



UNIT 6.

ACTIVITY 2: DEPLOY DOCKER ON VPS

Web Applications Deployment
CFGS DAW

Important: this activity is not mandatory and does not compute for the final grade.

Author: Carlos Cacho
Reviewed by: Lionel Tarazón
Reviewed by: Pau Miñana
2020/2021

INDEX

1 INTRODUCTION.....	3
2 INSTALL DOCKER.....	4
3 DEPLOY PHP APPLICATIONS.....	6
4 DEPLOY JAVA APPLICATIONS.....	10
5 DEPLOY NODE.JS APPLICATIONS.....	15
6 NGINX SERVER BLOCKS LOAD BALANCER.....	16
7 EXTRA: KITEMATIC.....	19

UT06. WEB APPLICATIONS DEPLOYMENT

ACTIVITY: DEPLOY DOCKER ON VPS

1 INTRODUCTION

In this activity we are going to simulate a deploy of different applications using Dockers on a VPS (Virtual Private Server).

We will create 4 Nginx server blocks to deploy applications written in simple PHP, Laravel, Java and Node.js. To do so, we will use different Dockers to deploy the applications and one Docker with the Nginx server to do the server blocks balancer.

To simulate the VPS create a **new Linux virtual machine (Ubuntu or similar)**. Make sure it's a 64 bits machine and operating system. 20 GB of disk space should be enough.

Remember we need to have an Internet connection in the virtual machine to download everything. As we did in previous activities, make sure to set the virtual machine's network to **Bridge Adapter** and configure the operating system's network to an IP address of your local network.

In my case, I am going to use the IP address 192.168.1.2, netmask 255.255.255.0 and gateway 192.168.1.1. Leave the DNS in automatic

The screenshot shows a network configuration window titled 'Cableada'. It has tabs for 'Detalles', 'Identidad', 'IPv4', 'IPv6', and 'Seguridad'. The 'IPv4' tab is active. Under 'Método IPv4', 'Manual' is selected. Under 'Direcciones', the first row shows 'Dirección' as 192.168.1.2, 'Máscara de red' as 255.255.255.0, and 'Puerta de enlace' as 192.168.1.1. The 'DNS' section has a toggle for 'Automático' which is turned on. The 'Rutas' section also has a toggle for 'Automático' which is turned on. There are buttons for 'Cancelar', 'Cableada', and 'Aplicar' at the top.

2 INSTALL DOCKER

Now, we have to install Docker Engine Community following the installation guide:

<https://docs.docker.com/engine/install/ubuntu/>

Install using the repository

Before you install Docker Engine - Community for the first time on a new host machine, you need to set up the Docker repository. Afterward, you can install and update Docker from the repository.

SET UP THE REPOSITORY

1. Update the `apt` package Index:

```
$ sudo apt-get update
```

2. Install packages to allow `apt` to use a repository over HTTPS:

```
$ sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  software-properties-common
```

3. Add Docker's official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Verify that you now have the key with the fingerprint

9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88 , by searching for the last 8 characters of the fingerprint.

```
$ sudo apt-key fingerprint 0EBFCD88

pub   rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid           [ unknown] Docker Release (CE deb) <docker@docker.com>
sub   rsa4096 2017-02-22 [S]
```

4. Use the following command to set up the **stable** repository. To add the **nightly** or **test** repository, add the word `nightly` or `test` (or both) after the word `stable` in the commands below. [Learn about nightly and test channels.](#)

```
$ sudo add-apt-repository \
  "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
  ${lsb_release -cs} \
  stable"
```

INSTALL DOCKER ENGINE - COMMUNITY

1. Update the `apt` package index.

```
$ sudo apt-get update
```

2. Install the *latest version* of Docker Engine - Community and containerd, or go to the next step to install a specific version:

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

4. Verify that Docker Engine - Community is installed correctly by running the `hello-world` image.

```
$ sudo docker run hello-world
```

```
adminvps@Dockervps:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:d1668a9a1f5b42ed3f46b70b9cb7c88fd8bdc8a2d73509bb0041cf436018fbf5
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.
    (amd64)
 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:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

As you can see, **`docker run`** command download and install container images hosted on dockerhub, so you can use any public image from there.

If you want to give SU privileges to docker so `sudo` isn't needed to be included with docker commands you can use the command **`sudo usermod -aG docker USERNAME`**. Without this command you'll need to add "sudo" to docker commands in this activity

Be careful with **docker run** command, as it actually means “install”; any time you write **docker run** a new copy of that container is installed. To stop/run previously installed container **docker stop/start CONTAINER_ID** are the commands to go.

