

# Application development with Docker, Kubernetes, and OpenShift

OpenShift Evangelists

# Slides Used in Class

[Slides](#) we usually and these workshop notes are released under [Creative Commons License](#) allowing free use as long you provide attribution.

## Application development with Docker, Kubernetes, and OpenShift

### Welcome

Greetings and welcome to tutorial for using the all-in-one VM (A1VM). By the time you have finished this tutorial you will have gotten your hands nice and dirty playing with OpenShift. Our overall goals for this document is:

1. Introduce you to running applications in containers
2. Give you enough information to launch into learning more OpenShift

### FYI

- You can keep using this VM to play with
  1. Docker
  2. Kubernetes
  3. OpenShift
  4. PostgreSQL with some advanced features
- This VM is not for production use and is community supported. You can go to the following places for more help
  1. StackOverflow and tag with OpenShift
  2. [The Google Group](#)
  3. The #openshift or #openshift-dev channels on freenode IRC

# Installing the Vagrant Box

## Preface

We build this all-in-one from a specific Github repository.

<https://github.com/openshift-evangelists/vagrant-origin>

You can always build an all-in-one using the scripts and instructions in that repository. Those instructions will allow you to build from any origin repository or branch. This is a great way to play deeply with the source code or try out the head of the repository without waiting for us to do a build. You can also turn on or off different features depending on what you do in the configuration files. One example that the OpenShift Evangelists do with those scripts is to create fully loaded VMs for workshops with all the Docker images pre-downloaded.

With that said, 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 install.

1. Install [VirtualBox](#). Vagrant doesn't work with the 5.1 versions
2. Install [Vagrant 1.8.4](#). There is a nasty SSH bug in the 1.8.5 and various other issues.

You can try other combinations of this software if you want to try but we only certify Vagrant 1.8.4 and VirtualBox 5.0.latest. If all else fails go with this combination.

VirtualBox is a requirement. We know some of you have parallels and vmware and other virtualization tech but we only built the image for VirtualBox. Our 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. Every time 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 to 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 exercises.

1. [GitHub account](#), if you don't have one

## Using the box on Atlas

On your machine make a directory called *os\_workshop*. In that directory do the following commands:

```
C:\_os_workshop> vagrant init openshift/origin-all-in-one  
C:\_os_workshop> vagrant up --provider=virtualbox
```

There is a 2.53 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.

They will always be on the Origin latest release page

<https://github.com/openshift/origin/releases>

We are using 1.3.0 release of OpenShift Origin.

---

## 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 you see this then you have installed everything properly.

If you break the installation you just need to do the following steps:

```
C:\_os_workshop\vagrant>vagrant destroy --force
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 you 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 up a modern application.

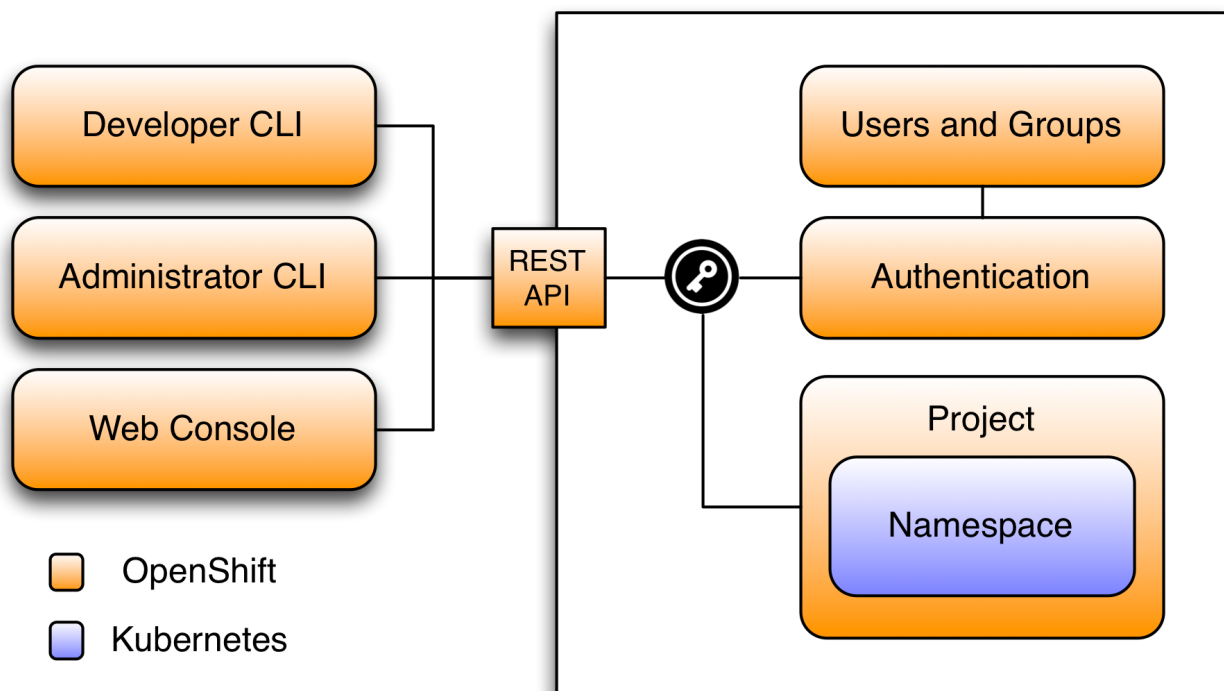
To accomplish this there are required pieces to the platform. We use the Docker container engine to handle just the running of the containers. Above that there is Kubernetes, the Open Source project for orchestrating, scheduling, and other tasks for actually building real applications out of Docker containers.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



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.

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
```

## TIP

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 password. 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

