Seminar 04. Deployment Technologies

TECHNOLOGIES OF NETWORKED INFORMATION SYSTEMS, 2016-17

ETSINF – DSIC - UPV

The material of this seminar is composed of a descriptive part with examples, and another in a separate document[1], that gathers reference manuals on Docker, tackling both command options and configuration files directives.

Goals of seminar 4

- Apply several concepts introduced in the Unit on "Service Deployment", with a level of sufficient detail to understand their scope
- Introduce some use cases about current technologies
- Illustrate an example that includes all necessary steps to reach a complete deployment
- Cover the requirements for the last laboratory project.

CONTENT

[1] *Additional documentation on Docker*

1 Introduction

Unit “Service Deployment” presented:

- A model of distributed applications integrated by autonomous components
- Multiple aspects to consider in a deployment, standing out:
    - Component specification and configuration
    - Deployment descriptor
- An example of deployment
- The problematic of hosting a deployment, introducing concepts and modalities related to Cloud Computing (CC), as well as a concrete case (Windows Azure)

This seminar tries to strengthen these concepts by means of a deployment case, with sufficient details to apply it in the laboratory. The concrete technologies to be used are:

- Applications (in NodeJS) that communicate by means of messaging (ØMQ)
- Any other additional software being required by those applications
- Systems (LINUX) on which those applications (and their required modules) will be run

These pieces (application + requirements + system) are gathered forming a component.
The technology that we will use has reached a point in which there is a consensus about its usefulness and applicability, but their implementations might be unstable given the speed of current evolution. It is worth noting that the documentation, characteristics and examples can be strongly tied to a given release version of these tools.
Regarding Docker (an implementation of container technology), we have tried that all the material and information collected here were compatible with **version 1.12**, since it is a crucial point in that implementation. It is easy to find old documentation and incompatible examples with this version. If we read any documentation, we must consider that:

- When it mentions *swarm*, it must describe *Docker in swarm mode*, instead of its previous release: *swarm-kit*
- When it mentions *compose*, the *docker-compose.yml* configuration file must use its **version 2**.

Please be careful with these two potential sources of error: (1) using swarm-kit instead of swarm mode, or (2) using an old docker-compose tool unable to process version 2 docker-compose.yml files.

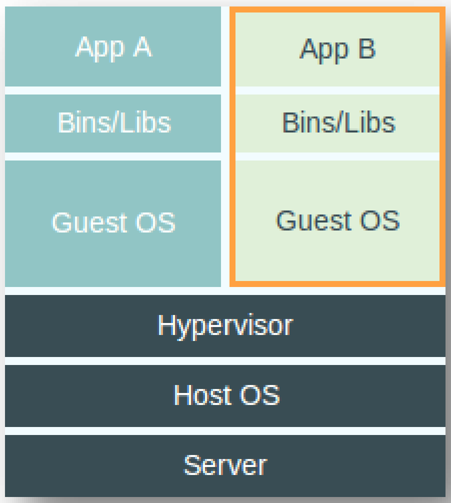
1.1 Provisioning

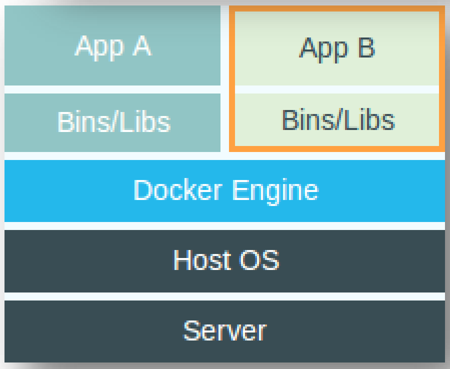
Provisioning consists in booking and providing the infrastructure being needed by a distributed application.

- Reserve specific resources for each component instance (processor + memory + storage)
- Reserve resources for the intercommunication between components.

That needed infrastructure may be provided as *a pool* of interconnected **virtual machines**.

- The component and its requirements are deployed and will be run on a virtual machine





In our case, we choose a light *version*[2] of virtual machine, called **container**

- The guest system coincides with that of the host. Because of this, operating systems do not require any duplication or emulation.
- The host must have a container management software, that provides some services of isolation and replication

Each component is run on a container.

1.2 Configuration of components

For each component there is a specification of its configuration that includes

- The software to execute
- The dependencies to be resolved

Our previous choice about using containers implies that the software of the component…

- …has to be compatible with the host operating system
- …may need some additional software (libraries, tools, etc.)
- …must be correctly configured and initialised

These tasks must be considered at image creation time, when the container configuration file is written.

1.3 Deployment plan

As it was described in Section 3 of the "Service Deployment" Unit, the list of actions to execute in order to carry out a deployment may be specified as an algorithm or plan

- In case of an automation tool, there is an application that interprets (*orchestration*) the specification, carrying out those actions
- In its absence, we must develop a program that implements all the actions specified in the deployment plan

Manual deployment is only valid for simple cases

- It is unmanageable for a middle-sized distributed application.

