Learn Git and GitHub without any code!

Using the Hello World guide, you'll start a branch, write comments, and open a pull request.

Read the guide
The goal of this lab is to install and use Docker to become familiar with Linux based containers and handle some of the common use cases around it. By the end of this lab you will have created a Web application comprised of a number of micro-services.

Lab Writers and Trainers

- Bruno.Cornec@hpe.com
- Rene.Ribaud@hpe.com

Table of Contents

- Docker Lab Contents
  - Lab Writers and Trainers
  - Objectives of the Docker Lab
  - Reference documents
  - Note on Linux commands
- Environment setup
  - Proxy consideration
  - Docker installation
    - Ubuntu installation
    - CentOS installation
    - Check installation
- Using Docker
  - The first container
  - The second container
- Configuring owncloud in a container
- Using Docker compose
  - Installing Docker compose
  - Our first docker-compose.yml file
  - Going further with docker-compose.yml
- Using docker-machine to create Docker hosts
- Using Docker Swarm
  - Installing Docker Swarm
  - Installing on CentOS 7
  - Installing the engine in the Cloud
  - Using Docker Swarm to make our configuration available and scalable
    - CentOS 7
    - Ubuntu
- Deploy a cloud native application.
  - Objectives

Objectives of the Docker Lab

At the end of the Lab students should be able to install Docker, use the CLI to create a new image, a container, launch an application in it, store data, configure the network.

This Lab is intended to be trial and error so that during the session students should understand really what is behind the tool. Blindly following instructions is not an effective way to learn IMHO. You've been warned ;)

Expected duration : 120 minutes

Reference documents

When dealing with the installation and configuration of Docker, the first step is to check the reference Web site http://docker.io/.

At the start of each section there is an estimate of how long it will take to complete.
**Note on Linux commands**

If you are familiar with Linux, you can skip this section. If not please read to understand some commands.

In a number of places, the lab uses the Linux here pattern to create text files, as in the following example,

```
# cat > fileToCreate << EOF
  Text line 1
  Text line 2
EOF
```

This command will create the text file `fileToCreate` and populate it with the lines of text that follow up but not including the `EOF` keyword.

You can display the content of the created file with the command `cat fileToCreate`.

In order to append text to the file, the first `>` can be replaced with `>>`.

If you prefer, you can edit the files using `vim` or `nano` text editors.

**But I'm a poor lonesome Windows cowboy!**

Well in that case, first condolences, then read the fine document made by Michael Mayer at [https://github.com/bcornec/Labs/blob/master/Docker/LabOnWindows.md](https://github.com/bcornec/Labs/blob/master/Docker/LabOnWindows.md) to get help.

**Environment setup**

Estimated time: 15 minutes

**Lab setup**

Please refer to the instructions available at [https://github.com/bcornec/Labs/blob/master/ENVIRONMENT.md](https://github.com/bcornec/Labs/blob/master/ENVIRONMENT.md)

Note that Docker is also providing the possibility to have a web based access to an on demand infrastructure for 4 hours. This is available at [http://play-with-docker.com](http://play-with-docker.com). Check that as a fallback you can use it and create up to 5 nodes with it. Become familiar to use this or your platform of choice for the rest of the Lab.

**Proxy consideration**

This lab is usually run in our environment that has a direct access to the Internet. If you want to run this lab on your site behind a corporate proxy, you will have to configure your Linux distribution and Docker to access the Internet via your proxy.

The following instructions are available for a Centos 7 distribution, because instructions on the Docker part are systemd based.

1. Get the proxy IP and port.
2. Make sure your host can resolve the proxy address using `nslookup <proxy>`, if not use the proxy IP.
3. Configure your Linux package manager to go through the proxy by exporting the `http_proxy` and `https_proxy` environment variables:

   ```
   export http_proxy=http://<proxy name or ip>:<proxy port>
   export https_proxy=http://<proxy name or ip>:<proxy port>
   ```

4. Configure Docker daemon to use the proxy as explained by this document: [https://docs.docker.com/engine/admin/systemd/#http-proxy](https://docs.docker.com/engine/admin/systemd/#http-proxy) (in short add `Environment="HTTP_PROXY=http://proxy.example.com:800" HTTPS_PROXY=http://proxy.example.com:800"` to your `[services]` section)

5. Build Dockerfile with the appropriate arguments.

   ```
   docker build --build-arg http_proxy=http://<proxy name or ip>:<proxy port> https_proxy=http://<proxy name or ip>:<proxy port> ...
   ```
Docker installation

Docker is available externally from [http://docs.docker.com/linux/step_one/](http://docs.docker.com/linux/step_one/) or using your distribution packages, or from github at [https://github.com/docker/docker](https://github.com/docker/docker) Version 19.03 is the current stable release. This lab requires at least version 1.7.

Ask to your instructor which Linux distribution will be used for the Lab (Docker in Docker, Ubuntu or CentOS). Then refer to the corresponding instructions below.

Other distributions should be as easy to deal with once the same packages have been installed using the package manager as they should be available directly (Case of most non-commercial distributions such as Debian, Fedora, Mageia, OpenSUSE,...). Follow the instructions from [https://docs.docker.com/engine/installation/](https://docs.docker.com/engine/installation/)

Docker in Docker installation

As the docker image already provides the required packages, you just need to check that docker is running and at the correct minimal version. Go to the Check installation paragrah below for that.

Ubuntu installation

If you work on an Ubuntu environment for the Lab, you may want to use apt to do the installation of Docker with all its dependencies. As Ubuntu provides an old version of Docker, we will use a PPA providing a more up to date version:

### 16.04 and later

```
# sudo apt-get update
# sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
# curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
# sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
# sudo apt-get update
# sudo apt-get install docker-ce
```

### 14.04

```
# apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 36A1D7869245C8959DF966E98D8576A8B88B21E9
# echo deb https://get.docker.io/ubuntu/ docker/main > /etc/apt/sources.list.d/Docker.list
# apt-get update
# apt-get install lxc-docker
```

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
aufs-tools cgroup-lite git git-man liberror-perl patch
Suggested packages:
    brtrfs-tools debootstrap lxc rinse git-daemon-run git-daemon-sysvinit git-doc
git-el git-email git-gui gitk gitweb git-arch git-gzr git-cvs git-mercurial
git-svn diffutils-doc
The following NEW packages will be installed:
aufs-tools cgroup-lite docker git git-man liberror-perl patch
0 upgraded, 7 newly installed, 0 to remove and 0 not upgraded.
Need to get 7,640 kB of archives.
After this operation, 46.9 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://fr.archive.ubuntu.com/ubuntu/ trusty/universe aufs-tools amd64 1:3.2+20130722-1.1 [92.3 kB]
Get:2 https://get.docker.io/ubuntu/ docker/main lxc-docker-1.7.0 amd64 1.7.0 [4,962 kB]
..."
Debian installation

Docker is providing deb packages to help you install the Engine on your Debian distribution:

```bash
# wget -O- https://apt.dockerproject.org/gpg | apt-key add -
# echo deb https://apt.dockerproject.org/repo debian-stretch main > /etc/apt/sources.list.d/Docker.list
# apt-get update
# apt-get install docker-engine
```

This procedure should also work for Ubuntu based distributions.

