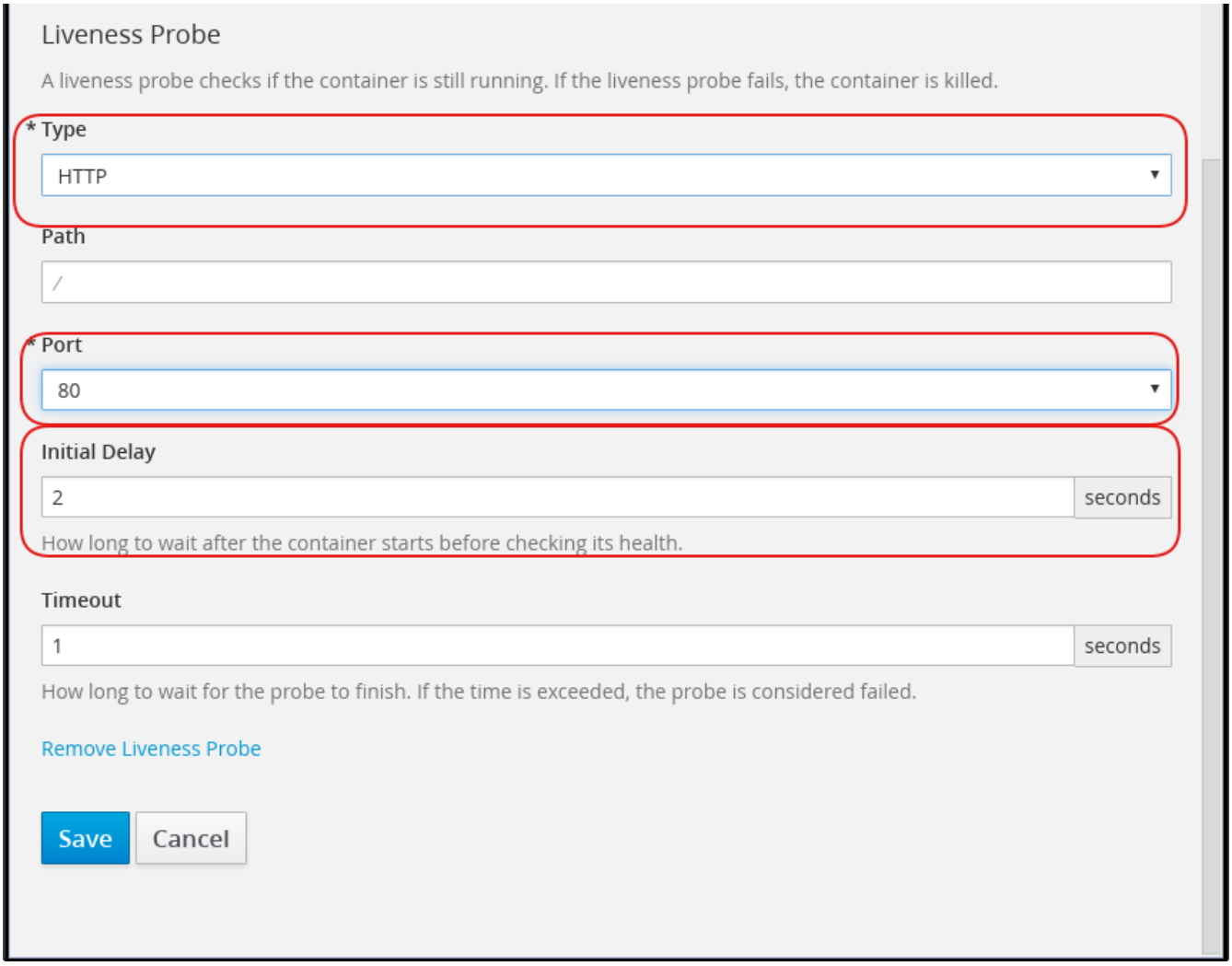


You will then see a list with one deployment, go ahead and select it. You are now looking at the details for your deployment. In the top left there is an actions button, go ahead and select it, and then choose "Edit Health Checks"



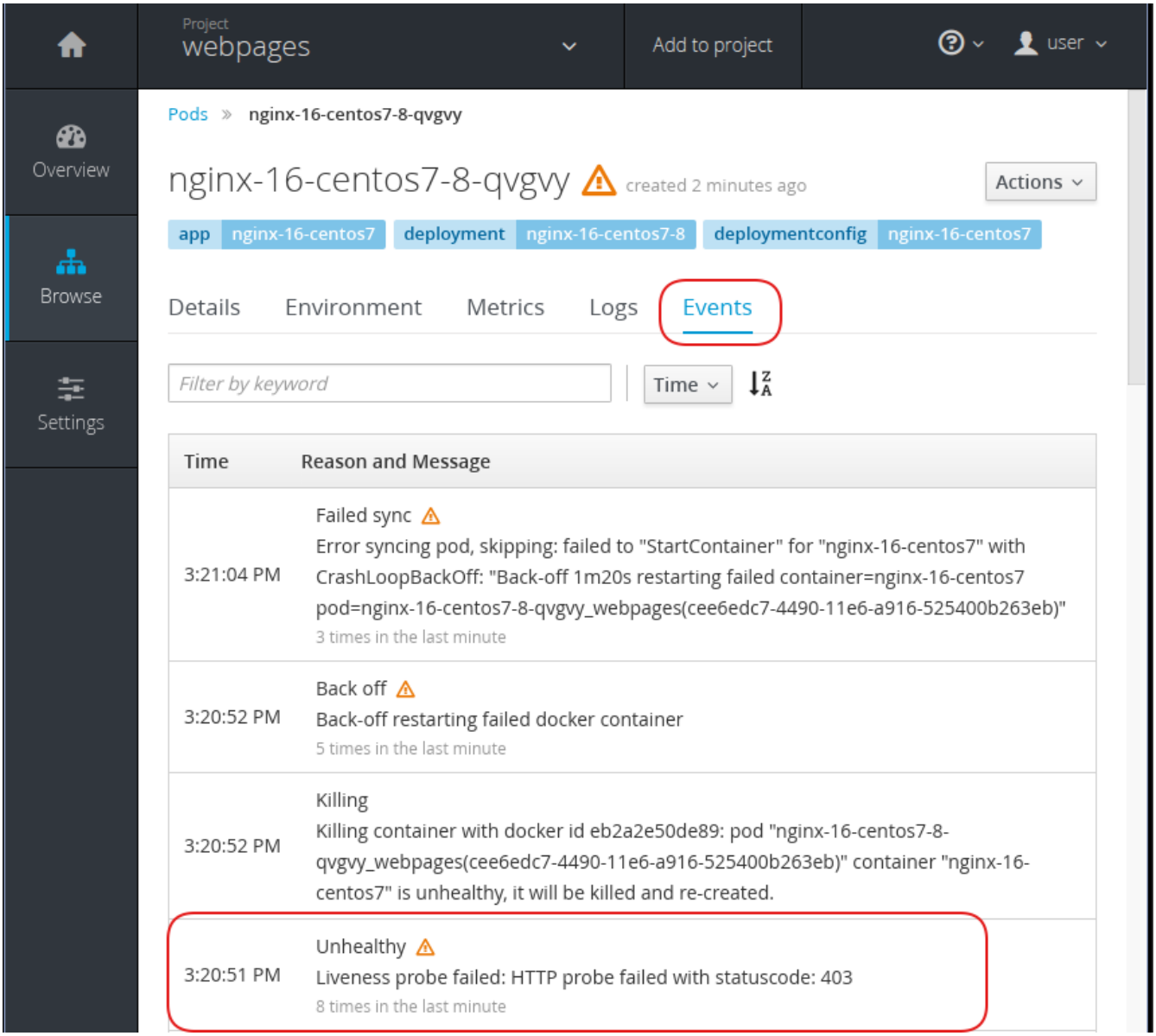
On the landing page you see a nice description for each of the probe types. Let's go ahead and click on the link for "Add Liveness Probe". This will add a form to the page that allows us to define a liveness probe.