[2] See second illustration

2 FROM VIRTUAL MACHINES TO CONTAINERS

At present, virtualisation technologies have allowed implementing services with flexibility, but those services should ensure that the effort being needed in a complete installation is profitable. For instance, a minimum virtualised installation may need 500MB…

- It is not practical as a base for a service that provides the current time (*too luggage*)
- But it is profitable as a base for an e-mail service for multiple users. Therefore, it is profitable for "large" services

The current implementations in container technology, like Docker, are sufficiently mature to be a valid alternative to virtual machines

- Advantage: assuming that an image of 1GB used by 100MVs requires 100GBs…
    - If there are 900 immutable MBs, 100 Docker containers will consume 0,9+0,1*100=10,9GBs -> containers reduce **space**
        - Containers consume roughly between 10 and 100 times less resources than their virtual counterparts.
    - If the immutable part is already "loaded", containers save us this time (90%) to initiate <u>each</u> Docker instance -> containers save **time**
- Inconvenient (like any container system)
    - Less flexible than VMs
    - Imperfect isolation between containers can cause interferences and security problems

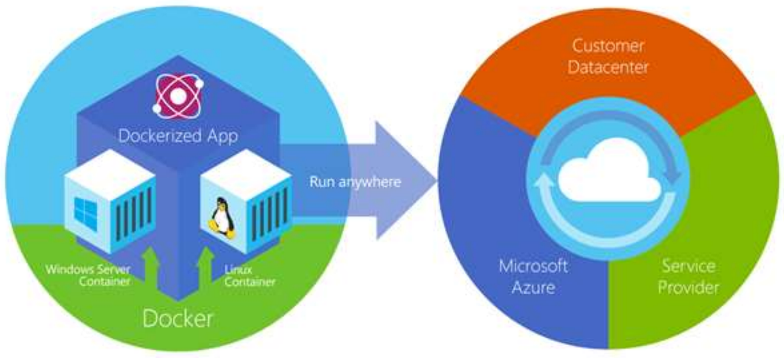
Container technology is so light that it makes possible the "virtualisation" **of an application**, independently of its weight.

3 Introduction to Docker

Docker provides an API to execute processes in an isolated form. With this base, a PaaS system may be built.

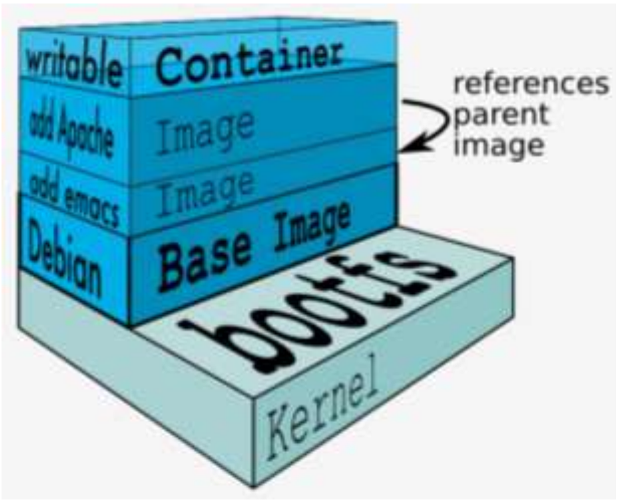
The implementation of Docker was originally based on LXC, but later this dependency has been softened for interacting with other container implementations. In a second stage *libcontainer* was used.

- At present, it is based on *run*.
- With Microsoft, a native implementation for Windows has been built.



With regard to LXC, Docker adds:

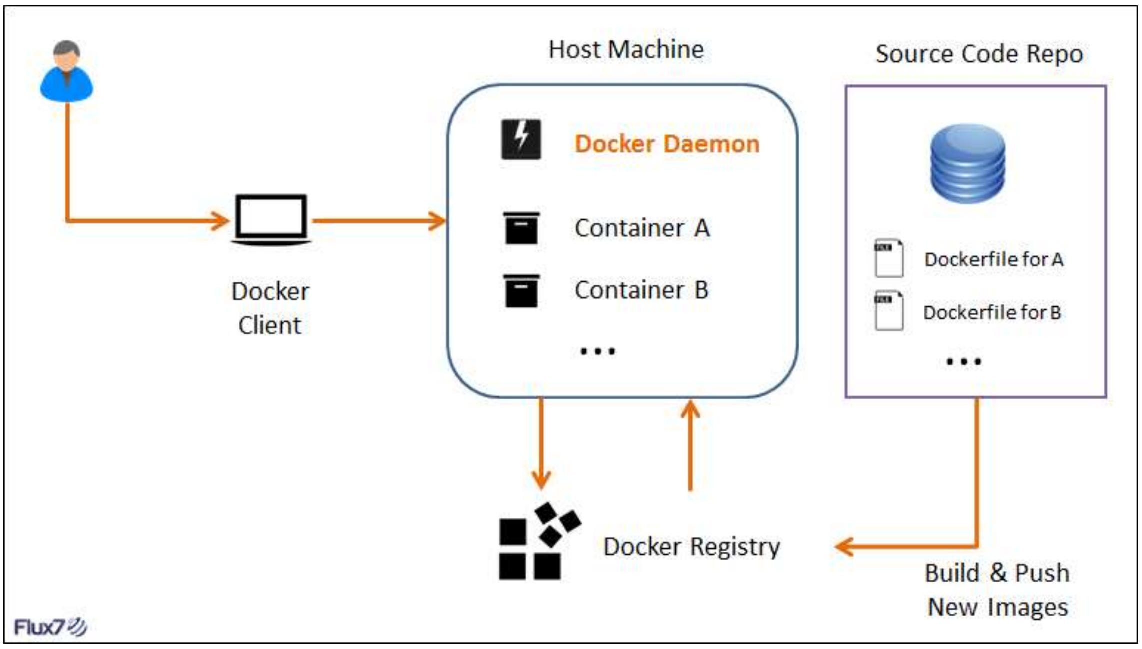
- AuFS: File system with read-only layers that can be shared
  - Updates on an image generate new layers.
- New functionality:
  - Automatic construction
  - Version control (Git)
  - Sharing by means of public repositories



The Docker ecosystem has of 3 components:

1. **Images** (component constructor)
   - The Docker images are basically read-only templates used for container generation.
2. **Repository** (component distributor)
   - There is a common repository where images may be uploaded and shared (hub.docker.com)
3. **Containers** (component executor)
   - They are created from images, and contain all what our applications need to execute
   - Can convert a container in an image using `docker commit`

Internally Docker consists of a daemon, a repository and a client command (docker)



3.1 Operation

```
docker run -i -t image program
```

- Action: run = build and execute (-i -t for interactive)
- Image: e.g., ubuntu
- Program: e.g., /bin/bash

Steps:

1. Download the image (ubuntu) from *the Docker Hub*
2. Create the container.
3. Reserve a filesystem and add a read-write layer.
4. Reserve a network interface or a bridge to communicate with the host.
5. Reserve an internal IP address.
6. Execute the specified program (/bin/bash)
7. Log the application output.

3.2 Docker from the CLI

The crucial element is the client docker command to be used for interacting with the daemon:

```
docker COMMAND [OPTIONS] [arg...] -H, --host=[]
```

Usually root privileges are required for using it

Types of commands:

1. Life cycle control (run, start, stop, …)
2. Informative (logs, ps, …)
3. Access to the repository (pull, push, …)
4. Other (cp, export, …)

Recommendable (in addition to docker run):

5. docker info
6. docker pull *image*

7. docker ps | ps -a
8. docker images

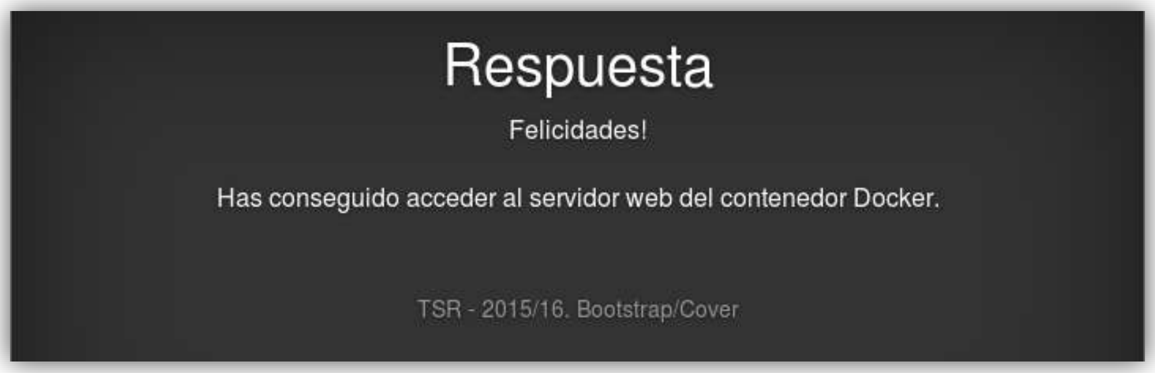
```
docker history httpd
IMAGE        CREATED      CREATED BY                                      SIZE
9a0bc463edaa 3 weeks ago  /bin/sh -c #(nop) CMD ["httpd-foreground"]      0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) EXPOSE 80/tcp                 0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) COPY file:761e313354b918b6c   133 B
<missing>    3 weeks ago  /bin/sh -c set -x && buildDeps=' bzip2 c        29.17 MB
<missing>    3 weeks ago  /bin/sh -c #(nop) ENV HTTPD_ASC_URL=https://    0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) ENV HTTPD_BZ2_URL=https://    0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) ENV HTTPD_SHA1=5101be34ac4    0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) ENV HTTPD_VERSION=2.4.23      0 B
<missing>    3 weeks ago  /bin/sh -c apt-get update && apt-get install    41.15 MB
<missing>    3 weeks ago  /bin/sh -c #(nop) WORKDIR /usr/local/apache2    0 B
<missing>    3 weeks ago  /bin/sh -c mkdir -p "$HTTPD_PREFIX" && chown    0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) ENV PATH=/usr/local/apache    0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) ENV HTTPD_PREFIX=/usr/loca    0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>    3 weeks ago  /bin/sh -c #(nop) ADD file:23aa4f893e3288698c   123 MB
```