```
osboxes@osboxes:~$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
NAMES
d53b3a2d3d61   hello-world    "/hello"                 2 minutes ago   Exited (0) 2 minutes ago
elegant_einstein
212db2bbc381   hello-world    "/hello"                 6 minutes ago   Exited (0) 6 minutes ago
sweet_saha
80181b9cef8d   pauminyana/php "/usr/sbin/apache2 -..." 2 weeks ago     Exited (0) 2 weeks ago
hopeful_goldwasser
0ff313293a99   sergarb1/bda-ceedcv-oracle-xe-11g:latest "/bin/sh -c /start.sh"    3 weeks ago     Exited (137) 5 seconds ago
bda-ceedcv-oracle-xe-11g
osboxes@osboxes:~$ docker start 0ff313293a99
0ff313293a99
```

Command **docker ps -a** is used to show installed containers so you could know their ID and status. Without option -a, **docker ps** it just shows active (started) containers.

Another useful command, **docker images**, shows already downloaded or built images in your docker folder and their size (not installed containers). As you can see, Ubuntu 16 basic image we'll use next takes just 131 MB.

```
osboxes@osboxes:~$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
pauminyana/php      latest       dc7c6d4f3036     3 weeks ago     311MB
ubuntu              16.04       9499db781771     6 weeks ago     131MB
hello-world         latest      bf756fb1ae65     12 months ago   13.3kB
sergarb1/bda-ceedcv-oracle-xe-11g latest       026e57a70881     2 years ago     3.71GB
```

3 DEPLOY PHP APPLICATIONS

So, with docker we can search and install container images and configure them to our liking but that's not all. Even if we can find a lot of images that could fit our needs, we can build our own ones too. We'll start with really basic builds, but with enough experience, your builds could be as complete as you could need.

First we are going to deploy a simple PHP application (we'll just deploy the info page). To do so we have to create a directory called **php** which will contain the Dockerfile needed to create the Docker image.

```
adminvps@Dockervps:~$ mkdir php
adminvps@Dockervps:~$ cd php
```

```
adminvps@Dockervps:~/php$ gedit Dockerfile
Dockerfile (~/.php) - gedit
Open [icon] Save
FROM ubuntu:16.04

RUN apt-get update -y
RUN apt-get install -y git curl apache2 php7.0 libapache2-mod-php7.0 php7.0-mcrypt php7.0-mysql

RUN rm -rf /var/www/*
ADD src /var/www/html

RUN a2enmod rewrite
RUN chown -R www-data:www-data /var/www
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_LOCK_DIR /var/lock/apache2

EXPOSE 80

CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

¿What is Dockerfile? <https://docs.docker.com/engine/reference/builder/>

Docker can build images automatically by reading the instructions from a `Dockerfile`. A `Dockerfile` is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build` users can create an automated build that executes several command-line instructions in succession.

The explanation of the instructions is the following:

FROM ubuntu:16.04

The Docker image will use Ubuntu 16.04 from the Docker Hub.

RUN apt-get update -y

RUN apt-get install -y git curl apache2 php7.0 libapache2-mod-php7.0 php7.0-mcrypt php7.0-mysql

Install in our image all the packages we need to work.

*RUN rm -rf /var/www/**

ADD src /var/www/html

Remove all the files and folders inside `/var/www/` (apache default webpage) and add `src` folder in dockerfile location to `/var/www/html`. We'll create `src` folder later with php app to deploy.

RUN a2enmod rewrite

Enable the module `rewrite` for rewrite URLs in a more friendly way.

RUN chown -R www-data:www-data /var/www

Assign the user `www-data` and the group `www-data` to the directory `/var/www`.


```
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_LOCK_DIR /var/lock/apache2
```

Create all the environment variables we need.

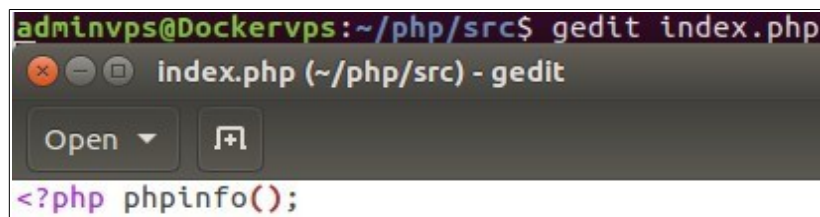
EXPOSE 80

The container will listen to the port 80; it does not actually publish the port, just a warning about which ports are intended to be published. For actually mapping ports we'll see the use of `-p` on `docker run` at image installation.


```
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

Execute Apache in foreground when the container launches.

Now we create the directory called `src` with the php application (in this case just `index.php` with the info page) so it can be added to the image:



And now, we create the Docker image from the directory php:

 I tag the image using my Dockerhub username just in case I want to push it to Dockerhub. You can use your own user or any other name tag.

`docker build -t imagename .` (do not forget last dot)

```
osboxes@osboxes:~/php$ docker build -t pauminyana2/php .
Sending build context to Docker daemon 3.584kB
Step 1/14 : FROM ubuntu:16.04
--> 9499db781771
Step 2/14 : RUN apt-get update -y
--> Using cache
--> f6f516bd66d2
Step 3/14 : RUN apt-get install -y git curl apache2 php7.
--> Using cache
--> 90365a82ac43
```



```

Step 13/14 : EXPOSE 80
---> Using cache
---> a1a664c02d7e
Step 14/14 : CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
---> Using cache
---> dc7c6d4f3036
Successfully built dc7c6d4f3036
Successfully tagged pauminyana/php:latest

```

Now we can see that the image we created is available; actually it already was available in this activity screenshots, as you must have seen, that's why Ubuntu image was already there.

