

# OpenShift Hands-on lab in GCE

Self-link: <http://bit.ly/openshift4kb8s>

Author: Rafael Benevides [@rafabene](#)

[Introduction](#)

[Installing OpenShift](#)

[The example app](#)

[Creating a Persistent Volume](#)

[Deploying MySQL](#)

[Building the frontend using S2I \(Source-to-image\)](#)

[Deploying Guestbook-service from the web console.](#)

[Installing a S2I image](#)

[Creating the application using the console](#)

[Deploying Helloworld-service using a Binary build](#)

[Adding a Health Check](#)

[Using web console](#)

[Using command line.](#)

[Advanced builds](#)

[Pipeline Build](#)

[GitHub webhook](#)

[Blue/Green deployments](#)

[A/B testing](#)

[What's next?](#)

[OpenShift](#)

## Introduction

Duration: 5:00

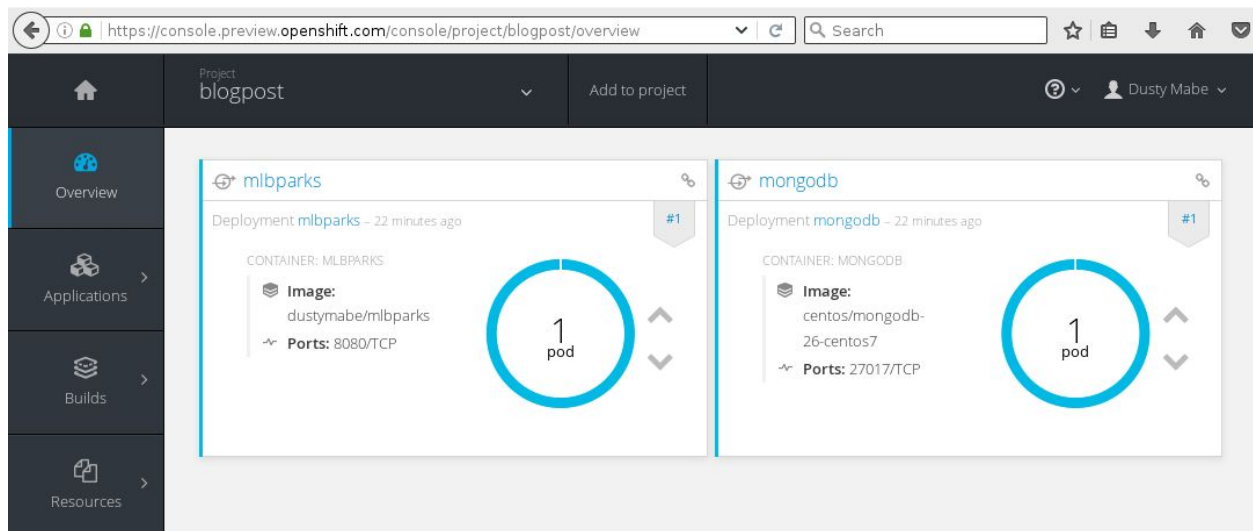
[OpenShift Origin](#) is an open source project maintained by Red Hat and the community. It was built on top of Google's Kubernetes and adds many features that makes it the

perfect PaaS (Platform as a Service) for different kind of applications (Monoliths, Microservices, Databases, Streams, etc). [OpenShift Container Platform](#) is the supported version of Kubernetes for the Enterprises.

Here are some features added by OpenShift on top of Kubernetes:

- RBAC - Role Based Access Control
- Security Context Constraints - that control the actions that a pod can perform and what it has the ability to access.
- Templates - enable a single JSON file to configure various Kubernetes resources (Pods, Replication Controllers, Services).
- Web console
- Automation: Source to image / Build pipelines
- Internal registry
- Routing

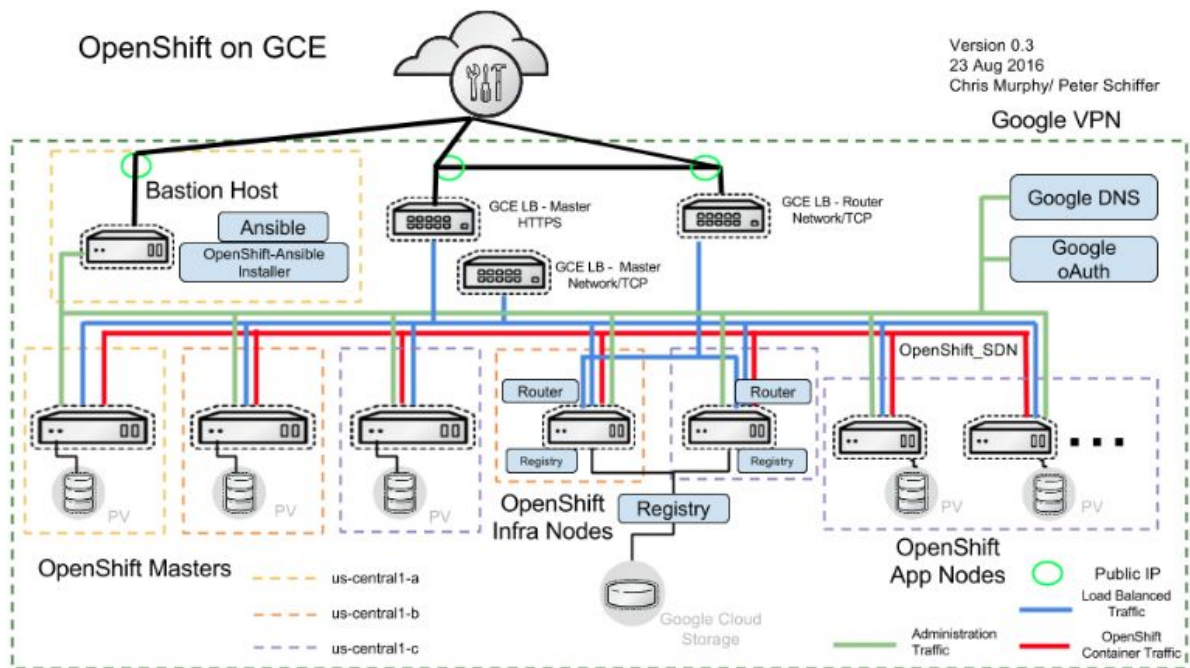
In this lab, we will explore these features and learn why and how OpenShift makes Kubernetes ready for enterprises.



## Installing OpenShift

Duration: 20:00

Installing an OpenShift cluster in Google Cloud Engine for production use requires a lot of different resources as you can see in the diagram below.



However, for development/testing purposes, we will create a Google Compute instance machine that will act like a single OpenShift master/node cluster. This small/personal cluster will allow us to explore and try OpenShift features.

If you want to know more details about the recommended installation of OpenShift Container Platform in Google Cloud Platform, please refer to <https://access.redhat.com/articles/2751521>

Now, let's create our Google Compute instance.