Switch the port to 80 and the initial delay to 2 seconds. Then click "Save".

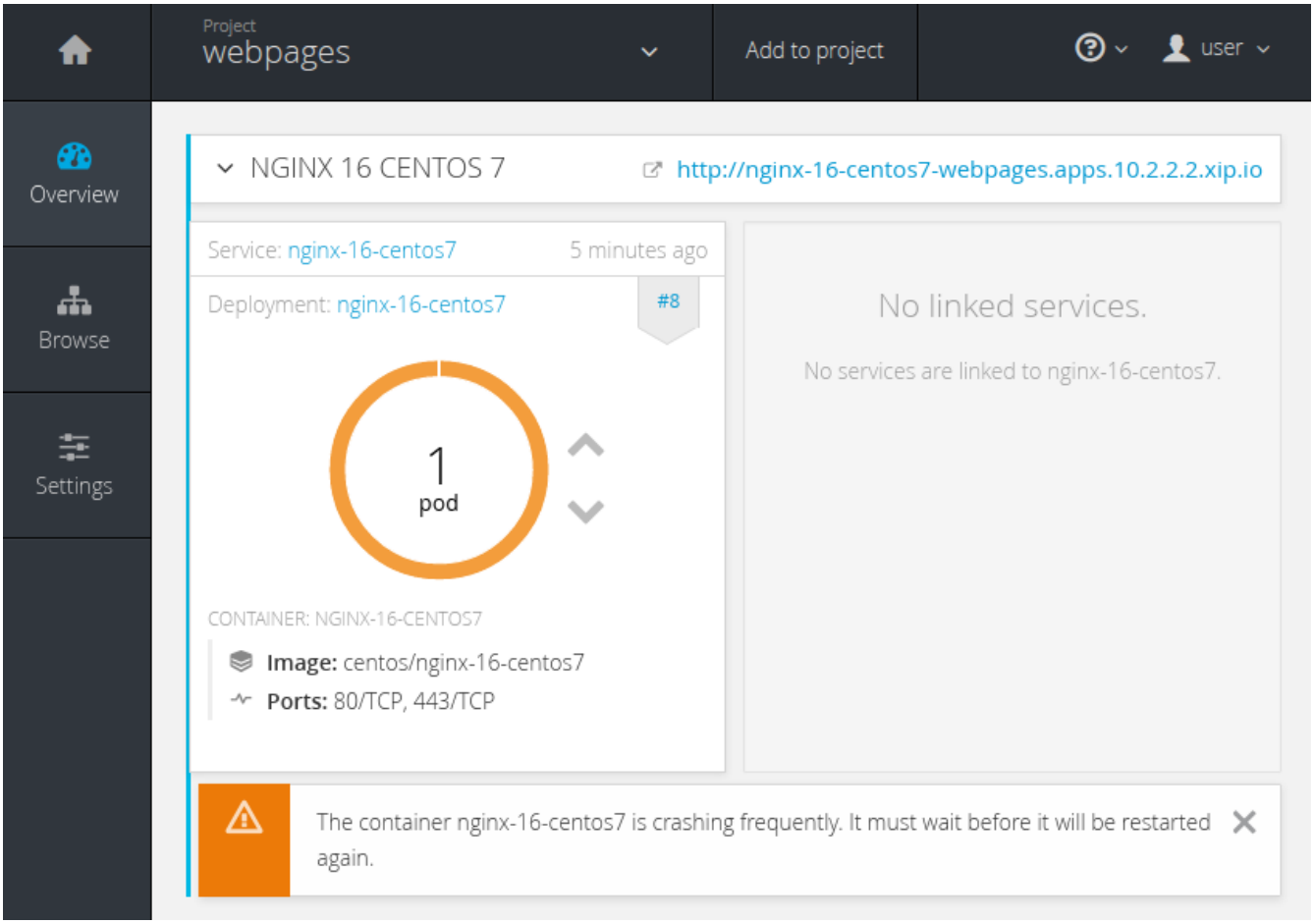


Now go back to the Overview page for your project. You will see a new pod come up and then the old one go down. You will also the deployment number increment, since we forced a new deployment.

Go ahead and click on the pod and go to it's detailed view. This time click on the "Events" tab on the detail page. You will see some interesting information.



Our liveness check is failing. And after it fails for a while, OpenShift says "This is broken, I am not going to try and deploy this anymore". If you go back to the overview page you can see that the circle has turned orange meaning there is an error.

