Introduction to OpenShift V3 for Developers

Prerequisites:

This document assumes you have the latest version of Vagrant and VirtualBox installed on your machine.
Tested with:
Fedora 21, Vagrant 1.7.2, VirtualBox 4.3.30
Fedora 22, Vagrant 1.7.2, VirtualBox 4.3.32

This document and all the binaries you need can be found here:
http://bit.ly/v3devs

You may also need to enable vt-x in your system bios under the virtualization setting to enable multi-core virtual machines. This is apparently turned off by default on Lenovo Thinkpads (model W541 confirmed) laptops.

Adding the Vagrant Box

Brief Version

assumes you downloaded the box file and the Vagrantfile to the same directory

1. $ vagrant box add --name openshift3 <box file you just downloaded.box>
2. $ vagrant up
3.

Wordy Version

Download the .box file in the directory mentioned above. At the time of this writing the file is approximately 2.2 gigabytes in size so be prepared for a long download and put it in a place with enough space. First you need to import the vagrant box

$ **vagrant box add --name openshift3 <location of your box image.box>**
{this will take a while}

The Vagrantfile tells Vagrant how much memory, CPUs, and which port mappings to perform when starting up the virtual machine.

If everything is set up properly you should be able to just do (MAKE SURE YOU DO THIS IN THE SAME DIRECTORY AS YOUR Vagrantfile)

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
   default: Adapter 1: nat
==> default: Forwarding ports...
   default: 80 => 1080 (adapter 1)
   default: 443 => 1443 (adapter 1)
   default: 5000 => 5000 (adapter 1)
   default: 8080 => 8080 (adapter 1)
   default: 8443 => 8443 (adapter 1)
   default: 22 => 2222 (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
   default: SSH address: 127.0.0.1:2222
   default: SSH username: vagrant
   default: SSH auth method: private key
   default: Warning: Connection timeout. Retrying...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Machine already provisioned. Run `vagrant provision` or use the
`--provision`
==> default: to force provisioning. Provisioners marked to run always will
still run.
```

If this is your first time running this command on your own machine you may need to wait while it imports the image. When it's done you have a complete version of OpenShift V3 along with a Docker registry running in a VM on your local machine. We are going to use this VM with the command line and web interface as if it was a server running out somewhere on the "internets", so you will have same experience as a developer whether you are using this VM or on a hosted version of OpenShift V3.

NOTE:

There appears to be a bug in Vagrant where on some computers, improper keys are set up in SSH leading to the following error when trying to start the image:

```
default: Warning: Connection timeout. Retrying...
default: Warning: Authentication failure. Retrying...
default: Warning: Authentication failure. Retrying...
default: Warning: Authentication failure. Retrying...
```

This is due to a bug in Vagrant :
https://github.com/mitchellh/vagrant/issues/5186

There are some places that seem to deal with the workaround, including the post above.
http://magento.stackexchange.com/questions/76500/magento-u-virtual-machine-warning-authentication-failure-retrying

We will continue to research this issue and see what we can do. Please follow the instructions above to try and make this work.

Command Line Tools:

Download the appropriate binary for your work machine and extract the oc file and put it in a directory in your path (or you can add the extraction location to your path). The command line utilities all start with the title openshift-origin-v* and then end with the name of the platform (Darwin is for Mac). Once you complete you extracting the file you should be able to do:

```
$ oc version
oc v1.0.7
kubernetes v1.2.0-alpha.1-1107-g4c8e6f4
```

Don't be concerned if the version numbers are greater than the ones listed here, it just means I forgot to update the document when I updated the binaries for download.

Lab 1: Smoke Test and Quick Tour

The OpenShift master for this all in one image uses a self-signed SSL certificate. This means we need to bypass the certificate check when logging in through the CLI and accept the security warning when logging in through the web console. Your connection will be secured, we just don't have a certificate from a recognized signing authority.

Note: if you already have a ~/.kube/config file on your machine make sure to rename it something else. If you don't your command line tools may try to connect to a different machine.

Command Line

$ oc login

```
OpenShift server [https://localhost:8443]:   #leave blank
The server uses a certificate signed by an unknown authority.
You can bypass the certificate check, but any data you send to the
server could be intercepted by others.
Use insecure connections? (y/n): y

Authentication required for https://localhost:8443 (openshift)
Username: admin
Password:  password
Login successful.

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

  * default (current)
  * turbo
```

$ oc project turbo

```
Now using project "turbo" on server "https://localhost:8443".
```

Congrats, you are now authenticated to the OpenShift server. OpenShift uses a token or Oauth2 based authentication. By default your auth token will last for 24 hours. There is more information about login and its configuration on the Origin Documentation Site.

We are working in a project named *Turbo*. Projects are a top level organizational concept. An OpenShift project allows a community of users (or a user) to organize and manage their content in isolation from other communities. Each project has its own resources, policies (who can or cannot perform actions), and constraints (this project is allowed this much quota, etc.). Projects act as a "wrapper" around all the application services and endpoints you (or your teams) are using for your work.

We are just going to exercise some commands to make sure things are working as expected. We will cover terminology later

The exact details regarding IP addresses may be different when you invoke `oc status`.

$ oc status

```
In project Turbo Sample (turbo)

service database (172.30.170.146:5434 -> 3306)
database deploys docker.io/openshift/mysql-55-centos7:latest
       #1 deployed 7 minutes ago

service frontend (172.30.203.123:5432 -> 8080)
frontend deploys origin-ruby-sample:latest <-
       builds git://github.com/openshift/ruby-hello-world.git with
turbo/ruby-20-centos7:latest
       #1 deployed 59 seconds ago

To see more information about a service or deployment config, use 'oc describe service
<name>' or 'oc describe dc <name>'.
```

```
You can use 'oc get pods,svc,dc,bc,builds' to see lists of each of the types described above.
```

Let's look at the routes that are available to our application.

$ oc get routes