CentOS installation

If you work on a CentOS 7 environment for the Lab, you may want to use yum to do the installation of Docker with all its dependencies. Add the repo provided by the Docker project (which is requiring 7.2 at least, but not by CentOS if you use that variant):

```bash
# cat > /etc/yum.repos.d/docker.repo << EOF
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
# yum install docker-engine
```

```
Loaded plugins: fastestmirror
Determining fastest mirrors
* base: mirror.denit.net
* extras: centos.mirror.triple-it.nl
* updates: mirrors.supportex.net
base | 3.6 kB 00:00:00
dockerrepo | 2.9 kB 00:00:00
extras | 3.4 kB 00:00:00
updates | 3.4 kB 00:00:00
(1/5): base/7/x86_64/group_gz | 166 kB 00:00:00
(2/5): extras/7/x86_64/primary_db | 149 kB 00:00:00
(3/5): dockerrepo/primary_db | 34 kB 00:00:00
(4/5): updates/7/x86_64/primary_db | 2.7 MB 00:00:00
(5/5): base/7/x86_64/primary_db | 5.9 MB 00:00:00
Resolving Dependencies
---> Package docker-engine.x86_64 0:18.06.0.ce-1.el7.centos will be installed
---> Processing Dependency: docker-engine-selinux >= 18.06.0.ce-1.el7.centos.x86_64
---> Processing Dependency: libltdl.so.7()(64bit) for package: docker-engine-18.06.0.ce-1.el7.centos.x86_64
---> Running transaction check
---> Package docker-engine-selinux.noarch 0:18.06.0.ce-1.el7.centos will be installed
---> Package libltdl-ltdl.x86_64 0:2.4.2-22.el7_3 will be installed
---> Finished Dependency Resolution
Dependencies Resolved
```
# systemctl start docker

### Check installation

If you're not using the Docker in Docker environment:

```bash
# systemctl status docker
```

Now for all versions, check that the correct version is installed and operational:

```bash
# docker --version
```

Docker version 18.05.0-ce, build f150324

Note: The version could be different on your system.

```bash
# docker info
```

Containers: 0
Running: 0
Paused: 0
Stopped: 0
Images: 0
Server Version: 18.06.1-ce
Storage Driver: devicemapper
Pool Name: docker-253:2-130978-pool
Pool Blocksize: 65.54 kB
Base Device Size: 10.74 GB
Backing Filesystem: xfs
Data file: /dev/loop0
Metadata file: /dev/loop1
Data Space Used: 11.8 MB
Data Space Total: 107.4 GB

Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: null host bridge
  Operating System: Red Hat Enterprise Linux Server 7.2 (Maipo)
  OSType: linux
  Architecture: x86_64
  CPUS: 6
Total Memory: 15.39 GiB
Name: lab3.labossi.hpintelco.org
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
WARNING: bridge-nf-call-iptables is disabled
WARNING: bridge-nf-call-ip6tables is disabled
Now that the software has been installed, we'll use it to create and manage containers.

## Using Docker

Estimated time: 15 minutes.

### The first container

In order to be able to manage a first container, the easiest approach is to import an existing one, before creating your own.

For that we will refer to the public Docker registry which contains thousands of ready to be consumed containers:

```bash
# docker run hello-world
```

**Unable to find image 'hello-world:latest' locally**

latest: Pulling from hello-world

a8219747be10: Pull complete

91c95931e552: Already exists

hello-world:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.

Digest: sha256:aa03e5d0d5553b4c3473e89c8619cf79d3680bba1d8681cf5daeb82aab55838d

Status: Downloaded newer image for hello-world:latest

Hello from Docker.

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (Assuming it was not already locally available.)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```bash
$ docker run -it ubuntu bash
```

For more examples and ideas, visit:

http://docs.docker.com/userguide/

So we've got a success! Of course, we do not really go far, but what can you expect from an hello-world example ;-) 

However, we can get some info on our modified Docker environment:

```bash
# docker images or docker image ls
```

**Note:** Command line has been refined in order to be more consistent, the first command is the legacy one. The second command is the new behavior.

### docker ps -a or docker container ls -a

```bash
CONTAINER ID  IMAGE               COMMAND             CREATED             STATUS              PORTS
NAMES
4dba332aec93d hello-world         "/hello"            14 minutes ago      Exited (0) 14 minutes ago

# docker rm 4dba332aec93d
dba332aec93d
```
So we see that we now have an image which has been downloaded from the Docker public registry, and that a container has been instantiated from that image and is not running anymore. The `rm` command allows to delete the container (but of course not the image which remains available)

### The second container

In order to have a more interesting environment, we'll now look for existing container images in the public Docker registry, and choose to use a fedora image on our host environment:

```
# docker search fedora
```

Once the container image has been downloaded we can view it in our catalog of images:

```
# docker images
```

This content is called an image and will serve as the base to create the operational container (here based on Fedora) in which we will process data:

```
# docker run -ti cc510acfcd70 /bin/bash
```

```
Fedora release 28 (Twenty Eight)
```

```
# dnf install -y wget
```

```
Installing:
wget x86_64 1.19.5-1.fc28 updates 719 k
```

Transaction Summary

Install 1 Package

Total download size: 719 k
Installed size: 2.8 M

Downloaded Packages:
wget-1.19.5-1.fc28.x86_64.rpm 8.7 MB/s | 719 kB 00:00

Total 780 kB/s | 719 kB 00:01

Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
Preparing : wget-1.19.5-1.fc28.x86_64 1/1
Installing : wget-1.19.5-1.fc28.x86_64 1/1
Running scriptlet: wget-1.19.5-1.fc28.x86_64 install-info: No such file or directory for /usr/share/info/wget.info.gz
Verifying : wget-1.19.5-1.fc28.x86_64 1/1

Installed:
wget.x86_64 1.19.5-1.fc28

Complete!

[root@ad9b474525d0 /]# `uname -a`

Linux ad9b474525d0 3.16.0-41-generic #57~14.04.1-Ubuntu SMP Thu Jun 18 18:01:13 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux

If you're on a CentOS distribution it will rather be:

Linux ad9b474525d0 3.10.0-327.el7.x86_64 #1 SMP Thu Oct 29 17:29:29 EDT 2015 x86_64 x86_64 x86_64 GNU/Linux

So you checked that your container behaves like a Fedora 28 distribution. Only the kernel is shared between the Docker host and the Docker container. Open another console to view how this container has been created and is seen:

# docker ps

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ad9b474525d0</td>
<td>ded7cd95e059</td>
<td>/bin/bash</td>
<td>12 minutes ago</td>
<td>Up 12 minutes</td>
<td></td>
</tr>
</tbody>
</table>

If you logout of the container, you'll see how Docker manages that fact:

[root@ad9b474525d0 /]# `exit`

# docker ps

# docker ps -a

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ad9b474525d0</td>
<td>cc510acfcd70</td>
<td>/bin/bash</td>
<td>23 minutes ago</td>
<td>Exited (0) 4 seconds ago</td>
<td></td>
</tr>
</tbody>
</table>

So your container is seen as stopped and you have to use the -a option to see it in the history of containers created, but not active anymore.
Re-create a new container based on the same image, connect to it and look at the packages installed. Check what Docker sees. Use the previous commands to perform these tasks.

Answer the questions:

1. Can you download a web page with wget in your container? Why? Which steps are needed now? Why?
2. Can you connect back to your first container? (Hint: use docker start to re-enable your dead container and docker attach to re-enter in it)
3. Feel free to call the trainer if something is unclear or if you want to ensure you understand all points.

### Configuring owncloud in a container

Estimated time: 60 minutes.

Based on the work done in the Docker Dojo during a Grenoble Docker Meetup (cf: https://github.com/Enalean/docker-dojo/tree/master/owncloud).

Owncloud is a web based application providing services such as calendar data or file sharing e.g. When we want to contain an application such as owncloud, there are a certain number of aspects to take in account and solve:

1. installing the application and its dependencies in the container
2. allow IP configuration for remote access to the application
3. allow data persistence at each invocation of the container
4. allow configuration data persistence at each invocation of the container One possibility would be to run the container from an image and launch the various commands in the container (as we've done previously). We could put that in a script and launch it systematically when we instantiate a container from an image, or rebuild a prepared image to be instantiated later. But there is a better way to achieve what we want to do, and this is by using the automation process by Docker with the Dockerfile.

The Dockerfile is a way to describe all the operations required to create an image from an initial empty one and stacking all the operations to build at the end the final image ready to be instantiated and consumed and thrown away Let's start our Dockerfile by creating a simple container from a base image and just installing some software components useful for our environment, and build an image from that:

```
# cat > Dockerfile << EOF
FROM centos:6
RUN yum install -y httpd
EOF
```

Note: Use cat Dockerfile to see the file content.