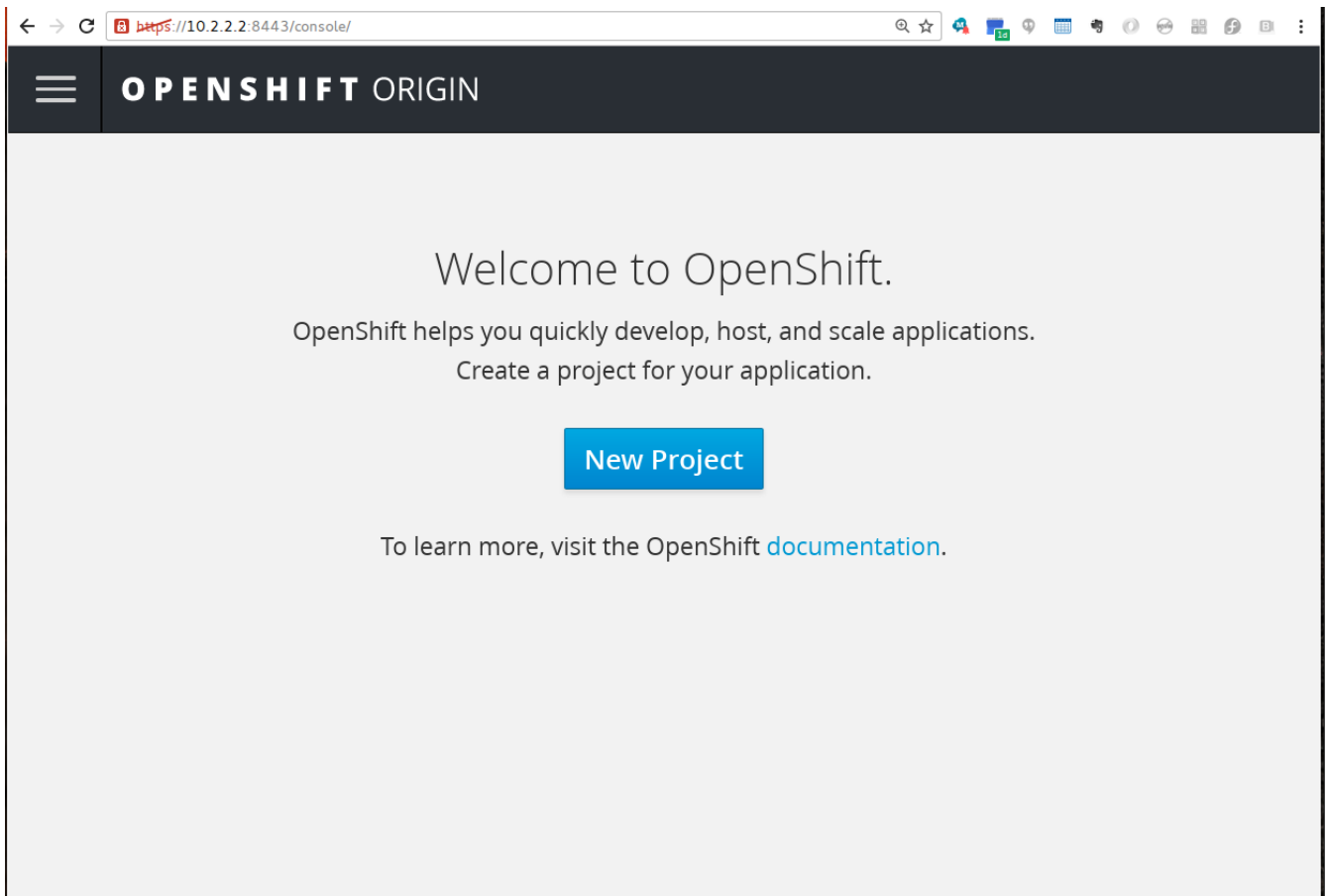
```
Login successful.

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 problem 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"

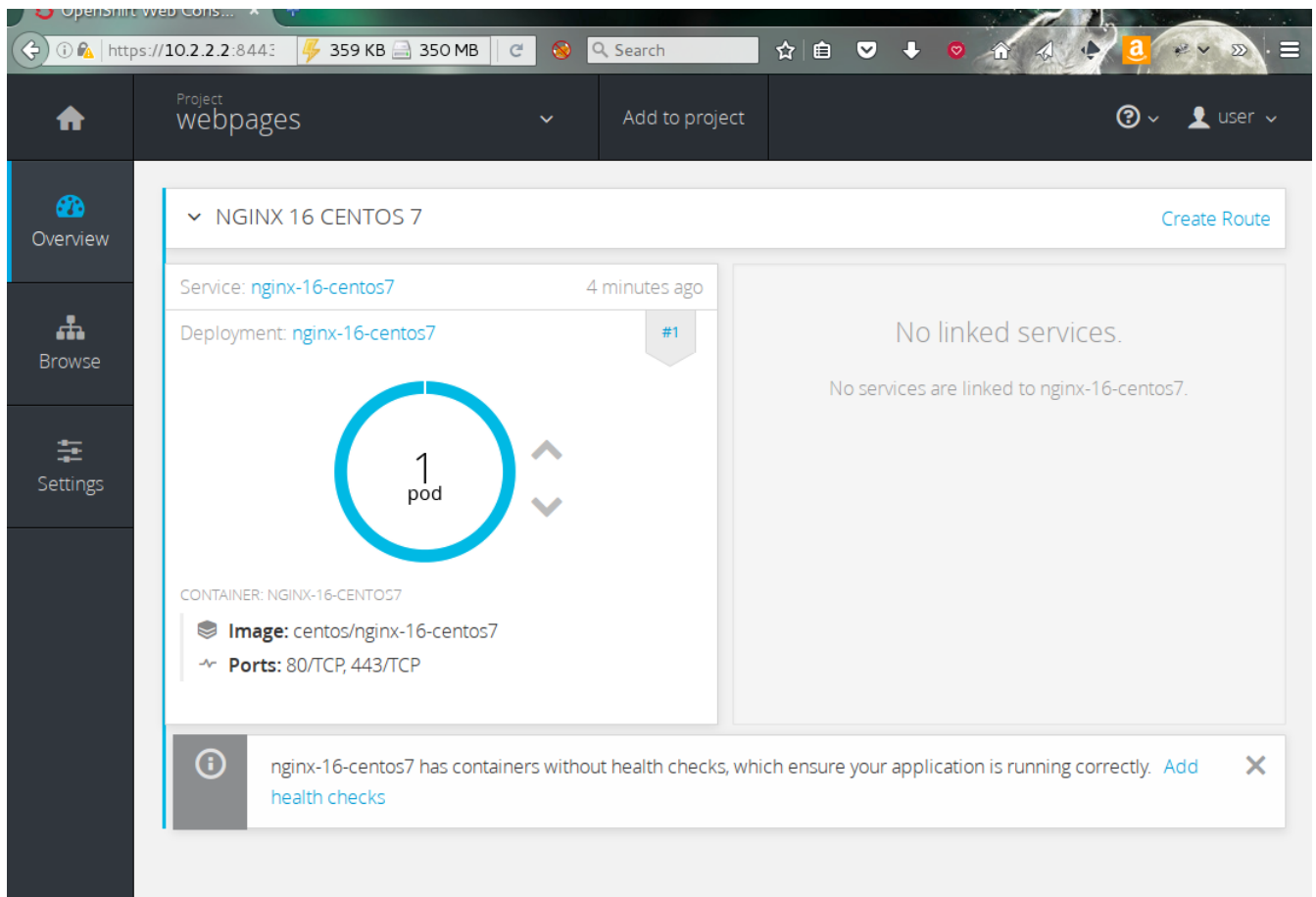
    * An image stream will be created as "nginx-16-centos7:latest" that will track this image
    * This image will be deployed in deployment config "nginx-16-centos7"
    * Ports 443/tcp, 80/tcp will be load balanced by service "nginx-16-centos7"
    * Other containers can access this service through the 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 your 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):



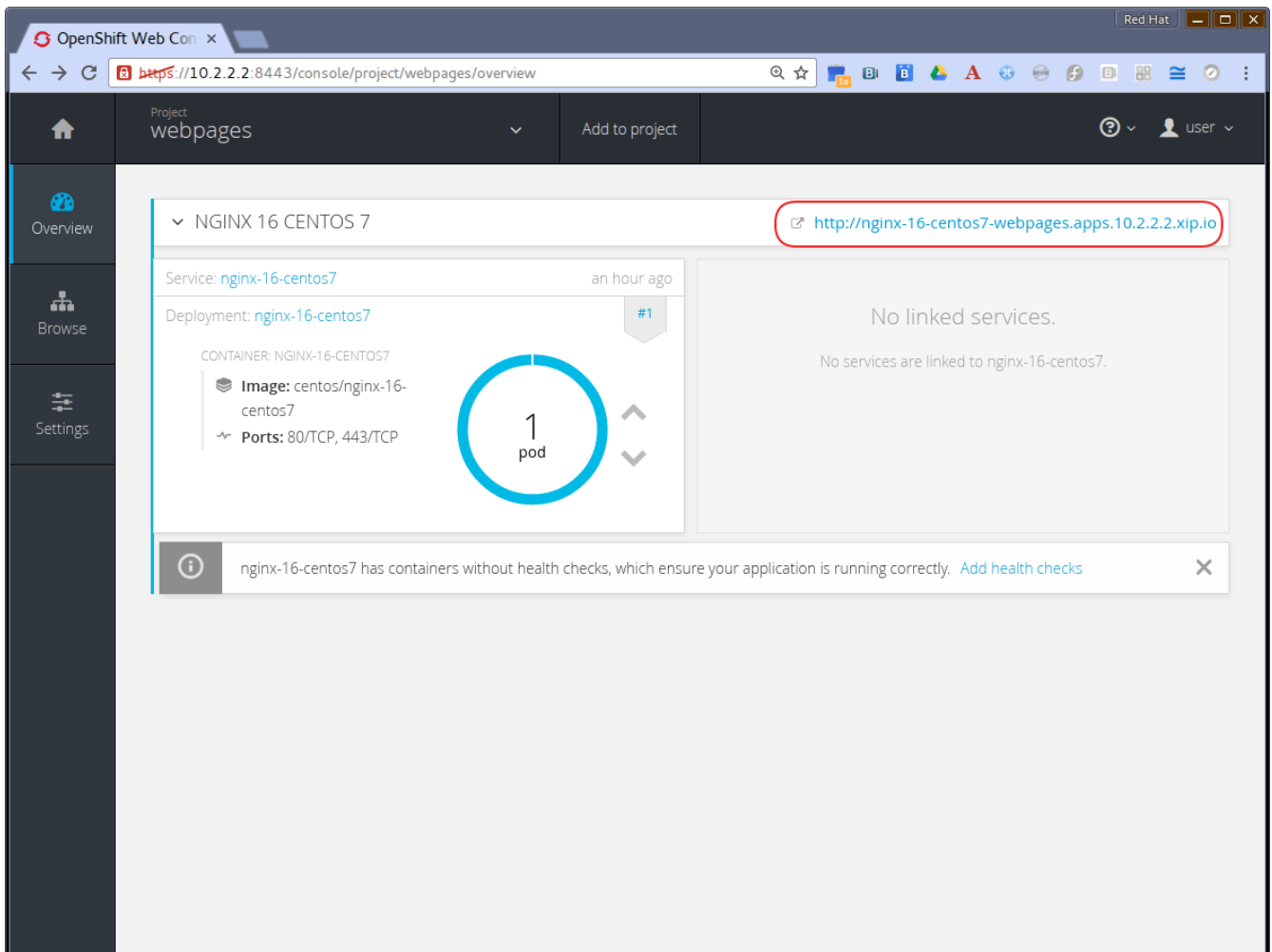
Now is a good time to reflect back on the [the architecture diagram](#) I explained earlier and think about the objects created above. Start from the pod and work our way up to *Services* and *Deployment Configurations*