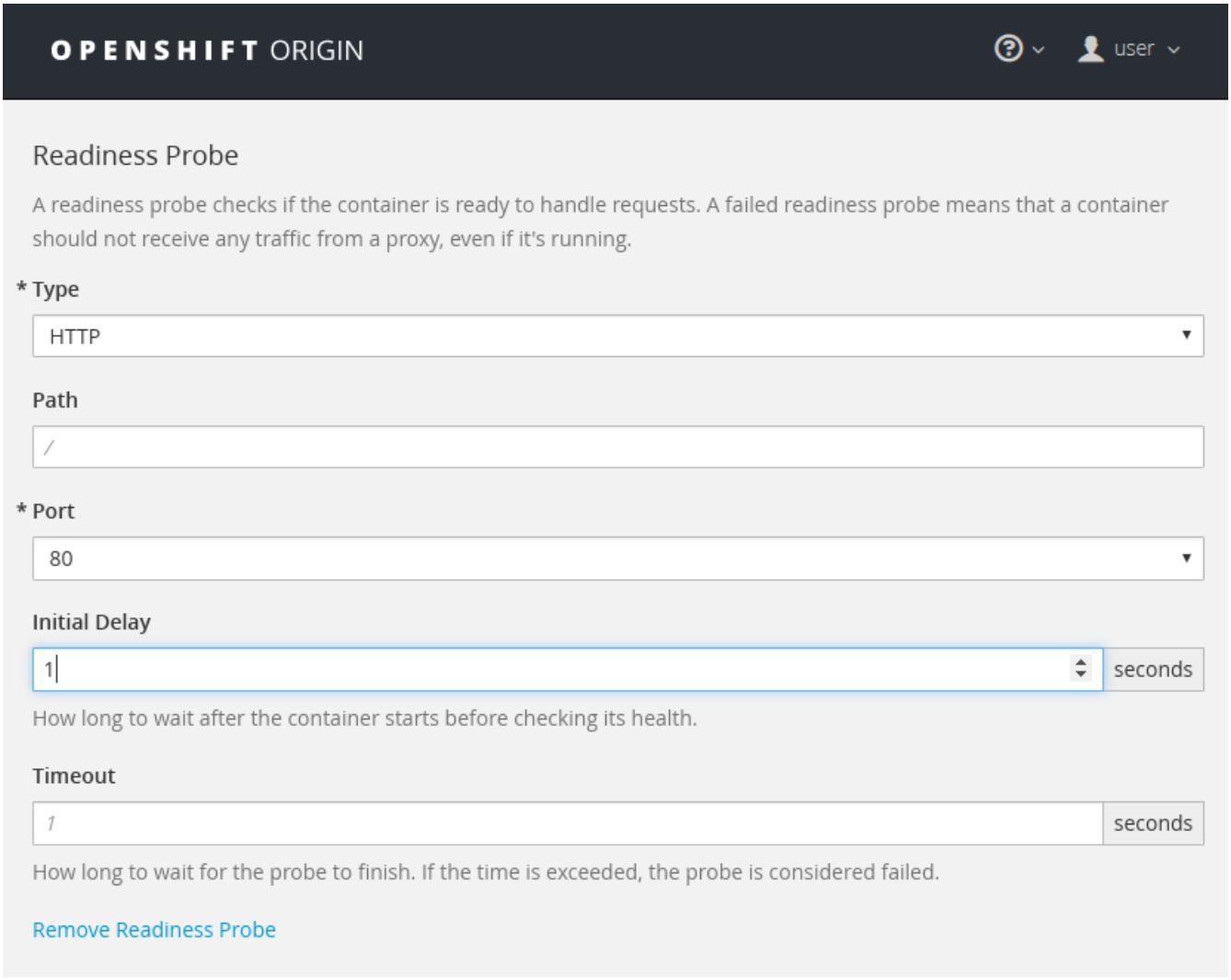


NOTE	Can you guess why this isn't working even though NGINX is running. I gave you a hint at the end of the last exercise. I will answer this in class.

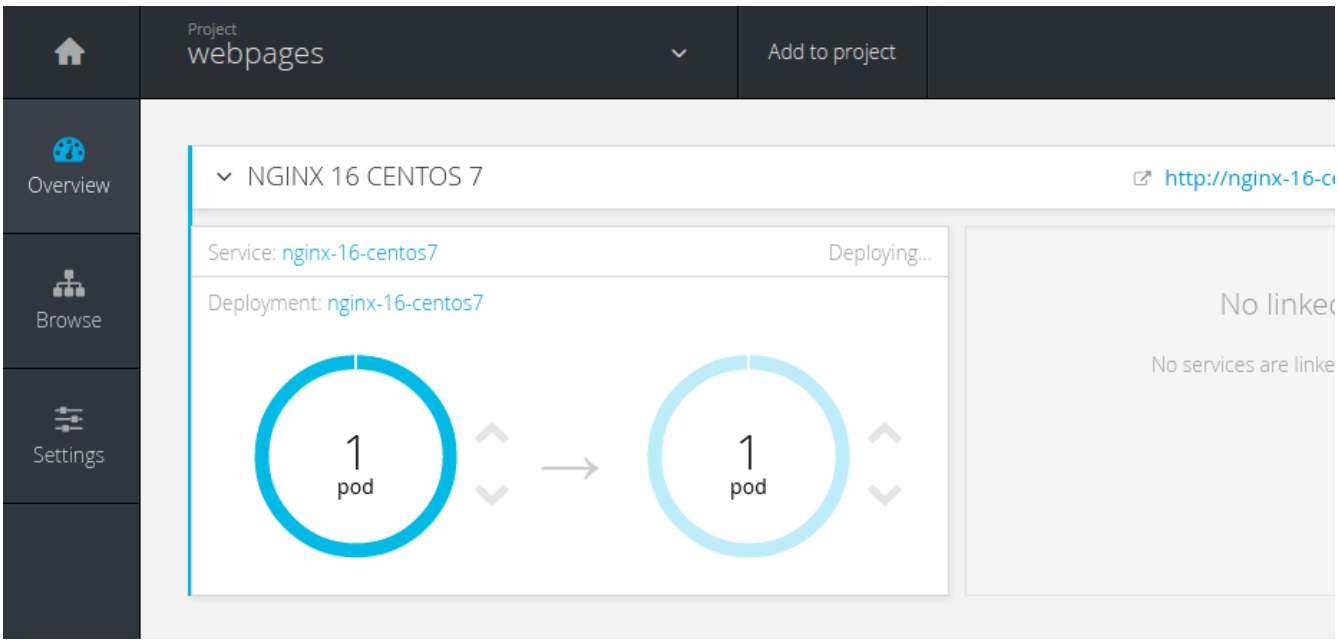
Let's go ahead and fix the liveness probe. Go back to the "Edit Health Checks" page that I showed you above. Change the probe type to "TCP Socket" and then make sure the port is 80. Go ahead and click save. You will see that the pod gets redeployed and then it comes up blue and stays blue this time.