```
# docker build .
```

```
Sending build context to Docker daemon  12.8 kB
Sending build context to Docker daemon
Step 0 : FROM centos:6
  --> a085304e4e74
Step 1 : RUN yum install -y httpd
  --> Running in ec382fd6f21b
Loaded plugins: fastestmirror
Setting up Install Process
Resolving Dependencies
  --> Running transaction check
  --> Package httpd.x86_64 0:2.2.15-39.el6.centos will be installed
  --> Processing Dependency: system-logos >= 7.92.1-1 for package: httpd-2.2.15-39.el6.centos.x86_64
  --> Processing Dependency: apr-util-ldap for package: httpd-2.2.15-39.el6.centos.x86_64
  --> Processing Dependency: /etc/mime.types for package: httpd-2.2.15-39.el6.centos.x86_64
  --> Running transaction check
  --> Package apr.x86_64 0:1.3.9-5.el6_2 will be installed
  --> Package apr-util-ldap.x86_64 0:1.3.9-3.el6_0.1 will be installed
  --> Package apr-util-lstdap.x86_64 0:1.3.9-3.el6_0.1 will be installed
```

Labs/Docker at master · bcornec/Labs https://github.com/bcornec/Labs/tree/master/D...
--- > Package httpd-tools.x86_64 0:2.2.15-39.el6.centos will be installed
--- > Package mailcap.noarch 0:2.1.31-2.el6 will be installed
--- > Package redhat-logos.noarch 0:60.0.14-12.el6.centos will be installed
--> Finished Dependency Resolution

Dependencies Resolved

<table>
<thead>
<tr>
<th>Package</th>
<th>Arch</th>
<th>Version</th>
<th>Repository</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>httpd</td>
<td>x86_64</td>
<td>2.2.15-39.el6.centos</td>
<td>base</td>
<td>825 k</td>
</tr>
<tr>
<td>Installing for dependencies:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apr</td>
<td>x86_64</td>
<td>1.3.9-5.el6_2</td>
<td>base</td>
<td>123 k</td>
</tr>
<tr>
<td>apr-util</td>
<td>x86_64</td>
<td>1.3.9-3.el6_0.1</td>
<td>base</td>
<td>87 k</td>
</tr>
<tr>
<td>apr-util-ldap</td>
<td>x86_64</td>
<td>1.3.9-3.el6_0.1</td>
<td>base</td>
<td>15 k</td>
</tr>
<tr>
<td>httpd-tools</td>
<td>x86_64</td>
<td>2.2.15-39.el6.centos</td>
<td>base</td>
<td>75 k</td>
</tr>
<tr>
<td>mailcap</td>
<td>noarch</td>
<td>2.1.31-2.el6</td>
<td>base</td>
<td>27 k</td>
</tr>
<tr>
<td>redhat-logos</td>
<td>noarch</td>
<td>60.0.14-12.el6.centos</td>
<td>base</td>
<td>15 M</td>
</tr>
</tbody>
</table>

Transaction Summary

<table>
<thead>
<tr>
<th>Install</th>
<th>7 Package(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total download size:</td>
<td>16 M</td>
</tr>
<tr>
<td>Installed size:</td>
<td>19 M</td>
</tr>
</tbody>
</table>

Downloading Packages:

<table>
<thead>
<tr>
<th>Total</th>
<th>1.0 MB/s</th>
<th>16 MB</th>
<th>00:15</th>
</tr>
</thead>
<tbody>
<tr>
<td>warning: rpmts_HdrFromFdno: Header V3 RSA/SHA1 Signature, key ID c105b9de: NOKEY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Importing GPG key 0xC105B9DE:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Userid : CentOS-6 Key (CentOS 6 Official Signing Key) <a href="mailto:centos-6-key@centos.org">centos-6-key@centos.org</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Package: centos-release-6-6.el6.centos.12.2.x86_64 (@CentOS/$releasever)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From : /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Running rpm_check_debug</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Running Transaction Test</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction Test Succeeded</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Running Transaction

Warning: RPMDB altered outside of yum.

| Installing | apr-1.3.9-5.el6_2.x86_64 | 1/7 |
| Installing | apr-util-1.3.9-3.el6_0.1.x86_64 | 2/7 |
| Installing | httpd-tools-2.2.15-39.el6.centos.x86_64 | 3/7 |
| Installing | apr-util-ldap-1.3.9-3.el6_0.1.x86_64 | 4/7 |
| Installing | mailcap-2.1.31-2.el6.noarch | 5/7 |
| Installing | redhat-logos-60.0.14-12.el6.centos.noarch | 6/7 |
| Installing | httpd-2.2.15-39.el6.centos.x86_64 | 7/7 |
| Verifying | httpd-2.2.15-39.el6.centos.x86_64 | 1/7 |
| Verifying | httpd-tools-2.2.15-39.el6.centos.x86_64 | 2/7 |
| Verifying | apr-util-ldap-1.3.9-3.el6_0.1.x86_64 | 3/7 |
| Verifying | apr-1.3.9-5.el6_2.x86_64 | 4/7 |
| Verifying | redhat-logos-60.0.14-12.el6.centos.noarch | 5/7 |
| Verifying | mailcap-2.1.31-2.el6.noarch | 6/7 |
| Verifying | apr-util-1.3.9-3.el6_0.1.x86_64 | 7/7 |

Installed:

httpd.x86_64 0:2.2.15-39.el6.centos

Dependency Installed:

apr.x86_64 0:1.3.9-5.el6_2
apr-util.x86_64 0:1.3.9-3.el6_0.1
apr-util-ldap.x86_64 0:1.3.9-3.el6_0.1
httpd-tools.x86_64 0:2.2.15-39.el6.centos
mailcap.noarch 0:2.1.31-2.el6
redhat-logos.noarch 0:60.0.14-12.el6.centos

Complete!

--> 358657a2b6b0

Removing intermediate container ec382fdfsfd21bb

Successfully built 358657a2b6b0

# docker images

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>VIRTUAL SIZE</th>
</tr>
</thead>
</table>

11 sur 30 22/08/2019 à 09:50
So we can verify that a new CentOS 6 image has been downloaded and based on it a new image has been created (without name nor tag, just an ID) containing httpd installed with its dependencies. Check it by instantiating a container based on that image and launching httpd in it:

```
# docker run -ti 358657a2b6b0 /bin/bash
```

```
[root@babbfd33b239 /]# httpd
```

httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.5 for ServerName

Note: The above message is just a warning, as an evidence, you can see the processes running with the next command.