```
NAME             HOST/PORT         PATH     SERVICE   LABELS
route-edge   www.example.com                frontend   template=application-template-stibuild
```

The simple explanation for how routes work is:

1. A request comes in to the OpenShift server over port 80 (which we have mapped to 1080 on our localhost)
2. The router looks at the HTTP header for the host entry
3. The router send the request on to the service that corresponds to that host

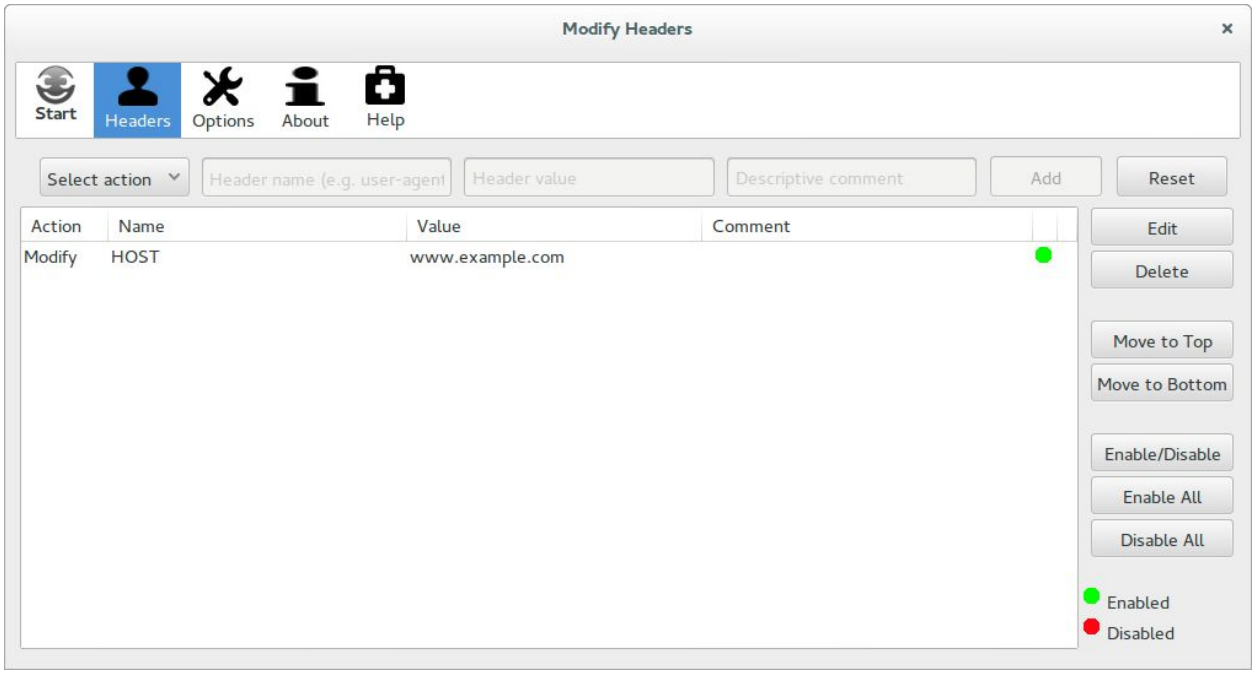
As you can see, our application will be available at www.example.com.

Application in Browser

If you want to get this to work with a browser on your host machine, you can do this by modifying the headers or editing your hosts file on your machine. For these instructions we will use a browser plugin since it is the easiest to explain and make work cross platform. That said, if you are comfortable editing your hosts file then this can actually be a very good solution as well.

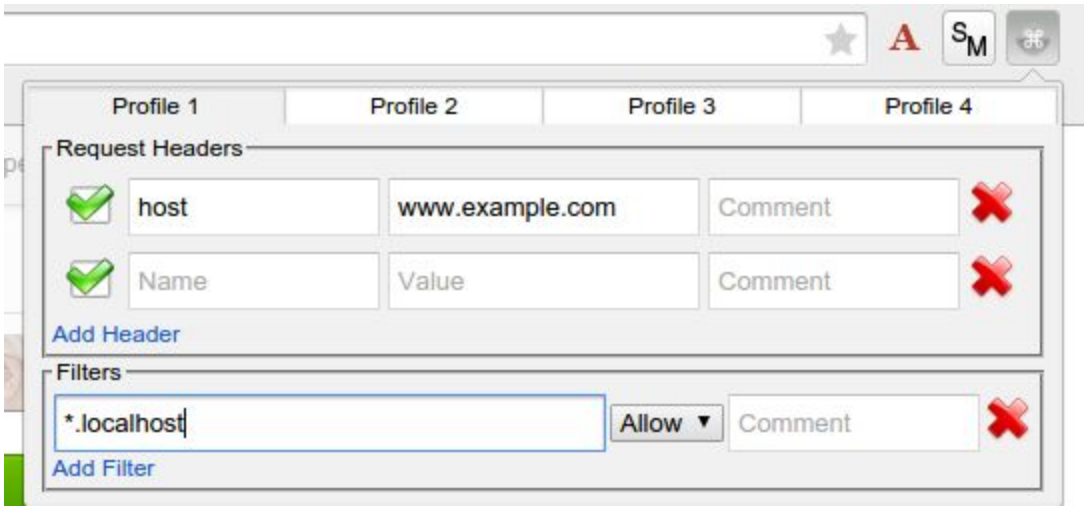
Note: If you decide to modify the hosts file on your machine, this can be found in the following directory on Microsoft Windows: *C:\Windows\System32\Drivers\etc* and at */private/etc/hosts*. Also, if you are comfortable editing the host file then it is assumed you know how to configure it properly - otherwise please use the header plugins listed below.

In Firefox I have tested the "Modify Headers" extension and in Chrome I used "ModHeader" by hao1300. Both of these extensions allow you to add or overwrite headers to HTTP requests. For example, here is what the "Modify Headers" interface looks like in Firefox:



Once I want to use this header I click the greyed-out start icon in the top left and it changes all host headers in Firefox to www.example.com. In the browser I type in https://localhost:1443 but the page that is show is the *frontend* service. For both browsers you may see a security warning for an invalid certificate due to our cert not being signed by a known authority. Please proceed past the security warning to see the page. **Don't forget to press Start when you ready to apply the rules.**

Here is what the "ModHeader" looks like in Chrome to achieve the same effect. **PLEASE NOTE: as of the recent update this extension no longer works at all - do not use it.**



Congrats we now have everything working and you have seen a fully functioning application!

Now it is time to dig in some more on how to work with Docker images and OpenShift

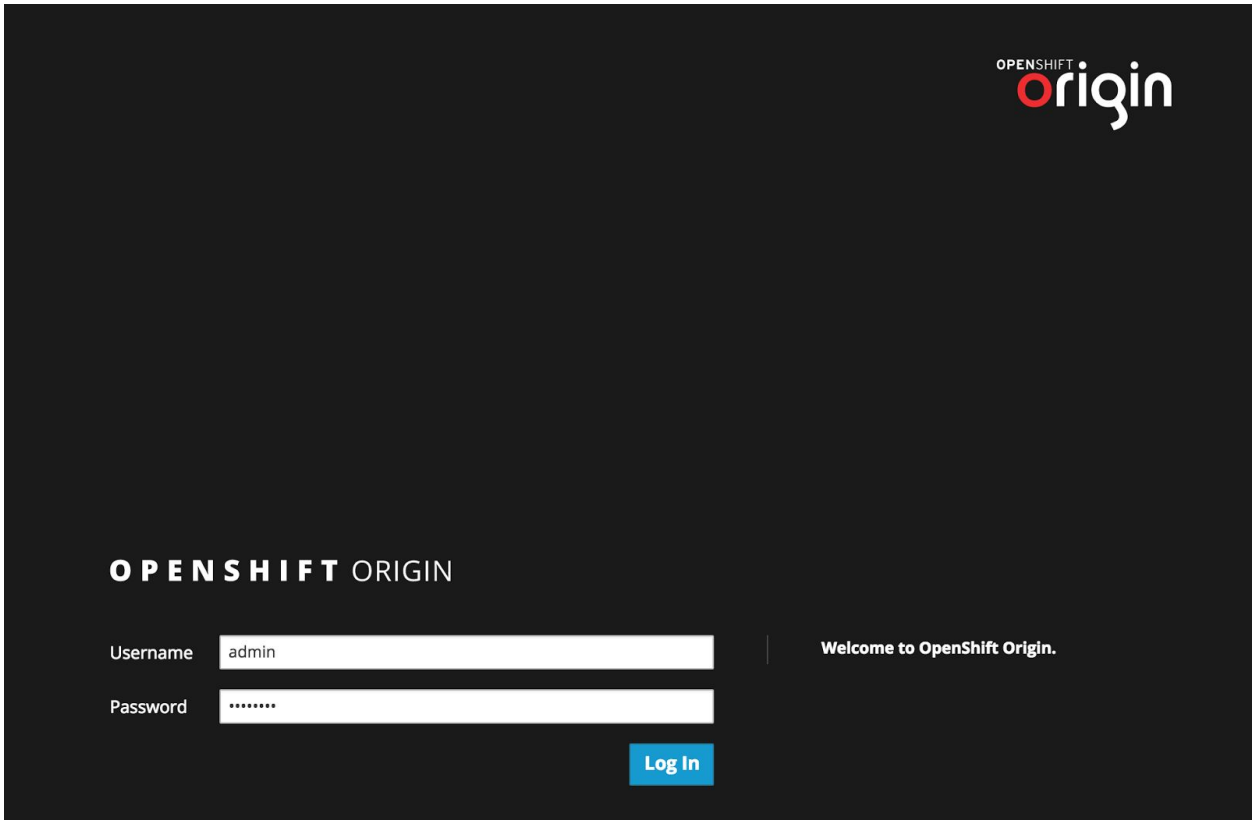
Web Console [Note these screenshots are 1.0.4 which is different from the version you downloaded - i.e. not as good]

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

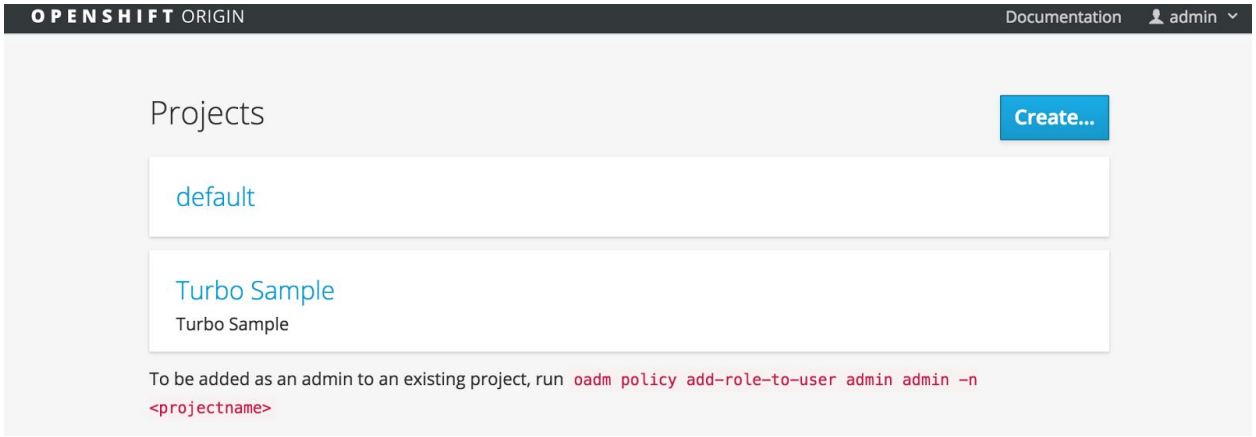
https://localhost:8443

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

Username: admin
Password: password

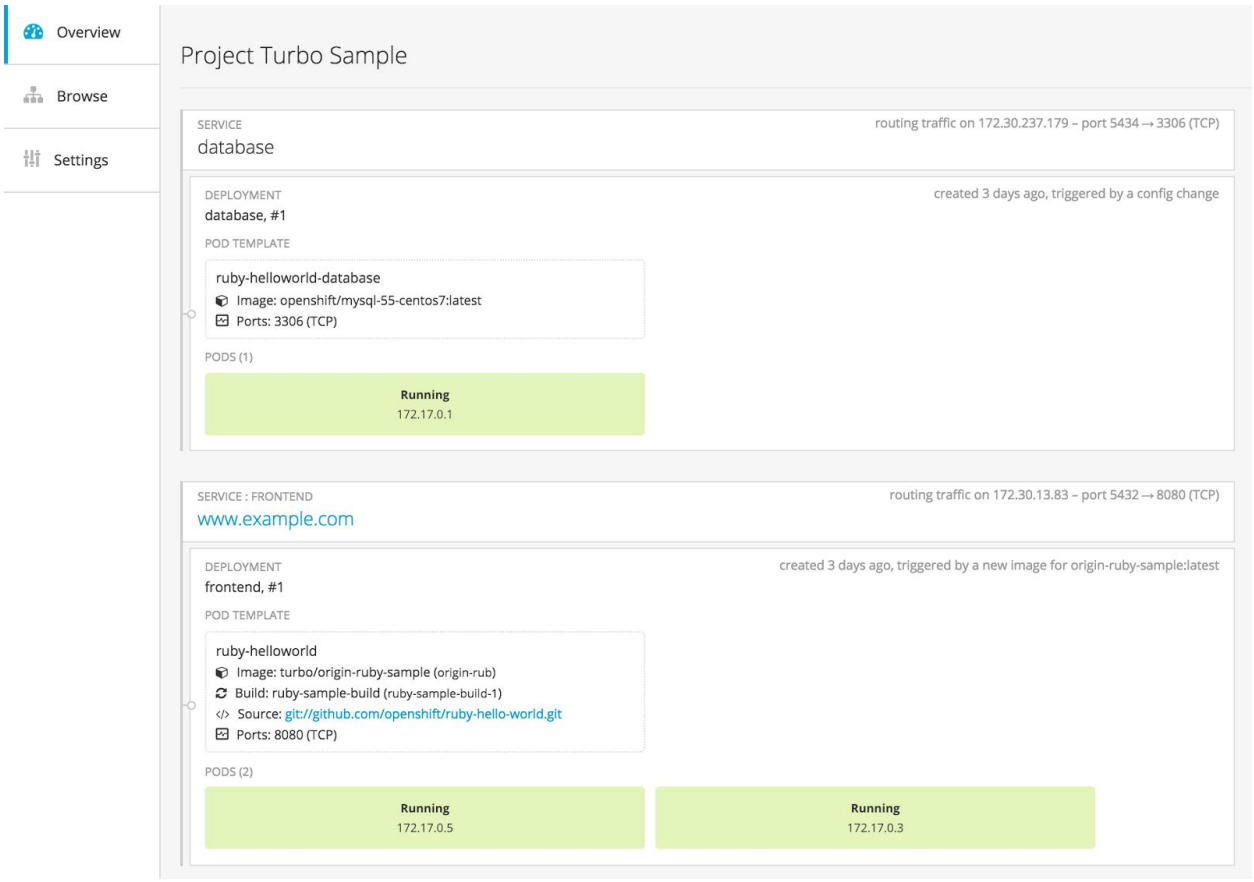