## 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.

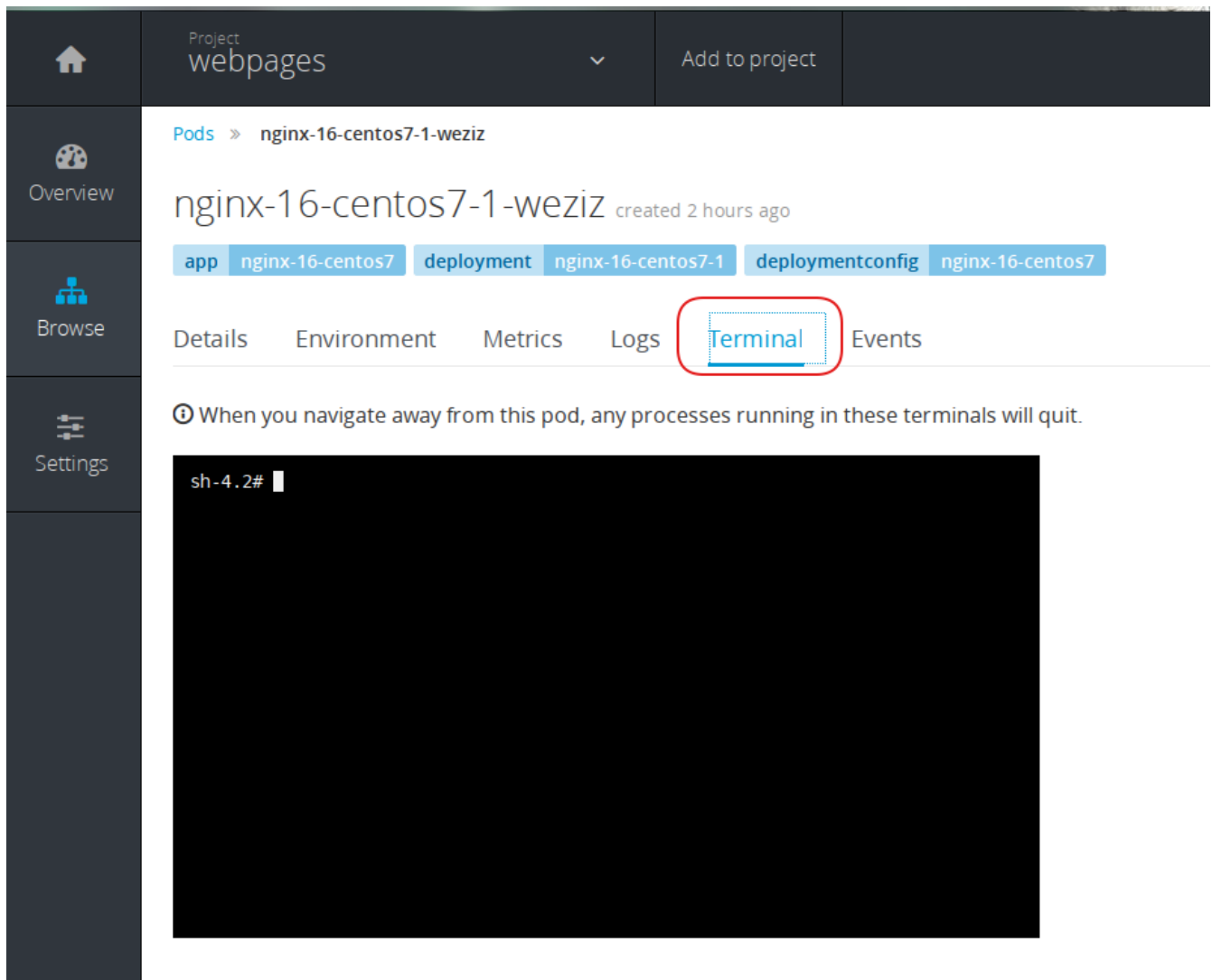
When you get back to the overview screen you will now have a URL up in the top of the box, go ahead and click (but be ready for sadness).



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.

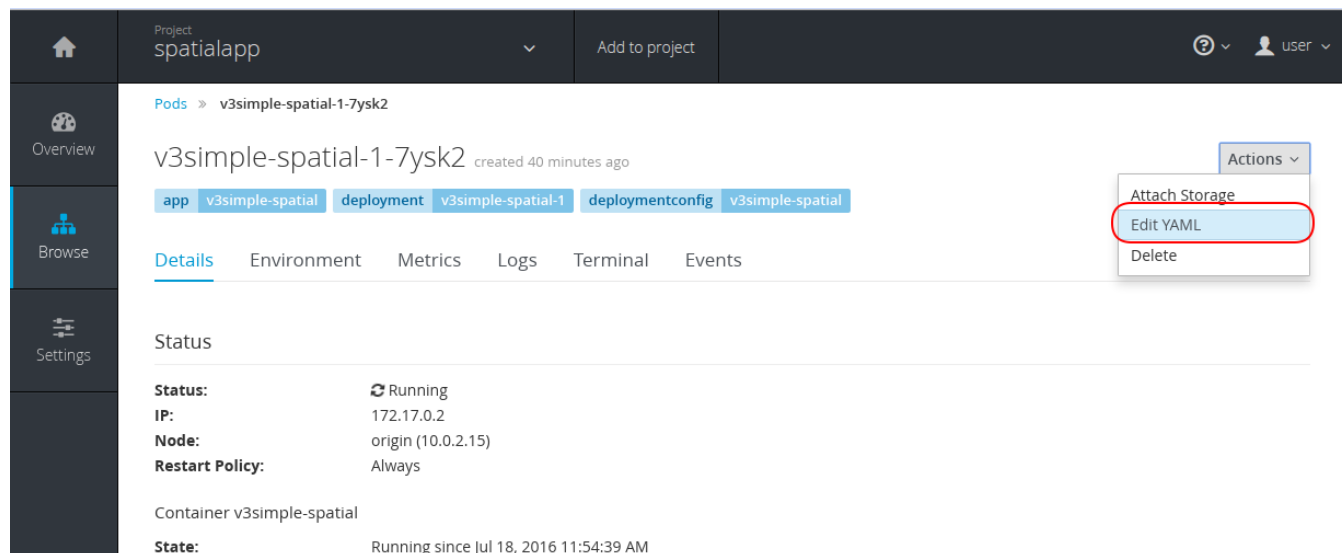


# Templates and OpenShift/Kubernetes Declarative Syntax

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 lets 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 namespace 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 parameterized 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.

2. Create a new project to import our project into