```
[root@babbfd33b239 /]# ps auxw |grep htt
```

```
root        14  0.0  0.0 175336  6360 ?        Ss   14:36   0:00 httpd
apache      15  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
apache      16  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
apache      17  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
apache      18  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
apache      19  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
apache      20  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
apache      21  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
apache      22  0.0  0.0 175336  3824 ?        S    14:36   0:00 httpd
root        24  0.0  0.0   9728  2128 ?        S+   14:37   0:00 grep htt
```

```
[root@babbfd33b239 /]# exit
```

```
# docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                      PORTS
NAMES
babbf033b239        358657a2b6b0        "/bin/bash"         About a minute ago   Exited (0) 16 seconds ago
sharp_hodgkin
[...]
```

```
# docker diff babbf033b239
```

```
C /root
A /root/.bash_history
C /tmp
C /var
C /var/log
C /var/log/httpd
A /var/log/httpd/access_log
A /var/log/httpd/error_log
C /var/run
C /var/run/httpd
A /var/run/httpd/httpd.pid
```

```
# docker history 358657a2b6b0
```

```
IMAGE        CREATED        CREATED BY                            SIZE
358657a2b6b0 13 minutes ago /bin/sh -c yum install -y httpd       71.75 MB
a005384ae474 12 days ago    /bin/sh -c #(nop) CMD ["/bin/bash"]   0 B
fb0c588de0c8 12 days ago    /bin/sh -c #(nop) ADD file:36d5dedfe 203.1 MB
f1b18cd84249 10 weeks ago   /bin/sh -c #(nop) MAINTAINER The Cent 0 B
```

So we checked that we can launch the httpd server from inside an instantiated container made from our image. We also checked how our image was built. Note that the image built is 72 MB larger than the base CentOS 6 one (shown by history) and has sensible modifications shown by the diff command. It's a good start, but now we would like to have our httpd server started automatically with our container creation. And have attribution accordingly ;-)
# cat >> Dockerfile << EOF

MAINTAINER myself@mydomain.org
CMD httpd
EOF

# docker build .

Sending build context to Docker daemon 12.8 kB
Sending build context to Docker daemon
Step 0 : FROM centos:6
  ---> a085db84e4e7
Step 1 : RUN yum install -y httpd
  ---> Using cache
  ---> 358657a2c5f0
Step 2 : MAINTAINER myself@mydomain.org
  ---> Running in d5b4613f9ab
  ---> 1c930b9d227a
Removing intermediate container d5b4613f9ab
Step 3 : CMD httpd
  ---> Running in b9f6b0c35de95
  ---> 76ce1da7808
Removing intermediate container b9f6b0c35de95
Successfully built 76ce1da7808

You can remark that all the first steps are very quick. This is because Docker caches steps, and will not repeat them unless the Dockerfile changes. You can modify the Docker file by putting the MAINTAINER command as the second line and re-launch the build. You'll see that in that case Docker invalidates its cache and restarts. Now start a container from that image to check the web server is indeed started

# docker run -ti 76ce1da7808

httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.6 for ServerName

1. What happened ? Why don't you get a prompt ?
2. Use docker ps to see the status of your container and docker logs to see what happened.
3. Try to adapt the Dockerfile to solve that issue. Discuss with your trainer if you're stuck !

# perl -pi -e 's|D httpd|D /usr/sbin/apachectl -DFOREGROUND -k start|' Dockerfile
(This magic command replaces the launch of the httpd command by the apachectl one with the right options. If you are using CentOS 7 as the host OS, you will have to install perl via yum).

1. Try to use a browser (you may want to install lynx) to connect to your web server. Can you do it ?
2. Which IP address do you point to ? You can use docker exec to get the IP address for the container.

By default, the container ports are not exposed outside of the container. So you can't use your host OS to access your isolated webserver. If you are running the container locally navigate to http://localhost in the host browser. If the container is running on lab instrutute navigate to http://10.3.222.x (as directed by the instructor).

You will have to explicitly open container port 80 to allow access to the web server running in the container. This will require changes to the Dockerfile:

# cat >> Dockerfile << EOF

EXPOSE 80
EOF

Note : This is not mandatory, as this is only metadata for the image. However it will help people to know the exposed port of your image.

# docker build .

[...]
Successfully built 04d9c18da22a
# docker run -d -p 80:80 04d9c18da22a

# docker ps

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>04d9c18da22a</td>
<td>c1c58f087482</td>
<td>'/bin/sh -c '/usr/sbin/httpd -b 0.0.0.0:80-&gt;80/tcp'</td>
<td>4 seconds ago</td>
<td>Up 3 seconds</td>
<td></td>
</tr>
<tr>
<td>0.0.0.0:80</td>
<td>0.0.0.0:80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now that we have exposed the port, we're able to launch our container in daemon mode (-d) and redirect the local port 80 to the container port 80. Our web server is listening on port 80 in the container. Repeat the earlier attempt to connect to the web server in the container. You should see a CentOS based page on your host distribution.

It's now time to add some content to our web server! Modify again the Dockerfile to add owncloud to our image:

```bash
RUN yum install -y tar bzip2
ADD https://download.owncloud.org/community/7.0.owncloud-7.0.15.tar.bz2 /var/www/html/
# Add this only if before docker engine 17.03
# RUN cd /var/www/html/ && tar xvfj owncloud-7.0.15.tar.bz2 && rm -f owncloud-7.0.15.tar.bz2
EOF
```

We can directly point to a URL, Docker will download the content and extract it in place. Try now to connect to your owncloud instance. The URL is 

http://10.3.222.X/owncloud

if we host the lab or

http://localhost/owncloud

if you run it locally.

What happens?

1. What should you do next to solve the issue? **Discuss with your trainer if you're stuck!**

Hint, you probably need to add the owncloud dependencies to be able to launch it. Open your Dockerfile and add the following line after the last `ADD`

```bash
RUN yum install -y php php-dom php-mbstring php-pdo php-gd
```

With that you should be able to use owncloud! (Note that you need to use that version with CentOS 6 for a PHP dependency management). But we're not done yet!!! If you log on to your owncloud instance, and start customizing it (login/passwd for admin, storage path), you'll have errors first, that we'll fix later on and then if you `Docker stop` and `Docker rm` the container to relaunch it again, of course, none of this customization will be kept as it's not part of your container content.

So we now have to deal with storage management for our Docker container. First we need to solve the error generated when you tried to configure your owncloud instance. We had rights issues. Use the following command to help solve the issue:

```bash
# docker exec b42f9f6f1034 ls -al /var/www/html
# docker exec b42f9f6f1034 ps auxww | grep httpd
```
The principle is that the owner of the httpd process should have the rights on the owncloud directory to read and store files there. **So modify your Dockerfile accordingly and retest**.

Now you should be able to customize your owncloud instance and start using it.

By now you have probably remarked that the ADD order is done each time, without any benefit from the cache management of Docker. Also you have to each time deal with IDs for containers and images, which is not that convenient. Let's fix that.

Download the owncloud tar file in your directory and modify the ADD line:

```
wget https://download.owncloud.org/community/7.0/owncloud-7.0.15.tar.bz2
perl -pi -e 's|ADD https://download.owncloud.org/community/7.0/owncloud-7.0.15.tar.bz2|COPY owncloud-7.0.15.tar.bz2|' Dockerfile
```

```
docker build -t owncloud .
```

Next time you re-run the build, the cache effect is optimal. Also you now have tagged your image and use it by its name:

```
docker images
```

```
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
owncloud latest de9663de44b2 7 minutes ago 568.5 MB
```

It would be great if you could persist the content from one run to another. Yes, you can ;-) For that, you need to attach a local directory of your host to your container, and point the setup of your owncloud to that directory instead of the one under `/var/www/html/owncloud`. Create a `/data` directory on your host, mount it in your container under `/data`, and then point your setup to it:

```
mkdir -p /data
date > /data/myfile.txt
cat >> Dockerfile << EOF
VOLUME /data
EOF
docker build -t owncloud .
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
29c8f5ca3d76 de9663de44b2 /bin/sh -c '/usr/sbinowncloud' 18 minutes ago Up 18 minutes 0.0.0.0:80->80/tcp prickly_jang
```

```
docker stop 29c8f5ca3d76
docker rm 29c8f5ca3d76
docker run -d -p 80:80 -v /data:/data owncloud:latest
```

Now reload the owncloud configuration page in your browser, but this time configure the data folder as in the following screen shot:
If you encounter issues you need to adapt your environment so that the apache user is allowed to write on to the /data directory.

Your current Dockerfile should look like this at that point:

````
FROM centos:6
#FROM fedora:latest
RUN yum install -y httpd
MAINTAINER myself@mydomain.org
RUN yum install -y tar bzip2
COPY owncloud-7.0.15.tar.bz2 /var/www/html/
RUN cd /var/www/html/ && tar xvfj owncloud-7.0.15.tar.bz2 && rm -f owncloud-7.0.15.tar.bz2
RUN yum install -y php php-dom php-xmlstring php-pdo php-gd
VOLUME /data
RUN chown -R apache:apache /var/www/html/owncloud /data
CMD /usr/sbin/apachectl -DFOREGROUND -k start
EXPOSE 80
```'

Move the example text file you created earlier to your ownCloud Documents folder so you can see the file and view the file in ownCloud.