After you have authenticated to the web console, you will be presented with a list of projects that your user account has permission to work with as shown in the following image:

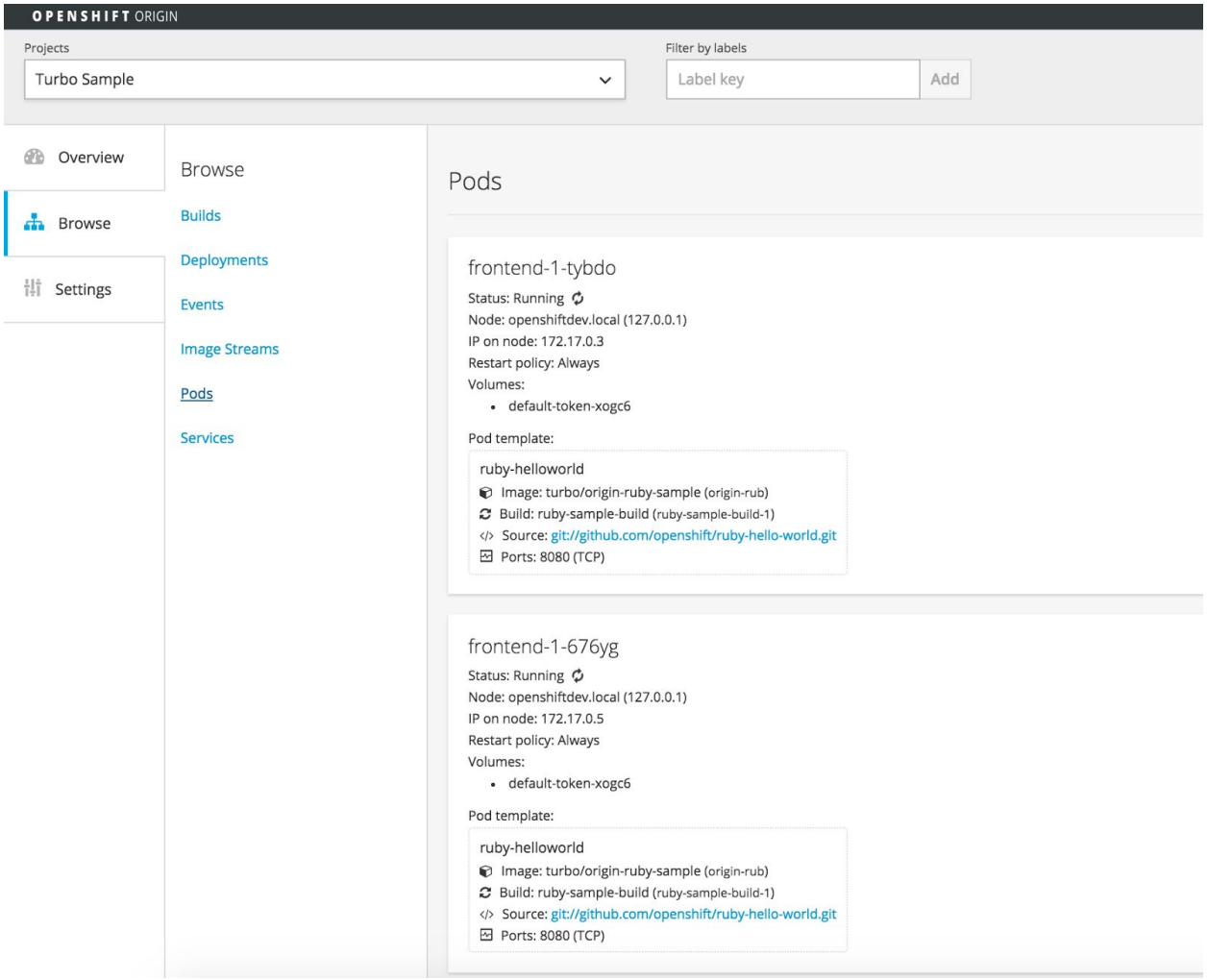


Click on the *Turbo Sample* project to view the operations that you can perform on a specific project. After you click on *Turbo Sample* you will be presented with the project overview page

which will list all of the services and deployments that you have running as part of your project. For this example, you will see a database deployment (mysql) and a frontend that is deployed to two pods.



Once you have digested the information on the overview page, click on the *Browse* tab on the left hand side of the screen:



Go ahead and play around a bit more with the web console to get familiar with the functionality. However, we will be using the command line tools for the majority of this lab. Please also remember, you may be confused by some of the concepts you have been exposed to, we promise we will explain them all in due time. This is just a smoke test to make sure things are working.

Lab 2: Deploy a docker image

Background: Container and Pods

Before we start digging in I need to explain the how containers (Docker images and instances) and pods are related. I am assuming since you are in this class you are very familiar with how Docker works so I am not going to cover this; I am not going to cover basic Docker commands or even the ideas of containers. It turns out that with OpenShift you don't use any Docker commands at all and instead interact with OpenShift directly. If you are not familiar, there are a lot of references on how to use containers in general and Docker in particular. Here are some references for you to read to become more familiar:

https://docs.docker.com/introduction/understanding-docker/
https://www.codementor.io/docker/tutorial/what-is-docker-tutorial-andrew-baker-oreilly
http://developerblog.redhat.com/2014/05/15/practical-introduction-to-docker-containers/

In OpenShift, the smallest deployable unit is a Pod. A pod is a group of one or more Docker containers and they are guaranteed to be on the same host. From the doc:

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

Containers within a pod can be from the same Docker image, though this is considered bad practice. The general idea is for a pod to contain a "server" and any auxiliary services you want to run along with that server. Examples of containers you might put in a pod are, an Apache HTTPD server, a log analyzer, and a file service to help manage uploaded files.

OpenShift uses containers and pods throughout its entire architecture. A pod is actually a container with containers running inside it. OpenShift also creates a pod to carry out any source build. Whenever you specify a build, a pod is used to hold the build and create the resulting Docker image. We will talk about builds much later in the labs.

Let's look at the description for one of the pods we deployed earlier on in the sample application.

$ oc get pods