Open the Google Cloud Shell by clicking on the icon on the top right of the screen:



Click **Start Cloud Shell**

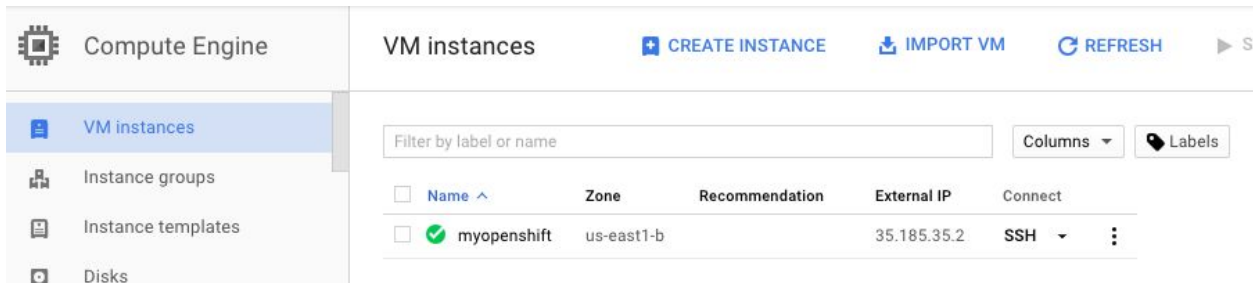
In the Google Cloud Shell type the following commands:

```
$ gcloud compute firewall-rules create allow-all --allow=tcp:0-65535

$ gcloud compute instances create --image-family=centos-7 \
  --image-project=centos-cloud --machine-type n1-standard-4 \
  --zone us-east1-b --boot-disk-size=200GB myopenshift
```

```
rafabene@openshift-msa:~$ gcloud compute instances create --image-family=centos-7 --image-project=centos-cloud --image=centos7 --zone=us-east1-b --machine-type=n1-standard-2 --tags=openshift
Created [https://www.googleapis.com/compute/v1/projects/openshift-msa/zones/us-east1-b/instances/myopenshift].
NAME          ZONE          MACHINE_TYPE  PREEMPTIBLE  INTERNAL_IP  EXTERNAL_IP  STATUS
myopenshift   us-east1-b    n1-standard-2          10.142.0.3   35.185.6.148  RUNNING
rafabene@openshift-msa:~$
```

Once the second command completes, you should see a Virtual Computer called "myopenshift" created. **Click in the SSH link.**



When you click on SSH, a web shell will be opened. Now, we will setup our CentOS Linux to run a containerized version of OpenShift.

To do that, we need:

- Install docker
- Modify docker configuration to allow 172.30.0.0/16 as an insecure registry
- Download oc binary
- Execute "oc cluster up"

Let's do it. Execute the following commands **in the myopenshift SSH terminal** that you previously opened:

```
$ sudo su -

# Installing docker and other tools
$ yum install -y docker git wget maven

# Modifying docker configuration
$ echo "INSECURE_REGISTRY='--insecure-registry 172.30.0.0/16'" >> /etc/sysconfig/docker

$ systemctl enable docker; systemctl start docker

# Downloading oc binary
$ wget https://github.com/openshift/origin/releases/download/v1.4.1/openshift-origin-client-tools-v1.4.1-3f9807a-linux-64bit.tar.gz
```

```
$ tar -zxvf openshift-origin-client-tools-v1.4.1-3f9807a-linux-64bit.tar.gz
```

```
$ mv openshift-origin-client-tools-v1.4.1+3f9807a-linux-64bit/oc /usr/local/bin/
```

Now, return to "VM Instances" menu of "[Computer Instances](#)" page and note the external IP address. We need to use it to create a dynamic DNS entry using nip.io.

[NIP.IO](#) allows you to map any IP Address in the following DNS wildcard entries:

- 10.0.0.1.nip.io maps to 10.0.0.1
- app.10.0.0.1.nip.io maps to 10.0.0.1
- customer1.app.10.0.0.1.nip.io maps to 10.0.0.1
- customer2.app.10.0.0.1.nip.io maps to 10.0.0.1
- otherapp.10.0.0.1.nip.io maps to 10.0.0.1

NIP.IO maps **<anything>.<IP Address>.nip.io** to the corresponding **<IP Address>**, even 127.0.0.1.nip.io maps to 127.0.0.1

Name	Zone	Recommendation	External IP	Connect
myopenshift	us-east1-b		35.185.35.2	SSH

In our example, the External IP is 35.185.25.2. We will now execute "**oc cluster up**" using 35.185.25.2.nip.io as a hostname:

```
$ oc cluster up --host-data-dir=/mydata --use-existing-config=true  
--routing-suffix=app.<external_ip>.nip.io --public-hostname=<external_ip>.nip.io
```

Oc will use docker to download OpenShift as a container and execute it. At the end, you can access OpenShift in the following URL: [https://<external\\_ip>.nip.io:8443/console/](https://<external_ip>.nip.io:8443/console/)

Example: <https://35.185.25.2.nip.io:8443/console/>

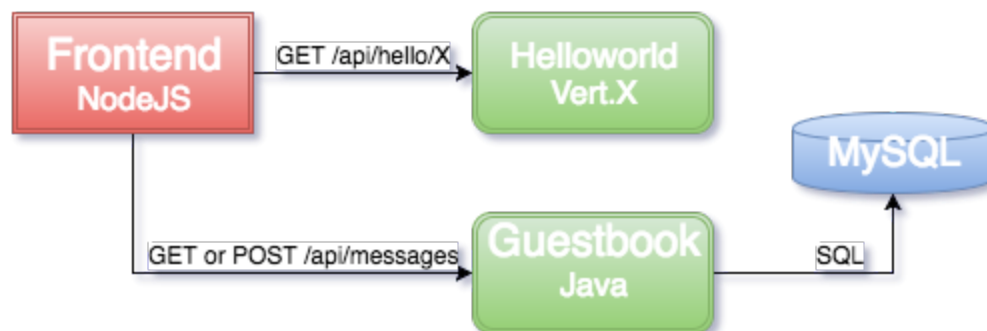
Accept the SSL certificate and proceed to the login page.

Login with the credentials **developer/developer**



## The example app

To allow us to explore the OpenShift features, we will use a variation of the Guestbook example that we used in the Kubernetes-lab. This example has a frontend using HTML5 and Javascript (client-side and server-side), and two microservices built using WildFly Swarm and VertX. It requires MySQL to store guestbook entries.



All these microservices need to be placed under the same namespace/project. A Kubernetes namespace provides a mechanism to scope resources in a cluster. In OpenShift, a project is a Kubernetes namespace with additional annotations.

Let's create the project to hold these microservices.

Execute:

```
$ oc new-project guestbook
```

## Creating a Persistent Volume

OpenShift uses [Kubernetes Persistent Volumes](#) to store data. The Persistent Volume is provided by the cluster admin because the Developer should not be concerned if the volume is a NFS, Google Persistent Disk, or any other [supported volume](#).

What we will do now, is log as a cluster administrator and create a Persistent Volume. Since this is a single machine cluster, we will create a directory in the server and use the hostPath plugin as the Persistent Volume.