````
# mv /data/myfile.txt /data/bruno/files/Documents
```'

Open the Documents folder in the ownCloud Web UI. Confirm that the myfile.txt example file is present and then view the contents to check that they match what you created earlier.
# docker ps

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>23fb18adf4f7</td>
<td>owncloud:latest</td>
<td>'/bin/sh -c '/usr/sbin...</td>
<td>5 minutes ago</td>
<td>Up 5 minutes</td>
<td>0.0.0.0:80-&gt;80/tcp</td>
</tr>
</tbody>
</table>

# docker stop 23f

23f

# docker rm 23f

23f

# docker run -d -p 80:80 -v /data:/data owncloud:latest

cca4a1776ef12b356615e69a29751202e0fb1af5dd64fec0b618d2a797b234e

1. At that point you should find again your data on your owncloud instance right? But what additional pain point do you have?

2. Knowing that the owncloud configuration data are located under /var/www/html/owncloud/config/config.php try to adapt the Dockerfile to solve that last issue. **Discuss with your trainer if you're stuck!** Note: there is more than one way to solve this.

## Using Docker compose

Docker compose is a tool part of the Docker ecosystem. It is used to run solutions split in multiple containers which is the case most of the time. This is mainly due to the Docker philosophy to use one container per service.

Another benefit is to define the container running parameters within a YAML configuration file.

### Installing Docker compose

If you're not in the Docker in Docker setup, Use the following commands:

```bash
# curl -L https://github.com/docker/compose/releases/download/1.9.0/docker-compose-$(uname -s)-$(uname -m) > /usr/local/bin/docker-compose
```

If that command fails and you're on MacOS X (because the File System is read-only, then use the target /opt/local/sbin/docker-compose instead)

```bash
# chmod +x /usr/local/bin/docker-compose
```

Check that the binary works by displaying the revision (providing you have /usr/local/bin in your path):

```bash
# docker-compose --version

docker-compose version 1.9.0, build 2585387 **
```

### Our first docker-compose.yml file

Now we have a working docker-compose, we need to create an application environment and our first `docker-compose.yml` configuration file.

Create the build environment by moving all our previous stuffs into a folder:

```bash
# mkdir owncloud
# mv Dockerfile owncloud-7.0.15.tar.bz2 config.php owncloud
```
Now we can create our configuration file. We will use the new v3.0 format instead of the legacy one. The v3.0 was created to extend functionalities and can be activated by specifying the release at the top of the file.

Note: Of course old docker-compose binaries don't manage v3.0, you can find support information here.

```
cat > docker-compose.yml << EOF
version: '3'
services:
  web:
    build: .
    volumes:
      - /data:/data
    ports:
      - "80:80"
EOF
```

The above file asks to docker-compose to define a web service that will be built from our Dockerfile, to expose port 80 and to map /data on the host to /data in the container.

We can now start our application using:

```
docker-compose up -d
```

```
Creating network "owncloud_default" with the default driver
Creating owncloud_web_1
```

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
2573be6f140f        owncloud_web        "/bin/sh -c '/usr/sb"
35 seconds ago       Up 34 seconds
          i"         0.0.0.0:80->80/tcp
owncloud_web_1
```

Our application starts and should work the same way as previously. However it is much simpler because we don't need to define ports and storage mapping using the command line, also the YAML file can be held in and this information can be managed in Configuration Management System.

You can also note that the container name is defined as application_service_number (owncloud_web_1)

Now stop the application:

```
docker-compose down
```

```
Stopping owncloud_web_1 ... done
Removing owncloud_web_1 ... done
Removing network owncloud_default
```

Check what happens to the container.

Ok that's cool, but it is not really a big change so far.

Going further with docker-compose.yml

If we look at our owncloud application, we are using an internal sqlite database. This was defined during the setup phase.

As mentioned during the setup (below), this is convenient for a limited installation, but for larger ones it is better to use mysql/mariadb or postgres.
In order to install owncloud on another database:

1. Wipe `config.php` to have the setup page proposed again by the application.
2. Add the `php-mysql` package to your Dockerfile in the relevant part.
3. Start the application but use `docker-compose up -d --build` to force the rebuild of the Dockerfile.
Instead of building our own mariadb container from scratch like we did for owncloud, we will use the official Docker one.

Of course it requires some information about the compose-file format. Documentation for this can be found here: https://docs.docker.com/compose/compose-file and the image itself there: https://hub.docker.com/_/mariadb

Try to modify `docker-compose.yml` to add a db service based on the mariadb official images.

1. We need to provide the database parameters fields (user, password etc...). Hint: Look at the mariadb container environment variables. Discuss with your trainer if you're stuck!

2. What is the hostname of our container ? Hint: Look at the links or preferred network directive to allow db container connection from the web container.

If you didn't manage to configure the mariadb container and use it with owncloud, then the additional content for your docker-compose.yml could be useful:

```yaml
db:
  image: mariadb
  environment:
    - MYSQL_ROOT_PASSWORD=password
    - MYSQL_DATABASE=owncloud
    - MYSQL_USER=owncloud
    - MYSQL_PASSWORD=owncloudpwd
```

We are now using a mariadb container, but the database content is inside the container. So this is the same story as before, we need to keep our data persistent.
1. Find out where are managed the db files.
2. Use a Docker volume to use them from the host.
3. Modify docker-compose.yml to do that. Hint: separate owncloud and db data under /data to avoid user rights conflicts.

If you manage to configure the mariadb container with persistent data your docker-compose.yml should look like this:

```yaml
version: '3'
services:
  web:
    build: .
  db:
    image: mariadb
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=owncloud
      - MYSQL_USER=owncloud
      - MYSQL_PASSWORD=owncloudpwd
    volumes:
      - /data/db:/var/lib/mysql
    networks:
      - oclan
  owncloud:
    build: .
    volumes:
      - /data/owncloud:/data/owncloud
    ports:
      - "80:80"
    networks:
      - oclan
    depends_on:
      - db

networks:
  oclan:
    driver: bridge
```

Try to change the listening port inside your docker-compose.yml file and perform a `docker-compose -up -d`

You can notice that only the services that need to be modified are recreated.

You may like to try to allow scalability for your application by scaling the web service

```
# docker-compose scale web=2
```

Detect whether this is working or not and why. If not, we’ll find another way to solve this.

Bonus, you can try to update the docker-compose.yml file to add an ha-proxy instance in front of the web services.

**Using docker-machine to create Docker hosts**

Depending on the context of the Lab, you may already have enough machines available (5) to run the Swarm part, or you may need to create them. In that case, continue with this part, if not, skip to the next one.

docker-machine is a Docker tool that can be used to deploy Docker hosts on various cloud platforms (AWS, Azure, Digital Ocean, Openstack, etc...). We will use this tool to deploy 5 nodes that will be used later in the Swarm part. Docker machine simply deploys a server on your favorite provider and then installs the latest release of Docker Engine.

