OPENSHIFT®

by Red Hat®

HPE

POINTNEXT

Patrick D'appollonio

Hewlett-Packard Enterprise USA

HPE Datacenter Care - Center of Excelence

OpenShift fundamentals

Things you need to know to use OpenShift

OpenShift?

OpenShift is an all-in-one solution to orchestrate workloads based on containers. It uses **Kubernetes** (from Google) internally as well as **Docker** to perform, among other features:

- Application builds
- Deployments
- Scaling
- Health management
- Orchestration
- Self-service platform

How OpenShift works

A combination of multiple application elements makes the standard OpenShift framework. With it you can deploy Docker containers to be automatically managed by OpenShift. You'll also get some extra benefits like **if one of the OpenShift nodes goes down, the pods are going to be scheduled in a different node**.

The goal is to have Docker containers that follow the containerized mantra: **stateless applications** that use services and apps by calling their APIs.

Docker requires a mindset change!

To Use Docker / OpenShift, we need a mindset change

Normal applications are based on the premise of single, monolithic apps with huge codebases and multiple entrypoints. Docker is a bit different: each Docker container **can only perform one operation at a time**. This operation can be running a web server, or running a database, **but not both**.

This doesn't mean that **you can simply create a second Docker container with the Database on it**, and link them to make them think they're on the same SDN.

There are ways to make old monolithic apps work in Docker containers, but this is never the proper way to go: it'll make the overall process slower and painful.

Docker containers are meant to be disposable 🚮, which means you can destroy and recreate ♻️ a new one at any given time without actually affecting your workload. This also means that the application should handle shutdowns and restarts gracefully, as well as have **disposable storage** or a plan to **store data elsewhere**, and just **assume the filesystem as *read-only***.

🚨 Let's say it again...

> ... the application **should handle shutdowns and restarts gracefully**, as well as have **disposable storage** or a plan to **store data elsewhere** and just **assume the filesystem as *read-only***

... Why⁉

Thinking about this approach, it makes really easy to think about container orchestration: **by having no penalty on destroying and recreating containers** orchestration becomes easy.

Docker 101: Fundamentals

Things you need to know to use Docker

Docker philosophy

✅ **Small, lightweight Docker containers** where each container has a single responsibility.

✅ **Avoid multiple services within a single container**, avoiding a *single point of failure*.

✅ Focused on a **microservices concept**: the app must be developed based on modular components.

💻 Building Docker containers

Each Docker container starts on a basic image, which is a descriptive way to define **how the platform will look**. Usually, this means starting from `scratch` -- which is the name of the most basic Docker image -- and add continuously the things we need, like folder structure, applications and everything else.

Every time we add something to the default, basic image, we're creating a **layer** which is commited

The Kernel for this Operating environment is provided by the **Host system**.

ℹ️ More technical details

Docker uses a filesystem technology called *copy-on-write*. This allows Docker to **quickly spin up Docker containers**, since the files inside the container point to the actual files in the host operating system.

Due to the filesystem mentioned above, the layers are also pointers to other copies in the filesystem. A final image **is a copy of all the combined layers in a certain point in time**: this means that even when we use the last layer actively, the contents of the previous layer remain within the same image.

Let's work with Docker containers

Example Docker container

Let's create a simple demo with a Docker container that **runs an Apache server in a Docker container**, using `debian` as a base OS. This should be the basic `Dockerfile`. Also create an `index.html` file in the same directory, with a `Hello, world!` inside:

```
FROM debian
MAINTAINER "Patrick D'appollonio" "dappollonio@hpe.com"

RUN apt-get update && apt-get install -y apache2 \
        && apt-get clean && rm -rf /var/lib/apt/lists/*

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80
ADD ["index.html","/var/www/html/"]

ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

When built by running `docker build`, we will see something like this...

```
$ docker build -t apache-server .
Sending build context to Docker daemon 2.048 kB
Step 1/9 : FROM debian
latest: Pulling from library/debian
c75480ad9aaf: Pull complete
Status: Downloaded newer image for debian:latest
 ---> a2ff708b7413
Step 2/9 : MAINTAINER "Patrick D'appollonio"
           "dappollonio@hpe.com"
 ---> Running in d39942078595
 ---> f61720845da6
Removing intermediate container d39942078595
Step 3/9 : RUN apt-get update && apt-get install -y apache2
           && apt-get clean && rm -rf /var/lib/apt/lists/*
 ---> Running in 4c3d349da87f
# wall of text removed for brevity
 ---> f07e09f2be55
Removing intermediate container 4c3d349da87f
```

```
Step 4/9 : ENV APACHE_RUN_USER www-data
 ---> Running in 4c0596830520
 ---> fb87ed5b4a10
Removing intermediate container 4c0596830520
Step 5/9 : ENV APACHE_RUN_GROUP www-data
 ---> Running in 4392c6e918ea
 ---> affed0ca64ad
Removing intermediate container 4392c6e918ea
Step 6/9 : ENV APACHE_LOG_DIR /var/log/apache2
 ---> Running in 3bacc24149d3
 ---> 7d386cdbacee
Removing intermediate container 3bacc24149d3
Step 7/9 : EXPOSE 80
 ---> Running in f3ae03b26805
 ---> d13c0b38acc4
Removing intermediate container f3ae03b26805
Step 8/9 : ADD index.html /var/www/html/
 ---> 187dbc08fba0
Removing intermediate container 5d8006768bc5
Step 9/9 : ENTRYPOINT /usr/sbin/apache2ctl -D FOREGROUND
 ---> Running in 7cd47dd6d8c0
 ---> 2e703f601881
Removing intermediate container 7cd47dd6d8c0
Successfully built 2e703f601881
```

Dissecting the Output:

- The first step **pulled down the Docker `debian` image from the public** [Docker Registry](#).
- All the steps of the `Dockerfile` **are commited against a source control platform built-in in Docker**. This is the *copy-on-write* technology mentioned earlier. Each commit gets a commit SHA: the first one is `a2ff708b7413`.
- Each one of the steps generates a **temporary "Docker image"** used to create that step, merged into the master flow, and then removed from the run flow -- but **not removed from the system**, to use it as a cache later on.

What happens if we run the same `docker build` command again?

It finishes pretty quickly!

```
Step 1/9 : FROM debian
latest: Pulling from library/debian
Status: Image is up to date for debian:latest
 ---> a2ff708b7413
Step 2/9 : MAINTAINER "Patrick D'appollonio"
           "dappollonio@hpe.com"
 ---> Using cache
 ---> 26cdb0b65b3a
Step 3/9 : RUN apt-get update && apt-get install -y apache2
           && apt-get clean && rm -rf /var/lib/apt/lists/*
 ---> Using cache
 ---> 4ae3b13c68ed
Step 4/9 : ENV APACHE_RUN_USER www-data
 ---> Using cache
 ---> fb87ed5b4a10
Step 5/9 : ENV APACHE_RUN_GROUP www-data
 ---> Using cache
 ---> affed0ca64ad
```

```
Step 6/9 : ENV APACHE_LOG_DIR /var/log/apache2
 ---> Using cache
 ---> 7d386cdbacee
Step 7/9 : EXPOSE 80
 ---> Using cache
 ---> d13c0b38acc4
Step 8/9 : ADD index.html /var/www/html/
 ---> Using cache
 ---> 187dbc08fba0
Step 9/9 : ENTRYPOINT /usr/sbin/apache2ctl -D FOREGROUND
 ---> Using cache
 ---> 2e703f601881
Successfully built 2e703f601881
```

Docker is using the **cache mechanism**, that's why it builds so quickly and each step finishes with a `Using cache` message: Each step, since we already did it and it's stored in the source control platform inside Docker **can be reused by just calling it by name** (SHA).

✅ Let's run our new server:

```
docker run -p 80:80 apache-server

# "-p 80:80" defines that I want to listen on the host
# machine on port 80, and route that port to the port 80
# inside the container.