```

osboxes@osboxes:~/php$ docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
pauminyana/php	latest	dc7c6d4f3036	3 weeks ago	311MB
ubuntu	16.04	9499db781771	6 weeks ago	131MB
hello-world	latest	bf756fb1ae65	12 months ago	13.3kB
sergarb1/bda-ceedcv-oracle-xe-11g	latest	026e57a70881	2 years ago	3.71GB

Now we run (install) a container of the image:

docker run -p 80:80 imagename

-p hostportnumber:virtualportnumber option to bind host machine ports to container virtual ports (If using VM as suggested, actually host machine port it's virtual too, but it doesn't really matters).

```

osboxes@osboxes:~/php$ docker run -p 80:80 pauminyana/php
AH00558: apache2: Could not reliably determine the server's fully qualified domain name,
globally to suppress this message

```

Do not close this terminal (the container would stop). Now we can see our application deployed and we can connect from our physical machine as they are in the same network:



Remember, to start or stop already installed containers use ***docker start ID*** or ***docker stop ID***, not docker run.

4 DEPLOY JAVA APPLICATIONS

Now we are going to deploy Java applications in a Tomcat sever. In this case instead of building the image we are going to pull the official tomcat image from DockerHub. We are going to use version 8.5.32 because latest is unstable.

docker pull tomcat:8.5.32

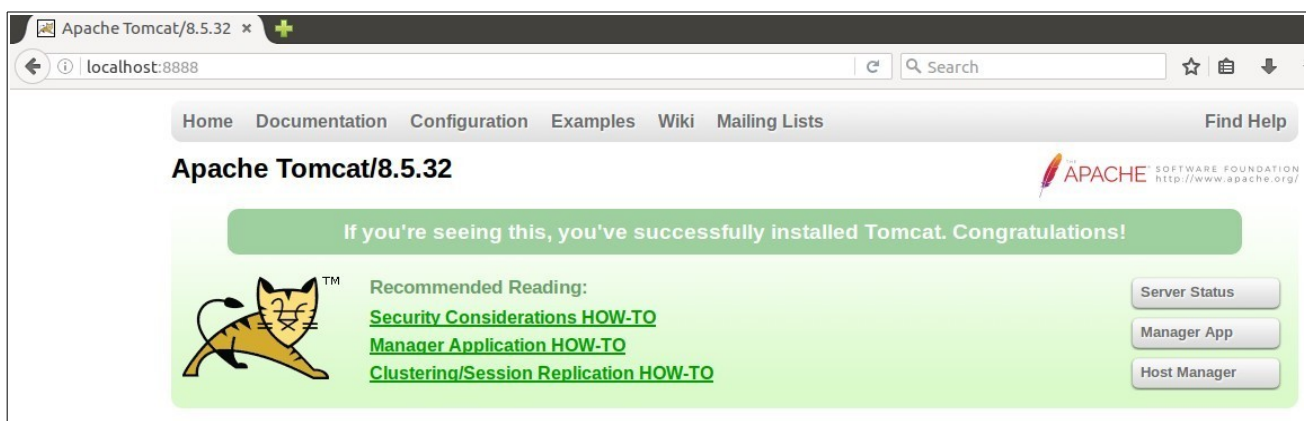
```
adminvps@Dockervps:~$ docker pull tomcat:8.5.32
8.5.32: Pulling from library/tomcat
55cbf04beb70: Pull complete
1607093a898c: Pull complete
9a8ea045c926: Pull complete
1290813abd9d: Pull complete
8a6b982ad6d7: Pull complete
abb029e68402: Pull complete
d068d0a738e5: Pull complete
42ee47bb0c52: Pull complete
ae9c861aed25: Pull complete
60bba9d0dc8d: Pull complete
15222e409530: Pull complete
2dcc81b69024: Pull complete
Digest: sha256:bbdb0de8298ab7281ff28331a9e4129562820ac54e243e44c3749f389876f562
Status: Downloaded newer image for tomcat:8.5.32
docker.io/library/tomcat:8.5.32
```

Lets run the container exposing its 8080 port (Tomcat's default port) to the machine's 8888 port.

docker run -p 8888:8080 tomcat:8.5.32

🔊 Important: Do not close the terminal, or restart tomcat container.

Then we can access the Tomcat server using localhost:8888



To deploy we can use the **Manager App**, but if we click on the **Manager App** button now we do not have permissions. One reason is that we have not enabled any user to access it, and the other reason is that since Tomcat 8 access is only allowed from the local machine by default. Therefore we need to change both things in Tomcat configuration so that later we can deploy outside of the Docker.