```
NAME                          READY     REASON       RESTARTS   AGE
database-1-ofy3w              1/1       Running      1          3d
frontend-1-676yg              1/1       Running      1          3d
frontend-1-tybdo              1/1       Running      1          3d
ruby-sample-build-1-build     0/1       ExitCode:0   0          3d
```

Note: In the following command, replace database-1-ofy3w with the correct name from your deployment

$ oc get pods database-1-ofy3w -o json


```
{
   "kind": "Pod",
   "apiVersion": "v1",
   "metadata": {
       "name": "database-1-ofy3w",
       "generateName": "database-1-",
       "namespace": "turbo",
       "selfLink": "/api/v1/namespaces/turbo/pods/database-1-ofy3w",
       "uid": "1681ada8-1603-11e5-b37c-080027c5bfa9",
       "resourceVersion": "768",
       "creationTimestamp": "2015-06-18T21:43:42Z",
       "labels": {
           "deployment": "database-1",
           "deploymentconfig": "database",
           "name": "database"
       },
       ……
```


Note: The above output has been truncated in this document for space considerations.

Exercise 1: Get a "vanilla" docker image to work on OpenShift

Let's start by doing the simplest thing possible - get a plain old Docker image to run inside OpenShift. To do this we are going to use the Fedora Apache HTTPD image (https://registry.hub.docker.com/u/fedora/apache/) to run since it is so easy to test if it works. This is incredibly simple to do:

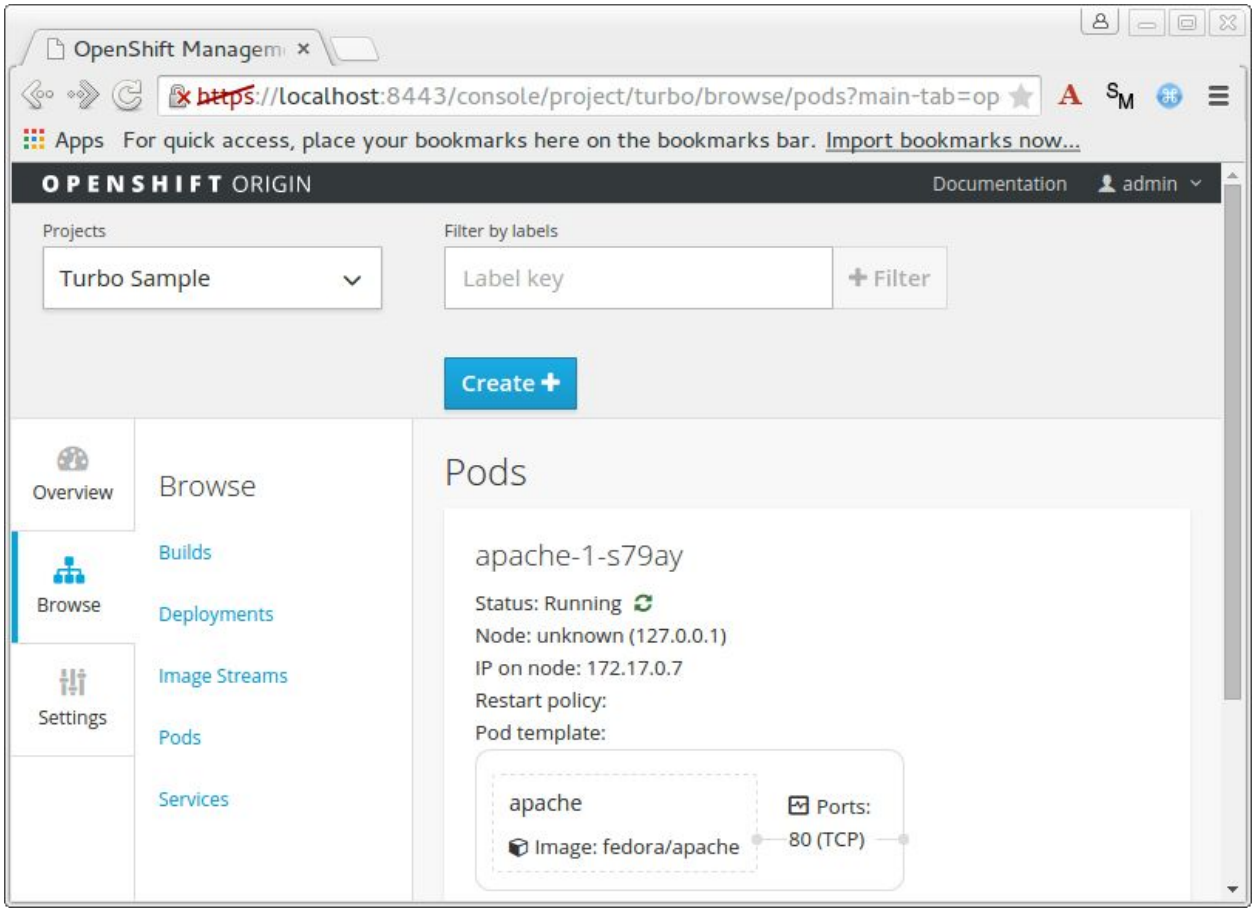
1. Create a new project to hold our our fedora/apache container:

   $ oc new-project myapache
2. Use the new-app command to pull the fedora/apache and have it deploy it (please note the IP address the service receives, as you will be using it later).

   $ oc new-app fedora/apache

   ```
   imagestreams/apache
   services/apache
   deploymentConfigs/apache
   Service "apache" created at 172.30.141.9:80 to talk to pods
   over port 80.
   ```

   This may take a while to complete depending on the speed of your connection since the Docker image for fedora/apache has to be downloaded and run. You can check on the status of this download and deploy by either going into the web console -> select project myapache -> Browse -> Pods. Under status you will see pending with the arrows circling (rather than running).



Or you can use the oc command line tools and keep checking on the pod status:

$ oc get pods --watch