```
oc new-project newproject
```

3. 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
```

4. 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
```

5. 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!

# Adding a Database

Now it's time to get to the part of the workshop you have all been waiting for - getting PostGIS up and running!

## Making a project

Let's make a new project for this application:

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

That's it - you now have a project "spatialapp" and all the subsequent commands will affect this project until you switch to another project.

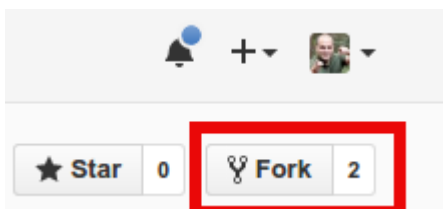
## Preparing your GitHub repository

The only reason we are working with the GitHub repository here and not in the code pieces is because we need to get the DDL files to our machine. If we weren't using these DDL files you could skip right to the "Generating Database Pieces" section.

The source code for this project can be found here:

<https://github.com/thesteve0/v3simple-spatial>

Now **BEFORE** you go and *git clone* it down to your machine, you need to fork it. See that button up towards the top right that says fork on it - click it (you did login with your github account right?)



Now you should be looking at URL that replaces *thesteve0* with your github userid:

[https://github.com/<your\\_userid>/v3simple-spatial](https://github.com/<your_userid>/v3simple-spatial)

This repository is set up for several different learning paths and so it has several different versions of `app.py` (the file that runs in a web app for Python). We need to rename the current `app.py` to `1_app.py`. It's very easy to do from the web interface:

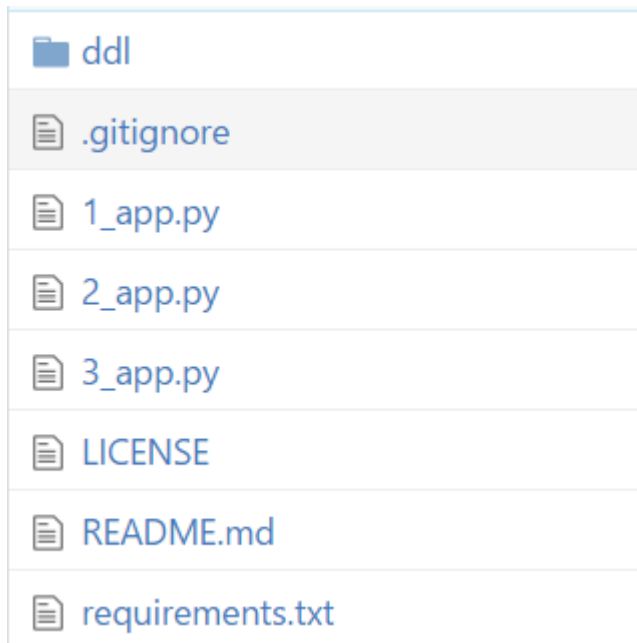
1. click on the `app.py` file
2. click on the pencil icon on the far right of the page
3. change the name of the file to `1_app.py` (a box right above the edit file box)
4. Scroll down to the **Commit changes** area on the bottom of the page and enter a message

5. Hit the Commit Changes button

There, you just changed the name of the file in your repository.

Now go back to the top level of your repository and follow the same process to rename *3\_app.py* to *app.py*.

In the end your repository should look this:



Go ahead and clone this repository to your machine.

## Generating the Database Pieces

In the terminal, please go to the location on your machine where you cloned the Crunchy Solutions repository. Go to the `|examples|openshift|workshop` directory. We are going to use a bunch of files in this directory but let's go ahead by creating our master instances:

```
> oc new-app -p CCP_IMAGE_TAG=centos7-9.5-1.2.2 -f master.json
> oc new-app -p CCP_IMAGE_TAG=centos7-9.5-1.2.2 -f replicas.json
```

That's it - thanks to the work by Jeff you now have a master-replica PostGIS database setup. You can see it in the web overview for the project now. One of the cool things besides the replica is Jeff added some other containers to the master pod (log collection and metric collection). I will talk through the pieces in class.

## Loading the Database

Let's go ahead and load up the database in the master with some DDL. In the `v3simple-spatial` repository <https://github.com/thesteve0/v3simple-spatial.git> there is already a SQL file with all our DDL statements. Please clone the repository and then change into the root of the repository.

```
# we need to get the master pod - it will have a different name on your machine
```

```
> oc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
master	3/3	Running	0	13m
replica-dc-1-m46ri	1/1	Running	0	4m

```
#Look for the container named pg - we will need this below
```

```
> oc describe pods pgmaster
```

```
> oc rsync ./ddl master:/tmp/. -c pg
```

You may receive a warning that rsync is not found on your machine but the command line tool will fall back to other methods to try and copy the files over. We have now put the DDL file in */tmp/ddl* in the Master postgis pod.

Let's shell into that same pod and load the data (you can also do this from the terminal in the web browser but then you can use copy and paste)

```

> oc rsh -c pg master
sh-4.2$ psql -l -h 127.0.0.1
                                List of databases
  Name      | Owner   | Encoding | Collate | Ctype | Access privileges
-----+-----+-----+-----+-----+-----
 postgres   | postgres | SQL_ASCII | C       | C     |
 template0  | postgres | SQL_ASCII | C       | C     | =c/postgres      +
            |          |          |         |       | postgres=Ctc/postgres
 template1  | postgres | SQL_ASCII | C       | C     | =c/postgres      +
            |          |          |         |       | postgres=Ctc/postgres
 userdb      | postgres | SQL_ASCII | C       | C     | =Tc/postgres     +
            |          |          |         |       | postgres=Ctc/postgres+
            |          |          |         |       | testuser=Ctc/postgres
(4 rows)

sh-4.2$ psql -h 127.0.0.1 -f /tmp/ddl/parkcoord.sql userdb
CREATE TABLE
CREATE INDEX
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1...

sh-4.2$ psql -h 127.0.0.1 userdb
userdb=# select count(*) from parkpoints;
 count
--
    547
(1 row)

userdb=# \q

```

You have now loaded your database with a bunch of points for national parks in the US and Canada. The really amazing part comes next. Go ahead and go to the overview for the project. Go ahead and click on the circle for the *pg-slave-rc* which will bring you to a listing with a single pod. Go ahead and click on that link. On the page for the pod, click on the terminal tab:

The screenshot shows the Kubernetes dashboard interface. On the left is a sidebar with navigation links: Projects, Overview, Browse (selected), and Settings. The main area displays the details for a pod named 'pg-slave-rc-1-obd02', which was created 2 days ago. At the top, there are tabs for 'deployment', 'pg-slave-rc-1', 'deploymentconfig', 'pg-slave-rc', 'name', and 'pg-slave-rc'. Below these are tabs for 'Details', 'Environment', 'Metrics', 'Logs', 'Terminal' (highlighted with a red box), and 'Events'. The 'Status' section shows the pod is 'Running' with IP '172.17.0.11', node 'origin (10.0.2.15)', and a 'Restart Policy' of 'Always'. The 'Container server' section shows it is 'Running since Apr 25, 2016 12:28:25 PM' with 'Ready: true' and 'Restart Count: 0'.

On the resulting page you need to click twice on the terminal area to give it focus BUT you are now in terminal in the running pod - slick.

In that terminal go ahead and type the following commands:

```
sh-4.2$ psql -h 127.0.0.1 userdb
psql (9.5.2)
Type "help" for help.

userdb=# select count(*) from parkpoints;
count
---
547
(1 row)
```

Do you REALIZE what just happened. We entered data into the Master DB and it was automatically replicated over to the slave DB and did 0 work to make sure that would happen.

## Time for More Replication Magic

Let's take this to even another level. In the web console, go back to the overview again and then click on the little up arrow next to the slave pods:



The screenshot shows the Kubernetes Dashboard interface. On the left is a sidebar with navigation links: Projects, Overview, Browse, and Settings. The main area displays the 'Overview' for the 'Project spatialapp'. At the top, there's a 'Filter by label' input and an 'Add' button. Below this, two service cards are visible. The first card is for 'SERVICE pg-master-rc' with 'DEPLOYMENT: PG-MASTER-RC, #1', showing a blue circle with '1 pod'. The second card is for 'SERVICE pg-slave-rc' with 'DEPLOYMENT: PG-SLAVE-RC, #1', also showing a blue circle with '1 pod'. To the right of each pod circle are up and down arrow icons. In the 'pg-slave-rc' card, the up arrow icon is highlighted with a red rectangular box. To the right of the pod circles, a 'CONTAINER: SERVER' box lists 'Image: crunchydata/crunchy-postgres' and 'Ports: 5432/TCP'.

The number inside the circle will increment to 2 and then the blue circle will fill in the rest of the circle. You now have 2 replicas running. If you click on the circle again you will see the list of the two pods. If you click on the new pod and then do the commands above you will see that it has already been replicated to the new replica.

In the next section we will write an application to use the master and the replicas. Make sure you have cloned the v3simple-spatial repo. to the local machine.

# Beginning Your Application

Let's go ahead and start our own project using Python. I chose Python for this workshop because I thought it would be easiest for you folks to understand regardless of your coding background. The development pattern you are going to use today will apply to any language you use for development in OpenShift.

## Bringing in the Code

### Making a project

Let's make a new project for this application:

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

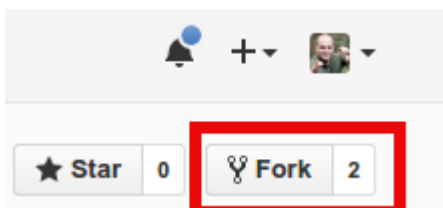
That's it - you now have a project "spatialapp" and all the subsequent commands will affect this project until you switch to another project.