We need to start the container in interactive mode to be able to run bash commands and do the changes. So, if you haven't restarted container first type **control+C** to stop it and restart with **docker start CONTAINER_ID**; then, to open a terminal INSIDE the container environment run:

docker exec -it CONTAINER_ID /bin/bash (execute bash in container with interactive mode)

```
osboxes@osboxes:~$ docker ps -a
CONTAINER ID   IMAGE                                NAMES
0b6beac6d9e2   tomcat:8.5.32                      upbeat_dhawan
2c10c6f66bd7   pauminyana/php                     hopeful_fermat
0ff313293a99   sergarb1/bda-ceedcv-oracle-xe-11g:latest
49158->1521/tcp, 0.0.0.0:49157->8080/tcp bda-ceedcv-oracle
osboxes@osboxes:~$ docker start 0b6beac6d9e2
0b6beac6d9e2
osboxes@osboxes:~$ docker exec -it 0b6beac6d9e2 /bin/bash
root@0b6beac6d9e2:/usr/local/tomcat#
```

To create a user in Tomcat we have to modify **/usr/local/tomcat/conf/tomcat-users.xml** file. But first we need to install a command-line editor (a graphical editor such as gedit will not work, as container hasn't GUI).

apt-get update

apt-get install nano

nano conf/tomcat-users.xml

Go to the end of the file and add role "manager-gui" and user "tomcat" as in the example below.

(You can use NOTE commentaries just above the end to copy-paste lines, but **Ctrl+c/Ctrl+v** won't work, use mouse or **Ctrl+shift+c/Ctrl+Shift+v**)

```

adminvps@Dockervps: ~
GNU nano 2.7.4 File: conf/tomcat-users.xml Modified

NOTE: The sample user and role entries below are intended for use with the
examples web application. They are wrapped in a comment and thus are ignored
when reading this file. If you wish to configure these users for use with the
examples web application, do not forget to remove the <!-- ... --> that surrounds
them. You will also need to set the passwords to something appropriate.
-->
<!--
<role rolename="tomcat"/>
<role rolename="role1"/>
<user username="tomcat" password="<must-be-changed>" roles="tomcat"/>
<user username="both" password="<must-be-changed>" roles="tomcat,role1"/>
<user username="role1" password="<must-be-changed>" roles="role1"/>
-->
<role rolename="manager-gui"/>
<user username="tomcat" password="s3cret" roles="manager-gui"/>
</tomcat-users>

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

```

Type **Ctrl+O** to save and **Ctrl+X** to exit nano.

Now we are going to allow connections from any machine. To do so we need to modify file `/usr/local/tomcat/webapps/manager/META-INF/context.xml`:

nano webapps/manager/META-INF/context.xml

You'll see something like this:

```

adminvps@Dockervps: ~
GNU nano 2.7.4 File: context.xml

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
-->
<Context antiResourceLocking="false" privileged="true" >
  <Valve className="org.apache.catalina.valves.RemoteAddrValve"
    allow="127\.\d+\.\d+\.\d+:::1|0:0:0:0:0:0:0:1" />
  <Manager sessionAttributeValueClassNameFilter="java\.lang\.(?:Boolean|Integer|S
</Context>

```

We have to change the **Valve's** option **allow** (we can see that the regular expression is for an address IP that starts with 127, it means, localhost) to this

regular expression: `^.*$` (match, from beginning to end, any character that appears zero or more times).

```
adminvps@Dockervps: ~
GNU nano 2.7.4 File: context.xml Modified

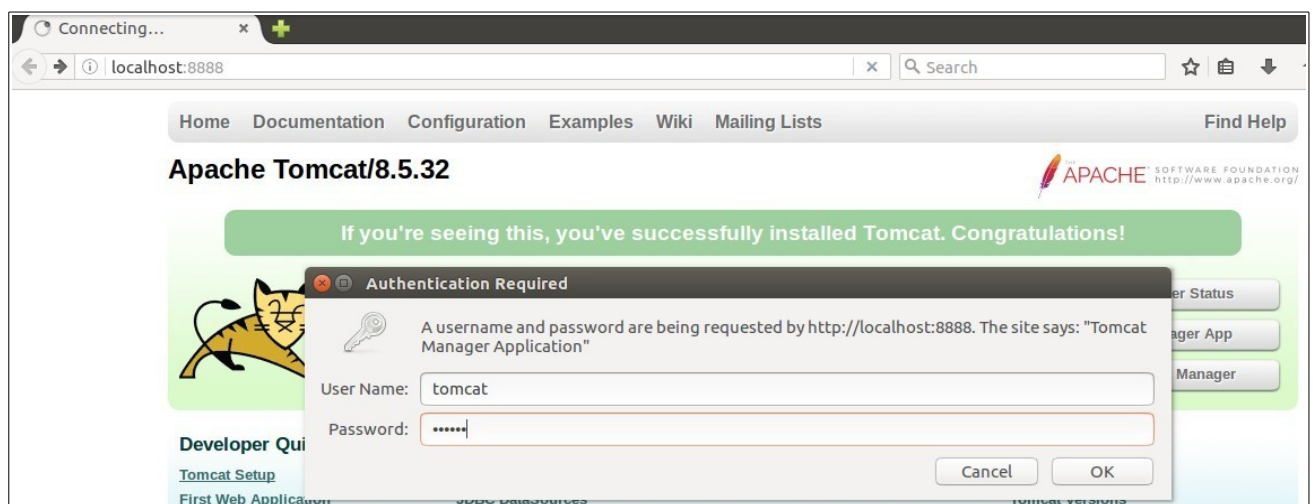
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
-->
<Context antiResourceLocking="false" privileged="true" >
  <Valve className="org.apache.catalina.valves.RemoteAddrValve"
    allow="^.*$" />
  <Manager sessionAttributeValueClassNameFilter="java\\.lang\\.(?:Boolean|Integer|
</Context>
```