```
POD                 IP           CONTAINER(S)    IMAGE(S)         HOST
      LABELS                                          STATUS CREATED
apache-1-s79ay   172.17.0.7   apache        fedora/apache
openshiftdev.local/127.0.0.1
deployment=apache-1,deploymentconfig=apache    Pending    3 minutes
```

If it completes almost immediately it is because we cached the Docker image for fedora/apache in the VMs Docker image cache. So upon doing this command Openshift just spins it up.

A long download only happens the first time someone creates a service with that Docker image on that OpenShift node. After that if someone asks for that image again and their pod is on the same node, then OpenShift will check to make sure if the image is the latest. If it is not then it will pull it again, otherwise it will reuse the one in the node's local docker image cache.

1. vagrant ssh into the box and curl the URL

   $ vagrant ssh # This SSHs you into the VM

   ```
   Last login: Thu Apr 30 05:23:39 2015 from 10.0.2.2
   [vagrant@openshiftdev ~]$ curl 172.30.141.9 # This IP is the IP
   of the Apache service - it will probably be a different octet
   in your case
   Apache
   [vagrant@openshiftdev ~]$ logout #Logout of your SSH session in
   the VM
   ```

Note: If you are using Windows, you may not have SSH available on the command line. If this is the case, use the onscreen instructions to configure your SSH client to connect to the vagrant box.

WINNING! These are the only commands you need to run to get a "vanilla" docker image deployed to OpenShift 3. This should work with any docker image that follows best practices, such as defining an ***EXPOSE*** port and a ***CMD*** to execute on start. You may be wondering why this doesn't work in a web browser? That is because by default we do not expose services or create routes by default. Don't worry though, we will cover that later in this lab.

Note: The "new-app" command currently only creates a service for the first EXPOSED port in the Docker image.

Background: Services

You can see that when we ran the new-app command, OpenShift actually created several resources behind the scenes in order to handle deploying and running this Docker image. First, it made a Service, which identifies a set of pods that it will proxy and load balance. Services assign an IP address and port pair that, when accessed, redirect to the appropriate back end (pods).

The reason you care about services is they basically act as a proxy/load balancer between your pods and anything that needs to use the pods that is running inside the OpenShift environment. For example, if you needed more Apache HTTPD servers to handle the load, you could spin up more Apache HTTPD pods behind the service and the incoming requests to the service would not notice anything different except that the service was now doing a better job handling the requests.

There is a lot more information about services, including the JSON format to make one by hand, in the online documentation. Let's take a quick look at the JSON for this service here:

$ oc get services apache -o json


```
{
        "kind": "Service",
        "apiVersion": "v1beta3",
        "metadata": {
        "name": "apache",
        "namespace": "myapache",
        "selfLink": "/api/v1beta1/services/apache?namespace=turbo",
        "uid": "e217f200-ef7d-11e4-bd48-080027c5bfa9",
        "resourceVersion": "164",
        "creationTimestamp": "2015-04-30T21:14:27Z"
        },
        "spec": {
        "ports": [
```

```
        {
                "name": "apache-tcp-80",
                "protocol": "TCP",
                "port": 80,
                "targetPort": 80
        }
        ],
        "selector": {
        "deploymentconfig": "apache"
        },
        "portalIP": "172.30.156.66",
        "sessionAffinity": "None"
        },
        "status": {}
}
```


Let's also get the json for the pod so we can see how OpenShift chose to wire them together.

$ oc get pods -o json


```
{
        "kind": "List",
        "apiVersion": "v1beta3",
        "metadata": {
        "resourceVersion": "17932"
        },
        "items": [
        {
        "kind": "Pod",
        "apiVersion": "v1beta3",
        "metadata": {
                "name": "apache-1-s79ay",
                "generateName": "apache-1-",
                "namespace": "myapache",
                "selfLink": "/api/v1beta1/pods/apache-1-s79ay?namespace=turbo",
                "uid": "e317d3be-ef7d-11e4-bd48-080027c5bfa9",
                "resourceVersion": "17931",
```