3.3 Some examples of use

Practical sections, like this, can be only checked at our virtual computers of `portal-ng`

3.3.1 Minimal web server on Fedora

In this section we try to take contact with a manual deployment, made in a progressive way. The service to be deployed, at the end, will consist in a web server connected to port 80 that provides a welcome page to the browser (URL `http://localhost/`):



Respuesta

Felicidades!

Has conseguido acceder al servidor web del contenedor Docker.

TSR - 2015/16. Bootstrap/Cover

This first case has been designed to be used in the laboratory; it serves us to test our infrastructure: software, configuration, ports, etc. The steps[3] being needed to this aim require that we firstly find out how to build this service. Abbreviating, our list of requirements is:

1. A compatible basic installation for a web server (e.g., Fedora and the APACHE server). It is necessary to foresee how to install that server on a Fedora image.
2. Contents to *populate* the directory with server documents (e.g., the content of `misitio.tar.gz` that may be found in the folder of Seminar 4 in PoliformaT[4]). It

[3] Based in http://linoxide.com/linux-how-to/interactively-create-docker-container/
[4] Also in `tsr/lab3/misitio_sem4_demo` from the asigDSIC folder in the labs

includes folders with pages, styles, etc. It is necessary to know where to place each one of these pieces.

3. Pathname of the generated service. If we put in operation a browser in the virtual machine, what is the URL being needed for accessing the container service? We have to indicate in the host (our virtual machine in the lab) that one of its ports (80) is associated to a container port (also, 80).

Let us proceed stepwise, but without losing the global perspective. Let us first connect to our virtual machine. It has all the necessary Docker software and an Internet connection.
We execute:

```
docker run -i -t fedora bash
```

In this command we wish to execute an image called "fedora[5]". Like Docker does not have locally any image with such identifier, it checks for that image in the central repository (*docker hub*). There, it finds the image[6] and brings it (204.4MBs). When it completes its set-up, it runs the bash shell in the container and waits for commands from the keyboard.
We use inside the container the order `dnf` (since we are in Fedora!) That it is a package installer. First we put it up-to-date (`dnf update`) and afterwards we install the APACHE server (package `httpd`)

```
(Inside the container) dnf update
(Inside the container) dnf install httpd
```

In the last tests, the first action needed to bring 94 packages (59MBs), and the second installed 7 packages (4.7MBs). An enormous advantage of Docker is that it saves local copies that may be later reused, so that a new image would require zero time for installing the same packages.
We do not need anything else in this container, so we go out of it and go back to our virtual machine.

```
(Inside the container) exit
```

In general, all what we do in containers is volatile except when we say the contrary.
Now we need to find out the container identifier. As we have already terminated its execution, we will look for a container based on fedora that has recently finished (some minutes at most)

```
docker ps -a
//We choose the most recent based on fedora
// We copy the first digits of the column CONTAINER_ID
// And we do a commit
docker commit CONTAINER_ID fedora-24-httpd
```

As a result, if we run `docker images,` we obtain two elements: the fedora image downloaded from the docker hub, and ours fedora-24-httpd that we have created locally.

```
docker images
```

This completes our first requirement. Let us now start the second one (directory population). We copy misitio.tar.gz to our virtual machine and decompress it, generating a folder called

[5] It is a Linux distribution derived from RedHat
[6] In fact it is a family in which each fedora member has an associated release number. If we omit it, it will bring the latest release.

misitio. Now we have to copy those files to the container. These operations may be automated using a Dockerfile[7] configuration file. Its content will be:

```
FROM fedora-24-httpd
ADD misitio/index.html /var/www/html/index.html
ADD misitio/css /var/www/html/css
ADD misitio/js /var/www/html/js
ADD misitio/fonts /var/www/html/fonts
EXPOSE 80
ENTRYPOINT [ "/usr/sbin/httpd" ]
CMD [ "-D", "FOREGROUND" ]
```

In this same directory we should do:

```
docker build --rm -t misitio .
```

The point at the end of the command is mandatory! With this command, Docker reads the `Dockerfile` in the current directory and generates a new image called `misitio` deleting the intermediate layers[8] that may be generated.