Save, close nano, and type `exit` to quit container's terminal mode. Actually we'd need to restart tomcat to apply changes, but it's just easier restart container itself.

```
root@0b6beac6d9e2:/usr/local/tomcat# exit
exit
osboxes@osboxes:~$ docker stop 0b6beac6d9e2
0b6beac6d9e2
osboxes@osboxes:~$ docker start 0b6beac6d9e2
0b6beac6d9e2
```

Now if we try to access Tomcat's **Manager App** we can log in using the tomcat user previously created.

User: tomcat Password: s3cret



The same will happen if we connect from our physical machine (they are in the same network).

And finally we are going to deploy a Java application. The easiest way to do it is with a war file. For that we are going to download the file *appJava.war* from the virtual classroom and then choose the file in the deploy section:

🔊 A *war* (Web Application Resource or Web application ARchive) file is a file used to distribute a collection of JAR-files, JavaServer Pages, Java Servlets, Java classes, etc.

Deploy

Deploy directory or WAR file located on server

Context Path (required):

XML Configuration file URL:

WAR or Directory URL:

WAR file to deploy

Select WAR file to upload: No file selected.

Then click on *Deploy* button. Now we can see the application has been deployed:

Applications				
Path	Version	Display Name	Running	Sessions
/	None specified	Welcome to Tomcat	true	<u>0</u>
<u>/appJava</u>	None specified		true	<u>0</u>
<u>/docs</u>	None specified	Tomcat Documentation	true	<u>0</u>
<u>/examples</u>	None specified	Servlet and JSP Examples	true	<u>0</u>

We can access to the application at *localhost:8888/appJava/NewServlet* or with any machine connected to our network via VM's ip

5 DEPLOY NODE.JS APPLICATIONS

Now we are going to deploy Node.js applications. First of all we have to stop the tomcat



container:

docker stop CONTAINER_ID

Lets clone a simple Node.js app from GitLab located in https://gitlab.com/lionel_ceedcv/appnode.git

git clone https://gitlab.com/lionel_ceedcv/appnode.git

Inside the appnode folder the project has three files: Readme.md, app.js and package.json. In this directory we are going to create the Dockerfile we will use to create a Docker image. I'll let to yourselves the understanding of the following dockerfile instructions

```
FROM node:14
WORKDIR /app
COPY package.json ./
RUN npm install npm@7
ADD . /app
EXPOSE 8000
CMD ["npm", "start"]
```

Now we create the Docker image and install the container mapping port 8000

docker build -t imagename .

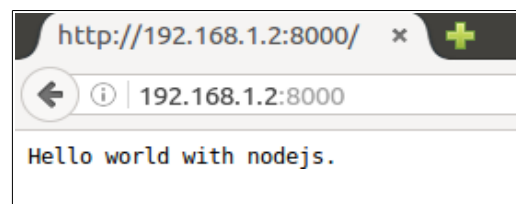
docker run -p 8000:8000 imagename


```

osboxes@osboxes:~/nodejs/appnode$ docker build -t pauminyana/nodejs .
Sending build context to Docker daemon 67.07kB
Step 1/7 : FROM node:14
Successfully built bb1525e9cb23
Successfully tagged pauminyana/nodejs:latest
osboxes@osboxes:~/nodejs/appnode$ docker run -p 8000:8000 pauminyana/nodejs
> hello-world@0.0.1 start /app
> node app.js

```

And we can see the application deployed:

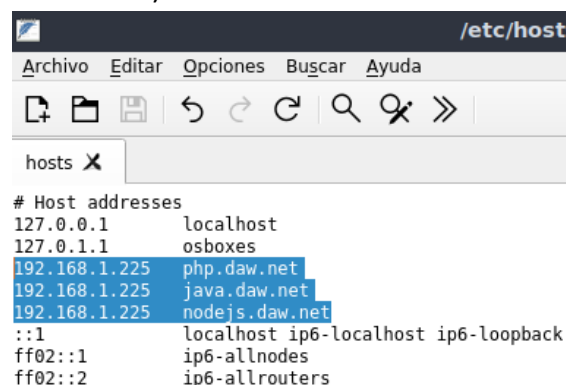


6 NGINX SERVER BLOCKS LOAD BALANCER

Finally we are going to create a balancer for the PHP, Java and Nodejs applications. We are going to create a reverse proxy configured in Nginx. For that we are going to use the [jwilder/nginx-proxy](#) project which is useful to do this balancer.