Execute:

```
$ mkdir -p /data/pv001  
$ chmod 777 /data/pv001/  
$ chcon -Rt svirt_sandbox_file_t /data/pv001/  
$ oc login -u system:admin  
$ oc create -f  
https://raw.githubusercontent.com/redhat-developer-demos/kubernetes-lab/master/kubernetes/mysql-pv.yaml
```

Now you can login again with the developer credential:

```
$ oc login -u developer -p developer
```

## Deploying MySQL

Duration: 15:00

Just like kubectl, you can create resource specifying yaml file. What we will do now is create a PVC (Persistent Volume Claim) to store mysql data.

To use a Persistent Volume, we need to claim it first. A Persistent Volume was already provided by the OpenShift Cluster admin in the previous chapter. You, as a Developer, don't need to worry where the data will be stored ( NFS, GCEPersistentDisk, AWSElasticBlockStore, AzureFile, etc). You, as a Developer, just need to request an amount of space to store the data. This request is made through the usage of a PersistentVolumeClaim. You can explore the contents of the file here: <https://github.com/redhat-developer-demos/kubernetes-lab/blob/master/kubernetes/mysql-pvc.yaml>

Next, create the PersistentVolumeClaim using the oc command:

```
$ oc create -f https://raw.githubusercontent.com/redhat-developer-demos/kubernetes-lab/master/kubernetes/mysql-pvc.yaml
```

Now that we have a PersistentVolumeClaim, we can create a ReplicationController and a Service for MySQL.

Let's take a look on the MySQL ReplicationController. Note that it holds a reference to the PersistentVolumeClaim that you created with the name **mysql-pvc**.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: mysql
...
spec:
...
  spec:
    containers:
      - name: mysql
        image: openshift/mysql-56-centos7
...
    ports:
...
    volumeMounts:
      # name must match the volume name below
      - name: mysql-persistent-volume
        # mount path within the container
        mountPath: /var/lib/mysql/data
    volumes:
      - name: mysql-persistent-volume
        persistentVolumeClaim:
          claimName: mysql-pvc
```



To create MySQL *ReplicationController* and *Service* resources, execute:

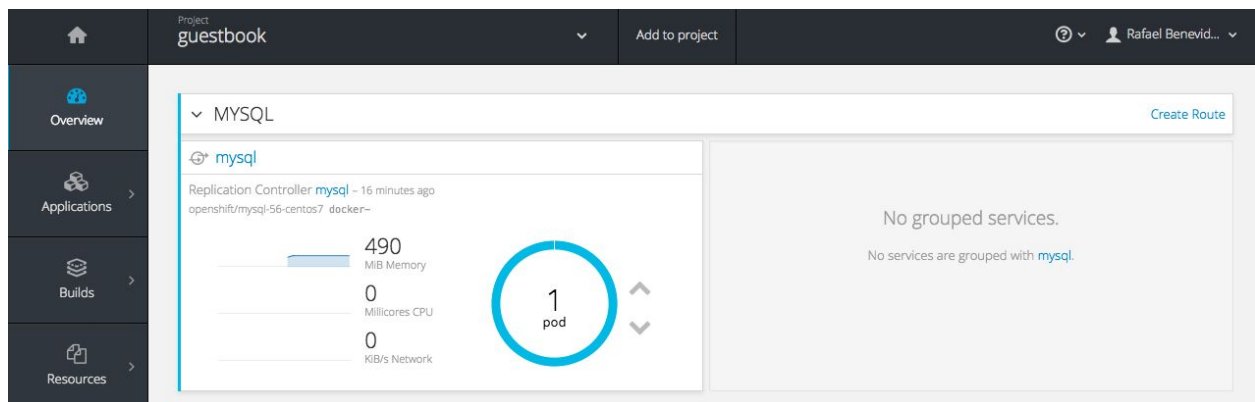
```
$ oc create -f
https://raw.githubusercontent.com/redhat-developer-demos/kubernetes-lab/master/kubernetes/mysql-rc.yaml -f
https://raw.githubusercontent.com/redhat-developer-demos/kubernetes-lab/master/kubernetes/mysql-service.yaml
```

After you have created the MySQL resources, just like **kubect**l, you can type **oc get pods** and **oc logs**.

```
$ oc get pods
NAME      READY   STATUS    RESTARTS   AGE
mysql-... 1/1     Running   0           9m

$ oc logs -f mysql-....
Starting local mysqld server ...
Waiting for MySQL to start ...
...
```

You can also open the OpenShift console and explore its dashboard in the address [https://<external\\_ip>.nip.io:8443/console/](https://<external_ip>.nip.io:8443/console/)



Take some minutes to explore this interface.

- Click on the *Service* and *ReplicationController* links and see the details.
- Click on the POD number and see the list of PODS. Select the executing POD to explore.

- Explore the POD details and the other tabs (Environment, Logs, Terminal and Events)
- In the right top corner, click in the gray *Actions* button and check that you can also edit the YAML file.

Pods » mysql-c4707

mysql-c4707 created 24 minutes ago Actions ▾

app mysql lab **kubernetes-lab**

Details Environment Metrics Logs Terminal Events

Status	Template
<p><b>Status:</b>  Running</p> <p><b>IP:</b> 10.1.0.245</p> <p><b>Node:</b> ip-172-31-3-143.ec2.internal (172.31.3.143)</p> <p><b>Restart Policy:</b> Always</p> <p>Container mysql</p> <p><b>State:</b> Running since Jan 9, 2017 7:11:55 PM</p> <p><b>Ready:</b> true</p> <p><b>Restart Count:</b> 0</p>	<p>CONTAINER: MYSQL</p> <ul style="list-style-type: none"> <li> <b>Image:</b> openshift/mysql-56-centos7</li> <li> <b>Ports:</b> 3306/TCP (mysql)</li> <li> <b>Mount:</b> mysql-persistent-volume → /var/lib/mysql/data</li> <li> <b>Mount:</b> default-token-tjaca → /var/run/secrets/kubernetes.io/serviceaccount</li> <li> <b>CPU:</b> 60 millicores to 1 core</li> <li> <b>Memory:</b> 307 MiB to 512 MiB</li> </ul> <p>Volumes</p>

## Building the frontend using S2I (Source-to-image)

Duration: 15:00

OpenShift S2I allows to create container images from source code. It produces a ready-to-run image by injecting application source into a container image and assembling a new image.

For compiled languages like Java, the S2I will compile, package and deploy your application in the application server. Even an interpreted language like NodeJS, where the build-time and run-time environments are typically the same, the S2I process can download the dependencies and start the application for you.

We will build the application frontend (NodeJS) using the S2I approach.

Later in this lab, we will need to perform some updates in the frontend code, so before moving further, **fork** the <https://github.com/redhat-developer-demos/kubernetes-lab> to your own Github profile.

Inside the [Kubernetes-lab repository](#), there's a folder called *frontend/* with the frontend source code. Note that this is a NodeJS application and that the dependencies specified in the [package.json](#) file are not included in the Github repository.

The S2I image will download source code from the Github, download the dependencies specified in the package.json file and create a new image and deploy it in the internal OpenShift registry.

OpenShift allows to create an application with a single command:

```
$ oc new-app --name frontend
nodejs~https://github.com/<your-github-profile>/kubernetes-lab --context-dir=frontend
```

Note that we've specified the name of the application (frontend), then the builder image (nodejs) followed by the github repository and the context directory that the builder image will look for the sources. If we didn't specify the parameter `--context-dir`, OpenShift would use the root of the github repository.

After you executed this command, OpenShift automatically created:

- An **ImageStream** - A place in the internal OpenShift registry to store the images produced by the S2I
- A **BuildConfig** - To store information how to build this application (using S2I), where the source code is stored (Github) and where to deploy the resulting image (The ImageStream name)
- A **DeploymentConfig** - To store the information about the execution of that application like the environment variables, number of Pod replicas, readiness and liveness probes, etc
- A **Service** - A Kubernetes Load Balancer between all Pods.

During the build, you can follow the logs using the command:

```
$ oc logs -f bc/frontend
```

You will notice that the builder image (nodejs) will download all NodeJS dependencies specified in the package.json, and create/push a new image to the OpenShift internal registry.

Note that the application at this moment is visible only inside the OpenShift Cluster. However, OpenShift provides an external proxy that allows you to access Web applications externally. This feature is called Route.

To create a Route, we need type the following command:

```
$ oc expose svc/frontend
```

Now point your browser to: [http://frontend-guestbook.app.<external\\_ip>.nip.io](http://frontend-guestbook.app.<external_ip>.nip.io) - Amazing, right?

**Attention: The frontend app will show a javascript dialog with an error message because we still need to deploy the backend microservices.**

## Deploying Guestbook-service from the web console.

Duration: 15:00

Now that we know how to create an application using openshift client (oc new-app) from the command-line. We will learn how to do the same using the web console.

### Installing a S2I image

To do this, we will deploy the guestbook-service, which is a [WildFly Swarm](#) application.

The OpenShift instance that we are using doesn't support WildFly Swarm S2I, so we need to install the S2I image for WildFly Swarm. There's a GitHub repository available at <https://github.com/wildfly-swarm/sti-wildflyswarm/> that contains a S2I image. Let's build this image from the provided github repository.

Execute:

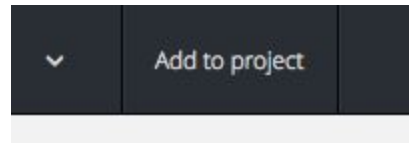
```
$ oc create -f
https://raw.githubusercontent.com/wildfly-swarm/sti-wildflyswarm/master/1.0/
wildflyswarm-sti-all.json

#Follow the S2I build
$ oc logs -f bc/wildflyswarm-10-centos7-build
```

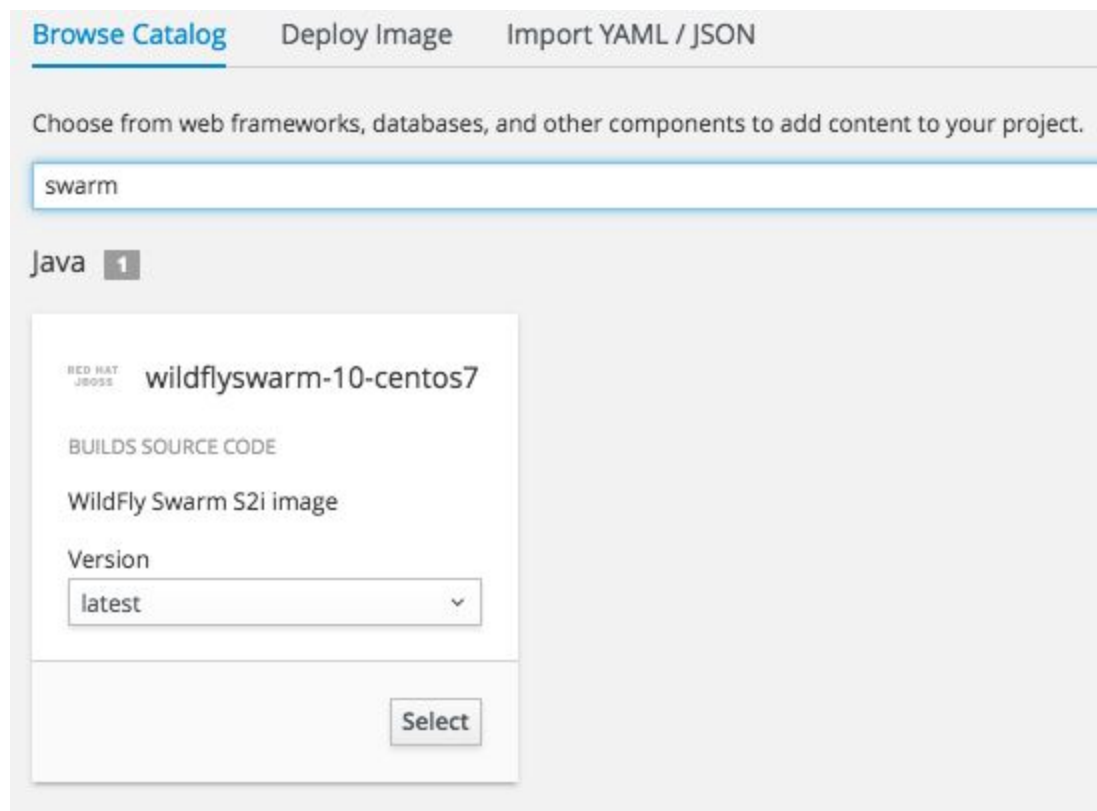
Once the S2I build has finished, we can now proceed to the creation of the application using the OpenShift console.

### Creating the application using the console

First, open the OpenShift console again, select your project and click on the "Add to Project" button that is located in the top black header bar of OpenShift console.



You will be presented to a catalog of supported languages and technologies that can be deployed in OpenShift. Use the Filter and type "swarm" to locate all builder images that are capable of building WildFly Swarm applications. Your screen should look like:



Click on the "Select" button of "**wildflyswarm-10-centos7**" builder image. This image will create an WildFly Swarm application running on a CentOS Linux.

In the next screen you just need to inform the application name and the Git repository URL.

But wait! The [Github repository](#) stores the guestbook-service inside a folder. We can't specify only the Github repository and expect that OpenShift will find the proper folder with the sources.

Just like we did in command line, we need to specify the context dir. To do that in the web console, we need to click in the ["Show advanced routing, build, deployment and source options"](#) to tweak the build and deploy of this application.

Fill the fields with the following definition:

**Name:** guestbook-service

**Git Repository URL:** <https://github.com/<your-github-profile>/kubernetes-lab>

**Context-dir:** guestbook-service

**Create a route to the application:** unchecked - We don't need a route to this microservice. We will access it from the frontend.

We also need to define an Environment variable that will limit the execution of the the JVM to 512MB. This is used in the run script of the s2i:

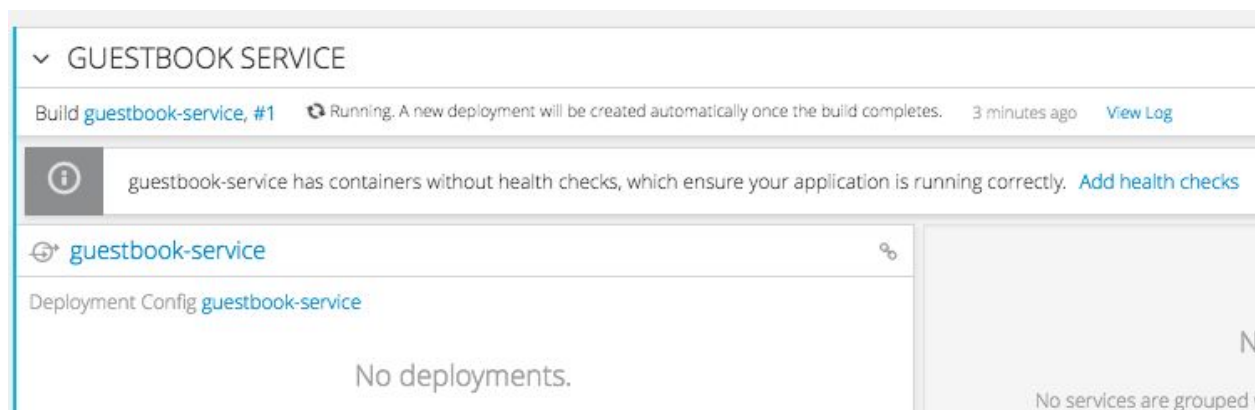
<https://github.com/wildfly-swarm/sti-wildflyswarm/blob/master/1.0/s2i/bin/run#L15>

### Deployment Configuration -> Environment Variable (Runtime only):

**Name:** SWARM\_JVM\_ARGS

**Value:** -Xmx512m

Leave the default values for the other fields and click on "Create" button. In the following screen, click on ["Continue to overview."](#) link to be taken the Overview page.



You will notice there that the Guestbook service is being built. You can click on the **"View log"** link to see the build log. Note the download of Maven dependencies. Once it finishes, the resulting image will be published in the OpenShift ImageStream and the application will be executed.

## Deploying Helloworld-service using a Binary build

Duration: 15:00

OpenShift also allows you to create an application based on an existing Dockerfile.

The helloworld-service was built using Vert.x, which is a toolkit for building reactive applications on the JVM. To build and deploy this OpenShift application, we could use a Vert.x S2I again just like we did for the guestbook-service (WildFly Swarm). However, we will use a Dockerfile provided by this application so you can see how to perform the build using Dockerfiles.

Inspect the [Dockerfile](#) for this service.