### Adding Code and Doing a Build

The source code for this project can be found here:

<https://github.com/thesteve0/v3simple-spatial>

Now **BEFORE** you go and *git clone* it down to your machine, you need to fork it. See that button up towards the top right that says fork on it - click it (you did login with your github account right?)



Now you should be looking at URL that replaces *thesteve0* with your github userid:

[https://github.com/<your\\_userid>/v3simple-spatial](https://github.com/<your_userid>/v3simple-spatial)

In the middle of the page you will see a button that may say **SSH** on it. If it does, please change that to **HTTPS** and then click the clipboard to copy the URL in the box.



Our code is ready. Let's fire off the build and deploy. In your terminal enter the following command:

```
> oc new-app python@sha256:bb47c81~https://github.com/theesteve0/v3simple-spatial.git
--> Found image 87d749a (4 weeks old) in image stream "python" in project "openshift"
under tag "3.5" for "python:3.5"

Python 3.5
-----
Platform for building and running Python 3.5 applications

Tags: builder, python, python35, rh-python35

* A source build using source code from https://github.com/theesteve0/v3simple-
spatial.git will be created
* The resulting image will be pushed to image stream "v3simple-spatial:latest"
* This image will be deployed in deployment config "v3simple-spatial"
* Port 8080/tcp will be load balanced by service "v3simple-spatial"
* Other containers can access this service through the hostname "v3simple-
spatial"

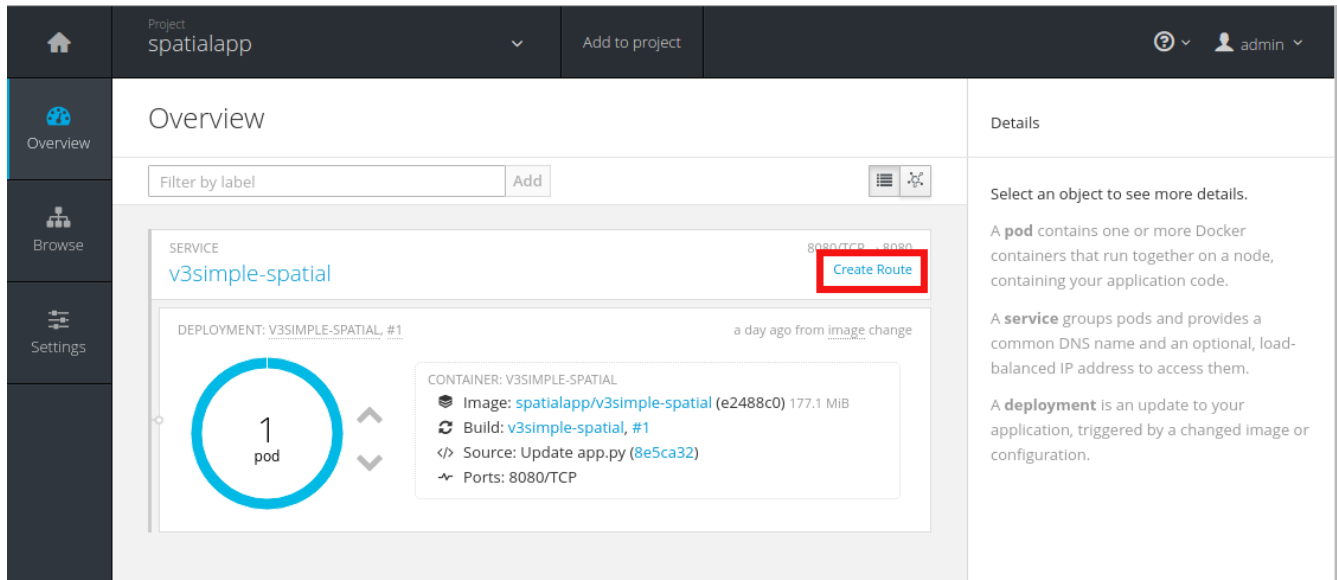
--> Creating resources with label app=v3simple-spatial ...
    imagestream "v3simple-spatial" created
    buildconfig "v3simple-spatial" created
    deploymentconfig "v3simple-spatial" created
    service "v3simple-spatial" created
--> Success
    Build scheduled, use 'oc logs -f bc/v3simple-spatial' to track its progress.
    Run 'oc status' to view your app.
```

What we just did is told OpenShift - take this [Docker Image](#) that knows how to build a standard Python application layout and combine it with this source code and produce a new Docker Image. As part of this process the *new-app* command knows we will need some other OpenShift objects to actually run that resulting image. The command goes ahead and makes those objects as well.

## Looking at what we built.

At this point we are going to switch back to the web terminal since it is easier to look at web sites in a browser. Go ahead and go into your browser and go to the projects page. Once you are looking at all your projects go ahead and click on *spatialapp*.

Your screen should look something like this:



You can see that we have the 1 pod running our image that was derived from our source code and our builder image. There is also other metadata in the box for the pod that we can return to later. For now I want you to click on the *Create Route* button that is highlighted in red. This will create a URL where we can see the web page for our pod. By default, nothing is exposed to the outside world and you have to choose to expose it.

Just click *Create* on the next page that comes up - all the defaults are fine for our us. You could have also done this at the command line with

```
> oc expose service v3simple-spatial
```

Now when you come back to the Overview page there is a URL for the service ending in .xip.io, go ahead and click it! You will be greeted by the following amazing web content:

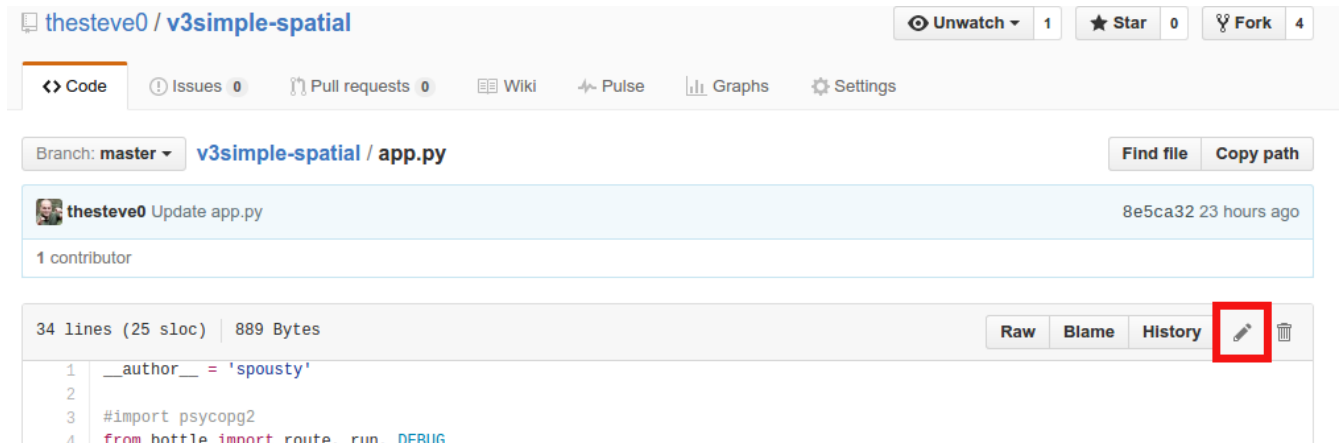
## hello OpenShift Ninja without DB

You have now all built and deployed containers with a working URL - give yourselves a pat on the back.