But first of all, we must stop all our prior containers. You can even remove them with `docker rm CONTAINER_ID`, as we are going to reinstall them with a little change. You could even remove images you won't need (**DON'T DO IT**, we are still using them), if there's not any container of them installed, with `docker rmi IMAGE_ID` or `docker rmi IMAGE_NAME`.

Now we have to create the subdomains for our server blocks. To configure them faster we are going to use the file `/etc/hosts` of our virtual machine VPS. So we edit the file adding three lines, one for each server block and all three with the ip address of the server (in my case 192.168.1.225):



```

# Host addresses
127.0.0.1    localhost
127.0.1.1    osboxes
192.168.1.225 php.daw.net
192.168.1.225 java.daw.net
192.168.1.225 nodejs.daw.net
::1         localhost ip6-localhost ip6-loopback
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters

```

Now we are going to pull the docker image:

docker pull jwilder/nginx-proxy

```
osboxes@osboxes:/$ docker pull jwilder/nginx-proxy
Using default tag: latest
latest: Pulling from jwilder/nginx-proxy
bb79b6b2107f: Pull complete
111447d5894d: Pull complete
a95689b8e6cb: Pull complete
1a0022e444c2: Pull complete
32b7488a3833: Pull complete
c45cf71bc68c: Pull complete
00d28348da58: Pull complete
190f1720abb7: Pull complete
a99149d671c6: Pull complete
c7e9f170fd7c: Pull complete
accc49186289: Pull complete
e67d699eb6f8: Pull complete
Digest: sha256:695db064e3c07ed052ea887b853ffba07e8f0fbe96dc01aa350a0d202746926b
Status: Downloaded newer image for jwilder/nginx-proxy:latest
docker.io/jwilder/nginx-proxy:latest
```

And now we install the image on port 80:

docker run -d -p 80:80 -v /var/run/docker.sock:/tmp/docker.sock jwilder/nginx-proxy

🔊 We use the options:

- d: to start the container in detached mode. By design, containers started in detached mode exit when the root process used to run the container exits.
- v: mounts the current working directory into the container as a volume;
This way we can create shared folders between docker container and host machine, just as we do with ports: -v hostfolderpath:virtualfolderpath

To configure the balancer we have to create again a container for each server block (*php.daw.net*, *java.daw.net* and *nodejs.daw.net*).

docker run -d -e VIRTUAL_HOST=serverblockname imagename

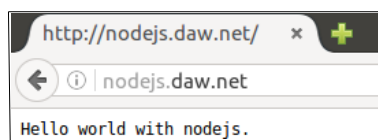
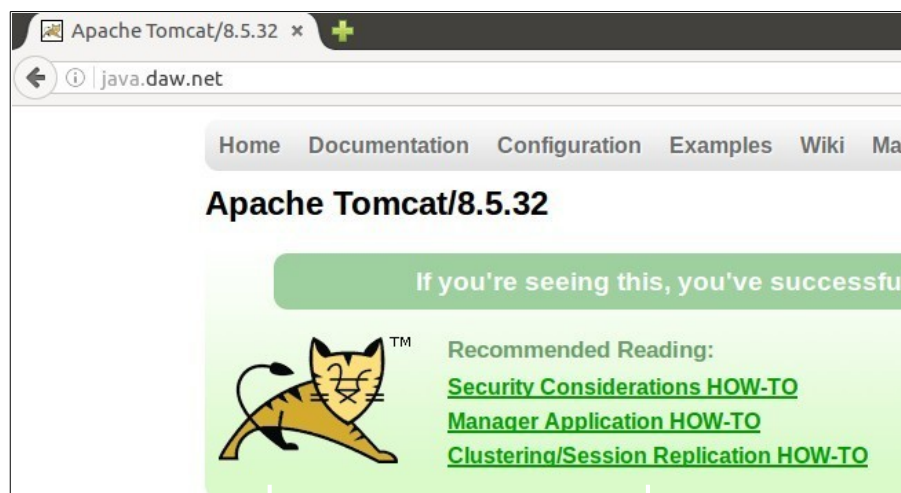
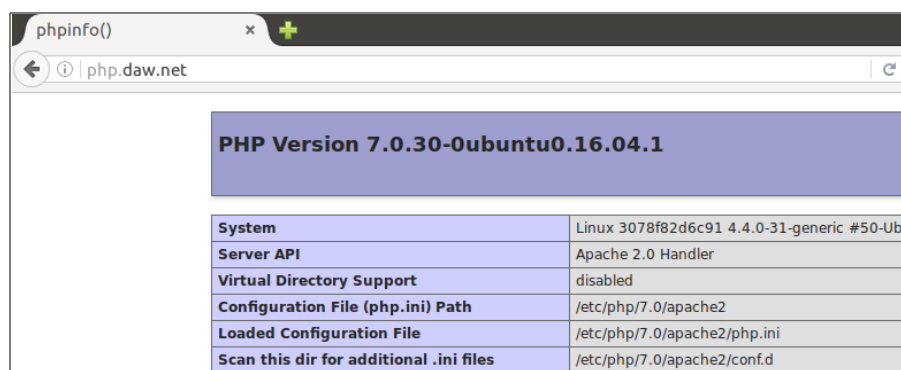
```
osboxes@osboxes:/$ docker run -d -e VIRTUAL_HOST=php.daw.net pauminyana/php
204b3be639b85795b3fbec99bda1a937b97860388e3e280a099cc37f13362877
osboxes@osboxes:/$ docker run -d -e VIRTUAL_HOST=java.daw.net tomcat:8.5.32
1db92f10dea18b8be690e02b0c9b669ab296c903bca8df2342d7c7f629223ba4
osboxes@osboxes:/$ docker run -d -e VIRTUAL_HOST=nodejs.daw.net pauminyana/nodejs
5191258acc3bb177c59cf0a33fb0eec38f48933f6fd95856c961300dfdc24d46
```