# "apache-server" is the name of the docker container
# created with "docker build"
```

Then, we can go into `http://localhost/` and see our result: the Apache2 server running in a Docker container.

We can also get a terminal inside the container

Let's explore our container, make sure it's running by executing `docker ps`. You should see a container ID, say, `ac2d34e3fbcc` and a name `apache-server` after we launched it in the previous step. Grab the ID and then run the following:

```
$ docker exec -it ac2d34e3fbcc bash
root@ac2d34e3fbcc:/#
```

We got a prompt inside the Docker container! Let's explore a couple of things...

Checking inside the container

Checking the details of the Linux distribution:

```
# uname -a
Linux ac2d34e3fbcc 4.4.0-81-generic 104-Ubuntu SMP Wed
Jun 14 08:17:06 UTC 2017 x86_64 GNU/Linux
```

Let's also check our file we passed to the Container:

```
# cat /var/www/html/index.html
<h1>Hello, world!</h1>
```

We can definitely modify the file:

```
# echo "<h1>Goodbye!</h1>" > /var/www/html/index.html
```

🚨 ... We have one problem

By modifying the contents of the containers, since their storage is ephemeral, then the moment we exit `bash` and we don't save the changes -- commit them -- then we will lose them 🚮.

🚛 Images vs Containers

To understand how to save our changes, we need to understand first the difference between a Docker *container* and a Docker *image*.

- **A Docker container** is a running environment created from a Docker image that specifies a set of dependencies, applications and folder structure to run.
- **A Docker image** is a declarative way to construct an environment based on a description on how this looks like.

Think about the **Docker Image** as the recipe to cook something, and the **Docker Container** being the cake already baked.

So, if we take our running **Docker Container** and we stop it, then we create a new one based on our **Docker Image** `apache-server` then that new container *will never have the changes we did* to the running container, **since the "recipe" doesn't include our change**.

What we need to do is:

- Exit the container gracefully with `CTRL+pq`.
- Take the current **Container ID** (in our example, `ac2d34e3fbcc`).
- Commit the changes in `ac2d34e3fbcc` **to a new Docker Image** (each change is a new image, remember?)

```
$ docker commit ac2d34e3fbcc server-changed
sha256:07fd4afe0955053a833de4a3f25fd23441222063048 4c63b49ece
```

Inspecting containers running:

```
$ docker ps
CONTAINER ID   IMAGE   COMMAND                    CREATED
ac2d34e3fbcc   demo    "/usr/sbin/apache2..."   24 minutes ago

STATUS         PORTS                 NAMES
Up 24 minutes  0.0.0.0:80->80/tcp    peaceful_dubinsky

# you can also print the containers stopped, by executing
# "docker ps -a"
```

Inspecting images created:

```
REPOSITORY       TAG      IMAGE ID       CREATED          SIZE
server-changed   latest   07fd4afe0955   14 minutes ago   206 MB
apache-server    latest   2e703f601881   39 minutes ago   206 MB
debian           latest   a2ff708b7413   6 days ago       100 MB

# you can also see intermediate images (the ones used to
# build our server) by running "docker images -a"
```

Let's play with `docker-http-server`

github.com/patrickdappollonio/docker-http-server

`docker-http-server` internals

`docker-http-server` is a container built with a bare-bones binary file, **based on the Alpine Linux distribution** -- pretty popular in the *Docker world* due to how small it is, and how the base image is built by just copying files, which is just **one layer**.

By default, `docker-http-server` will just redirect to its Github page where the source code is stored, but, if you mount a volume inside the container, **you can serve any static file you want**, like a simple HTTP server with just one dependency: Docker.

Get the image by doing:

```
docker pull patrickdappollonio/docker-http-server
```

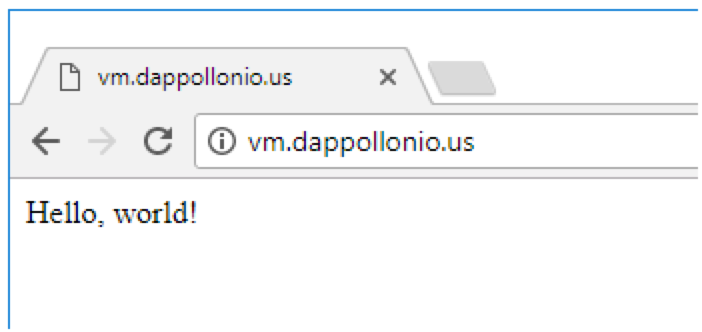
Now, let's use our current directory which should still have the `index.html` file with the `Hello, world!` message on it. We will mount the current directory for the Container to serve. This directory should also contain our previous `apache-server` `Dockerfile` which serves our purpose of serving static files. Let's trigger a run of our server:

```
$ ls
Dockerfile  index.html

$ docker run -p 80:5000 -v $(pwd):/html \
      patrickdappollonio/docker-http-server
2017/06/27 15:43:03 Starting HTTP Server. Listening at
"0.0.0.0:5000"
```

🌎 Opening the Browser and seeing the result:

Once we open our browser, and due to Internet standards, the first file served in an HTTP server must be `index.html` or `index.htm`. In the browser, when we point to the local machine's IP or hostname, we will see:



We can also inspect the `Dockerfile` by going to `127.0.0.1/Dockerfile` (or replacing `127.0.0.1` with the hostname of the machine running Docker).

Since this is a Docker container that prints whatever there is on the `/html` folder inside the container, but being that folder a **mounted volume** we can make changes to the content of the folder locally, and **those changes will be reflected *as soon* as we refresh the page**. Try it!

Exploring `docker-compose`

docs.docker.com/compose

`docker-compose`?

`docker-compose` is a simple tool that will allow us the simplicity of orchestrating Docker containers in a simple way. This tool **allows us to use YAML in a declarative way to define the state of our containers** and then execute them as instructed.

Docker for Mac, Docker for Windows and Docker Toolbox already comes with Compose inside. For Linux, **go to the Github releases page for Docker Compose and run the two commands offered there**: it'll download and move `docker-compose` to your `$PATH`.

Example of a `docker-compose.yaml` file

Running our previous server is pretty simple: You just define an arbitrary name, in this case, `http-server` (it used to be called `peaceful_dubinsky` before!), the image we want to run, a port mapping as well as the volumes we will mount. We can also request the container to be restarted if it crashes for any reason...

```yaml
version: '2'

services:
  http-server: # an arbitrary name for our running container
    image: patrickdappollonio/docker-http-server
    ports:
      - 80:5000 # the port mapping specified before with "-p"
    volumes:
      - .:/html # the dot indicates the current directory
    restart: always
```

To run the `docker-compose` defined environment, we just do:

```
$ docker-compose up
Creating network "development_default" with the
default driver
Creating development_http-server_1
Attaching to development_http-server_1
http-server_1  | 2017/06/27 15:57:06 Starting HTTP Server.
Listening at "0.0.0.0:5000"
```

Note that `docker-compose` did 3 things:

- It created a network called `development_default`
- Created the actual container `development_http-server_1`
- Attached the output to the container `development_http-server_1`
- Run the container in the foreground, printing the `stdout` / `stderr` on screen.

📣 Software-defined Networks (SDN)

Internally, **all of the Docker orchestrators out there will create SDNs**. The networks are meant to be used for service discovery as well as name resolution if the containers are linked (not by default).

Assuming our previous `docker-compose.yaml`, if we add another extra container, the network created between both of them **will allow them to see each other *if they're linked*** ❗ by using the "running name" (in our case, `development_http-server_1`).

Say we have a second container named `example`. `example` can do `curl http://development_http-server_1:5000` and it'll get to our `index.html` file, using the SDN created by `docker-compose`. Still, if we try from the host machine to `curl` the container in the same way, **our computer won't know how to route that request**.

Trying it out!

Update your `docker-compose.yaml` file to look like this:

```yaml
version: '2'

services:
  http-server: # an arbitrary name for our running container
    image: patrickdappollonio/docker-http-server
    ports:
      - 80:5000 # the port mapping specified before with "-p"
    volumes:
      - .:/html # the dot indicates the current directory
    restart: always

  ubuntu:
    image: demo
    restart: always
    links:
      - "http-server"
```

Then run it by executing `docker-compose up -d` (note the `-d` part!).

When running `docker ps`, it should show both our `patrickdappollonio/docker-http-server` running as well as our `apache-server` image. Let's get a terminal into our `apache-server` by getting the ID of the running container with `docker ps`, and then install cURL, then query our secondary container:

```
$ docker ps
... grab the ID of the Apache server container,
as well as the "name" of the docker-http-server,
we will assume ID "6c26e55795fa" and name
"development_http-server_1"...