## Doing Code Changes With a Build

While that web page is AWESOME, you might actually want to change it. OpenShift gives you the ability to change the code in a "hot swap" fashion for rapid iteration development or in a more complete build cycle which creates a new image. For now we will do the more complete build cycle and the "hot swap" later.

Go ahead to your GitHub fork and edit the app.py file right there in GitHub. When looking at the GitHub repo., click on app.py and then on the next page click on the pencil icon in the top right.



Then on line 11 change the text to whatever you would like. I recommend something like "Steve is the best instructor EVER!" but your tastes may vary (be wrong).

After you are done with your change go ahead and scroll down the page. If you want to add comments or something to your commit go ahead, but everyone needs to press the big GREEN button. You have now committed your change to your GitHub repository.

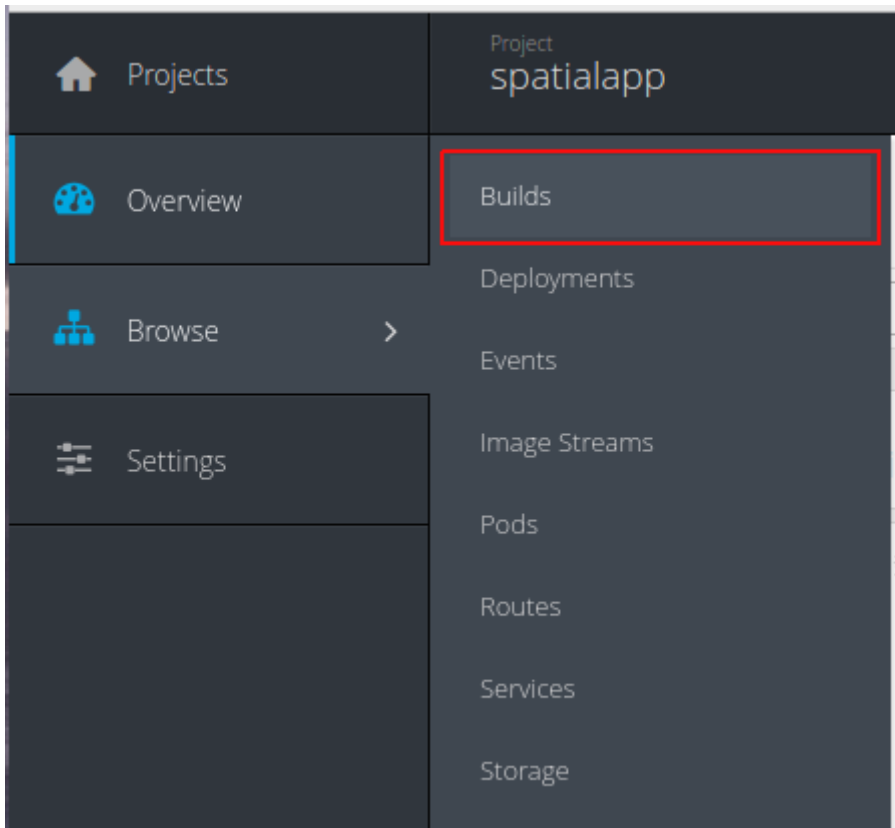
If our Vagrant image had an IP that was publically accessible, we could actually set up a web hook trigger in GitHub to tell OpenShift to build on commit to master. Instead we have to fire off the build manually. There are two ways to do this:

1) At the command line you can do:

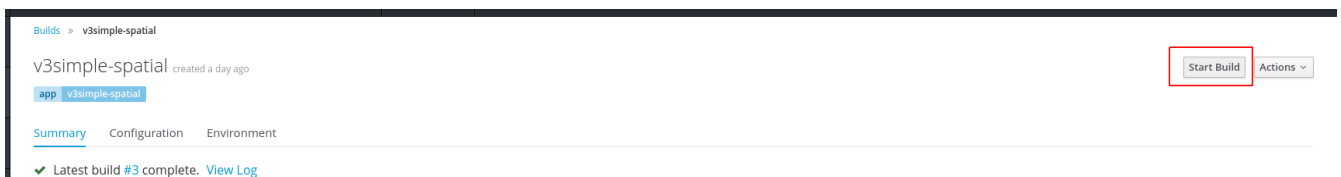
```
#get a listing of the build configs (bc)
> oc get bc
NAME                TYPE      FROM      LATEST
v3simple-spatial    Source    Git       1

> oc start-build v3simple-spatial --follow --wait
```

2) You can also use the web console. Click on Browse in the left menu and then choose the builds.



There will only be one build listed on the page, go ahead and click it. Then on the next page in the upper right you will see a button "Start Build", go ahead and click it.



On the next page you can look at the what is happening by clicking on the "View Log" link.

[Builds](#) » v3simple-spatial

✓ **Build v3simple-spatial-4 has started.**

v3simple-spatial created a day ago

app v3simple-spatial

[Summary](#) Configuration Environment

🔄 Latest build #4 is running [View Log](#)  
started a few seconds ago

In the terminal on the resulting page you can watch the entire build and push progress: just like you could in the command line with the --follow flag.

With either the command line or the web console, when you go back to the Overview for your project you will notice that the number next to the deployments has incremented because we have actually done a build AND deployment as part of this process.

You are now certified in the Pousty School of Docker and Cloud management as having completed build and deploy MASTERS!

In the next section we look under the hoods to the declarative syntax that OpenShift uses under the hoods. After that we will move on to add PostGIS to your application.