```
FROM fabric8/java-jboss-openjdk8-jdk:1.1.7

ENV JAVA_APP_JAR helloworld-service-fat.jar
ENV AB_OFF true

ADD target/helloworld-service-fat.jar /app/
ADD config.json /app/

EXPOSE 8080
CMD []

ENTRYPOINT ["java", "-Dvertx.disableFileCPResolving=true ", "-Xmx256m", "-jar",
"/app/helloworld-service-fat.jar"]
```

Note that it adds the file **target/helloworld-service-fat.jar** to the folder **/app/**, but the file helloworld-service-fat.jar is not stored inside Git repository since it would be a bad practice.

So first we need to build the fat jar using Java, then we can ask OpenShift to build the image for us. Let's now build the fat jar. Clone the git repository, change to the source folder and execute the maven package command.

```
$ git clone https://github.com/redhat-developer-demos/kubernetes-lab
$ cd kubernetes-lab/helloworld-service/
$ mvn package
```

Now that we have created the file `target/helloworld-service-fat.jar`, we can now create a build inside OpenShift to use the existent Dockerfile and create an ImageStream.

```
$ oc new-build --name helloworld --binary
```

Note that this command created an ImageStream and a Build Configuration called "helloworld". You can execute "**oc describe bc/helloworld**" and note that this "BuildConfig" strategy shows "Docker". Also note the "Output to" field showing "ImageStreamTag helloworld:latest"

```
[root@myopenshift helloworld-service]# oc describe bc/helloworld
Name:          helloworld
Namespace:     guestbook
Created:       28 minutes ago
Labels:        build=helloworld
Annotations:    openshift.io/generated-by=OpenShiftNewBuild
Latest Version: 5

Strategy:      Docker
Output to:     ImageStreamTag helloworld:latest
Binary:        provided on build
```

Now we can start the build of this image using the local resources (fat jar and Dockerfile). Execute:

```
$ oc start-build helloworld --from-dir=. --follow
```

At the end of the build, the resulting Image will become available on an "ImageStream" called helloworld. You can check the existing ImageStreams by executing "**oc get is**". Note the ImageStreams from other applications and the helloworld ImageStream that we just built.

Now that we have an image available in the OpenShift internal registry, we can use the "oc new-app" command to create the helloworld-service using the following syntax:

```
$ oc new-app --name helloworld-service-vertx -i helloworld
```

This will create a new application called "helloworld-service-vertx" using the ImageStream called "helloworld".

Return to the application at [http://frontend-guestbook.app.<external\\_ip>.nip.io](http://frontend-guestbook.app.<external_ip>.nip.io) and try it. **The application should be working without any error this time.**



# Adding a Health Check

Duration: 5:00

Do you remember how to add Health Checks in Kubernetes? We needed to run "kubectl edit deployment ..." and manually edit the YAML file to enable the Readiness/Liveness probes? In OpenShift we can do it easily by the web console or command line. Let's try first using the web console.

## Using web console

Return to the Overview page by clicking on the "Overview" link on the left hand side of the OpenShift console. You will notice the following warning in the guestbook-service *"guestbook-service has containers without health checks, which ensure your application is running correctly. "*

Click on ["Add Health Checks"](#) to learn how to create Health Checks from the web console and click on ["Add Readiness probe"](#).

Fill the fields with the following definition:

**Type:** HTTP

**Path:** /api/messages

**Port:** 8080

**Initial Delay:** <empty>

**Timeout:** 1

Click on ["Save"](#). This will cause a new deployment (using the defined ReadinessProbe).

The specified Readiness probe will do a GET request in the /api/messages resource, which will connect to the deployed MySQL to get the existing messages. If the the WildFly Swarm application returns no error (40x or 50x), it means that the application is deployed and connected to MySQL. It means that it's ready to serve requests.

## Using command line.

You might also have noticed that the frontend and helloworld applications also complain about the lack of Health checks with the following message in the OpenShift web console: *"... has containers without health checks, which ensure your application is running correctly"*.

What we will do now is use the oc command to set the ReadinessProbe for both applications.

Execute:

```
$ oc set probe dc/frontend --readiness --get-url=http://:8080/  
  
$ oc set probe dc/helloworld-service-vertx --readiness  
--get-url=http://:8080/api/hello/health
```

Easy, right?

## Advanced builds

Duration: 20:00

Do you know that you can also use Jenkins to build your application? What we will do now is explore this feature called [Pipeline Build](#).

### Pipeline Build

When you deploy a BuildConfig that uses the JenkinsPipeline as strategy, OpenShift will automatically deploy a Jenkins in your project to build the application using the Pipeline that you also specified in the BuildConfig.

Take a look in the following BuildConfig file:

"frontend-pipeline" BuildConfig file. (You will copy and paste this)

```
{  
  "kind": "BuildConfig",  
  "apiVersion": "v1",  
  "metadata": {  
    "name": "frontend-pipeline"  
  },  
  "spec": {  
    "triggers": [  
      {  
        "type": "GitHub",  
        "github": {  
          "secret": "secret101"  
        }  
      }  
    ]  
  }  
}
```

```

    },
    {
      "type": "Generic",
      "generic": {
        "secret": "secret101"
      }
    }
  ],
  "runPolicy": "Serial",
  "strategy": {
    "type": "JenkinsPipeline",
    "jenkinsPipelineStrategy": {
      "jenkinsfile": "node('maven') { \n stage 'build'\n
openshiftBuild(buildConfig: 'frontend', showBuildLogs: 'true')\n stage 'deploy'\n
openshiftDeploy(deploymentConfig: 'frontend')\n}"
    }
  }
}
}

```

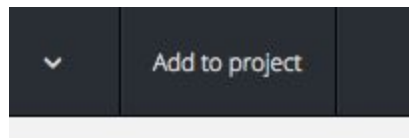
This file declares a BuildConfig called "frontend-pipeline" that uses the JenkinsPipeline strategy. It also specifies a simple Pipeline script in the "jenkinsfile" field. This Pipeline script is really simple, but you can enhance it later.

For more details about the OpenShift Jenkins plugin, visit:

<https://github.com/openshift/jenkins-plugin>

Now, let's deploy this build config. Access your OpenShift console in the address

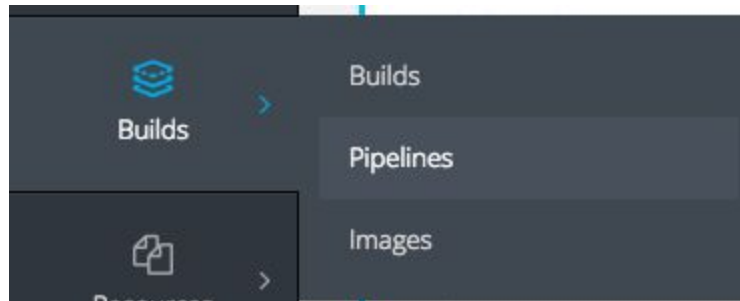
[https://<external\\_ip>.nip.io:8443/console/](https://<external_ip>.nip.io:8443/console/). Select your project and click on the "Add to Project" button that is located in the top black header bar of OpenShift console.



Now select "Import YAML/JSON" tab. Copy and paste the "frontend-pipeline" BuildConfig file in the green table above. Next, click on the "Create" button. Note that Jenkins will be deployed automatically in your project.

## GitHub webhook

Now, let's integrate this Pipeline with GitHub. Visit the menu Build -> Pipelines.



Then, click on the "[frontend-pipeline](#)" link and open the "[Configuration](#)" tab. Copy the Github Webhook URL.

frontend-pipeline created a few seconds ago Start Build Actions ▾

[Summary](#) [Configuration](#)

---


Configuration


**Build Strategy:** Jenkins Pipeline


**Jenkinsfile:**