$ docker exec -it 6c26e55795fa bash

root@6c26e55795fa:/# apt update && apt install curl -y
... output ommited for brevity ...

root@6c26e55795fa:/# curl development_http-server_1:5000
Hello, world!
```

`docker-compose` allows us to create environments in an easy way

You can stop all the running containers created by `docker-compose` by running:

```
docker-compose down
```

`docker-compose` is the closest way to define environments similar to what OpenShift uses. But **OpenShift is heavily loaded with extra features, as well as different feature names**. Still, the features of the containerization engine works pretty much like `docker-compose`.

One extra example!

WordPress + MySQL in `docker-compose`:

```
version: '2'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_DATABASE: wordpressdb
      MYSQL_USER: exampleuser
      MYSQL_PASSWORD: examplepass

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - 80:80
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_NAME: wordpressdb
      WORDPRESS_DB_USER: exampleuser
      WORDPRESS_DB_PASSWORD: examplepass

volumes:
  db_data: # this will mount to a docker-configured volume
```

Let's talk about OpenShift 👍

RedHat OpenShift options:

- **OpenShift Online:** The hosted application from RedHat, you'll receive updates and patches as they come in. RedHat manages everything related to this platform so you only have to worry about your code.
- **Container Platform:** The Container Platform is the future plan for RedHat, even though we already have some of it in version 3.4. Managing OpenShift environments has proven difficult, overall because of ***so many different tools you need to keep it running***, so a containerized approach makes sense as well as makes it easier to deploy and maintain.

- **Dedicated / Enterprise:** OpenShift Enterprise is based on the mixes of the Container Platform as well as "*Origin*", it include some extra business features as well as support from RedHat, nice integration with other RedHat tools, as well as an streamlined installer.
- **OpenShift Origin:** Origin is the source of the whole OpenShift ecosystem. All the development happens in Origin, and then is packed and streamlined to the different ecosystems, by first being *dogfooded* in the OpenShift online platform. Origin develops the whole process and ecosystem, ***it's free to use*** **and it includes 99% of the features of OpenShift Enterprise**, with the exception of those business features that requires a RedHat subscription or they're built as paid modules by RedHat.

OPENSHIFT

Most of the OpenShift **features** between versions use the same codebase, so most of the time **using the documentation of one will give you the answers you're looking *for your own installation***.

Components of OpenShift

A minimal OpenShift installation is based on a couple of main applications running:

- **Docker engine:** This will manage the Docker container platform, as well as the Docker Registry features.
- **Kubernetes:** The core of the platform: this is the app that will handle and manage the container lifecycle inside OpenShift.
- **Docker registry:** It's separated from Docker, because Docker by itself doesn't include a registry server. OpenShift needs an internal OpenShift registry server to maintain a temporary copy of the builds.

- **Etcd:** A key-value datastore to persist certain cluster details / state across all of the OpenShift platform.
- **OpenShift router:** The OpenShift router is based on HAProxy, it's the application running on the master nodes which will take a request from an external account and route it through the OpenShift platform directly to the container that's supposed to serve it.
- **OpenShift STI / S2I:** This is an extra feature of OpenShift called *Source-to-Image*. What it does is, given a Git repo and an unknown source code, it'll take the code, detect the stack and build the project for distribution in an appropiate way. By default, no runtime is installed, but OpenShift allows an easy way to get Java, NodeJS and Ruby.

There's a video presentation here about how to use S2I to easily iterate building applications with OpenShift

- **Deployer:** The *semi-HA* feature of OpenShift is, if either a change in the codebase for an S2I project or the container / pod went down, the Deployer will redeploy a new container. This is the *stubborn* piece of the software.
- **Docker SDN:** The software-defined network based on the Docker technology. Every time a container is created, Docker will create a software-defined network which may or may not use for communication purposes between containers. By linking two containers, you're specifying that they should resolve each other by container name as the host name.
- **Authentication:** The current OpenShift authentication is based on `HTPasswd` which you can use to create development accounts in the OpenShift installation.

- **Web Console:** The Web Console is the easy-to-use way to manage the OpenShift installation. You can do any development task from the UI, like deploying new code, manage the number of pods running (upscaling / downscaling) as well as URI Endpoints for running pods.
- **The `oc` command:** The `oc` command is a CLI application to manage both the development flow as well as the administration flow of OpenShift. The `oc` command is the most complete way of OpenShift administration and automation.
- **REST API:** To extend the power of the OpenShift platform, OpenShift also has an API you can use for things like deployments, S2I and so on.

A side comment...

From now on, we will use the word "*pod*" interchangeably with "container"... **A "pod" is the name Kubernetes gives to Docker containers being orchestrated by the Kubernetes scheduler**.

Additionally **we will be using an *OpenShift Origin* instance to run some commands and execute some actions**. It's usually safe to assume though that most of the OpenShift Enterprise features are available in Origin. After all, Enterprise *is just a fork* with Business Capabilities and better support than Origin.

OpenShift CLI tools

💾 CLI tools

Openshift **includes a series of CLI tools that can be used to manage and control the OpenShift environment**. As an OpenShift administrator, having control of the whole environment in your fingertips is essentially what you need to keep operations working.

There's going to be two CLI tools we will use during this training: the `oc` tool and the `oadm` tool.

The `oc` tool manages and control all areas related to OpenShift management, **such as project, applications and routes**.

The `oadm` tool is used on **more advanced tasks related to the OpenShift internals**, such as the router or the private registry, as well as giving certain apps some extra "powers" based on the administrative privileges it'll have.

Getting familiar with the OpenShift installation

Depending on the installation method, OpenShift may have been installed **using bare-bones configuration using the `openshift` CLI application**, or using **the Ansible installer** which generates most of the configuration for you, by offloading the proper parameters to the `openshift` app.

In any case, there will be a configuration folder with all the OpenShift parameters. Usually, the common locations for these settings are:

- **Container platform:** `/opt/openshift/master.local.config/`
- **Enterprise:** `/etc/openshift/master/` or `/etc/origin/master/`
- **Origin:** `/etc/origin/master/` (Ansible), `/opt/openshift/` (self)

There's an *ongoing effort* to **centralize all of them in a single** place no matter what version of OpenShift you have installed. From now on, we will call the config folder `$OPENSHIFT_CONFIG`.

Inside the `$OPENSHIFT_CONFIG` there will be two main useful files:

- The OpenShift Master configuration file, `master-config.yaml`: **This file holds the entire OpenShift configuration**, from endpoints to access control to routing. We will be working with this file a lot.
- An `admin.kubeconfig` configuration file: Whenever you're working with OpenShift as an administrator, **you need to log in into the cluster using the Kubernetes certificates**, stored on this file.

🔑 Logging in as the OpenShift Administrator

By default, and not even considering the login strategy -- more on this later -- **there's one user account already set up to manage the OpenShift installation**. This account is called `system:admin` -- and you'll note later on that all administrative accounts have the `x:y` notation, separated by a colon.

The issue? ❗ **There's no password to log in to it**. So in order to log in as `system:admin` -- which we will call "OpenShift Administrator" from now on -- **you'll have to give the OpenShift `oc` client the Kubernetes certificates**.

To log in, use:

```
$ oc login -u system:admin \
        --config=$OPENSHIFT_CONFIG/admin.kubeconfig
```

When logged in as `system:admin`, you'll be greeted with something like this:

```
Logged into "https://master.example.com:8443" as
"system:admin" using existing credentials.

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

  * default
    kube-system
    logging
    management-infra
    openshift
    openshift-infra

Using project "default".
```

Most of the alleged "projects" here are just OpenShift / Kubernetes internals that **it's advisable to never touch, unless you really know what you are doing**.

Types of OpenShift accounts

There are 7 different account types in OpenShift, called **"roles"**:

1. `admin`: A project manager. An admin user will have rights to view any resource in the project and modify any resource in the project except for quota.
2. `basic-user`: A user that can get basic information about projects and users.
3. `cluster-admin`: A super-user that can perform any action in any project. They have full control over quota and every action on every resource in the project.
4. `cluster-status`: A user that can get basic cluster status information.

5. **`edit`**: A user that can modify most objects in a project, but does not have the power to view or modify roles or bindings.
6. **`self-provisioner`**: A user that can create their own projects.
7. **`view`**: A user who cannot make any modifications, but can see most objects in a project. They cannot view or modify roles or bindings.

An OpenShift administrator can add and remove multiple roles to specific users with the `oadm` command:

```
oadm policy add-role-to-user <role> <username>
oadm policy remove-role-from-user <role> <username>
```

Sometimes, the "roles" can be given for an specific project or like in the case of the `view` role, **you might want to extend it to all available projects**. Depending on this kind of broadness, a given user can have either "local policies" or "cluster policies":

- **Local Policies:** Roles that are scoped to a given project. Roles that exist only in a local policy are considered *local roles*.
- **Cluster Policies:** Roles that are applicable across all projects. Roles that exist in the cluster policy are considered *cluster roles*.

To add cluster roles to a given user, you can execute, similar as before:

```
oadm policy add-cluster-role-to-user <role> <username>
oadm policy remove-cluster-role-from-user <role> <username>
```

Finally, you can also create your own roles, but that goes out of the scope of this training. Proper documentation and *how-tos* are available in the OpenShift documentation website.

Custom roles are based on verbs and actions against a project or cluster, but they require modifying an knowing the YAML specification for each action.

🔐 Configuring user accounts and logging in

OpenShift by default uses a login strategy called `AllowAll`. It essentially means that 🚨 **anyone using an arbitrary username and password can log in** as a basic user, create projects and manage them, as well as implement things like pod management -- confined to its own project / namespace -- or build control.

When setting it up, the new login mechanism, the most basic one recommended by RedHat is `HTPasswd`. **It uses the Apache authentication methodology** which is a file stored on the master filesystem which contains username and hashed passwords, one per line.

A simple `htpasswd` file looks like this:

```
developer:$apr1$/G3YyYZt$glgEKoCYXS0QKwxV8gAoc.
developer2:$apr1$dNzEE.q3$qfhN1eKI5jSNr0zaQ2J/g.
developer3:$apr1$9fXKJsKp$QJn24LrSJlK.xMsxyWdV21
```

You can remove the lines related to an specific user to prevent them from accessing the platform.

A word of caution though: authentication and authorization *are technically separated*. In different words, removing an user account **will not force him to log out** since it's already authenticated inside OpenShift.

To remove his privileges from the OpenShift environment, ***before* or *after* you remove the entry from `htpasswd`**, you can execute the `oadm` command:

```
oadm policy remove-user <username>
```

Managing the Login Strategies in OpenShift

Like I mentioned before, **the standard OpenShift installation allows anyone to Log In**. This is usually intentional to allow closed, internal platforms to quickly iterate and develop new features quickly. The configuration at `$OPENSHIFT_CONFIG/master-config.yaml` looks like this:

```
oauthConfig:
  ...
  identityProviders:
  - challenge: true
    login: true
    mappingMethod: claim
    name: anypassword
    provider:
      apiVersion: v1
      kind: AllowAllPasswordIdentityProvider
```

Implementing the new `HTPasswdIdentityProvider` which is built-in in OpenShift **is as easy as changing the configuration parameters**:

```
oauthConfig:
  ...
  identityProviders:
  - challenge: true
    login: true
    mappingMethod: claim
    name: demo_htpasswd_provider
    provider:
      apiVersion: v1
      kind: HTPasswdPasswordIdentityProvider
      file: /etc/origin/master/htpasswd
```

Note that there's a `file` configuration which points to the `htpasswd` file we will set up now. Changing the `identityProviders` settings require an OpenShift master restart so you can issue a `systemctl restart origin-master` to apply the changes.

🔐 Creating a developer account

Let's create a developer account in OpenShift. Since our OpenShift installation has already configured the `HTPasswd` login strategy then we will add the new users we want them to log in.

First, we need to install the `httpd-tools` which include the `htpasswd` application: `yum install httpd-tools`, then, let's go ahead and create the `developer` account with password `oc2017`:

```
$ htpasswd -c $OPENSHIFT_CONFIG/htpasswd developer
New password:
Re-type new password:
Adding password for user "developer"
```

You'll see that `htpasswd` will prompt you for the password.

Now let's check our work, by checking that, effectively, **the user account *did got* registered**:

```
$ cat $OPENSHIFT_CONFIG/htpasswd
developer:$apr1$/G3YyYZt$glgEKoCYXS0QKwxV8gAoc.
```

To add more accounts, you can run the same command as before, just omitting the `-c` flag, which stands for "create":

```
$ htpasswd $OPENSHIFT_CONFIG/htpasswd developer2
New password:
Re-type new password:
Adding password for user "developer2"

$ cat $OPENSHIFT_CONFIG/htpasswd
developer:$apr1$/G3YyYZt$glgEKoCYXS0QKwxV8gAoc.
developer2:$apr1$dNzEE.q3$qfhN1eKI5jSNr0zaQ2J/g.
```

To delete accounts, you can:

- Edit the file and remove the line where the user is declared
- Run `htpasswd -D $OPENSHIFT_CONFIG/htpasswd <user>`

Remember that by removing them here **it won't log them out from their current session**, you should also remove their privileges by issuing:

```
oadm policy remove-user <user>
```

🔑 Login in as an User without being in the Master

The last step of managing accounts is **allowing other users to log in from their workstations with ease**. This is pretty simple because everything happens using the CLI tool `oc`. Depending on your login strategy, sometimes it's possible to log in with a Token, but given our `HTPasswd` authentication, **we will log in with plain username and password, then exchange those for a token**.

The first thing we need though is to download the CLI tools. To do so, you can download them from the OpenShift Origin Github page, at github.com/openshift/origin/releases/latest. The file you need is the `openshift-origin-client-tools`. Download the zip for your platform, extract it and move the `oc` binary somewhere in your `$PATH`.

Once you have the `oc` binary for your platform, we can log in remotely:

```
$ oc login -u developer https://master.example.com:8443
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://master.example.com:8443
(openshift)
Username: developer
Password: ******
Login successful

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

    oc new-project <projectname>

Welcome! See 'oc help' to get started.
```

Since **the OpenShift installation uses a 📜 self-signed certificate created during the installation**, we get the message regarding this, which we can discard for now. You can customize the certificate later on by changing the settings in the `$OPENSHIFT_CONFIG/master-config.yaml` file, `servingInfo` section.

The login flow also tells us we have *no* projects which we will address shortly. To log out, you can easily do `oc logout` and it'll remove any tokens stored locally.

📂 Creating OpenShift projects

An OpenShift project is the minimal object that holds a set of OpenShift elements needed to run our applications. **A project also serves as a namespace** to control and manage all the applications created inside the project.

We already saw when logging in that creating projects is easy, you just have to execute:

```
$ oc new-project <project-name>
```

Giving `developer2` admin access to `<project-name>`

One of the things we will also need to manage within our project is allowing access from other users to our own projects. Internally, we're administrators of our project too, so you can use the `oadm` tool (or also `oc adm` which will also work) to manage who can use and how, our different projects.

In an specific project, to grant an user permissions, you can do:

```
dev1: $ oc adm policy add-role-to-user <role> <username>
```

Where `<role>` can be one of the many roles. You can revert that by doing:

```
dev1: $ oc adm policy remove-role-from-user <role> <username>
```

Let's create a demo project to toy with. Issue the following command:

```
$ oc new-project demo
Now using project "demo" on server "https://master.example.com:8443".

You can add applications to this project with the 'new-app' command.
For example, try:

   oc new-app centos/ruby-22-centos7~https://github.com/openshift/ruby-ex.git

to build a new example application in Ruby.
```

You'll also see that OpenShift offers you a boilerplate example, **a Ruby on Rails app with a MySQL database you can deploy in one command**. This little example here has *way more features* that we would expect to see in a first glance, so we won't run that command yet.

And just as a reminder, you can change projects by running `oc project <project-name>`.

🚢 Deploying our first container!

Let's do some real work and start orchestrating containers now. **Let's run a container in our brand-new environment**. We will work with a simple one first, a `Hello, OpenShift!` container to see how everything plays together.

The container we will deploy is `patrickdappollonio/hello-docker`. While OpenShift already provides an `openshift/hello-openshift`, **we will use the `Hello, Docker!` one later down the road**, so it's better to familiarize with it for future reference.

Let's test it first, let's download the container and run it first locally, to see what it does, and then let's push it to OpenShift. In your workstation do:

```
$ docker pull patrickdappollonio/hello-docker
Using default tag: latest
latest: Pulling from patrickdappollonio/hello-docker
e5458d100942: Pull complete
Digest: sha256:5ef638877a3b5aa05575be4dc4624f6bde600e22e...
Status: Downloaded newer image for
patrickdappollonio/hello-docker:latest

$ docker run -p 80:8000 patrickdappollonio/hello-docker
2000/01/01 01:01:01 Starting Server. Listening at ":8000"
```

Then **go to your browser and navigate to the IP address where you're running the container** -- usually `localhost`. You should see a message on screen:

```
Hello, 0f449a6e0aa5!
```

`patrickdappollonio/hello-docker` is a minimal, bare-bones Docker container that **only has one layer**, the HTTP server. It's based from the `scratch` image, which means **there's no shell, no directory structure, no nothing**, just the HTTP server as a compiled executable **which uses system calls for everything** -- making it possible to run it using the `scratch` image, although this is usually not possible for big applications or interpreted languages.

The ID you see when browsing into that container is the Docker container's `$HOSTNAME`, which in Docker defaults to the ID of the container. There are some configurations available, such as changing the display name for the `Hello, <name>!`.

Let's stop the running container by pressing `CTRL+c` and then start it again, this time, with a different name (note the `-e` parameter):

```
$ docker run -p 80:8000 -e NAME=OpenShift \
        patrickdappollonio/hello-docker
2000/01/01 01:01:01 Starting Server. Listening at ":8000"
```

If, in another terminal we make a request to the IP address holding the Docker engine, we will see:

```
$ curl http://localhost/
Hello, OpenShift!
```

Obviously, you can change the `$NAME` to be anything you want and it'll be printed when someone make an HTTP request to that hostname.

🚢 Pushing `hello-docker` to OpenShift

Now, going back to the fun parts, let's push this container to OpenShift from your workstation. To do so, let's verify the namespace / project we are now, and if there's anything here, **then deploy our container**. Perform the following:

```
$ oc status
In project demo on server https://master.example.com:8443

You have no services, deployment configs, or build configs.
Run 'oc new-app' to create an application.
```

```
$ oc new-app patrickdappollonio/hello-docker
--> Found Docker image 68e56dd (20 hours old) from Docker
    Hub for "patrickdappollonio/hello-docker"