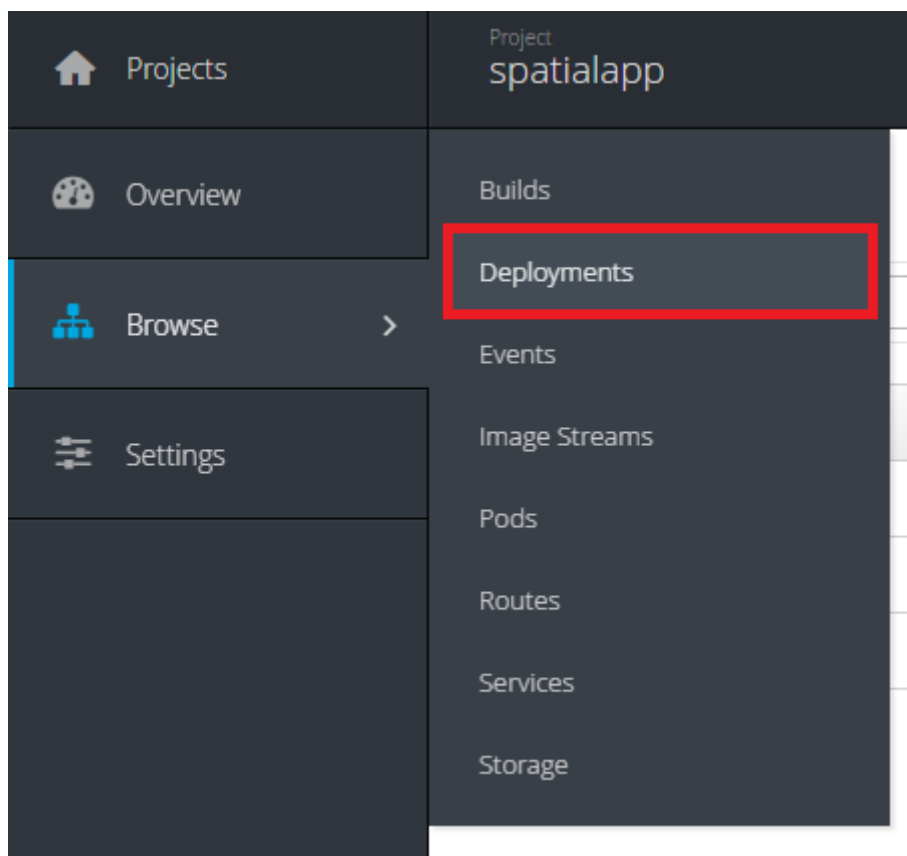
# Full Application with Database

To exercise the database some more we are going to change our python web application to read and write from the PostGIS master and slave services. When using OpenShift the way to handle the connection parameters to the DB are handled through environment variables. Rather than hard coding in IP addresses or username/passwords we set environment variables that actually point to those values.

Some of these are set by the platform "automagically" - these are usually network type variables like hosts to IP and port mappings. The others, that we need to handle manually right now are database name, username, and password. At the end of this section I will explain how we could have avoided having to manually do this.

So as I explained above we got the network pieces for free. Let's add the env variables to the deployment configuration (dc) for the python code. We add it to the dc so it becomes available to all pods controlled by the dc. We will start with the read operations from the DB so we will use the slave pods for those operations.

In your browser go to the Browse → Deployments



From there click on the deployment for the *REPLICA\_DC\_EXAMPLE*, called *replica\_dc* then click on the environment tab. From there you should see all the environment variables defined on the dc. We are interested in 3 of the variables: PG\_USER, PG\_ROOT\_PASSWORD, and PG\_DATABASE. You will need the names and values for both of these (please note that your values will be different). They are highlighted in red below:



Projects

Overview

Browse

Settings

Project

spatialapp

Deployments » pg-slave-rc

pg-slave-rc created 32 minutes ago

DetailsEnvironmentEvents

Container server

Name	Value
TEMP_BUFFERS	9MB
MAX_CONNECTIONS	101
SHARED_BUFFERS	124MB
MAX_WAL_SENDERS	20
WORK_MEM	9MB
PG_MASTER_HOST	pg-master-rc
PG_MASTER_PORT	5432
PG_MASTER_SERVICE_NAME	pg-master-rc
PG_MODE	slave
PG_MASTER_USER	master
PG_MASTER_PASSWORD	kBrTxWudYKJs
PG_ROOT_PASSWORD	keTPfgSIoQ78
PG_USER	testuser
PG_PASSWORD	eo3F1KP43rhp
PG_DATABASE	userdb

Now that we have these we just have to do a simple command:

```
> oc set env dc/simple-spatial PG_DATABASE=userdb PG_USER=postgres
PG_PASSWORD=password
deploymentconfig "v3simple-spatial" updated
```

This will force a redeployment of the pods to get the new environment variables. We have to do this redeployment because remember, containers are immutable.

With that step done we can now modify our python code. I have the completed example as 2\_app.py in the github repo. The steps we did to make this code change was

- Added the library to requirements.txt

```
psycopg2==2.6.1
```

- Added lines to the app.py to make the connection to the slave **service** with the proper credentials. Then on a GET request we return back some simple information from the database.

#### WARNING

This code is in NO WAY production type code. This code is trying to be as simple as possible so you learn the basic patterns. In a real app you would not load a DB connection on every request, you would check for exceptions, and you would have more optimized queries.

To make the code change there are two ways to proceed:

1. Either type or copy the code changes you see in 2\_app.py into app.py
2. Go ahead and rename app.py to 1\_app.py and then rename 2\_app.py to app.py

After either of those steps go ahead and do the following commands in the v3simple-spatial directory:

```
> git commit -am "new code"
> git push origin master
```

Then back in your web console you can fire off another build like you did in the last exercise. After the build and deploy is finished when you go to <http://v3simple-spatial-spatialapp.apps.10.2.2.2.xip.io/db> the first 10 entries in the table.

## Writing to Master

The great part of what we have set up is we can isolate our writes to master and our reads from the replica - which is why people usually set up replicas in the first place.

We have already set all the environment variables we need so really it is just editing the code. The only change we need to make to the new code is to have the connection be to the master rather than the replica. Other than that the connection parameters stay the same. In a more production

ready app you would probably use two different Postgresql accounts, one for reading and one for writing, which would require new environment variables.

I added code to randomly generate a name and the coords for a new point whenever you HTTP POST to the /db URL. Again this is really hacky code for a workshop - not production code. I will talk you through the code in class.

Remember, to make the code change there are two ways to proceed:

1. Either type or copy the code changes you see in 3\_app.py into app.py
2. Go ahead and rename app.py to 2\_app.py and then rename 3\_app.py to app.py

Don't forget to do the git commands I gave you above and then fire off another build.

Finally, to hit this URL you can either install a plugin for your browser or you can use cURL. By default browsers do an HTTP GET but we need a POST. There are plenty of plugins for Chrome and Firefox to help you do a HTTP Post - most of them have the word REST in them. Here is cURL syntax that will exercise the end point:

```
# -d says to do a POST and we leave the payload blank
curl 'http://v3simple-spatial-spatialapp.apps.10.2.2.2.xip.io/db' -d ''

# if you want to look at the output in a nicer format you can save it to HTML
curl 'http://v3simple-spatial-spatialapp.apps.10.2.2.2.xip.io/db' -d '' > index.html
```

If you use a browser plugin the URL stays the same and you just tell the plugin to use a POST.

The response will be the last 10 entries in the DB - which will include your latest entry. You can go ahead and POST several items and watch the new entries show up.

# Appendix: Here are some basic commands I found useful

How to build the documentation

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

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));
```