Readiness Check

Now that we have configured a liveness check let's go ahead and set up a readiness check. Again navigate back to the health checks configuration page. Go ahead and set up a readiness check. With a web application, ready means we can serve up the web pages we want, not just connect to a port. Therefore, we need to use a check that uses HTTP over port 80.



Once you click save, if you go back to the overview you will see a new deployment kick off but then the screen will stay like this for a while:



If you wait five minutes, the new deployment will go away and we will stay with the old pod. OpenShift won't continue the deployment until the new pod is ready to serve content. The same problem from above is the reason we are also failing here.

Conclusion

Given the current configuration of our Docker container image, we could take advantage of the liveness check but not the readiness check. With your own images you make, knowing these checks are available should help you in crafting an image that can take full advantage of the platform.

NOTE	Please delete the current project as we will no longer need it. We are now actually going to move on to working with code and databases.

Running Your Images as a Non-Root User

If you look at most of the images on Dockerhub they run as the root user - so if yours does as well you have a lot of company. Unfortunately this is a bad security practice. You don't run your desktop as a "root" user and especially not on your servers, so there is no reason you should do it on your containers. All those same reasons that warn against running as an account with admin priveliges still apply in Docker land.

Running in OpenShift Container Platform, OpenShift Online, and OpenShift dedicated requires that your container be able to run as a random non-admin userid. For our class we are going to use a simple example.

Making a CentOS Apache HTTPD Image

We are going to start with a CentOS image and Apache HTTPD to the image. Since you have already built Docker images before I will just give you the GitHub repo. with my simple image:

https://github.com/thesteve0/myDockerImages/tree/master/simple

Here is the Dockerfile with the comments removed:

```
FROM centos:centos7

MAINTAINER Steve Pousty <thesteve0@redhat.com>

RUN yum install -y --setopt=tsflags=nodocs httpd.x86_64 && yum clean all -y

EXPOSE 80

CMD /usr/sbin/apachectl -DFOREGROUND

#docker run -p 80:80 6e3c25bdca9b
```

You should be able to run this no problem on your machine. If we push this to Dockerhub we can actually run this in the A1VM since the OpenShift cluster does not enforce most of the security policies.

First we need to push the image to DockerHub

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a
Docker ID, head over to https://hub.docker.com to create one.
Username: thesteve0
Password:
Login Succeeded
$ docker push thesteve0/centoshttpd
```

Now with our command line we run this in OpenShift with the OC command line tools:

```
$ oc login  #your username and password can be anything you want
$ oc new-project mydockerimages
$ oc new-app thesteve0/centoshttpd
--> Found Docker image 6e3c25b (4 days old) from Docker Hub for
"thesteve0/centoshttpd"

    * An image stream will be created as "centoshttpd:latest" that will track this
image
    * This image will be deployed in deployment config "centoshttpd"
    * Port 80/tcp will be load balanced by service "centoshttpd"
      * Other containers can access this service through the hostname "centoshttpd"
    * WARNING: Image "thesteve0/centoshttpd" runs as the 'root' user which may not be
permitted by your cluster administrator

--> Creating resources with label app=centoshttpd ...
    imagestream "centoshttpd" created
    deploymentconfig "centoshttpd" created
    service "centoshttpd" created
--> Success
    Run 'oc status' to view your app.
```

Let's go ahead and discuss what we just did. Oc new-app realizes we have passed it a Docker image and does "the right thing" to get this image running in OpenShift. It creates:

1. An Image Stream
2. A Deployment Configuration
3. A Service

The Deployment Configuration uses the Image Stream to determin the Docker image to pull, pulls it, and then spins up the pod with our image running in it as a container.

You can also log into the web console to look at what we created:

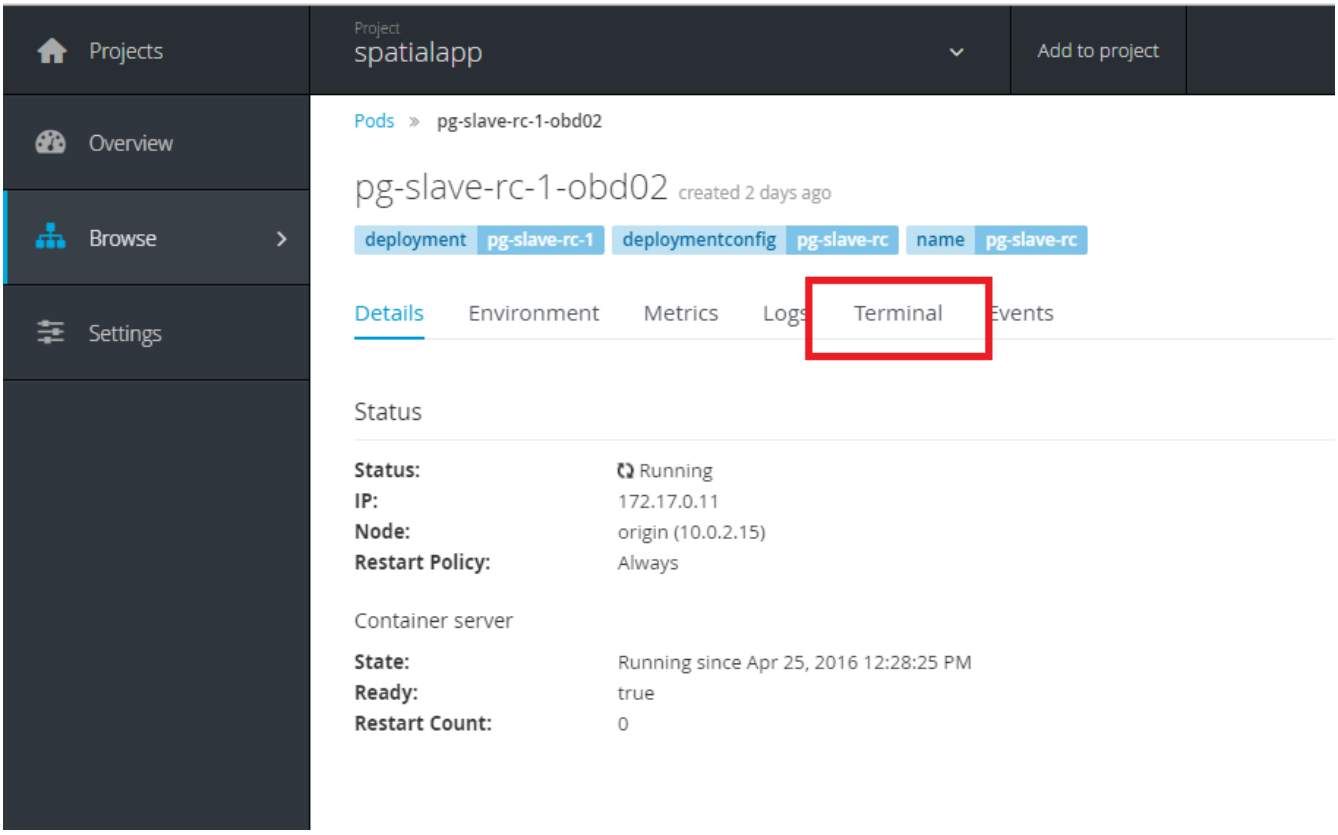
https://10.2.2.2:8443/console

Just use the same username and password as you used in the command line.

TIP	If your first deploy fails it may be because the Docker image does now download in time to the node. Please just trigger another deploy.

What is the User For Our Pod

In the web console - go to the running pod and then click on the terminal tab.



This is a terminal into your pod. Go ahead and click on the terminal and then type in `$ whoami`. You will see that you are running your container as root.

Switching the User

Our first task in getting this ready is making the image run as non-root. Actually to run in OpenShift you need the image to be ok with running as a randomly assigned user. We are going modify our Docker image to be a random users. There are several things we need to do to get there . Since we can't bind to port 80 as a non-root user we need to change the port for Apache HTTPD. We will also need to change the user we run as. Finally then we will need to see what this breaks.

Here is the Github repo with my new docker image:

https://github.com/thesteve0/myDockerImages/tree/master/simpleNoRoot

Let's go through the following items together

1. Modify httpd.conf to serve over port 8080. While we do this we will also modify logging to pipe to stdout
2. Expose 8080 rather than 80
3. Run as user 1001
4. This turns out to cause a problem with file permissions for creating the httpd.pid. Which then exposes a bug in Docker

At the end of this we now have a docker image that does not need root to run.

After we build, We can test this by running our docker image like so:

```
docker run -p 8080:8080 -u 1234 mydockerimage
```

If it runs no problem then we are all set to run on OpenShift.

We can build and push this to Dockerhub and then run it in OpenShift like last time.

```
$ oc new-app thesteve0/noroot
--> Found Docker image 439b966 (4 days old) from Docker Hub for "thesteve0/noroot"

    * An image stream will be created as "noroot:latest" that will track this image
    * This image will be deployed in deployment config "noroot"
    * Port 8080/tcp will be load balanced by service "noroot"
      * Other containers can access this service through the hostname "noroot"

--> Creating resources with label app=noroot ...
    imagestream "noroot" created
    deploymentconfig "noroot" created
    service "noroot" created
--> Success
    Run 'oc status' to view your app.
```

Voila, the root warning is gone. From my teams experience the biggest place we find problems with being non-root, or a random userid, is actually with file permissions

What if I need a user to be in the /etc/passwd table

Sometimes an application will look up the user in the password table which won't work given the way I have just shown you. For example you need a username and not just an id. To handle you can use NSS Wrappers to do the work in your image.

I am not going to talk through it much here but we will look at some example Docker images that take advantage of NSS wrappers.

Jenkins Image does this with NSS wrapper example https://github.com/openshift/jenkins/blob/master/1/Dockerfile.rhel7 OpenShift assigns a random UID when we start up jenkins and it runs this script (along with others) https://github.com/openshift/jenkins/blob/master/1/contrib/s2i/run#L29-L31

And this adds the random UID to the NSS passwords file so the app can find it there https://github.com/openshift/jenkins/blob/master/1/contrib/jenkins/jenkins-common.sh#L12-L29

Pulling New Images from Dockerhub Into OpenShift

As you iteratively work on your image you will probably push to Dockerhub often. To get OpenShift to pick up the new images you need to modify your DeploymentConfig to always pull new images on deployment.

By default the DeploymentConfig has its *imagePullPolicy* set to *AlwaysPull* if we use the tag latest (which we are doing in this case). This means everytime we do a deploy we do a pull request to Dockerhub (or whatever registry we are using) but we may not pull any bits if there are no changes. Here is some documentation.

Next steps

And with that we have:

1. Gotten our image to run as a random UUID
2. Made our image log to stdout

The final things we absolutely need to make it do is use volumes for changing content. So let's go to the next section and get going!

Working with Storage

Since Docker containers are immutable, any content that you want to change over the life of the container should be put in external volumes. As a Docker author you probably already know this. In OpenShift this doesn't change and also requires nothing special to make it happen.

In case you need a reminder here is the Docker documentation on storage volumes and their use.

Changing Our Image For Storage

The changes to our HTTPD image are minimal. First I change the permissions on the HTML directory (I looked at httpd.conf to determine the directory). This way we can make changes to the directory while the container is running and the content can remain.

Here is the Dockerfile that runs as a random UID and also uses volumes:

https://github.com/thesteve0/myDockerImages/blob/master/simpleNRVol/Dockerfile

I did two steps to enable the volume:

1. Made the content directory into a VOLUME.
2. Changed the permissions on /var/www/html

That's it to get it to work with OpenShift.

Run the Image in OpenShift

Let's go ahead and run this image in OpenShift.

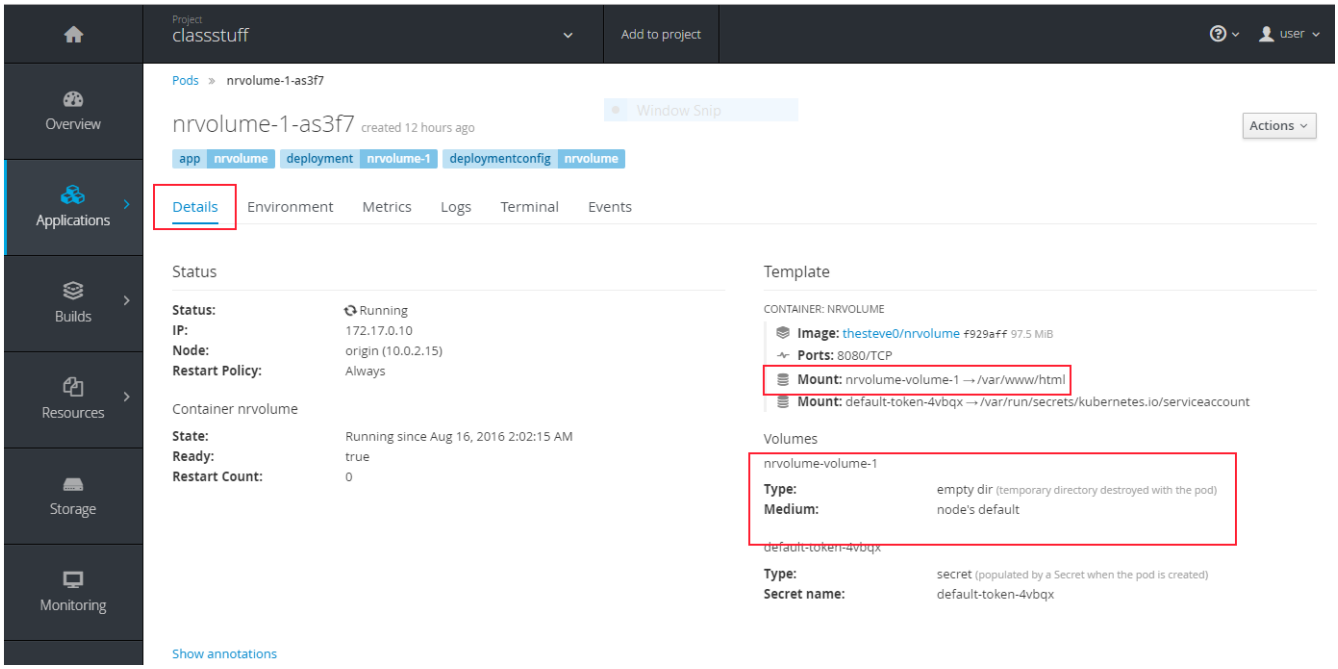
```
oc new-app thesteve0/nrvolume
```

Once this comes up we don't see any change to the content but we can again go to the terminal for the pod. Once inside the terminal we can do the following:

```
$ cd /var/www/html
$ ls
$ vi index.html
```

The CentOS base image includes Vi so we can go ahead and edit the file. After editing and saving, when we reload the page we can see our changes.

You may be wondering where did this file get saved. To explain that I need you to click on the Details tab again and look at the right side.



In the top red box we can see that OpenShift has mounted a Volume for /var/www/html and in the second red box we can see that OpenShift has mapped that volume to an emptyDir.

Watch what happens now when I vagrant SSH into the machine and kill the pod running the container.

To understand more of this I will do a very high level discussion of Volumes, Persistent Volumes, and Persistent Volume Claims in OpenShift/Kubernetes. A more thorough discussion of this topic could easily take a whole day.

Storage on OpenShift/Kubernetes

OpenShift inherits most of it's storage capabilities and functionality from Kubernetes so I am going to have this part of the discussion using Kubernetes.

There are different ways that a Kubernetes cluster can handle a volume claim by a Docker image. Depending on how the cluster is configured there can be up to 5 different types of volume mounts (look at the options under the -t flag). The settings for how volumes will be handled is typically on the DeploymentConfig, which is responsible for determining how pods will be deployed.

The easiest volume, if enabled by the cluster administrator, is an emptyDir. This setting means the volume will be created as an empty directory on the node hosting the pod. The volume is ephemeral and only lasts as long as the pod reamins on the node. If they pod is removed from the node all the data will be lost, though the data should be safe from a container crash

On the other hand, When setting up a cluster, a system administrator can make persistent volumes available to the cluster.

Persistent Volumes can be made from several different types of storage such as Ceph, iSCSI, EBS, or in the case of our VM, NFS. Once these are "attached" to the cluster, space can be requested from them by any of the pods.

If you want to see all the Persistent Volumes (pv) on the A1VM do the following commands:

```
$ oc login -u admin
Authentication required for https://10.2.2.2:8443 (openshift)
Username: admin
Password:
Login successful.
$ oc get pv
NAME      CAPACITY   ACCESSMODES   STATUS      CLAIM     REASON    AGE
pv01      10Gi       RWO,RWX       Available                       7d
pv02      10Gi       RWO,RWX       Available                       7d
pv03      10Gi       RWO,RWX       Available                       7d
pv04      10Gi       RWO,RWX       Available                       7d
pv05      10Gi       RWO,RWX       Available                       7d
pv06      10Gi       RWO,RWX       Available                       7d
pv07      10Gi       RWO,RWX       Available                       7d
pv08      10Gi       RWO,RWX       Available                       7d
pv09      10Gi       RWO,RWX       Available                       7d
pv10      10Gi       RWO,RWX       Available                       7d