    * An image stream will be created as
      "hello-docker:latest" that will track this image
    * This image will be deployed in deployment
      config "hello-docker"
    * Port 8000/tcp will be load balanced by
      service "hello-docker"
      * Other containers can access this service through
        the hostname "hello-docker"
    * WARNING: Image "patrickdappollonio/hello-docker"
      runs as the 'root' user which may not be permitted
      by your cluster administrator

--> Creating resources ...
    imagestream "hello-docker" created
    deploymentconfig "hello-docker" created
    service "hello-docker" created
--> Success
    Run 'oc status' to view your app.
```

If we ran `oc status` once again, we will see:

```
$ oc status
In project demo on server https://master.example.com:8443

svc/hello-docker - 172.30.70.134:8000
  dc/hello-docker deploys istag/hello-docker:latest
    deployment 1 deployed 2 minutes ago - 1 pod

View details with 'oc describe <resource>/<name>' or list
everything with 'oc get all'.
```

We can also get all the running pods to see if our container is running:

```
$ oc get pods
NAME                    READY     STATUS    RESTARTS   AGE
hello-docker-1-xs03r    1/1       Running   0          3m
```

This is great: our container is running, but **it's not possible yet to access it from the outside**. We need to route it!

🌎 Having a way to access our `hello-docker` pod

Now that we created our pod there are a lot of questions. We saw "image streams", "deployment config", "load balanced", and "service" which we will cover later on. For now, we want to see our code running, and to do so, **we need a way to access the container**.

OpenShift routes

An OpenShift route **is just a configuration given to OpenShift to allow traffic to the container through the OpenShift router** -- one of the core components of our installation -- running HAProxy. Creating a route for a given service is rather easy:

```
$ oc expose svc/hello-docker
route "hello-docker" exposed
```

Once a route is created, **there's going to be a public endpoint accessible to make HTTP calls to it**. This part depends on one of the OpenShift prerequisites, which is **having a *wildcard* A record pointing to the OpenShift router** -- usually in the Master -- ro route traffic through.

We will assume the following record, thinking that `master` is in the IP address `192.168.100.1`:

```
*.apps.example.com   IN   A   3600  192.168.100.1
```

This needs to be configured at a DNS level. Ask your Domain Name company for instructions on how to set them up.

Once the DNS configuration is properly set, then the format to get the URL to access our published projects is:

```
<service-name>-<project-name>.apps.example.com
```

In our case, our service name is `hello-docker`, and our project name is `demo`, so the ending URL will be: `http://hello-docker-demo.apps.example.com`. Let's see if it works:

```
$ curl http://hello-docker-demo.apps.example.com
Hello, hello-docker-1-xs03r!
```

It definitely works! And we can also see the name of the running Docker container / Kubernetes pod, `hello-docker-1-xs03r`. This matches the name given when we check with `oc get pods`.

Now, regarding routes, and any other OpenShift resource, we have to say that everything is customizable. You can execute `oc expose -h` to see the full flag set **where you can customize the names and parameters**.

But there's also another option to customize settings, so let's briefly deviate from our main core to talk about it: the `oc create` command.

ℹ️ The powerful `oc create` command

Pretty much all the OpenShift features you can think of **have an OpenShift command available to manage them** -- which is, *sadly*, not true for the OpenShift UI we will see later on -- but there's also an additional command, pretty powerful, called `oc create`.

In short, `oc create` allows you to create any OpenShift resource given a YAML definition for it. Said it in a different way: **if you know how to define OpenShift resources in YAML, you can create any resource using the CLI** by just executing `oc create -f filename.yaml` where `filename.yaml` will contain the definition for that resource.

Let's see an example...

Remove the route we just created by doing `oc delete route hello-docker`, then let's create the YAML specification for our route by doing a dry-run of the `oc expose` command, exporting the settings to a YAML file:

```
oc expose svc/hello-docker -o yaml --dry-run \
        > ~/hello-docker-route.yaml
```

By running this, we will save the route definition into a file we can access later on.

If we show the contents of the file, we should see this:

```
apiVersion: v1
kind: Route
metadata:
  creationTimestamp: null
  labels:
    app: hello-docker
  name: hello-docker
spec:
  host: ""
  port:
    targetPort: 8000-tcp
  to:
    kind: ""
    name: hello-docker
    weight: null
status:
  ingress: null
```

Let's do something funny: let's modify the route so it doesn't have the format `<service>-<project>.apps.example.com` but rather just `myapp.apps.example.com`. Modify `spec.host` and write `myapp.apps.example.com` inside the quotes...

```
apiVersion: v1
kind: Route
metadata:
  creationTimestamp: null
  labels:
    app: hello-docker
  name: hello-docker
spec:
  host: "myapp.apps.example.com"
  port:
    targetPort: 8000-tcp
  to:
    kind: ""
    name: hello-docker
    weight: null
status:
  ingress: null
```

Save the file, in this case, `~/hello-docker-route.yaml` and then use `oc create -f` to create the route:

```
$ oc create -f ~/hello-docker-route.yaml
route "hello-docker" created
```

Then let's verify everything works correctly:

```
$ oc get route hello-docker
NAME            HOST/PORT
hello-docker    myapp.apps.example.com

PATH      SERVICES       PORT        TERMINATION    WILDCARD
          hello-docker   8000-tcp                   None
```

And finally, test it out!

```
$ curl myapp.apps.example.com
Hello, hello-docker-1-xs03r!
```

Based on the same principle, **later on we will create full featured services by *merging* multiple YAML configuration files** to create *deployment configs*, *services*, *routes* and so on. For now, this little example will stay in our `demo` project just to continue checking it out!

OpenShift build methods

🔧 OpenShift Build Methods

Now that we've managed to understand **what is needed to run and deploy stuff in OpenShift**, it's time to move to more interesting places. Let's talk about ***build methods***. In OpenShift, you can orchestrate containers, but as you've already noticed, **you need a published container somewhere in a registry where you can pull the container down in OpenShift and run it**.

This is usually the case for pre-baked containers, and to be fair, **it's the easiest and most simple way to define containers in OpenShift** since it doesn't interfere with the normal operation flow of the company procedures.

But the truth is, sometimes this requires some extra "platforms" running, and it may not be the case. Let's explore other alternatives...

The current available 🔧 OpenShift **build methods** are:

- **From a Container Image:** If this is the case, is just a matter of publishing the image in a Docker Registry and then pull it down like we already did with `patrickdappollonio/hello-docker`.
- **From source code:** We have two options here...
  - **From a Git source code:** If we give OpenShift a Git source code, *either web or local*, it'll scan the codebase and find if there's a `Dockerfile`, if so, it'll build a container with those instructions.
  - **Detecting the language:** This feature is rather limited to, right now, Ruby, Java, NodeJS, PHP, Python and Perl. By looking at certain files it'll decide to use an specific "language builder" provided by RedHat.

> When working with the ***Detecting the language*** strategy, OpenShift **will first look for an image stream** that was registered to match the detected language, and if nothing is found, **it'll try to download the official Docker image for the given language**.
>
> If this would be a Python example, then the downloaded image will be `python:latest`.

✅ Testing the Git + Dockerfile strategy

Let's avoid the middleman **and go directly to the Source Code with a Dockerfile** to see how everything works. We will deploy a NodeJS application which is a simple `Hello world` printed on screen with a NodeJS application.

The source code is here:

```
https://github.com/patrickdappollonio/nodejs-in-docker
```

Now **let's create a new project just to hold this testing**, and leave the `hello-docker` in a different namespace for later use. Execute `oc new-project nodejs` to get the new namespace and switch there at the same time.

Then, let's create the app:

```
$ oc new-app https://github.com/patrickdappollonio/nodejs-in-docker.git
--> Found Docker image 867601d (5 days old) from Docker Hub for "node:boron"

    * An image stream will be created as "node:boron" that will track the source image
    * A Docker build using source code from
      https://github.com/patrickdappollonio/nodejs-in-docker.git will be created
      * The resulting image will be pushed to image stream "nodejs-in-docker:latest"
      * Every time "node:boron" changes a new build will be triggered
      * WARNING: this source repository may require credentials.
                 Create a secret with your git credentials and use 'set build-secret'
                 to assign it to the build config.
    * This image will be deployed in deployment config "nodejs-in-docker"
    * Port 8080 will be load balanced by service "nodejs-in-docker"
      * Other containers can access this service through the hostname "nodejs-in-docker"
    * WARNING: Image "node:boron" runs as the 'root' user which may not be permitted
      by your cluster administrator

--> Creating resources ...
    imagestream "node" created
    imagestream "nodejs-in-docker" created
    buildconfig "nodejs-in-docker" created
    deploymentconfig "nodejs-in-docker" created
    service "nodejs-in-docker" created
--> Success
    Build scheduled, use 'oc logs -f bc/nodejs-in-docker' to track its progress.
    Run 'oc status' to view your app.
```

We can track the progress of the build flow, which will clone the repository, run the `Dockerfile` and then generate a Docker container that will be published inside the Master's internal Docker Registry to be spread later on to all the nodes:

```
$ oc logs -f bc/nodejs-in-docker
Cloning "https://github.com/patrickdappollonio/nodejs-in-docker.git" ...
        Commit: ea839db5d34e2d13bb4a9820fccc666b64864079 (First upload.)
        Author: "Patrick D'appollonio <dappollonio@hpe.com>"
        Date:   Wed Jun 28 14:09:22 2017 -0600
Step 1 : FROM node@sha256:19de5403244485481fa1c2ddf47d47debabbe8094a80fb3ccab082
 ---> 867601d9565a
# steps 2 to 8 removed for brevity
Step 9 : ENV # some openshift env vars
 ---> Running in c88d549c12fa
 ---> 789539d70996
Removing intermediate container c88d549c12fa
Step 10 : LABEL # some openshift labels here
 ---> Running in 7d5ae20c5e8b
 ---> dec55a31e1ad
Removing intermediate container 7d5ae20c5e8b
Successfully built dec55a31e1ad
Pushing image 172.30.21.231:5000/nodejs/nodejs-in-docker:latest ...
Pushed 0/12 layers, 6% complete
# ommited for brevity
Pushed 12/12 layers, 100% complete
Push successful
```

📝 Reviewing what we got:

With the current flow we have from Git, we've got:

- A way to create OpenShift-schedulable containers **from source code**
- A way to generate containers **without publishing them to a public registry** or the Docker Hub
- A build process based on Docker containers **that will give us a final container**
- A full build process that, if defined in the `Dockerfile` **can also do testing before deploying**
- A way to track the build progress using `oc logs`
- A message telling us that the image is being published to our internal **private registry** installed on Master.

Wouldn't be nice that this flow could be *fully automated* so pushing a new change to the repository will trigger a new build and deploy?

Introducing Build Configs and Webhooks 🎉

OpenShift Webhooks allows any developer to continue building features for as long as the repo: a) maintains a way for OpenShift to track and validate the progress of the source code; and b) there's a webhook connected to the source control tool.

In our case, we can get a webhook for Github as well as a generic webhook that, if called, **it'll re-trigger a build**, including cloning the repository and building the project again from source.

This is all part of the OpenShift `BuildConfig`. In short, **Build Configs are a way to define how you want to build a project**. In our case, since the code is a Github repository, it knows that is part of a development process where people can continue to push code to that repository for as long as the project is being developed, **triggering builds if needed**.

🌎 How to get a Github / Generic WebHook

Since we already created our project from Git, this is easy: the `oc` command allows us to inspect the `BuildConfig` it was created for us in the app `nodejs-in-docker`. Check the details by running:

```
$ oc get bc/nodejs-in-docker
Name:           nodejs-in-docker
Namespace:      nodejs

# ... ommited for brevity ...

Strategy:       Docker
URL:            https://github.com/patrickdappollonio/nodejs-in-docker.git
From Image:     ImageStreamTag node:boron
Output to:      ImageStreamTag nodejs-in-docker:latest

Triggered by:           Config, ImageChange
Webhook GitHub:
        URL:    https://master.example.com:8443/oapi/v1/namespaces/nodejs/buildconfigs/
                        nodejs-in-docker/webhooks/Q3KHjB2lck8X6gnmHA1t/github
Webhook Generic:
        URL:            https://master.example.com:8443/oapi/v1/namespaces/nodejs/buildconfigs/
                                nodejs-in-docker/webhooks/MG_-mKfHLpUa0xxSxIBy/generic
        AllowEnv:       false

Build                   Status          Duration        Creation Time
nodejs-in-docker-1      complete        46s             2017-06-28 14:21:37 -0600 MDT
```

A couple of things to note here:

- The `Strategy` mentions what strategy is the `BuildConfig` following.
- The `URL` contains the Git repository where the code is coming from.
- **It is possible to build from an specific branch**, by appending `#branch-name` after the Git URL when creating a `new-app`.
- Since we're building starting from a `Dockerfile` which starts from its own Docker image -- in this case, `node:boron` -- then **there's also an automatic tracking mechanism** which rebuilds the project when this image from the Docker Hub gets updated.
- **The output is also a Docker Image**, in this case `nodejs-in-docker:latest`

🔑 How about building from Private Github repositories?

It is definitely possible to build from private Github repositories. To do so, you'll have to use the secrets management part of OpenShift which isn't covered in the basic training. Still, the steps to make `BuildConfig` work with private repos is:

```
# Generate an RSA key which you'll use for both Github and
# the OpenShift builder image. Make sure not to add a
# password and not to overwrite your existent keys:
ssh-keygen -t rsa -C "your_email@example.com"

# Then, add the key to the OpenShift secret storage as an
# `sshauth` secret:
oc secrets new-sshauth ghk \
    --ssh-privatekey=$HOME/.ssh/id_rsa
```

```
# Then add the new secret to the builder service account
# so when building, it can access this secret:
oc secrets add serviceaccount/builder \
    secrets/ghk

# Finally, modify your Build Config so it can use this
# new secret when building:
oc patch buildConfig <app-name> \
    -p '{"spec":{"source":{"sourceSecret":{"name":"ghk"}}}}'
```

By doing this, you're instructing the `BuildConfig` flow, when started, **to load the secret `ghk` and authenticate with it**.

We won't cover too much about *patching* or *secrets*, but the Documentation is always available for extra details. Secrets should be self explanatory, whereas ***patching* takes a JSON argument and adds it to the original JSON definition** -- we saw a YAML before, but it's possible to get JSON, it's just that *it gets tricky* to patch elements with YAML.

🔧 How about building an specific SHA commit?

Another neat feature we can do, and **it'll help us learn how to start builds**, is to build from an specific SHA commit. Obviously, to take advantage of this feature, our build process **needs to begin in a Github repository**.

Then, to start builds you just need to use the `oc start-build` command:

```
oc start-build \
          --commit="022d87e4160c00274b63cdad7c238b5c6a299265"
```

The `oc start-build` allows multiple options, like starting from a specific folder, or from a previous build. For a full list of options, check the extensive documentation. Triggering builds **requires a `BuildConfig` already available in OpenShift** to trigger.

🌎 Exposing our NodeJS app to the world

```
# let's expose our app to the world using a custom domain
# and using "oc create"
$ oc expose svc/nodejs-in-docker -o yaml \
    --hostname "expressapp.apps.example.com" \
    --dry-run > ~/nodejs-in-docker.yaml

# some elements were removed for brevity
$ cat ~/nodejs-in-docker.yaml
spec:
  host: expressapp.apps.example.com
  port:
    targetPort: 8080-tcp
  to:
    kind: Service
    name: nodejs-in-docker

$ oc create -f ~/nodejs-in-docker.yaml
route "nodejs-in-docker" created

$ curl expressapp.apps.example.com
Hello world
```

⁉️ WARNING: Image "abc" runs as the 'root' user

An extra consideration at the time of using OpenShift is to think thoroughly about what containers are allowed to run inside OpenShift. This, because in some cases, **some containers have instructions to define volumes which are part of the main Host OS**. By allowing them to run privileged, **you're also allowing any command to run in your Host OS**, like `chown` or `chmod`.

You can fix these containers **by modifying them and letting the caller to decide what files it should have access to**, or not to use those containers at all.

Additionally, **you can disable the verification check** and allow *any* privileged container to run in your OpenShift environment by doing:

```
oadm policy add-scc-to-user anyuid -z default
```

Where `anyuid` allows the container to impersonate any User ID needed and the `default` makes it so any container will be affected by this rule.

You can also define an `USER` value in the `Dockerfile` which will change the UID to a non-root account.

OPENSHIFT

Wrapping up OpenShift main concepts

So far, **we've talked about 5 main concepts**. Let's revisit them again to make sure we got them right:

- **Build Configs:** A `BuildConfig` is a set of instructions to explicitly tell OpenShift how an specific project should be built. It uses a *build strategy* to define how. We've seen the `Dockerfile` strategy as well as the *Source-to-Image*.
- **Image Streams:** An `ImageStream` is a OpenShift hook-like flow which tracks the changes on a Docker image published in a Docker registry -- either public or private. The most common flow is that if a change is detected, a build and deploy is triggered.

- **Deployment Config:** A `DeploymentConfig` is the result of a `BuildConfig`, in short, when a project is built, it needs to be deployed to OpenShift. The flow is controlled by a `DeploymentConfig`.
- **Services:** The `Services` are a software-defined network and load balancing strategy to Docker containers running an specific service. It can be our own project, or something like a Database, a Router or similar.
- **Pods:** A Pod is the minimum element in our OpenShift environment. It's strictly a Docker container running in Kubernetes.

The previous elements define an OpenShift application lifecycle.

Storage for OpenShift

How to make stateless containers have `r+w` access

💾 Storage

So far, we've seen how to deploy applications using S2I as well as Docker container easily. We also mentioned that **the official strategy when working with Docker containers is to have *stateless containers* that store data elsewhere**.

Sadly, **this is usually never the case with *self-hosted platforms*** where there are hard requirements or there aren't any cloud strategies. **In platforms like Amazon, this is easy, because you can use Amazon S3** (Simple-Storage-Service) which provides a set of APIs to store and retrieve data from specially created storage mechanisms. This, however, produces a *vendor lock-in*, since it forces you to use their APIs, **so a change in the storage mechanism usually means rewriting that portion** of the code.

Docker storage: Volumes and Mounted Volumes

Docker does include an strategy to mount host directories as part of the `docker run` flow. We saw it in action with `docker-http-server` where we overwrite the `/html` folder with our own in both Docker and Docker Compose.

There's also a secondary strategy built-in in Docker 1.12, which are **Docker Volumes**. In short, **Docker Volumes allows you to mount either network storage or securely-accessed local storage inside Docker containers**, and have transparently and seamlessly use that storage like if it were any filesystem.

This is a *huge* benefit because **you don't need to necessarily change the code to make it work**. An `mkdir -p /mnt/storage/demo && touch /mnt/storage/demo/hello.txt` will continue to work because, for the Docker container, it's a transparent operation. This doesn't mean, however, that *when* the Docker container gets destroyed, **the data stored outside the volume mounted / attached will get lost**.

Additionally, **this also allows to seamlessly attach network volumes** like `nfs` volumes or, in our case, Gluster. It will require us to talk though about `PersistentVolumes` and `PersistentVolumeClaims`.

OpenShift `PersistentVolumes` & `PersistentVolumeClaims`

💾 Persistent Volumes and Persistent Volume Claims

OpenShift expands the features provided by Kubernetes on Persistent Volumes and Volume Claims by provide business-grade features related to RedHat technologies. Let's define, *in Layman's terms* what both are:

- `PersistentVolume`: It's a way to tell the OpenShift / Kubernetes cluster **that we have storage available to use** in the Docker containers. It is like a pizza: the whole pizza is the `PersistentVolume`.
- `PersistentVolumeClaim`: Once the user needs storage, it will *request a slice of the pizza* by creating a `PersistentVolumeClaim`. It's basically **a way to request part of the storage from the Storage Pool**. It's up to the OpenShift Administrator to accept or reject their request.

Supported Storage Mechanisms

The standard OpenShift installation supports using storage mechanisms such as: `NFS`, `GlusterFS`, `Ceph RBD`, `OpenStack Cinder`, `AWS EBS`, `GCE Persistent Disk`, `iSCSI`, and `Fibre Channel`.

Other mechanisms *may be available* **by modifying the internal installation of OpenShift / Kubernetes** but this may lead to trouble in the future if an OpenShift update is available, since the transition won't be seamless.

`PersistentVolume` example with NFS

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv01
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /tmp
    server: 172.17.0.2
```

A `PersistentVolume` defines capacity based on the Kubernetes model and it also defines an access mode…

🔌 `PersistentVolumes` access modes

A defined `PersistentVolumeClaim` gets matched *automatically* with a `PersistentVolume` based on two main conditions: **the capacity requested, and the access mode available**. The available `accessModes` are:

Access Mode	CLI abbreviation	Description
`ReadWriteOnce`	`RWO`	Can be mounted as read-write by a single node
`ReadOnlyMany`	`ROX`	Can be mounted read-only by many nodes
`ReadWriteMany`	`RWX`	Can be mounted as read-write by many nodes

> ❗ Access Modes **are just a "label" in OpenShift / Kubernetes**. **They're not *enforced*** on OpenShift or Kubernetes' side, so it's the responsibility of the storage engine to enforce them and fail if needed.

`PersistentVolume` reclaim policy

Once a `PersistentVolume` is no longer in use or deleted, there's a reclaim policy which allows the Administrator to reclaim the used storage space. Unfortunately, the current OpenShift standard sets the policy as `Retain`, which means that it's up to the OpenShift Administrator to remove the unused files from the storage to regain space.

This is not because of a current issue of because of the feature was never thoroughly tested, but instead, because only two mechanisms suport the `Recycle` reclaim policy: `NFS` and `HostPath`.

Available `PersistentVolume` reclaim policies

Policy	Description
`Retain`	The administrator must manually reclaim the used space
`Recycle`	Basic scrub, as in `rm -rf /<volume>/*`

As a final note, a `PersistentVolume` description can be modified after it's created to change the reclaim policy.

`PersistentVolumeClaims`

Once the administrator has defined `PersistentVolumes`, now when creating OpenShift resources such as services, **the Developer can request storage using `PersistentVolumeClaims`**. A claim looks like this:

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "myclaim"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "1Gi"
  volumeName: "mypv01"
```

You can use the `oc create -f <filename>` command to create the claim.

Reviewing `PersistentVolume`s and `PersistentVolumeClaim`s

An user can review the `PersistentVolumeClaim`s he created by executing the `oc` command:

```
$ oc get pvc
NAME       STATUS    VOLUME    CAPACITY   ACCESSMODES   AGE
myclaim    Pending   mypv01    0                        1m
```

An Administrator can see both `PV` and `PVC`s:

```
$ oc get pvc --all-namespaces
NAMESPACE  NAME      STATUS   VOLUME   CAPAC. ACCESSMODES  AGE
example    myclaim   Pending  mypv01   0                   1m
```

If there's a match, the `STATUS` will be `Bound`.

Describing the `PersistentVolume`s will tell you **if they're bound or not to an specific claim**:

```
$ oc get pv
NAME     LABELS   CAPACITY   ACCMODE   STATUS   CLAIM
mypv01   map[]    5Gi        RWO       Bound    example / myclaim
```

In this case, in the `CLAIM` section it's listed both the namespace, `example` and the name of the claim which this volume is attached to, in this case `myclaim`.

You can also describe `PersistentVolumeClaim`s to see how they were set:

```
$ oc describe pvc/myclaim -n example
Name:           myclaim
Namespace:      example
StorageClass:
Status:         Pending
Volume:         mypv01
Labels:         <none>
Capacity:       0
Access Modes:
No events.
```

Mounting storage from the `PV` / `PVC` into the Pod

By having a bound `PVC` to a `PV`, then now it's possible to use that storage during the lifecycle of our Pod. To do so, you can create or modify a Pod and add the volume:

```yaml
apiVersion: "v1"
kind: "Pod"
spec:
  containers:
    - name: "example-pod"
      image: "patrickdappollonio/hello-docker"
      volumeMounts:
        - mountPath: "/var/www/html"
          name: "internal-volume-name"
  volumes:
    - name: "internal-volume-name"
      persistentVolumeClaim:
        claimName: "myclaim"
```

The container will mount the volume seamlessly inside /var/www/html, and the data will live inside the `PersistentVolume`, which in our case is an NFS server:

```
nfs:
  path: /tmp
  server: 172.17.0.2
```

Which means that any file created in the pod, like `/var/www/html/example.txt` will be stored at `172.17.0.2:/tmp/example.txt`. If a second container is created using the same Volume pointing to the same Claim, then **they will both share the files in the NFS server**. A full documentation on how to share the same volume across multiple pods is available here.

OpenShift Templates

Automating the Deployment of common Projects

📜 OpenShift Templates?

So far, you've seen that the whole overall process of creating projects in OpenShift **consists of pretty much repetitive tasks that you can definitely automate if there were a way to do so**. OpenShift offers the possibility of templating, which is a set of `Services`, `DeploymentConfigs`, `Routes`, `PersistentVolumeClaims` and so on which are part of a collection inside a template object.

When using this method, **you can tell OpenShift to create templates that any other user can, later on use** by changing the values you set as customizable. Those values **can be things like database usernames and passwords, certain paths, project names**, and so on.

Example OpenShift Template with Redis

```yaml
apiVersion: v1
kind: Template
metadata:
  name: redis-template
  annotations:
    description: "Description"
    iconClass: "icon-redis"
    tags: "database,nosql"
objects:
- apiVersion: v1
  kind: Pod
  metadata:
    name: redis-master
  spec:
    containers:
    - env:
      - name: REDIS_PASSWORD
        value: ${REDIS_PASSWORD}
      image: redis
      name: master
      ports:
      - containerPort: 6379
        protocol: TCP
parameters:
- description: Password used for Redis authentication
  from: '[A-Z0-9]{8}'
  generate: expression
  name: REDIS_PASSWORD
labels:
  redis: master
```

The template from the previous slide defines **a way to automate the creation of OpenShift pods that run Redis** by simply defining one object inside the `objects` list. Note that the `objects` array is pretty much embedding the YAML definition of a Pod, including even the `apiVersion`.

You can add **as many elements belonging to OpenShift** as you please inside the `objects` parameter, like `Services`, `DeploymentConfigs`, `Routes` and so on. It's up to you to mix-and-match the level of automation you decide to gain by doing so.

Then, to create the given template in OpenShift, **an administrator needs to run `oc create -f <template-file>.yaml` to make it globally available**. This process needs to run in the `openshift` namespace. If users want to make their own templates, they can also run `oc create` as well.

Later on, users can benefit from templates by creating a YAML with the parameters interpolated by issuing:

```
oc process -f <template-file> -p PARAMETER=value
```

Do note that the command above doesn't require you to have the template in OpenShift first, since you're specifying the template definition with the `-f` parameter. You can also use a pre-registered template by issuing:

```
oc process -p PARAMETER=value <template-name>
```

It's possible to list all of the parameters you can change in a template by running:

```
oc process --parameters <template-name>
```

We've seen that the previous commands just process the template into another YAML definition. But we also know that the `oc create` command is very powerful, and **allows you to pass YAML definitions to it, to create the actual services**. You can always **save the output of the `oc process` command to a file and then use `oc create`**, but you can also pipe the command as follows:

```
oc process -f <template-file> \
        -p PARAMETER=value | oc create -f -
```

Note the dash at the end of the `oc create` command, **which is a standard to receive details piped from another command**. The command above will both create the YAML definition and also use it to create all the required elements in the OpenShift environment.

Final template trick: create templates from already deployed projects

It's also possible to just create **one** project by hand with all the requirements, **and then just export everything as a template**. Eventually **you can modify the file, add your own parameters, set extra details and so on**, but it's a nice way to not start from scratch. To do so, execute the following:

```
oc export all --as-template=<template_name> \
        > my-new-template.yaml
```

This will take the entire project and export it as a template. It'll include: `BuildConfig`, `DeploymentConfig`, `ImageStream`, `Pod`, `Route`, `Service` as well as some other OpenShift extras not covered in this training.

OPENSHIFT

Health checks

Restarting pods which aren't working

Health checks

OpenShift relies on the Kubernetes probes to perform health checks against containers and decide what to do with them:

- **Liveness probe:** It's a way to detect if the container is still running. If this check fails, **it'll kill the container and based on the restart policy, it'll either restart it or keep it closed**.
- **Readiness probe:** Usually, some services may require a bit of a headstart before they can start accepting requests. This is usually true when you're loadbalancing pods, you might not want to route traffic to something that's not yet receiving traffic. **This probe checks against that and if it's not ready, it won't route traffic to it**.

Configuring Kubernetes probes

There are three ways to configure probes: just one for readiness, but two for liveness.

HTTP Checks: Kubernetes uses an HTTP request to detect the readiness of the pod:

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

Container execution check: Kubernetes will run a command inside the container and an exit status of `0` (zero) means it's live:

```
livenessProbe:
  exec:
    command:
    - "cat /tmp/health"
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

TCP Socket check: Kubernetes opens a tcp connection to the container. If it manages to establish a connection then it's considered "healthy":

```
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

OPENSHIFT

Pod autoscaling

Running multiple instances of a pod to handle more workload

Pod autoscaling

Currently, **OpenShift only supports one way to autoscale pods, which is based on CPU Usage**. Of course, you can manually scale pods by creating them yourself either using the UI or the `oc create` command with a YAML definition of a pod.

To autoscale a pod based on its CPU usage you should run the following:

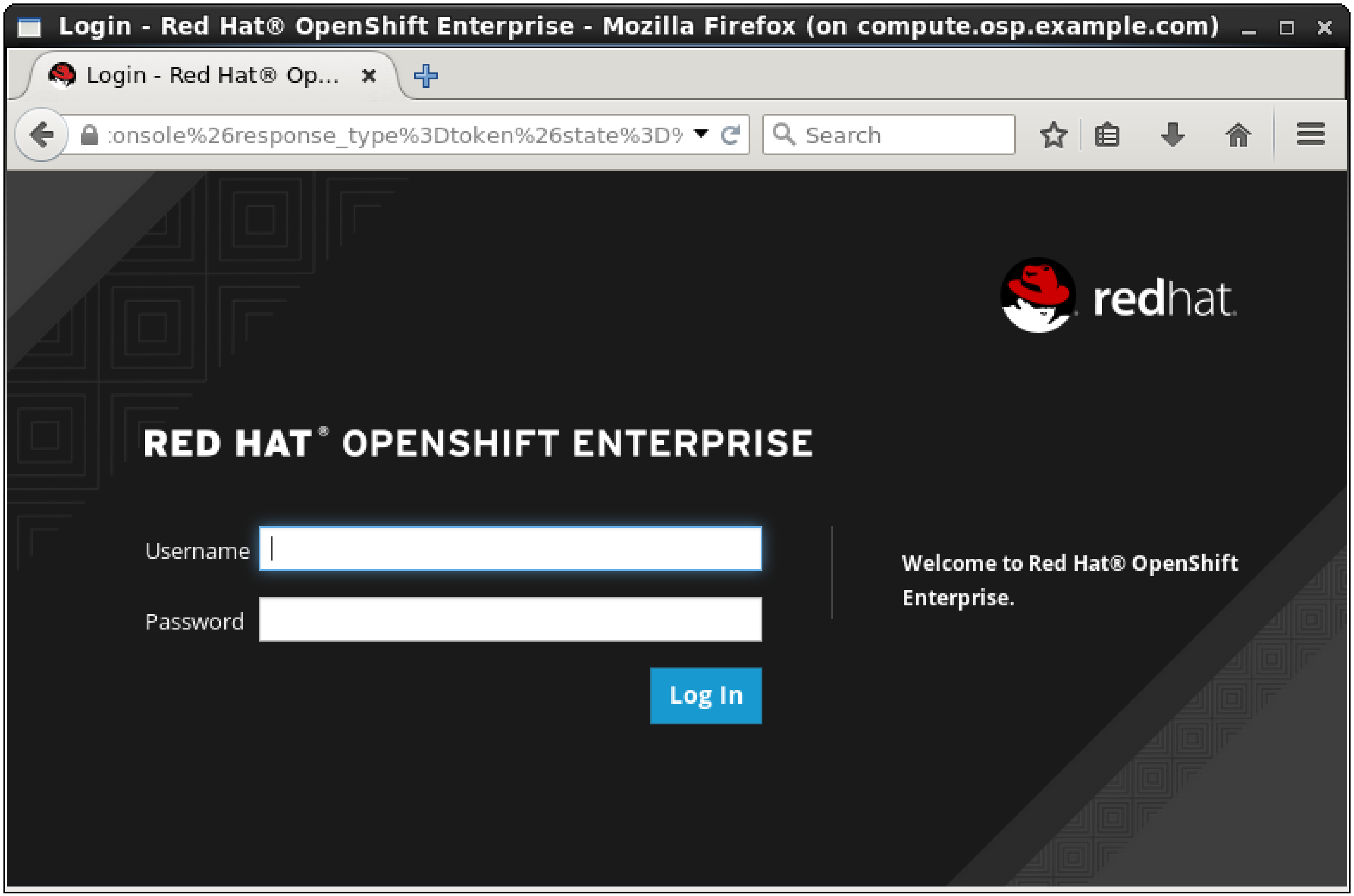
```
oc autoscale dc/<deployment-config-name> \
        --min 1 --max 10 --cpu-percent=80
```

This will autoscale the deployment configuration to add more pods if needed, based on a CPU usage of up to 80%.

OPENSHIFT

UI Walkthrough

Let's see a *button-driven* approach


Login - Red Hat® OpenShift Enterprise - Mozilla Firefox (on compute.osp.example.com)
Login - Red Hat® Op...
:onsole%26response_type%3Dtoken%26state%3D%
Search
redhat.
RED HAT® OPENSHIFT ENTERPRISE
Username
Password
Log In
Welcome to Red Hat® OpenShift Enterprise.

Final Part: Troubleshooting / Status Check

Check how healthy your OpenShift installation is

`oc adm diagnostics`

Every once in a while it's good to see what things are correct and which ones are wrong in our OpenShift installation. That's a rather easy task -- even when fixing it may not be as straightforward as detecting it -- given the fact that OpenShift includes a health check for the whole OpenShift ecosystem, as well as a full-run of the whole lifecycle of an OpenShift project.

To run it, execute the following in the master: `oc adm diagnostics`

The output is rather long and **it will report back everything that it doesn't *seem* right**. Some of those things may be related to orphan things that stayed even when the Developer removed the surrounding services, **which means that it's not necessarily harmful for your environment**, but instead, it's a report of things that may require a second look.

Finally, the OpenShift Administration Guide includes everything related to how to troubleshoot and solve problems with common stuff that may break due to unadverted changes, like:

- Router
- Nodes and synchronization
- Pod scheduling
- Deployment configuration
- Secrets

And so on. Like it was mentioned in the beginning, there difference between **OpenShift Origin**, **Container Platform**, **Enterprise** and **hosted** are still a thin line, **so *usually* the documentation for one works for all others**, unless stated otherwise.

HPE POINTNEXT

Questions?