```
            "creationTimestamp": "2015-04-30T21:14:28Z",
            "labels": {
                  "deployment": "apache-1",
                  "deploymentconfig": "apache"
            },
          ……..
```

Note: the above output has been truncated due to space considerations in this lab guide.

The service has a key:value selector -- “deploymentconfig:apache”. Any pod who has a label key “deploymentconfig” whose value is “apache” will be matched by this service, and that pod will be listed in the service’s endpoint table.

To see this in action, you can use the following command:
$ oc describe service apache

Let’s go ahead and learn about Deployment Configurations and Replication Controllers.

Lab 3: Scale to 3 Pods

Background: Deployment Configurations and Replication Controllers

While services provide routing and load balancing for pods which may blink in and out of existence, ReplicationControllers (RC) are used to specify and enforce the number of pods (replicas) that should be in existence. RCs can be thought of to live at the same level as Services but they provide different functionality above pods. RCs are a Kubernetes object.

OpenShift provides a “wrapper” object on top of the RC called a Deployment Configuration (DC). DCs not only include the RC but they also allow you to define how transitions between images occur as well as post-deploy hooks and other deployment actions. In general in

OpenShift you should interact with OpenShift objects. We don't prevent you from interacting with the Kubernetes objects but we put our objects there for a reason. The two native Kubernetes objects we did not wrap are Pods and Services.

In OpenShift it is possible to have:

1. Just a pod
2. Pod(s) + a separate RC/DC - good if you want scale up a bunch of jobs that have no inbound requirements, like fetching messages from a message queue and putting them in a database.
3. Just a pod + a separate Service - good if you want to make sure anything that wants to communicate with a pod doesn't have to hardwire an IP for individual pods
4. Pods + a separate Service + a separate RC/DC - this is what was created when we used 'oc new-app docker/image' to import the Docker image. This is the recommended configuration in most scenarios.

Exercise 2: Scaling up

Based on this information we can now look at the DeploymentConfig that was created when we imported our Apache image:

$ oc get dc

```
NAME      TRIGGERS                    LATEST VERSION
apache    ConfigChange, ImageChange   1
```

```
$ oc get dc apache -o json
{
   "kind": "DeploymentConfig",
   "apiVersion": "v1",
   "metadata": {
       "name": "apache",
```

```
        "namespace": "sample",
        "selfLink": "/osapi/v1beta3/namespaces/sample/deploymentconfigs/apache",
        "uid": "4b066875-1a02-11e5-8818-080027c5bfa9",
        "resourceVersion": "1545",
        "creationTimestamp": "2015-06-23T23:48:05Z"
    },
    ……..
```

Note: Truncated due to space considerations in this document.

$ oc get rc apache-1 -o json

```
{
        "kind": "ReplicationController",
        "apiVersion": "v1beta3",
        "metadata": {
        "name": "apache-1",
        "namespace": "turbo",
        "selfLink": "/api/v1beta1/replicationControllers/apache-1?namespace=turbo",
        "uid": "e22caa09-ef7d-11e4-bd48-080027c5bfa9",
        ……..
```

Note: Truncated due to space considerations in this document.

Given the above output, you should now understand how this controls the number of replicas. Even though RCs control the scaling of the pods, they are wrapped in a higher construct, DeploymentConfig, which also manages when, where, and how these Pods/RCs will be deployed.

Ultimately, OpenShift 3's autoscaling will involve monitoring of the status of an "application" and then manipulating the DCs accordingly. At the time of this writing, OpenShift 3 supports manual scaling (or through the API) and is incredibly easy. Let's scale our Apache Pod up to 3 instances. We need to do this through the deployment configuration :

$ oc scale --replicas=3 dc apache

```
$ oc get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)               SELECTOR
REPLICAS
apache-1     apache         fedora/apache:latest
deployment=apache-1,deploymentconfig=apache   3
```

```
$ oc get pods
NAME             READY     REASON    RESTARTS   AGE
apache-1-ah6e1   1/1       Running   0          14s
apache-1-czhz9   1/1       Running   0          14s
apache-1-s6k58   1/1       Running   1          1d
```

That's how simple it is to scale up your pods in a service. If you are also watching the web UI, you will see 2 new pods, show up Pending (grey) then Running (green). This scaling happens so fast that the changes will happen before you can switch screens or type the commands. The rapidness of this command is because all we are doing is just spinning up Docker containers that are already cached after the first deployment.

Lab 4: Expose as a URL

Background: Routes

By default, new-app assumes you do not want to expose this new service to the outside world. If you want expose a service as an HTTP endpoint you can easily do this with a route. OpenShift global router uses the *host* HTTP header to determine where to send the request. You can optionally define security, such as TLS, for the route.

Exercise 3: Adding a route

In our case, since we are working on localhost without a DNS server, we are going to trick the server by inserting a host header using the browser tools above. We use the 'oc expose' command:

./$ oc expose service apache --hostname=www.apache.com

```
NAME      HOST/PORT         PATH      SERVICE   LABELS
apache    www.apache.com              apache
```

Now in the browser header update please add a mapping for www.apache.com to localhost. Refer back to earlier in the exercises to see how to update the plugin to add the new host entry. You can also edit your hosts file and do that as well. Now go to http://localhost:1080 in your browser and you should see the word `Apache`.

Note: The reason we are using port 1080 instead of 80 for this lab is because we make the assumption that you may already have a service running on port 80 of your machine. For this reason, we have mapped to port 1080 to avoid any potential conflicts. You can see this mapping happen when after you fire the 'vagrant up' command. If you want to change this mapping you can edit your Vagrantfile and change the port mapping and restart your VM using Vagrant (vagrant halt; vagrant up) for the changes to be effective.

Lab 5: Deploy Python Code and a Docker Image

Background: Source to Image

We have seen how to deploy a Docker image, let's see how we can work with source code and builds with Docker images. We can use the new-app command to do a simple deploy of code with a docker image. The OpenShift teams have built some Docker images that are enabled for a more generic build mechanism, called Source-To-Image.

Source-To-Image (S2I) is another Open Source project sponsored by Red Hat. It's goal is to provide a simple, efficient, and stand-alone mechanism to add source code to a Docker image

and produce a Docker image that can run as-is. OpenShift is S2I enabled and can use it as one of it's build mechanisms (in addition to Docker build and custom build). A full discussion of S2I is beyond the scope of this class. However, if you would like to learn more about this project, please read the documentation.

All the programming language images in the OpenShift portion of the Docker Hub registry are S2I enabled.

Create the application

Start by creating a new project for just this code example:

```
$ oc new-project mycode
```

Fork application code on Github

OpenShift can work with git repositories in GitHub and if our "server" was accessible from the outside world, we could actually register webhooks to rebuild on any update to Github. We built a simple Python application with the Bottle framework and put it on Github.

Go ahead and fork the repo into your own Github account. Later in the lab, we want you to make a code change and then rebuild your application. If you know Python applications you will see that there is nothing special in our application - it is a standard, plain-old Python application.

Combine the code with the Docker image on OpenShift

The new-app command makes it very easy to combine a Github repository with a Docker image. Now that you have your own Github repository let's use it with OpenShift's Python S2I image.

Again, we use the new-app command:

$ oc new-app openshift/python-33-centos7~https://github.com/thesteve0/v3simplebottle.git

```
imagestreams/python-33-centos7
imagestreams/v3simplebottle
```

```
buildconfigs/v3simplebottle
deploymentconfigs/v3simplebottle
services/v3simplebottle
A build was created - you can run `oc start-build v3simplebottle` to start it.
Service "v3simplebottle" created at 172.30.72.255 with port mappings 8080.
```

Replace the part after the ~ with your github repository's URL. The ~ in this command acts as a shortcut to say "take this Docker image and combine it with this source code repository".

Now you should SSH into your OpenShift instance and then curl the ip and port mapping above.

$ vagrant ssh

```
Last login: Thu Jun 18 21:35:31 2015 from 10.0.2.2
[vagrant@openshiftdev ~]$ curl 172.30.72.255:8080
hello OpenShift Ninja
```

Create a route

Now let's go ahead and expose this service with a route, just as you did for the Apache application earlier.