```
node('maven') {
  stage 'build'
  openshiftBuild(buildConfig: 'frontend', showBuildLogs: 'true')
  stage 'deploy'
  openshiftDeploy(deploymentConfig: 'frontend')
}
```

Triggers

**GitHub Webhook URL:**   [Learn more](#)

**Generic Webhook URL:**   [Learn more](#)

**Manual (CLI):**   [Learn more](#)

Now we will configure GitHub. Visit the Webhooks page of your forked project in GitHub. The URL will be <https://github.com/<your-github-profile>/kubernetes-lab/settings/hooks>. You can find this page opening the forked project and clicking "Settings -> Webhooks".

Now click on the "Add Webhook button" and configure the payload URL with the URL that you copied from OpenShift. Do the following changes:

- Change the Content type to "application/json",
- Disable the SSL verification.

Then, click on the "[Add Webhook](#)" button.

**Payload URL \***

https://104.196.143.32.nip.io:8443/oapi/v1/namespaces/guestb

**Content type**

application/json

**Secret**

**Warning:** SSL verification is not enabled for this hook. [Enable SSL verification](#)

**Which events would you like to trigger this webhook?**

☒ Just the push event.

☐ Send me **everything**.

☐ Let me select individual events.

☒ **Active**  
We will deliver event details when this hook is triggered.

[Add webhook](#)

Now, do any modification in the index.html file in the frontend project and check that Github automatically notified your OpenShift instance that new changes have been introduced in the repository. Because of that OpenShift automatically started the Pipeline build for the frontend.

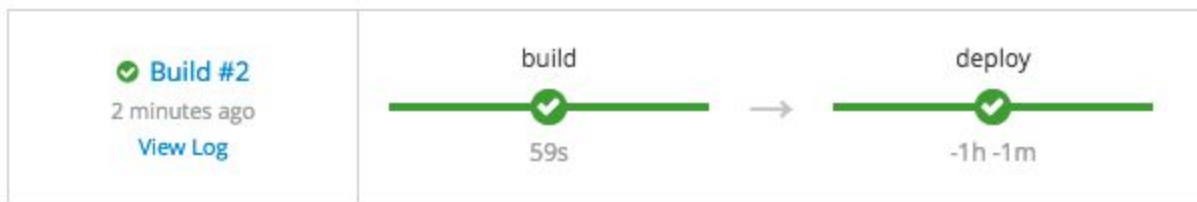
Tip 1: You can go to the following URL to edit it online:

<https://github.com/<your-github-profile>/kubernetes-lab/edit/master/frontend/index.html>

Tip 2: You can change the <h2>Say Hi</h2> title in the line 51

frontend-pipeline created 16 minutes ago

Recent Runs



[View History](#) | [Edit Pipeline](#)

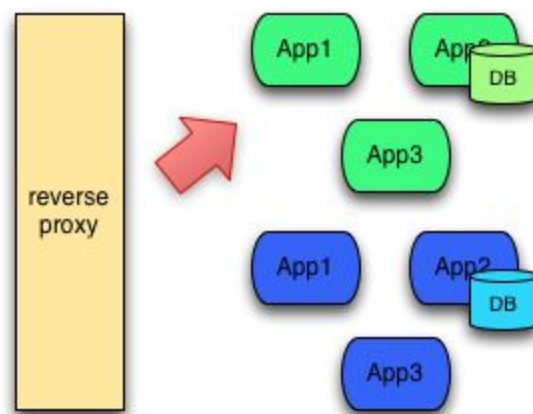
Wait the build and deploy to finish and access your frontend application again. Your changes should be visible to you at this moment.

## Blue/Green deployments

Duration: 15:00

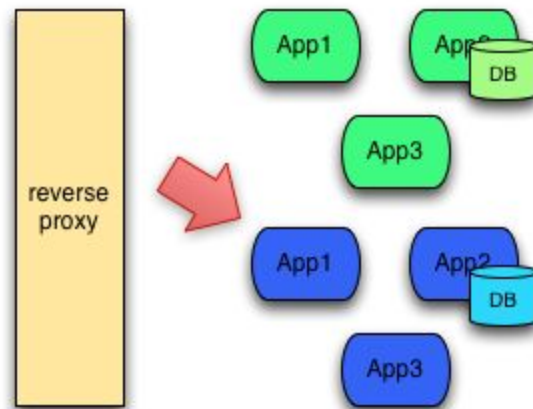
Blue/Green deployment is basically a technique for releasing your application in a predictable manner with an goal of reducing any downtime associated with a release. It's a quick way to prime your app before releasing, and also quickly roll back if you find issues.

Simply, you have two identical environments (infrastructure) with the “green” environment hosting the current production apps (app1 version1, app2 version1, app3 version1 for example):



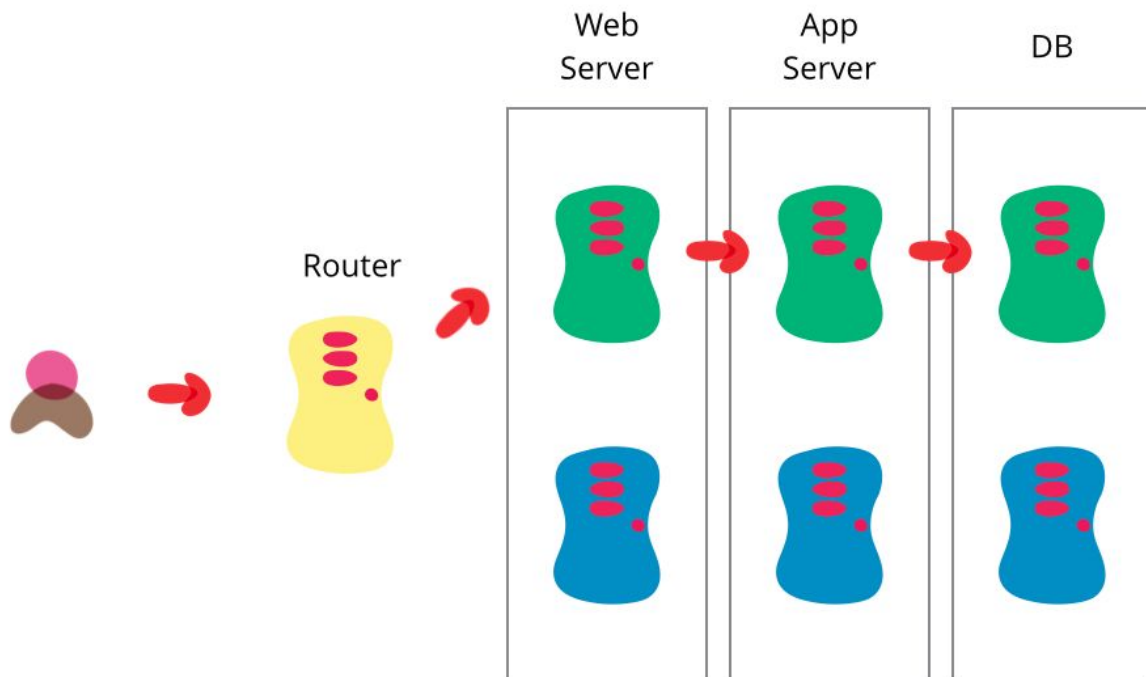
Now, when you're ready to make a change to app2 for example and upgrade it to v2, you'd do so in the “blue environment”. In that environment you deploy the new version of the app, run smoke tests, and any other tests (including those to exercise/prime the OS, cache, CPU, etc).

When things look good, you change the loadbalancer/reverse proxy/router to point to the blue environment:



You monitor for any failures or exceptions because of the release. If everything looks good, you can eventually shut down the green environment and use it to stage any new releases. If not, you can quickly rollback to the green environment by pointing the loadbalancer back.

Since OpenShift uses a Router to connect to the application, we can change the router destination from the "green" environment to the "blue" environment.



To show how this can be performed with OpenShift, we will deploy the original version of the frontend, using the "-blue" suffix, and using an existent Docker image that has been previously created.

Let's do it:

```
# Deploy a new application based on an existing Docker image
$ oc new-app --name frontend-blue rafabene/microservices-frontend:1.0

# Set ReadinessProbe for frontend-blue
$ oc set probe dc/frontend-blue --readiness --get-url=http://:8080/
```

The OpenShift route, as every other OpenShift/Kubernetes object, can be specified in YAML/JSON format. Let's take a look in the existing route called frontend.

Execute the following command to see the route defined using the JSON format.

```
$ oc get route frontend -o json
{
  "kind": "Route",
  "apiVersion": "v1",
  "metadata": {
    ...
  },
}
```



```
"spec": {
  "host": "frontend-guestbook.app.104.196.143.32.nip.io",
  "to": {
    "kind": "Service",
    "name": "frontend",
    "weight": 100
  },
  "port": {
    "targetPort": "8080-tcp"
  },
  "wildcardPolicy": "None"
},
....
}
```

Now we can use the "oc patch" command to send a JSON patch that modifies the destination of this route from "frontend" to "frontend-blue".

Execute:

```
$ oc patch route frontend -p '{"spec": { "to": { "name": "frontend-blue" } } }'
```

Look the OpenShift console and note that the route points now to the "frontend-blue" application that we just deployed. Open [http://frontend-guestbook.app.<external\\_ip>.nip.io](http://frontend-guestbook.app.<external_ip>.nip.io) and verify that your application (that you have previously changed using a deployment pipeline), has now returned to the original version.

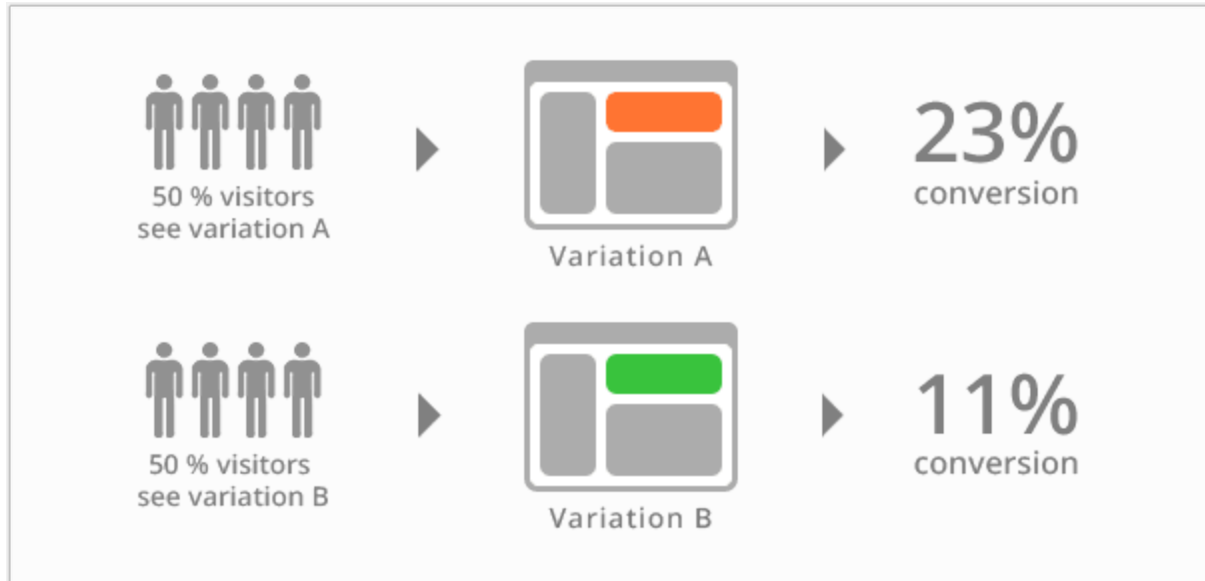
To return to the previous version, send another patch but point back the destination to "frontend".