The following command will deploy one node to our openstack environment but will not run it yet.

```
# docker-machine create --driver openstack --openstack-auth-url http://10.11.50.26:5000/v2.0 --openstack-flavor-name m1.small --openstack-image-name ubuntu1604 --openstack-username dockerlab --openstack-password Linux1 --openstack-tenant-name dockerlab --openstack-net-name private --openstack-floatingip-pool external-network --openstack-sec-groups default --openstack-ssh-user ubuntu dockerw1
```
In order to save time we will deploy 5 hosts in parallel with the following command.

```bash
for i in dockerm1 dockerm2 dockerm3 dockerw1 dockerw2; do
docker-machine create --driver openstack --openstack-auth-url http://10.11.50.26:5000/v2.0 --openstack-flavor-name m1.small --openstack-image-name ubuntu1604 --openstack-username dockerlab --openstack-password linux1 --openstack-tenant-name dockerm1 --openstack-net-name private --openstack-floatingip-pool external-network --openstack-sec-groups default --openstack-ssh-user ubuntu $i &
done
```

This will take around 5mn. You can list the machines installed with the command: docker-machine ls

To connect to a server you can use: docker-machine ssh <machine_name>

Docker CLI always uses the API. So you can configure the CLI to use a remote host instead of your local Unix socket. That way your client will act as usual but instead of managing your local engine, it will manage a remote one. Example, suppose you want to interact with the dockerm1 machine. Just type the following command:

```bash
# docker-machine env dockerm1
```

The above command will provide the env variable and the command to export them in the environment. So using

```bash
# eval $(docker-machine env dockerm1)
```

you can now work with Docker as usual, however all commands passed will operate on the remote host.

### Using Docker Swarm

Docker Swarm is, since version 1.12, part of Docker Engine. It is used to provide high availability for Docker containers.

A really complete and excellent workshop is available for Swarm at https://jpetazzo.github.io/orchestration-workshop We extracted lots of ideas from it to lead you towards a first understanding of Swarm.

We will deploy a 5 nodes (3 X master + 2 X workers) cluster.

Note : If you are late on this lab, you can just use 1 X master and 2 workers, but do not stop the master in further steps.

### Installing Docker Swarm

If you have a version prior to 1.13, then you'll need to install Docker Engine 1.13+ as the rest of this lab requires that version.

### Installing on CentOS 7

On CentOS 7 just add the repo file mentioned earlier in this Lab to get it.

### Installing the engine in the Cloud

If you followed docker-machine part, you can now use these machines to configure a Swarm cluster as you have the latest version available in them.

### Using Docker Swarm to make our configuration available and scalable

So now that we can orchestrate the creation of our 2 containers hosting our application, we would like to make it scalable and error proof. Let's try to look at which nodes are available on our cluster:

```bash
# docker node ls
```

Error response from daemon: This node is not a swarm manager. Use "docker swarm init" or "docker swarm join" to connect this node to swarm and try again.

Ok, so you need first to initiate a swarm cluster ! Let's do it on our node as instructed:

```bash
# docker swarm init
```
Swarm initialized: current node (82mm398ya9d4l1or2gsc4eoo) is now a manager.

To add a worker to this swarm, run the following command:
```
docker swarm join \
  --token SWMTKN-1-444fdgnkvchgo18ck8rewshx8bvvmcqygs61mvcu8b3978ps-4r5p96yudo5r1x6c4psxd1uyt \
  10.11.51.136:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

So use the previous advise to add your other nodes to the Swarm cluster as worker.

```
# docker node ls
```

```
<table>
<thead>
<tr>
<th>ID</th>
<th>HOSTNAME</th>
<th>STATUS</th>
<th>AVAILABILITY</th>
<th>MANAGER STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2cosbse8y5o1sl2zzr4o2tc06q</td>
<td>c11.labossi.hpintelco.org</td>
<td>Ready</td>
<td>Active</td>
<td></td>
</tr>
<tr>
<td>31n32lc4wjv9oskc6eyvxz9j</td>
<td>c6.labossi.hpintelco.org</td>
<td>Ready</td>
<td>Active</td>
<td>Leader</td>
</tr>
<tr>
<td>51kz6qmid4hj7pbrq552705</td>
<td>c7.labossi.hpintelco.org</td>
<td>Ready</td>
<td>Active</td>
<td></td>
</tr>
<tr>
<td>726rg8m9foksfhjg27tw0g</td>
<td>c8.labossi.hpintelco.org</td>
<td>Ready</td>
<td>Active</td>
<td></td>
</tr>
<tr>
<td>d8dfb2e8q8d3h8w4o3o5r88f</td>
<td>c10.labossi.hpintelco.org</td>
<td>Ready</td>
<td>Active</td>
<td></td>
</tr>
</tbody>
</table>
```

Check what you can see on each node. Also look at the output of the `docker info` command.

If you have problems with error messages like "Error response from daemon: Timeout was reached before node was joined." then your firewall may be blocking the ports that Docker Swarm uses. If you think this is the case, you may have firewalling issues ;-) 

**Configuring firewall on CentOS7**


I recommend that you run the following commands on all nodes to avoid firewalling issue in the rest of the Lab:

```
# firewall-cmd --add-port=2376/tcp --permanent
# firewall-cmd --add-port=2377/tcp --permanent
# firewall-cmd --add-port=7946/tcp --permanent
# firewall-cmd --add-port=7946/udp --permanent
# firewall-cmd --add-port=4789/udp --permanent
```

But you will probably have many issues with firewalld later on anyway, so it's worth disabling it now on all nodes to avoid solving unrelated issues and integration aspects with Docker iptables management (been there done that for hours !). And believe me, I don't like that :-( (so in our Lab we have peripheral firewall !). For that use:

```
# systemctl stop firewalld  
# systemctl restart docker
```

This previous command re-establish the NAT rules set up by Docker and that have been reset just previously.

**Configuring firewall on Ubuntu**

I recommend that you run the following command on all nodes to avoid firewalling issue in the rest of the Lab:

```
# ufw disable
```

Back to our normal program now !

Swarm has the notion of worker (hosting containers), manager (able to be also a worker and being a backup leader) and Leader (manager being in charge of the Swarm cluster).

In order to render our cluster highly available, we need to have an odd number of managers. Here we can promote 2 of our workers as managers. For that, we need to get another token, the manager one, instead of the worker one we used previously.

Note : If you deployed only 3 nodes, you can not add managers, so skip this part.

```
# docker swarm join-token -q manager
```

```
So now you have the right token, use it as previously on 2 of your nodes to promote them as managers.

Example:

```
# docker node promote c7.labossi.hpintelco.org
```

At the end you should get the following result:

```
# docker node ls

ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
2cosbse8y5o1sl22r4o2tc06q    c11.labossi.hpintelco.org  Ready   Active
3ln32lc4wcjposkc6ejyxxv29j * c6.labossi.hpintelco.org   Ready   Active        Leader
51kz6qmid4blq7pjbrq5527os    c7.labossi.hpintelco.org   Ready   Active        Reachable
726rg84phkohofjpn8p2ztfg    c8.labossi.hpintelco.org   Ready   Active        Reachable
d8df82e8qjd7sh78p4o30r8bf    c10.labossi.hpintelco.org  Ready   Active

There are many ways to do it, including using docker node update.

So now that we have a cluster running, it would be a good idea to launch containers on it. But in a Swarm cluster this means creating services. So let’s create a simple service to test our cluster:

```
# docker service create alpine ping 8.8.8.8
```

```
v12wk2jruwhlftfgv9xalaped
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>
verify: Service converged
```

```
# docker service ls

ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
v12wk2jruwhl        relaxed_morse       replicated          1/1                 alpine:latest

```

```
# docker service ps v12

ID                         NAME             IMAGE          NODE                      DESIRED STATE  CURRENT STATE
ERROR               PORTS
9aq9iq25ayhp1k11ems7ts1y  relaxed_morse.1  alpine:latest c6.labossi.hpintelco.org  Running        Running 35
seconds ago

Use the Docker commands to check how the container is behaving in your environment. Restart the Docker daemon on the leader node and look at the cluster behaviour.