```
$ oc expose service v3simplebottle --hostname=simple.example.com
```

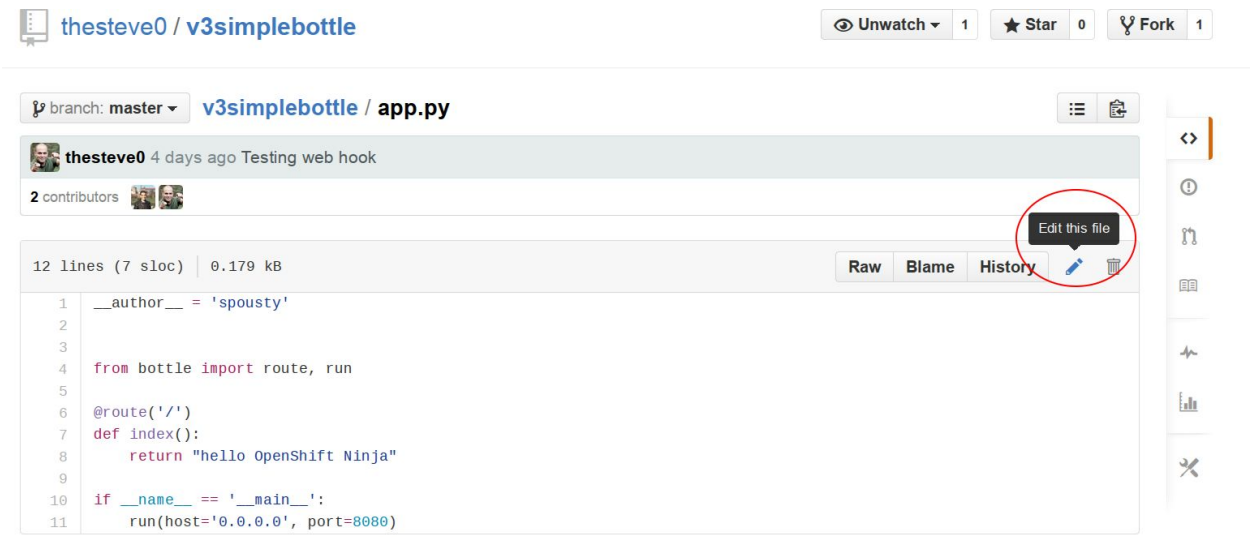
Go ahead and update the header plugin for your browser to set the host header to simple.example.com. Since we did not set up TLS with this route remember to hit http://localhost:1080.

Expose was "smart" enough to reroute the requests coming in over port 80 to the 8080 on the service. The next step is to make code changes and have them deployed to our application.

Code and redeploy

With OpenShift, teams or people who like working on Github will be really happy. All we need to do is update the code on Github and fire off a new build.

Go ahead and change the app.py file right on Github. Change the output text to say something new:



Then all we need to do is fire off a new build to get the change. This build should happen faster than the last one.

```
$ oc start-build v3simplebottle
```

You can see the build status by either going to the web page and watch the builds or do:

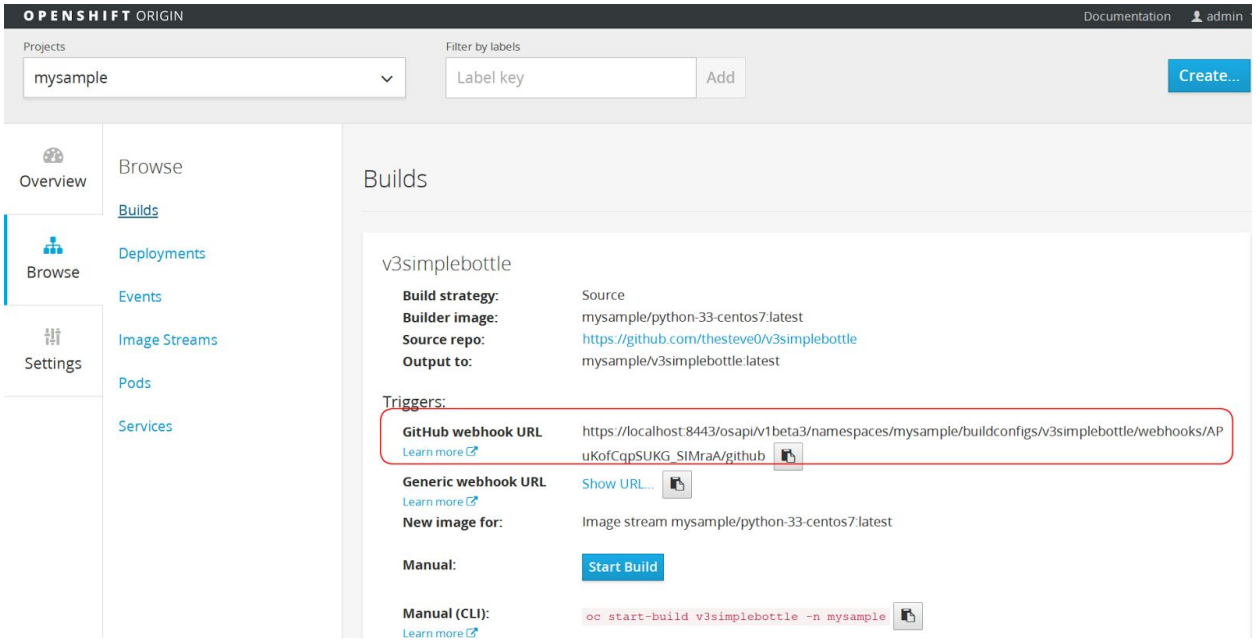
```
$ oc get builds --watch
```

When the build finishes you can look at the web page in your browser again and you should see the change. Winning!

Github webhooks

If our OpenShift server was visible on the internet, rather than just running on our local computer, we could have set a Github webhook to fire a build every time we commit our changes in Github.

In the OpenShift web console, under the builds for mysample you will see a URL for a Github webhook



You just need to add this to your Github repo (under the “Webhooks and Services” item for your repository) for this code and then you get automated builds on commit. Double Winning!

Lab 6: Add Postgresql to the Application

Many application go beyond just having a web server and involve some sort of database or other data storage. In this next example we are going to add Postgresql to our project and then rewire our application to talk to the database.

We are going to use the Postgresql Docker image produced by the OpenShift team - https://github.com/openshift/postgresql

By default, this will use the *EmptyDir* for data storage, which means if the pod disappears the data does as well. In a real application you would use a template to request a permanent volume for the database pods to use for their data storage. We will cover a template file, a file specifying the infrastructure and repositories used in your application, at the end of the lab.

When we use the new-app command this time we need to pass in some environment variables to be used in the container. These environment variables are required to set the username,

password, and name of the database. You can change the values of these environment variables to anything you would like:

```
$ oc new-app openshift/postgresql-92-centos7 -e POSTGRESQL_USER=user -e
POSTGRESQL_DATABASE=db -e POSTGRESQL_PASSWORD=steveisgreat
```

Again you can use the web console or the oc command to watch the progress of this command.