🔊 We use the option -e to set the environment variable VIRTUAL_HOST.

Here we are creating new containers for each Docker image. We need it to add VIRTUAL_HOST variable to work with server blocks and manage it and not mapping ports as we have done previously.

You could see that in Tomcat we do not have the configuration created before, so we would need to repeat that if actually want to deploy a java app. Now we can access to the three server blocks from inside the virtual machine.

Notice that the daw.net domains will only work from inside the virtual machine because we configured them in the /etc/hosts file. If you want the domains available from your physical machine you will need to use a DNS server.

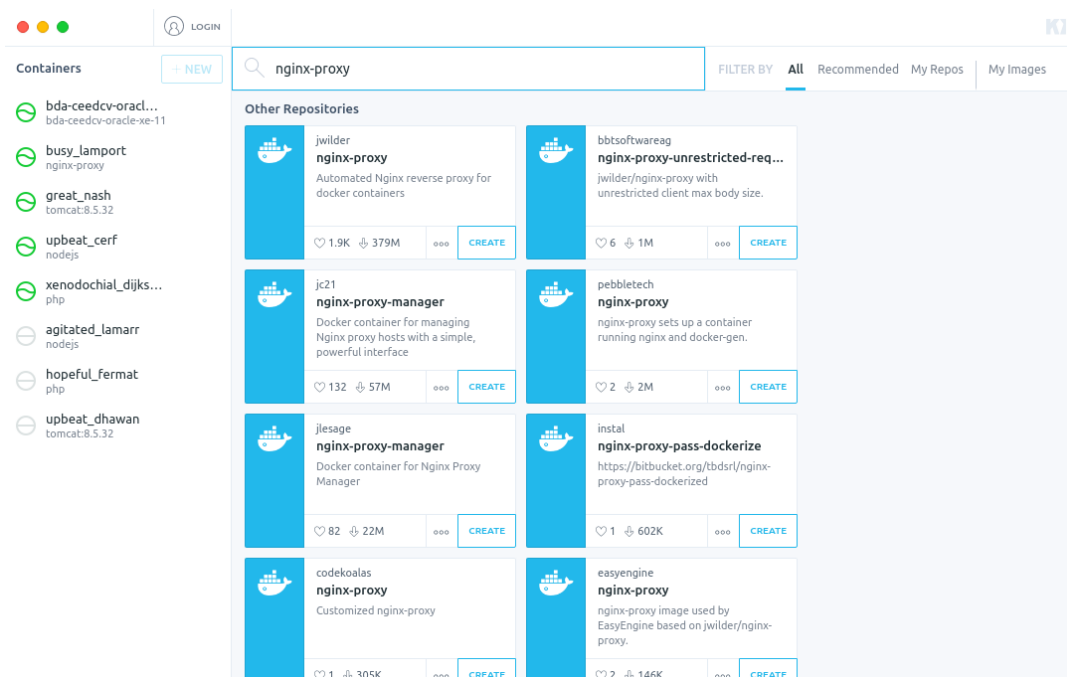


7 EXTRA: KITEMATIC

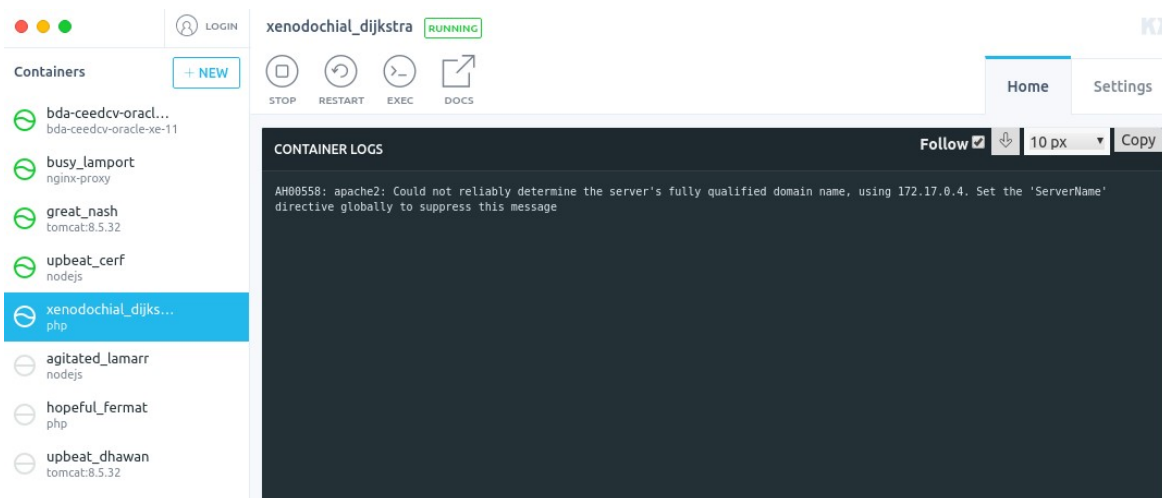