```
$ oc patch route frontend -p '{"spec": { "to": { "name": "frontend" } } }'
```

## A/B testing

Duration: 15:00

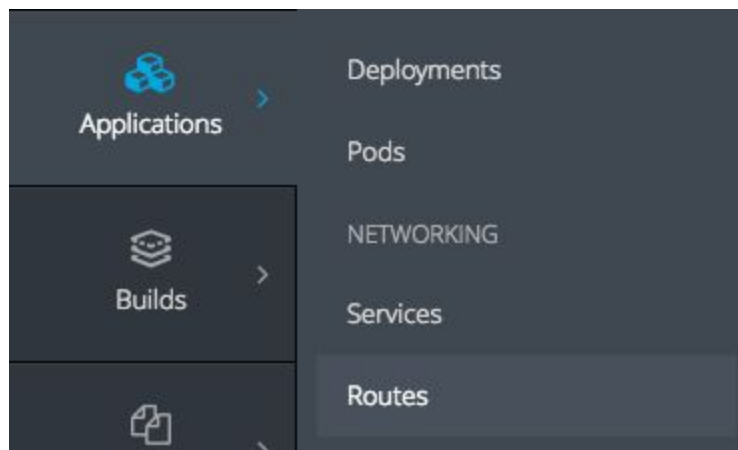
A/B testing is a way of testing features in your application for various reasons like usability, popularity, noticeability, etc, and how those factors influence the bottom line. It's usually associated with UI parts of the app, but of course the backend services need to be available to do this.



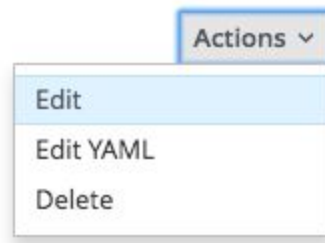
The difference between blue-green deployments and A/B testing is A/B testing is for measuring functionality in the app. Blue-green deployments is about releasing new software safely and rolling back predictably. You can obviously combine them: use blue-green deployments to deploy new features in an app that can be used for A/B testing.

We will see now how we can place two different versions of the same application to respond under the same OpenShift router.

First, visit the menu Applications -> Routes.



Click on the "frontend" route. In the next screen, select "Actions -> Edit" on the top right corner of the screen.



There is a link called "Split traffic across multiple services" that allows you to add another application in the same route. Click in this link and select "frontend-blue" with weight 100 as an "Alternate Service". Your screen should look like this.

**Path**

/

Path that the router watches to route traffic to the service.

**\* Service**

frontend

Service to route to.

**\* Weight**

100

Weight is a number between 0 and 256 that specifies the relative weight against other route services.

**Target Port**

8080 → 8080 (TCP)

Target port for traffic.

**Alternate Services**

**\* Service**

frontend-blue

Alternate service for route traffic.

**\* Weight**

100

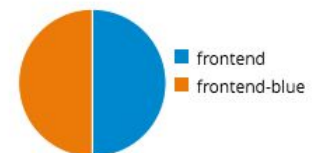
Weight is a number between 0 and 256 that specifies the relative weight against other route services.

Click on "Save". You will see that your route now is split 50/50 between "frontend" and "frontend-blue".

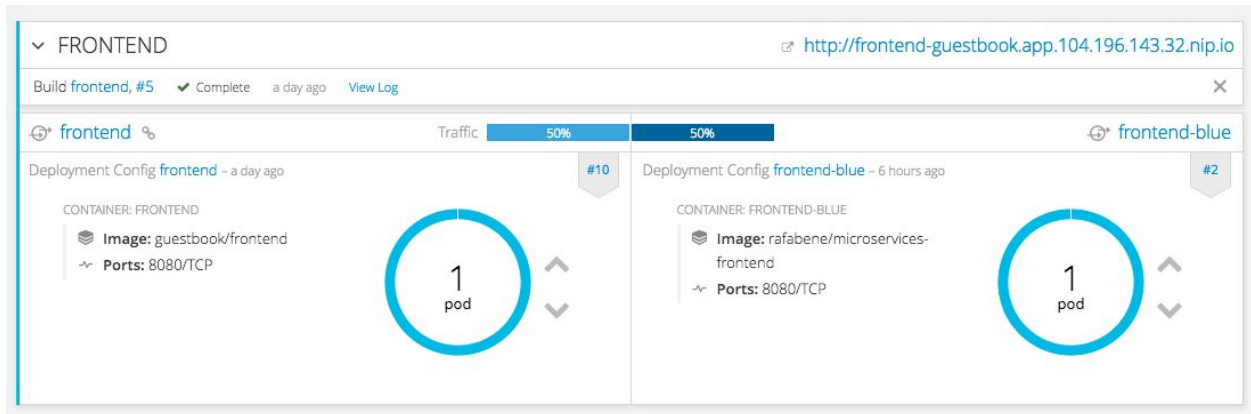
#### Traffic

This route splits traffic across multiple services.

Service	Weight
frontend	100
frontend-blue	100



This can also be perceived in the Overview page.



To try it, make a request using curl in the console. Type:

```
$ curl http://frontend-guestbook.app.<external_ip>.nip.io
# Each request will bring a different version of the application.
```

When you use the "curl" command multiple times, it brings different versions of the application each time because the curl command doesn't store any connection information. However if you access the same URL [http://frontend-guestbook.app.<external\\_ip>.nip.io](http://frontend-guestbook.app.<external_ip>.nip.io) in the browser, you will realize that you will always receive the same version. This is useful to show that people using the browser will always stay in the same version that they accessed for the first time.

## What's next?

### Codelab feedback

- The codelab was easy and useful
- The codelab was too complicated
- The codelab didn't go far enough

## OpenShift

- <https://www.openshift.org/>
- <https://www.openshift.com/>
- <https://www.openshift.com/dedicated/>
- <https://github.com/minishift/minishift>