```
$ oc get pods --watch

NAME                            READY  REASON       RESTARTS   AGE
postgresql-92-centos7-1-btiug   1/1         Running      1           15m
v3simplebottle-1-build          0/1    ExitCode:0    0           1h
v3simplebottle-2-build          0/1    ExitCode:0    0           25m
v3simplebottle-3-l3uxj          1/1    Running      0           25m
```

Using the PSQL command line utility in the container

To interact with our database we will use the OpenShift exec command which allows us to run arbitrary commands in our pods. In this example we need to use bash because we need to activate the software collection library that gives CentOS a newer version of PostgreSQL.

Note: Make sure to use the correct pod name for your postgresql container.

```
$ oc exec -tip postgresql-92-centos7-1-btiug -- bash -c psql
psql (9.2.8)
Type "help" for help.

postgres=# select * from pg_user;
usename  | usesysid | usecreatedb | usesuper | usecatupd | userepl |  passwd  |
valuntil | useconfig
----------+----------+-------------+----------+-----------+---------+----------
+----------+-----------
postgres |        10 | t            | t         | t         | t   | ******** |
      |
user  |     16384 | f          | f          | f         | f   | ******** |
|
(2 rows)
```

```
## the following command quits out of PostgreSQL. Since we also fired off exec
## with the psql command this will also exit the shell.
postgres=# \q
```

Making the Python Container work with the PostgreSQL container

By being in the same project our two services will share information about IPs and ports that are exposed. To see this you can use os exec to open a shell session in the Python pod and get the environment variables

Note: Make sure to use the correct pod name for your container.

```
$ oc exec -tip  v3simplebottle-9-g8jgm -- bash|grep POST
POSTGRESQL_92_CENTOS7_PORT=tcp://172.30.42.227:5432
POSTGRESQL_92_CENTOS7_SERVICE_HOST=172.30.42.227
POSTGRESQL_92_CENTOS7_SERVICE_PORT=5432
POSTGRESQL_92_CENTOS7_PORT_5432_TCP_PROTO=tcp
POSTGRESQL_92_CENTOS7_PORT_5432_TCP=tcp://172.30.42.227:5432
```

But we are missing information to make the connection. We need to add the environment variables for the user and the password from the Postgresql pod to the environment variables in the Python pod.

$ oc env dc/v3simplebottle -e POSTGRESQL_USER=user -e POSTGRESQL_DATABASE=db -e POSTGRESQL_PASSWORD=steveisgreat

This command will also trigger a new deploy with the added environment variables.

Check that these environment variables were applied to the DeploymentConfig and will be available on the new POD

$ oc describe dc v3simplebottle

Adding these environmental variables to the pods causes the images to redeploy so that the environment variables are available at the system level. Once the redeploy is done you can re-exec our ‘env | grep POST’ command and you should see the new environment variables. Note that you need to specify the new POD ID.

```
$ oc exec -ti  v3simplebottle-9-a8xom env| grep POST
```

I modified the application and made a new Github repository with the code:

https://github.com/thesteve0/v3simplebottle-db

There are two ways you can update your code to take advantage of the DB.

1. You can copy and paste the code I put into a new repository that uses the Pyscopg2 as the database connector. You will need to update requirements.txt and app.py
2. The niftier way to do it just change the repository in the BuildConfig (bc) for the Python project. We haven't talked about Build Configurations yet but the quick story for them is they specify the build for your application and containers. It is referenced in the Deployment Configuration.

$ oc get bc
NAME TYPE SOURCE
v3simplebottle Source https://github.com/thesteve0/v3simplebottle.git

Now we can just edit the bc directly, save the change, and fire off a new build and you should be golden.

```
$ oc edit bc v3simplebottle

# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this
file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: BuildConfig
metadata:
 creationTimestamp: 2015-06-24T23:39:28Z
 name: v3simplebottle
…
spec:
 output:
```

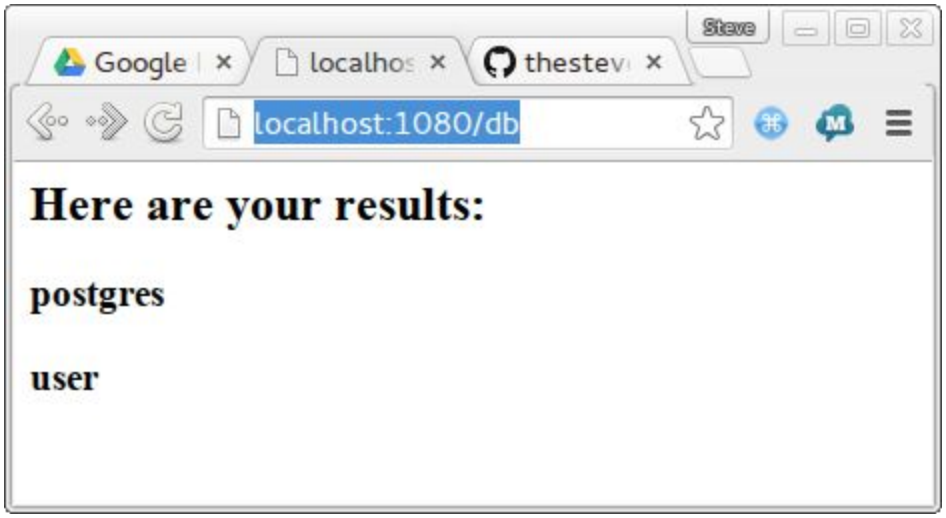
```
    to:
      kind: ImageStreamTag
      name: v3simplebottle:latest
  resources: {}
  source:
    git:
      uri: https://github.com/thesteve0/v3simplebottle-db.git
    type: Git
…
```

```
Note: On Windows, this should open notepad. On Mac and Linux machines this
should open VI. If you are not familiar with VI commands, can hit move the
cursor around as you normally would. When you want to start editing you can
type "i" for insert or "a" append. Make your changes. Then hist "esc" to exit
out of editing mode. The type "wq" and hit enter.
```

Once you update the value of the github repo. to the new repository, you can just fire off a new build:

```
$ oc start-build v3simplebottle
v3simplebottle-15
```

When it's finished, you need to go to a new URL to see the new output. If you look in the source code you can see that we do the DB call in a function that maps to the URL "/db". Please go to http://localhost:1080/db URL in your browser and witness the following:



Lab 7: Walk through a full template

Deleting Resources

Before we move on to other work, let's go ahead and clean up some of the resources on our OpenShift instance. The easiest way is to just delete the whole project.

$ oc delete project myapache

This gets rid of all the objects and resources used in the project.

To delete a pod(s) you actually need to the DeploymentConfig first. if you haven't deleted the myapache project you can just do this:

$ oc delete dc apache

This will delete the DeploymentConfig which will also remove the Replication Controller and the pods. You will still need to delete the services, image streams, and routes using the same general syntax:

$ oc delete <object type> <object name>

Using a template

Doing all these individual commands to create OpenShift objects can be tedious and error prone. You can actually put all of this configuration together into a template file which can then be processed to create a full set of services. In a template you may have parameters for certain values, such as DB username or password and they can be automatically generated at processing time.

Templates can actually be loaded on the server and then they will be available in the web console to chose when creating a new application. In order to add this template for anyone with access to the project, download the template to your local filesystem and then execute the following command ensuring that you point to the location of the template file and replace *mytemplate* with the correct name of your project:

$ oc create -f application-template-stibuild.json -n mytemplate

As our final exercise let's see the template that was used to created the application for our smoke-test of our setup.

https://github.com/openshift/origin/blob/master/examples/sample-app/application-template-stibuild.json

In order to add this template for anyone with access to the project, download the template to your local filesystem and then execute the following command ensuring that you point to the location of the template file and replace *mytemplate* with the correct name of your project:

$ oc create project template
$ oc create -f application-template-stibuild.json -n mytemplate