If you don't like terminal management of docker there are some options for search, download, run and manage containers through a simple graphical user interface. An example of that is Kitematic, that works with Mac, Windows and Ubuntu versions of docker, just download and install.

<https://github.com/docker/kitematic/releases>

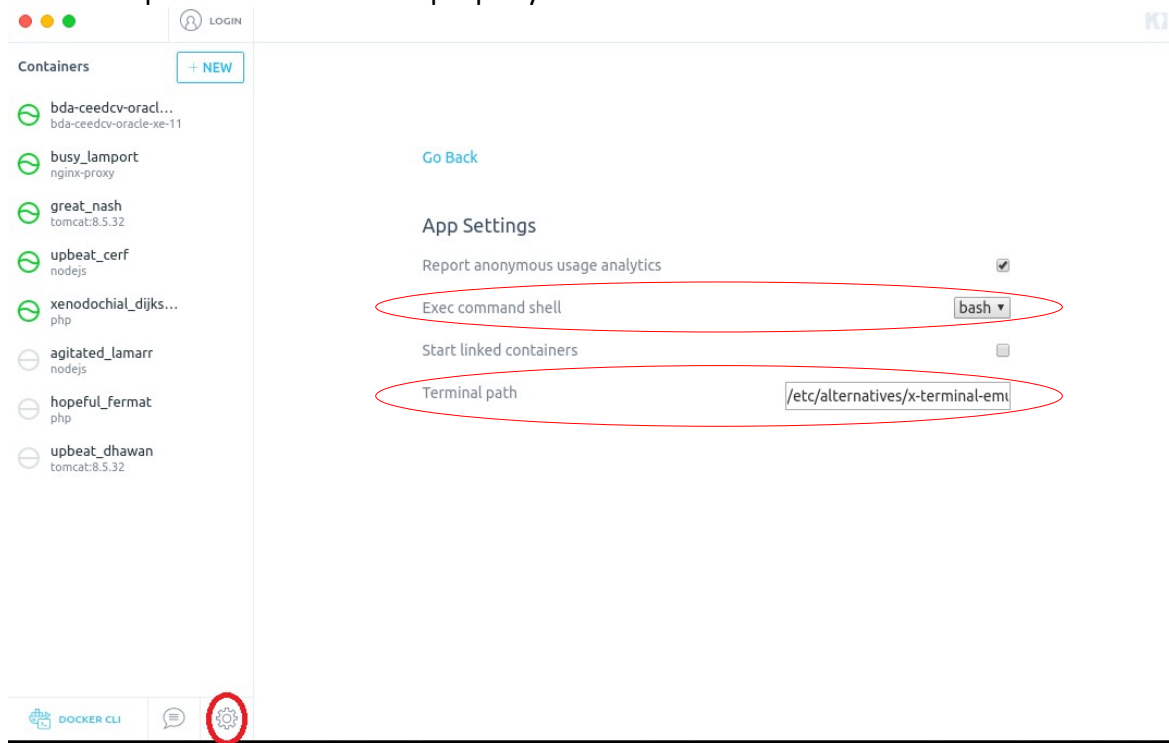
With it, you can view your installed and active containers, search images from dockerhub and manage your containers.



Select your container, start/stop it, **exec** to open internal container terminal.



You could need to go to config and change Exec command shell to bash and select the terminal path for exec to work properly



You also can go to settings to manage environment variables, ports, network or volumes. Better just to view them, those options better fixed on install/run