$ oc describe pv pv01
Name:       pv01
Labels:     <none>
Status:     Available
Claim:
Reclaim Policy: Recycle
Access Modes:   RWO,RWX
Capacity:   10Gi
Message:
Source:
    Type:   NFS (an NFS mount that lasts the lifetime of a pod)
    Server: localhost
    Path:   /nfsvolumes/pv01
    ReadOnly:   false
```

Developer can then make claims into the PVs by making a persistent volume claim (PVC). There is developer documentation on how to use PVCs with Volumes in Docker images.

In the case when the filesystem supports proper permissioning, like NFS, multiple pods can mount the same claim. this would be a good pattern to use with our HTTPD pod as we want all the pods to serve the same content. Here is a blog post by one of my coworkers which discusses using PVs and PVCs with an NGNIX docker image.

Moving on to Templates

Making Persistent Volume claim usually happens in a template so let's move on to that topic. Again, like storage templates could take a whole day to cover so I am only going to give a brief introduction. A Docker image by itself will unlikely be enough if you want people to be successful with your product. As an image maker you are most definitely going to want to make a template for

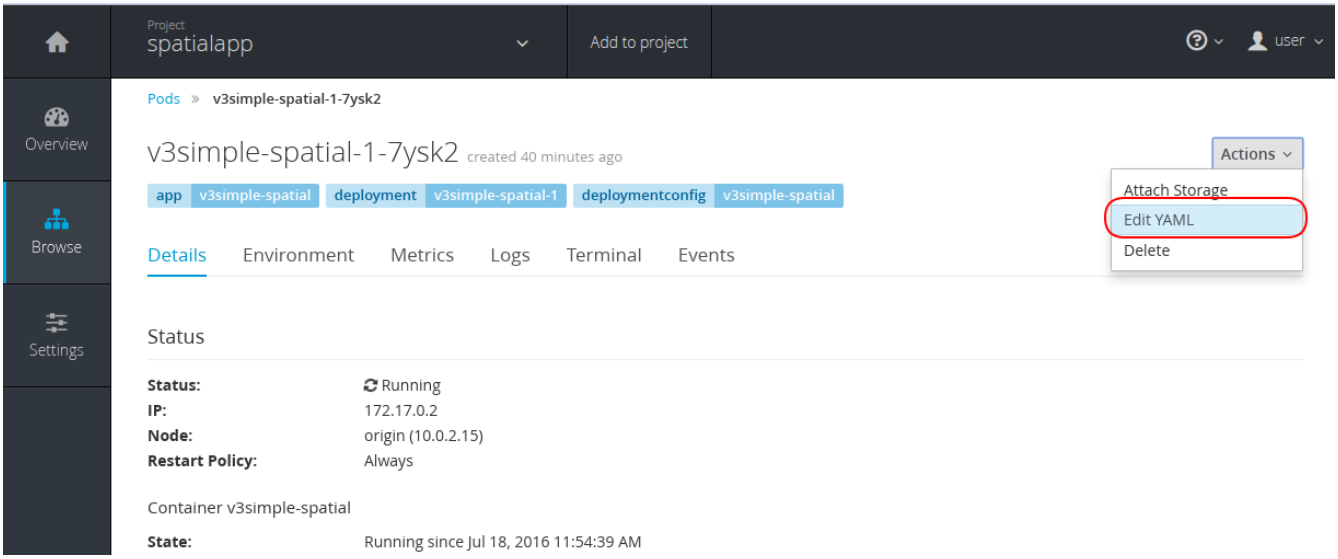
developers or system administrators to consume your Docker image.

Templates and OpenShift/Kubernetes Declarative Syntax

As I explained before, Kubernetes and OpenShift use a declarative model of the world - you tell the cluster what the truth is and the cluster then goes about making it so. OpenShift let's you declare the truth in a file, either YAML or JSON syntax, whichever you prefer. In this section we are going to VERY briefly look at some of these files and leave with pointers to more information if you want to dive in more.

The OpenShift Master exposes a Swagger API if you want to really dig in and see what is exposed. You can hit this URL https://10.2.2.2:8443/swaggerapi/api/v1

You can also look at any of the YAML for any of the resources in your names space by clicking on the top right drop down and selecting "Edit". This will let you live edit the YAML for any of your resources and apply the changes immediately. Here is an example showing you how to open the YAML for your pod:



We are not going to change any of the YAML right now but you can look at it. You will see a *labels* section, these are the labels for the pods that can be used by selectors on other resources.

The main docs for understanding more about the OpenShift/Kubernetes Objects starts on the OpenShift Documentation but then links out for further explanation on Kubernetes native objects.

Simple Examples

In the OpenShift Origin GitHub repository there is an example called "Hello OpenShift". I will walk through this with you in the class.

Inside this is probably the simplest JSON you can find, which defines a project:

https://github.com/openshift/origin/blob/master/examples/hello-openshift/hello-project.json

And there is also one to define a pod:

https://github.com/openshift/origin/blob/master/examples/hello-openshift/hello-pod.json

After talking through these you should have a fairly basic understanding of how these files are structured.

Full Templates

Templates take this basic structure and allow you the ability to create much more complete and interesting collection of resources, including full applications.

We don't have time in this workshop to really cover all that you can do with the a template, but one of the most interesting things is parameterized input. As a template developer you can make certain variables in your template, say database username and password, to parameters. This way you can do two things:

1. You can allow users to input their own values when they process the template. If you enable this feature then the user is required to input a value to make sure the template actually works.
2. If you want to provide an autogenerated default value for the paramterized field you can specify a generating expression to create a value. For example:

```
{
    "name": "ADMIN_PASSWORD",
    "description": "administrator password",
    "generate": "expression",
    "from": "[a-zA-Z0-9]{8}"
},
```

Back in GitHub there are different template examples put together by the OpenShift team. There are two which are particularly helpful:

1. **sample-app** - this template provides an example of setting up a Ruby application with a MongoDB database. It shows how to parameterize everything and wire things together. It also shows how to do different build types in OpenShift. If we are good on time I will quickly walk us through the S2I build example.
2. **Django-ex** - this a quickstart put together to help you get started with Django very quickly. Inside this project there is a template to get Django and PostgreSQL up and running in an OpenShift cluster. There are more of these templates, such as Node.js and Rails. Just look in the openshift/templates directory to see more example templates.

If you want to learn more about templates, there is documentation on them in the architecture section and in the developer guide.

Import/Export

One of the really useful byproducts of having a declarative API mean import and export becomes

very easy. Let's go ahead and export our simple application and then import it into a new namespace.

First we need to export the JSON (my preferred format) for our existing application. This is quite simple:

1. Export our current application as a template text file:

```
oc export is,dc,svc,bc,route --as-template=json -o json > simple_spatial.json
```

We have now created a template that lists all of our project resources, except pods, as a template. We do not export the pods because we actually want our project to recreate the new pods.

If we wanted to move the pods over exactly then we would have to put the containers we created into a registry our new project will have access to and then name the containers by name.

1. Create a new project to import our project into

```
oc new-project newproject
```

1. Open simple_spatial.json in your favorite text editor. After you do that do a global search and replace of "spatialapp" with "newproject". And remove the Sha signature from the item that looks like:

```
BEFORE
  - image: 172.30.182.216:5000/spatialapp/simple-
spatial@sha256:b0379d1e921087aa267a9bb5e4602d1ec51948f88dfd83c4d65f24c133654102

AFTER

- image: 172.30.182.216:5000/newproject/simple-spatial
```

1. Make sure you are in the newproject project and then just do the new-app command on simple-spatial.json

```
oc new-app simple_spatial.json
```

1. PROFIT!!!

You can use these steps to move between any OpenShift instances as well - as long as they can see the same git repos and Docker registries.

Now with that business done, let's add a database to our application!

Other Things You Should Know

Let's touch briefly on some of the ways you can allow users to alter settings (such as passwords or memory settings) in your Docker images. The primary way to do this is by using environment variables in your docker image for settings. Once you put these into your Docker image the user can set these in OpenShift using OpenShift environment variables, ConfigMaps, or Secrets. All three of these things are better means to handle setting information in your app that you might want to change at runtime.

Let's look at the Red Hat Software Collections Library's PostgreSQL docker image to see some examples of how to work with environment variables in your Docker image.

https://github.com/sclorg/postgresql-container/tree/master/9.5/root/usr/share/container-scripts/postgresql

Once you set up your image to accept these, the end user can use OpenShift environment variables, secrets, or configmaps to set these variables. Again, you may also want to put them into a template with sensible defaults but allow the user to overide the auto-generated or default values.

You Need to Brush Up On Shell Scripting

I hope it comes as no surprise to anyone that the simple Docker image we built for Apache HTTPD would not be realistic in a production scenario. Most of the production ready images I have seen launch their software using a shell script which in turn may call other shell scripts. You will most likely need to make these same kinds of shell scripts to accomplish any real world Docker images.

For example, here is the repository for CrunchyDB Docker containers, production ready PostgreSQL containers. They are probably our most advanced partner in making containers to run on OpenShift and Kubernetes.

https://github.com/CrunchyData/crunchy-containers

So we start with this file, which is the Dockerfile for RHEL 7:

https://github.com/CrunchyData/crunchy-containers/blob/master/rhel7/9.5/Dockerfile.postgres.rhel7

Which then calls this file:

https://github.com/CrunchyData/crunchy-containers/blob/master/bin/postgres/start.sh

Which then calls this file - which allows the user to set shared buffers and such:

https://github.com/CrunchyData/crunchy-containers/blob/master/bin/postgres/start-pg-wrapper.sh

which has lines that do sed on a temporary postgresql.conf

https://github.com/CrunchyData/crunchy-containers/blob/master/bin/postgres/start-pg-wrapper.sh#L94

Here is the postgresql.conf.template it is using

https://github.com/CrunchyData/crunchy-containers/blob/master/conf/postgres/postgresql.conf.template#L114

Source To Image

If your users will need to combine your image with some type of "source" to create a new image then you are going to want to look at the Source to Image(S2I) project. It is a separate project outside of OpenShift, but the OpenShift infrastructure knows how to take advantage of it.

S2I is typically leveraged by language runtimes, such as Python with Gunicorn or Tomcat. We could certainly do the scripts with our HTTPD image and it would allow us to keep our HTML files out of the Docker image and just combine the image and the files at build time into a new Docker image.

I will explain in class and then I will go through the wildfly s2i image. I chose this image particularly for the APM attendees in the class since Java is a good use case for them.

https://github.com/openshift-s2i/s2i-wildfly/tree/master/10.0

If we wanted a simpler example we could look at the S2I for PHP

https://github.com/sclorg/s2i-php-container/blob/master/5.6/s2i/bin/assemble

Troubleshooting images

The `oc debug` command makes it easy to obtain shell access in a misbehaving pod. It clones the exact environment of the running deployment configuration, replication controller, or pod, but replaces the run command with a shell.

So you would use it like:

```
$ oc debug dc/MyDeploymentConfig

or

$ oc debug mypod
```

This will deploy or run the pod, but instead of doing the run command at the end of the Dockerfile, it will just give you a bash prompt so you can poke around and see how things are set up.

Setting up your RHEL base VM

While being primed is great, there is also even more value in getting to a certified container. Red Hat now provides Red Hat Enterprise Linux (RHEL) for free for development purposes. The steps to getting the VM setup are:

1. Register at http://developer.redhat.com
2. Download the RHEL ISO from https://developers.redhat.com/downloads/
3. Take your favorite VM technology and make a RHEL VM (I used VirtualBox)
4. Log in as root and do the following commands:

```
$ subscription-manager register # use your username and password from above
$ subscription-manager attach # should just work
$ yum update
$ reboot
$ subscription-manager repos --enable=rhel-7-server-extras-rpms
$ subscription-manager repos --enable=rhel-7-server-optional-rpms
# Do all the steps neccessary to install the guest additions on VirtualBox
# Useful for when we try to build our docker image
```

Installing Docker

Right now there are two version of Docker you can install into RHEL. https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/7/single/getting-started-with-containers/#getting_docker_in_rhel_7

Until OpenShift 3.3 comes out please use 1.9:

```
$ yum install docker --exclude docker-1.10* --exclude docker-selinux-1.10*
```

Add registry.access.redhat.com to the Docker Client

TODO

Summary

Now you have a machine that is ready to build Docker images based on RHEL. Any Docker image you build on this box will get it's enablement from the underlying host OS. So now you can start your Dockerfile with:

```
FROM rhel:7.2
```

and it should just work. Here is an example of a Docker build file that should work for you - https://github.com/sclorg/rhscl-dockerfiles/blob/master/rhel7.s2i-base/Dockerfile

These instructions are for a very minimal RHEL install. You will probably find that you need other tools installed on the machine as well. Go ahead and add any other tools you want. Don't forget about EPEL for packages not found in the standard repos.

Appendix: Here are some basic commands I found useful

How to build the documentation

```
asciidoctor -S unsafe ~/git/workshops/index.asciidoc
```

How to delete the application pieces but not the DB pieces

```
oc delete is,dc,svc,bc v3simple-spatial
```

How to do a web request with cURL

```
curl 'curl 'http://v3simple-spatial-spatialapp.apps.10.2.2.2.xip.io/db' -d ''
' -d ''
```

How to insert a value into the table

```
Insert into parkpoints (name, the_geom) VALUES ('ASteve', ST_GeomFromText('POINT(-
85.7302 37.5332)', 4326));
```