This finalises the second requirement. Let us start the third one: to run the container providing access to its service.

```
docker run -p 80:80 -d -P misitio
```

This command starts the container, associating port 80 of the virtual machine to port 80 of the container. A browser in the virtual machine that accesses to URL `http://localhost` will see the illustration that appears to the beginning of this subsection.
To finish the container, let us find out its identifier with `docker ps`[9], and stop it with `docker stop` (using its identifier as an argument). Afterwards, let us reload this same web page in the browser. This should fail, since the browser won't be able to connect.

Remarkable conclusions

- The creation of a new image needs time and many resources. If the image is based on another that is held in the local repository, the system reduces the necessary steps and the consumption of resources.
- A complete Dockerfile automates the actions to be applied, reducing the risk of errors, but it may be hard to write. The method of proof and error by means of an interactive session is a good approximation. In case of problems, read its documentation!

3.3.2 Preparing a container to execute NodeJS applications with ZeroMQ

1. Launch Docker using an interactive Fedora image:

```
docker run –i –t fedora bash
```

2. From the shell of that container, let us run the following commands:

```
dnf install –y nodejs
dnf install –y make
dnf install –y zeromq-devel
```

[7] Dockerfile is described in one of the next sections
[8] Each layer is obtained as a result of the execution of an instruction in the Dockerfile.
[9] It does not need any options since the container is still running.

```
dnf install –y python gcc-c++
npm install zmq
```

3. Obtain the identifier or name of the container used in previous steps, using this command in the shell of our (host) system:

```
docker ps
CONTAINER ID   IMAGE    COMMAND   CREATED        ...  NAME
c7b87a37d012   fedora   "bash"    9 minutes ago       serene_hopper
```

4. Request a `commit` of the current contents of the container, generating a new image

```
docker commit serene_hopper zmq
```

Or, alternatively

```
docker commit c7b87 zmq
```

- With this, we have generated a Docker image called "zmq" for running node programs in containers.

5. Check this by means of `docker images`

3.4 Small tricks

1. **More recent identifiers**

```
alias dl='docker ps -l -q'
docker run ubuntu echo hello world
docker commit $(dl) helloworld
```

2. **Find out the IP address of a container**

```
docker inspect $(dl) | grep IPAddress | cut -d '"' -f 4
```

3. **Assign port**

```
docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf
0).HostPort}} {{end}}' <id_container>
```

4. **Locate containers by means of regular expression**

```
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d" "); do echo $i; done
```

5. **Show the environment in a container**

```
docker run --rm ubuntu env
```

6. **Terminate all running containers**

```
docker kill $(docker ps -q)
```

7. **Delete old containers**

```
docker ps -To | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```

8. **Delete stopped or exited containers**

```
docker rm -v $(docker ps -a -q -f status=exited)
```

9. **Delete hung images**

```
docker rmi $(docker images -q -f dangling=true)
```

10. **Delete all the images**

```
docker rmi $(docker images -q)
```

11. **Delete hung volumes**

```
docker volume rm $(docker volume ls -q -f dangling=true)
```

(In 1.9.0, dangling=false does not work and returns all the volumes)

12. **Monitor the consumption of resources when executing containers**

For finding out the CPU, memory and network usage of a single container, we can use:

```
docker stats <container>
```

For all the containers, ordered by ID:

```
docker stats $(docker ps -q)
```

Ditto ordered by name:

```
docker stats $(docker ps --format '{{.Names}}')
```

Ditto selecting those derived from an image:

```
docker ps -a -f ancestor=ubuntu
```

4 The Dockerfile configuration file

Docker may **build an image** from the instructions of a text file called Dockerfile, which should be placed in a folder that contains all the local elements needed for building the image

