

# OPENSIFT<sup>®</sup>

by Red Hat<sup>®</sup>

**HPE**  
POINTNEXT

**Patrick D'appollonio**

Hewlett-Packard Enterprise USA

HPE Datacenter Care - Center of Excellence



# 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

# 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 appropriate 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.

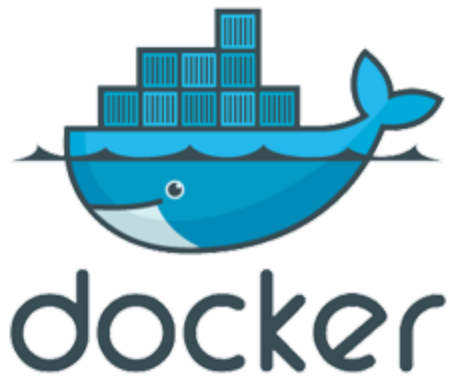
- **Web Console:** The Web Console is the easy-to-use way to manage the OpenShift installation. You can do any development task from the UI, like deploying new code, manage the number of pods running (upscaling / downscaling) as well as URI Endpoints for running pods.
- **The `oc` command:** The `oc` 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.

## How OpenShift works

A combination of all of the previously mentioned 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 thing 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 like**. 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 committed

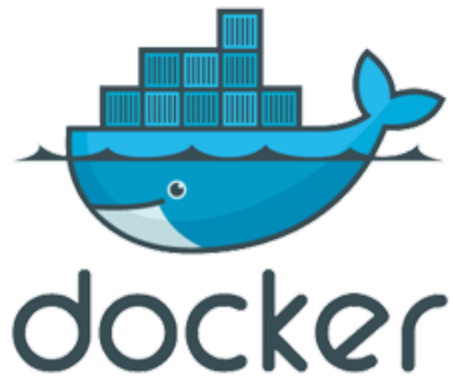
The Kernel for this Operating environment is provided by the **Host system**.



## More technical details

Docker uses a filesystem technology called *copy-on-write*. This allows 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` :

```
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, we will see something like this...

```
$ docker build .
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
... removed for brevity
---> f07e09f2be55
Removing intermediate container 4c3d349da87f
```

```
Step 4/9 : ENV APACHE_RUN_USER www-data
---> Running in a595a14a7414
---> 79272e685c7a
Removing intermediate container a595a14a7414
Step 5/9 : ENV APACHE_RUN_GROUP www-data
---> Running in df4daa26176b
---> 896f50ea4edb
Removing intermediate container df4daa26176b
Step 6/9 : ENV APACHE_LOG_DIR /var/log/apache2
---> Running in e503fa1ba60e
---> a39316289abd
Removing intermediate container e503fa1ba60e
Step 7/9 : EXPOSE 80
---> Running in 63b86325cfe2
---> fc49fed1f389
Removing intermediate container 63b86325cfe2
Step 8/9 : ADD index.html /var/www/html/
---> Running in 4us2e4k98qfk
---> v5blbws5kbsz
Step 9/9 : ENTRYPOINT ["/usr/sbin/apache2ctl",
                      "-D", "FOREGROUND"]
---> Running in nun9zl8381gi
---> mkr9twrh9zx
```

## Dissecting the Output:

- The first step pulled down the Docker `debian` image from the public [Docker Registry](#).
- Each layer has a commit SHA: the first one is `a2ff708b7413`.  
Docker works pretty much like Git: each command is committed against their filesystem.
- Each command is also a Docker container which gets stopped once it's done -- but it's not removed from your local computer storage.