You can scale that service:

```
# docker service update v12 --replicas 10
```

```
v12 overall progress: 10 out of 10 tasks
1/10: running   [==================================================>
2/10: running   [==================================================>
3/10: running   [==================================================>
4/10: running   [==================================================>
5/10: running   [==================================================>
6/10: running   [==================================================>
7/10: running   [==================================================>
8/10: running   [==================================================>
9/10: running   [==================================================>
10/10: running   [==================================================>
verify: Service converged
```

Check what happens. You can use docker ps on the current node, and on another node.
In order to help visualize the state of the Swarm cluster you can use the visualizer companion of Swarm. On the master node run the following:

```bash
# docker run -it -d -p 8080:8080 -v /var/run/docker.sock:/var/run/docker.sock manomarks/visualizer
```

or using the service notion:

```bash
# docker service create --name=viz --publish=8080:8080/tcp --constraint=node.role==manager --mount=type=bind,src=/var /run/docker.sock,dst=/var/run/docker.sock dockersamples/visualizer
```

And then connect your browser to it on port 8080. You should see something similar to the below image:

![Visualizer Image](https://c6labossi теплее.org:8080)

Here you can experiment on meshing, connecting to any node, should send you to the required application.

Now let's deploy our application on our cluster. With recent versions of docker-compose, there is the new notion of stack to orchestrate services. Adapt your docker-compose to use it following the below model:

```yaml
version: '3'
services:
  web:
    image: owncloud_web   # use previously generated image (used to be build: .)
    volumes:
      - /data/owncloud:/data/owncloud
      - /data/config:/var/www/html/owncloud/config
    ports:
      - "8000:80"
    networks:
      - oclan
```
depends_on:
  - db

db:
  image: mariadb
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=password
    - MYSQL_DATABASE=owncloud
    - MYSQL_USER=owncloud
    - MYSQL_PASSWORD=owncloudpwd
  volumes:
    - /data/db:/var/lib/mysql

networks:
  - oclan

oclant:
  driver: overlay # use overlay network

Note: The overlay network is a network that will use VXLAN technology to create a private network between hosts, which could be in the different subnets.

Note 2: behind the scene the init phase of swarm did a lot of complex things, VXLAN, security (everything is on top of TLS), meshing, load balancing. Also note that load balancing on physical nodes must be achieved by an external mechanism.

Really nice job!

Now start your stack:

```
# docker stack deploy -c docker-compose-v3.yml oc
```

Ignoring unsupported options: build, links

Creating service oc_web
Creating service oc_db

```
# docker service ls
```

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>MODE</th>
<th>REPLICAS</th>
<th>IMAGE</th>
<th>PORTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>dm2j7n185u53</td>
<td>oc_db</td>
<td>replicated</td>
<td>1/1</td>
<td>mariadb:latest</td>
<td></td>
</tr>
<tr>
<td>g26e0mhakd6x</td>
<td>oc_web</td>
<td>replicated</td>
<td>1/1</td>
<td>owncloud-web:latest</td>
<td>*:80-&gt;80/tcp</td>
</tr>
<tr>
<td>v12wk2jruwhl</td>
<td>relaxed_morse</td>
<td>replicated</td>
<td>10/10</td>
<td>alpine:latest</td>
<td></td>
</tr>
</tbody>
</table>

You may have some problems with this. Try to understand what happens and solve your issues. How many replicas are working? Where are the images to use? Which node can use them? Hint: use the command `docker stack services oc` to help diagnose. And as usual talk to your instructor!

So you will need to use a private registry here to help solving that issue.

We have deployed a Docker registry for you, available from a URL that will be provided by the instructor. (If you use the internal HPE Lab, then try lab7-2.labossi.hpintelco.org:5500 - If you want to create your own, use our scripts at https://github.com/bcornec/Labs/tree/master/Docker/registry)

You need to add the CA public certificate made on the registry to trust it. Download the CA from the registry web site:

**CentOS 7**

```
# curl -L http://lab7-2.labossi.hpintelco.org/ca.crt > /etc/pki/ca-trust/source/anchors/ca-registry.crt
```

```
# update-ca-trust
```

```
# systemctl restart docker
```

**Ubuntu/Debian**

```
# curl -L http://lab7-2.labossi.hpintelco.org/ca.crt > /usr/local/share/ca-certificates/ca-registry.crt
```

```
# update-ca-certificates
```

22/08/2019 à 09:50
Check that the registry runs as expected:
```bash
# curl -L https://<my-registry-fqdn>:5500/v2
```

Of course, each node needs to be configured identically.

In order to share the image between the nodes, you need to push it to this new registry, by using the appropriate tag. For example, you may use a command similar to
```bash
# docker tag owncloud_web:latest $(DOMAIN_NAME):5500/owncloud_web
```

And then you can push that image into our registry so it's available to other engines to use.
```bash
# docker push $(DOMAIN_NAME):5500/owncloud_web
```

Do the same with the mariadb service that you create afterwards following the same approach. Look at your stack status. Is everything working fine or not? What happens if you kill the httpd process? the mysql process? Explain what is happening.

Now for the storage it's more difficult as the volumes you want to mount should be, as the images previously, available on all engines so each container created on it can use these data. One way to solve this for the mariadb image is to use an NFS exported directory from your first node. Let's configure NFS on the first machine (10.11.51.136 in my case):
```bash
# yum install -y nfs-utils # CentOS
```

or
```bash
# apt-get install -y nfs-server # Ubuntu before 18.04
```

or
```bash
# apt-get install -y nfs-kernel-server # Ubuntu 18.04
```

Edit the exports file so it looks like:
```bash
# cat /etc/exports

/data/db        *.labossi.hpintelco.org(rw,no_root_squash,async,insecure,no_subtree_check)
/data/owncloud  *.labossi.hpintelco.org(rw,no_root_squash,async,insecure,no_subtree_check)
/data/config    *.labossi.hpintelco.org(rw,no_root_squash,async,insecure,no_subtree_check)
```

```bash
# exportfs -a
# systemctl start nfs # Ubuntu before 18.04
```

or
```bash
# systemctl start nfs-kernel-server
```

Install on other nodes nfs client and check that your NFS setup is correct.

Hint:
```bash
# yum install -y nfs-utils
```
```bash
# systemctl start rpc-statd # CentOS
```

or
```bash
# apt-get install -y nfs-common
# service rpc.statd start # Ubuntu
```

Now you can create a Docker volume that will be used by the containers launched with a service, by amending your docker-compose file which should now look like this:

```yaml
version: '3'
services:
  web:
    build: .
    image: lab7-2.labossi.hpintelco.org:5500/owncloud_web
```
volumes:
  - /data/owncloud:/data/owncloud
  - /data/config:/var/www/html/owncloud/config

ports:
  - "8000:80"

networks:
  - oclan
db:
  image: mariadb
  ports:
  - "3306:3306"        # note that this port is exposed for the following part

environment:
  - MYSQL_ROOT_PASSWORD=password
  - MYSQL_DATABASE=owncloud
  - MYSQL_USER=owncloud
  - MYSQL_PASSWORD=owncloudpwd

volumes:
  - dbvol:/var/lib/mysql

networks:
  - oclan:
    driver: overlay

volumes:
  dbvol:
    driver: local
    driver_opts:
      type: nfs
      o: addr=10.11.51.136,rw,nfsvers=4.1    # this is required to have locks as nfs V3 does not support lock
    required by mariadb
    device: ":/data/db"

Restart your stack:
# docker stack rm oc

# docker stack deploy -c docker-compose-v3.yml oc

Check they have now been created with:
# docker volume ls

BTW, you can see that Docker already transparently created many more volumes for you. Note that you have to do it on all the engines of your Swarm cluster for this method to work.