```
docker build <options> <folder_pathname>
```

General syntax: INSTRUCTION arguments

- By convention, the instructions are written in upper case
- They are executed in the order they are placed in the file

All Dockerfiles have to begin with **FROM**. This specifies the image taken as a base to build the new one.

- Syntax: FROM <image_name>

4.1 Dockerfile commands

You have of a complete reference in the "Additional documentation on Docker"

1. MAINTAINER: Sets the author of the image
    - Syntax: MAINTAINER <appoint author>
2. RUN: Executes a command (shell or exec), adding a new layer to the resultant image. The result is taken as a base for the next instruction
    - Syntax: RUN <command>
3. ADD: Copies files from a place to another
    - Syntax: ADD <origin> <target>
    - The origin can be a URL, a directory (copies all its content, but not the folder itself) or an accessible file in the context of this execution
    - The target is a pathname in the container
4. CMD: Provides default values in the execution of the container
        - It can be used only once (otherwise, the last one is considered)
    - Syntax: 3 alternatives
        - CMD ["executable","arg1","arg2"]
        - CMD ["arg1","arg2"]
        - CMD command arg1 arg2 …
5. EXPOSE: Specifies the port (or ports) in which the container listens to requests
    - Syntax: EXPOSE <port>
6. ENTRYPOINT: Configures a container as an executable
    - It specifies an application that will be run automatically each time a container is generated from this image
        - It implies that this will be the only purpose of this image
    - As in CMD, the last specified ENTRYPOINT is considered
    - Syntax: 2 alternatives
        - ENTRYPOINT ['executable', 'param1','param2']
        - ENTRYPOINT command param1 param2
7. WORKDIR: Sets the working directory for subsequent RUN, CMD and ENTRYPOINT commands.
    - Syntax: WORKDIR /route/to/working_directory

8. ENV: Assigns values to environment variables to be used by programs inside the container.
   - Syntax: ENV <variable> <value>
9. USER: Sets the UID under which the remaining instructions in the Dockerfile will be run.
   - Syntax: USER <uid>
10. VOLUME: Allows the access of the container to a host folder.
    - Syntax: VOLUME [‘/data']

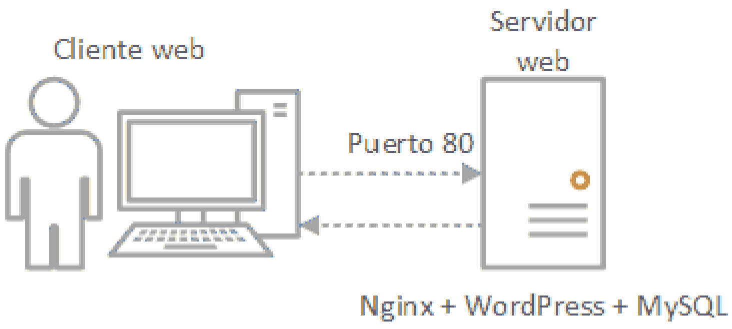
4.2 Dockerfile examples

You can find other examples in the additional documentation and in a lot of places in the web, as https://github.com/komljen/dockerfile-examples

4.2.1 Example 1 (WordPress on Ubuntu)

This first example is a web service based on WordPress installed on Ubuntu (https://github.com/eugeneware/docker-wordpress-nginx/blob/master/dockerfile). It requires:

- A web server (Nginx)
- Support for the PHP language
- A DB server (MySQL typically)



Designed for an only component

```
FROM ubuntu:14.04
MAINTAINER Eugene Ware <eugene@noblesamurai.com>
# Keep upstart from complaining
RUN dpkg-divert --local --rename --add /sbin/initctl
RUN ln -sf /bin/true /sbin/initctl
# Let the container know that there is no tty
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update
RUN apt-get -y upgrade
# Basic Requirements
RUN apt-get -y install mysql-server mysql-client nginx php5-fpm php5-mysql php-apc pwgen python-setuptools
curl git unzip
# WordPress Requirements
RUN apt-get -y install php5-curl php5-gd php5-intl php-pear php5-imagick php5-imap php5-mcrypt php5-
memcache php5-ming php5-ps php5-pspell php5-recode php5-sqlite php5-tidy php5-xmlrpc php5-xsl
RUN sed -i -e"s/^bind-address\s*=\s*127.0.0.1/bind-address = 0.0.0.0/" /etc/mysql/my.cnf
# nginx config
RUN sed -i -e "s/keepalive_timeout\s*65/keepalive_timeout 2/" /etc/nginx/nginx.conf
RUN sed -i -e "s/keepalive_timeout 2/keepalive_timeout 2;\n\tclient_max_body_size 100m/"
/etc/nginx/nginx.conf
RUN echo "daemon off;" >> /etc/nginx/nginx.conf
# php-fpm config
RUN sed -i -e "s/;cgi.fix_pathinfo=1/cgi.fix_pathinfo=0/g" /etc/php5/fpm/php.ini
RUN sed -i -e "s/upload_max_filesize\s*=\s*2M/upload_max_filesize = 100M/g" /etc/php5/fpm/php.ini
RUN sed -i -e "s/post_max_size\s*=\s*8M/post_max_size = 100M/g" /etc/php5/fpm/php.ini
RUN sed -i -e "s/;daemonize\s*=\s*yes/daemonize = no/g" /etc/php5/fpm/php-fpm.conf
RUN sed -i -e "s/;catch_workers_output\s*=\s*yes/catch_workers_output = yes/g"
/etc/php5/fpm/pool.d/www.conf
RUN find /etc/php5/cli/conf.d/ -name "*.ini" -exec sed -i -re 's/^(\s*)#(.*)/\1;\2/g' {} \;
# nginx site conf
ADD ./nginx-site.conf /etc/nginx/sites-available/default
# Supervisor Config
RUN /usr/bin/easy_install supervisor
RUN /usr/bin/easy_install supervisor-stdout
ADD ./supervisord.conf /etc/supervisord.conf
# Install Wordpress
ADD http://wordpress.org/latest.tar.gz /usr/share/nginx/
RUN mv /usr/share/nginx/html/5* /usr/share/nginx/wordpress
RUN rm -rf /usr/share/nginx/www
RUN mv /usr/share/nginx/wordpress /usr/share/nginx/www
RUN chown -R www-date:www-date /usr/share/nginx/www
# Wordpress Initialization and Startup Script
ADD ./start.sh /start.sh
RUN chmod 755 /start.sh
# private expose
EXPOSE 3306
EXPOSE 80
CMD ["/bin/bash", "/start.sh"]
```

4.2.2 Example 2 (NodeJS and ZMQ)

The actions of the previous example 3 (section 3.3.2) could be held in this Dockerfile:

```
#Take the latest Fedora distribution ace to base.
# Currently (November 22, 2016), it is Fedora 24.
FROM fedora
# Install the latest NodeJS on that distribution.  It is NodeJS v4.6.1.
# We may Use the dnf Fedora command to this end.
RUN dnf install -y nodejs
# Next step: Install the zeromq library. Its package is called "zeromq-devel" in the Fedora
# distribution.
RUN dnf install -y zeromq-devel
RUN dnf install -y python gcc-c++
# We still need the standard "make" command in order to run "npm".
RUN dnf install -y make
# Finally, let us run the npm command for installing the "zmq" module.
RUN npm install zmq
```

The image “zmq” generated Section 3.3.2 could now be built using...

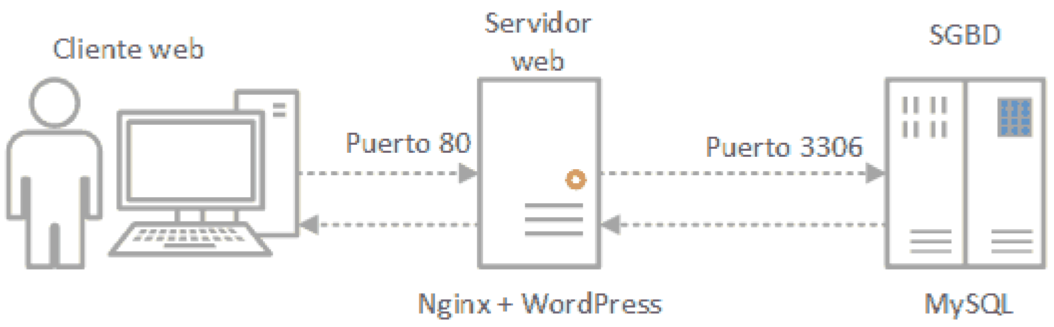
```
docker build –t zmq .
```

- The option “-t” allows to assign a label or name to the image being created.
- This command should be run in the folder where this Dockerfile is placed.

5 MULTIPLE COMPONENTS

The previous example about Wordpress mixes in a container several components, losing opportunities for scaling and improving availability

- A minimum improvement consists in differentiating the DBMS (MySQL), placing it in another container



This new deployment is more complex. There are some dependences that should be resolved:

- How does the first component (Nginx+WordPress) know the necessary details to connect to the second (MySQL)?
    - At least its IP is needed.
- Perhaps those dependences couldn't be resolved until that second component would have started.

Succinctly, the following steps are required…

1. Create the Dockerfile of both components
    - They are a subset of that showed previously
2. Initiate the second component, obtaining his IP
3. Initiate the first component, transmitting to it the IP of the second

However, this handmade approximation is inapplicable for medium or large deployments. In those other cases, we need:

- A language to generalise the description of deployments
    - Able to express components, properties and relations
    - e.g., Compose-YAML, OASIS-TOSCA
- Automate the execution of deployments
    - By means of an engine that executes the deployment according to that description
    - e.g., Docker-Compose, APACHE-Brooklyn

It is convenient to have some tools that facilitate the creation and simulation of these deployments, as https://lorry.io/

- Faults and errors are not welcome while a large deployment is being done!!

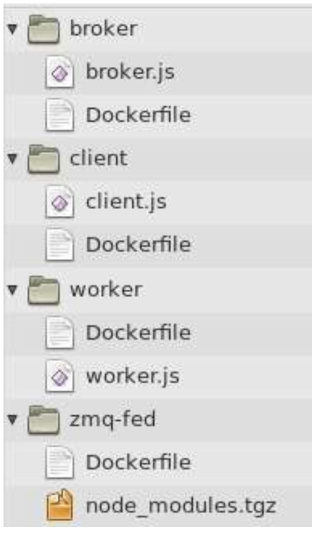
Docker admits "links" between containers. They may be created in an automated way using "docker-compose".

5.1 Example: client/ broker/worker with ZMQ

Example: Service implemented by means of a "client" component, a "broker" (type ROUTER-DEALER) and a "worker", which may be replicated as many times as necessary. Both clients and workers need as an argument the URL of a broker endpoint.

5.1.1 client.js

It requires the URL of the broker as an argument



```
// client.js
// Client process. It sends a random integer value to
// the servers, and it prints the returned result.