Is that now working as expected? If you use Docker 17.03+ you should have the docker service logs command now to help you diagnose your issue. If not, then tip is to use docker service ps <svc_id> to find on which host runs the service and then docker exec/logs on that host e.g. Also think to the /var/log/messages log file on your host.

Can you have access to the database with the mysql command from your host (install the command if you need it)? Check that the volume is mounted correctly in the container. Check that you can reach the mysql daemon from any host in the cluster. For mysql to work correctly using an NFS exported directory for its files, you will need to have the rpc.statd daemon running on all nodes of your cluster.

Create a temporary table in the owncloud database to check and then relaunch the service to verify the persistency of the DB. MariaDB hint:

# mysql -h <one node> -uowncloud -powncloudpwd

MariaDB [(none)]> use owncloud;

MariaDB [(owncloud)]> create table toto (id int);

MariaDB [(owncloud)]> show tables;

MariaDB [(owncloud)]> quit;

Once all this is solved, you can try dealing with the web frontend. Adopt a similar approach (NFS volume and service).

Check that the communication between owncloud and the DB works fine.

You may be affected as myself by remaining bugs with previous versions of docker, such as <https://github.com/docker/docker/issues/20486> or <https://github.com/docker/docker/issues/25981>, especially mixing tests with docker-compose and swarm. For me, the only way to turn around them was to reboot the full cluster completely.

Observe what happens when you restart a Docker service on a node hosting one of the 2 services.
We can scale out such a stateful application (while less interesting than a cloud native one) with many owncloud instances to support many users and spread the load across the Swarm cluster.

PLEASE, stop your services to avoid ports conflicts with the next part.

```bash
# docker stack rm oc
```

Now we'll see the adequation of Docker Swarm and Cloud Native applications.

## Deploy a cloud native application.

Let's explain first the application and its goal.

### Objectives

In this section, we will create a promotional lottery for an e-commerce site. All the software components are provided, you'll "just" have to perform a partial containerization of the service.

As the setup takes some time, we'll start with the instructions and then you'll have time to read the explanations.

First have access to the application we developed for this.

```bash
# yum install -y git
# git clone https://github.com/bcornec/cloud_native_app.git
# cd cloud_native_app
```

As you can see in the cloud_native_app directory created, the same application can be used for a Docker or an OpenStack usage (or combining them). The application is still a WIP/Demo, so don't worry with all the additional files and directories for now. Upstream is at [https://github.com/uggla/openstack_lab.git](https://github.com/uggla/openstack_lab.git) alongside its documentation.

We need first to run the application locally using the compose file, in order to create all the Docker images and to upload them into the registry.

```bash
# ./docker_services.sh
```

Drink a coffee, it's well deserved at that point, the composition takes a bit of time. Or stay looking at it to observe closely the magic of Docker automation ;-) Please start reading the following explanations in or to understand what we're building for you here.

A customer of a big e-commerce site receives a promotional email with a link to win a prize if they are lucky. The application detects whether the player already played or not, and whether he won already or not. Each status is kept with the date when it was performed. The application provides a button allowing the customer to play, in case he didn't already, and the result of the computation which happens behind the scene is given back to the customer: it is the nature of the article he has won, and the corresponding image is displayed in the interface. Mails are sent to admins when a winner is found.

That application is made of one Web page with 5 parts/micro-services: I, S, B, W and P:

- **I(dentification) service**: receives http request from customer (link with customer ID) and look for it into the DB.
- **S(tatus) service**: detect whether customer already played or not, status stored in the DB. It is using a messages bus to buffer requests.
- **B(utton) service**: button widget allowing the customer to play. Only when not already done.
- **W(orker) service** that computes whether the customer won or not (slow service on purpose with a REST API interface), called by B. If won, post an image representing what has been won into an object store with customer ID. Then post by e-mail via an external provider a message to admins (using a messages bus). Button is gray if the customer has already played. W and the DB are on a separate private network.
- **P(icture) service**: Look into the object store with customer ID to display the image of the customer's prize, empty if no image.

Each part of the web page is implemented as a micro-service. So the application supports nicely the death of any one of the 5 micro-services. The page is still displayed anyway, printing N/A when a micro-service is unavailable. In case of insufficient resources (as with the slow W micro-service), we will look at how to scale that application.
Please have a look at the `docker_services.sh` script and adapt what needs to be changed for your environment at the start in case of issues.

At the end of the script you should get a list of services running similar to the one below:

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>REPLICAS</th>
<th>IMAGE</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1empjc0n6a8u</td>
<td>w</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_w</td>
<td></td>
</tr>
<tr>
<td>1zsf6ruvvyjm9</td>
<td>i</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_i</td>
<td></td>
</tr>
<tr>
<td>3gasrkzgpp8w</td>
<td>b</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_b</td>
<td></td>
</tr>
<tr>
<td>3sc3qexaxkl1</td>
<td>redis</td>
<td>1/1</td>
<td>redis</td>
<td></td>
</tr>
<tr>
<td>4c512jiwymh</td>
<td>myownsvc</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/owncloud_web</td>
<td></td>
</tr>
<tr>
<td>5y71166s0a0v4</td>
<td>w2</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_w2</td>
<td></td>
</tr>
<tr>
<td>6ldkc0f12cz8</td>
<td>ping</td>
<td>global</td>
<td>alpine</td>
<td>ping 8.8.8.8</td>
</tr>
<tr>
<td>79jwqr43zyt2</td>
<td>web</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_web</td>
<td></td>
</tr>
<tr>
<td>7hyg6a0bbyq</td>
<td>db</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_db</td>
<td></td>
</tr>
<tr>
<td>9i4oqen03ak3</td>
<td>rabbit</td>
<td>1/1</td>
<td>rabbitmq:3-management</td>
<td></td>
</tr>
<tr>
<td>ag12qqf0ts417</td>
<td>tiny_curran</td>
<td>10/10</td>
<td>alpine</td>
<td>ping 8.8.8.8</td>
</tr>
<tr>
<td>a3jcrq0nywn8</td>
<td>s</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_s</td>
<td></td>
</tr>
<tr>
<td>cn82ua65by1</td>
<td>w1</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_w1</td>
<td></td>
</tr>
<tr>
<td>e6c6ngcxdxy2</td>
<td>p</td>
<td>1/1</td>
<td>lab7-2.labossi.hpintelco.org:5500/cloudnativeapp_p</td>
<td></td>
</tr>
</tbody>
</table>

In order to use the application you'll now have to connect to your system hosting the web application (in our case [http://c6.labossi.hpintelco.org/](http://c6.labossi.hpintelco.org/))

You should see a message in your browser saying:

Please provide a user id !

So now to use the application, you have to provide the id of the user who is playing to see his prize. Browse [http://c6.labossi.hpintelco.org/index.html?id=1](http://c6.labossi.hpintelco.org/index.html?id=1)

Check the availability of the application by restarting a docker daemon on a host running one of the containers the application is using. Check the micro-service behavior by stopping the 'i' micro-service, and then the 'p' micro-service. Reload the Web page each time to see what happens.

Try to make more connections. What is the problem encountered. Which micro-service is causing the issue. Scale that micro-service to solve the problem.

This is the end of this lab for now, we hope you enjoyed it.

Github issues and pull requests to improve this lab are welcome.