// Import the zmq Modulate.
var zmq = require('zmq');
// Count the number of replies.
var numReplies = 0;
// Total number of requests.
const reqs = 10;

// Check whether we have received the URL as an argument.
if (process.argv.length != 3) {
  console.error("The broker URL should be given as an argument!!");
  process.exit(1);
}
// Get the broker URL.
var url = process.argv[2];
console.log("Broker URL is %s.", url);
var req = zmq.socket('req');
// Connect to the broker.
req.connect(url);

// Handle the received replies.
req.on('message', function(msg) {
        // Get the reply value.
        var value = parseInt(msg);
        // Print the value.
        console.log("Returned value: %d", value);
        // Check whether we should terminate.
        if (++numReplies == reqs)
                process.exit(3);
});

// Send a sequence of 10 requests
for (var i = 0; i < reqs; i++) {
        // Generate a random integer value in the range 1..100
        var value = Math.floor(Math.random()*100 + 1);
        // Print the request.
        console.log("Request: %d", value);
        // Send it.
        req.send(value.toString());
}
```

5.1.2 broker.js

```
// broker.js
// Basic ROUTER-DEALER broker.

var zmq = require('zmq');
var router = zmq.socket('router');
var dealer = zmq.socket('dealer');

// Bind these sockets to their ports.
router.bindSync("tcp://*:8000");
dealer.bindSync("tcp://*:8001");
```

```
// The router simply forwards messages
// in both directions...
router.on('message', function(){
  var args = Array.apply(null, arguments);
  dealer.send(args);
});

dealer.on('message', function() {
  var args = Array.apply(null, arguments);
  router.send(args);
});
```

5.1.3 worker.js

It requires the URL of the broker as an argument

```
// worker.js
// It assumes that requests only carry an integer value.
// It returns the double of the incoming value.

// Import the zmq Modulate.
var zmq = require('zmq');

// Check whether we have received the URL as an argument.
if (process.argv.length != 3) {
  console.error("The broker URL should be given as an argument!!");
  process.exit(1);
}
// Get the broker URL.
var url = process.argv[2];
console.log("Worker: The broker URL is %s", url);

// Create the REP socket.
var rep = zmq.socket('rep');
// Connect to the broker.
rep.connect(url);

// Handle the received requests.
rep.on('message', function(msg) {
  // Get the incoming value.
  var value = parseInt(msg);
  // Double it!
  value *= 2;
  // Print the reply.
  console.log("Reply: %d", value);
  // Send the reply.
  rep.send(value.toString());
});
```

We use as a base a Docker image called “zmq-fed” with...

- NodeJS Installed on an image of some Linux distribution.
- The zeromq library properly installed.
- The zmq module installed and available for its use from node.
- On this image will be able to run any NodeJS program that uses ØMQ.

The Dockerfile being needed[10] is:

```
# Take the latest Fedora distribution ace to base.
# Currently (November 9, 2016), it is Fedora 24.
FROM fedora
# Install the latest Node.js on that distribution.
# This also installs npm.
RUN curl -sL https://rpm.nodesource.com/setup_6.x | bash -
RUN dnf install -y nodejs
# Next step: Install the zeromq library.
# Its package is called "zeromq-devel" in the Fedora
# distribution.
RUN dnf install -y zeromq-devel
# The following lines install the "node_modules" directory
# generated with a "npm install zmq" command on a system with
# the required tools. Those tools are not installed in the
# image being built here.
ADD ./node_modules.tgz /
```

Steps to be followed to use this service...

1. Create the Docker image for the <u>**broker**</u> component.

We need a Dockerfile similar to this:

```
FROM zmq-fed
RUN mkdir /zmq
COPY ./broker.js /zmq/broker.js
WORKDIR /zmq
EXPOSE 8000 8001
CMD node broker.js
```

- It exposes port 8000 for clients...
- ...and port 8001 for workers.

And we generate its container with this command:

```
docker build –t broker .
```

- We start the broker (**docker run**) and find out its IP by means of **docker inspect broker**

- We write down the IP that appears (e.g., *a.b.c.d*) and use it to write the Dockerfile of the other 2 components

2. Create the Docker image for the <u>**client**</u> component.

We need a Dockerfile similar to this:

```
FROM zmq-fed
RUN mkdir /zmq
COPY ./client.js /zmq/client.js
WORKDIR /zmq
# We assume that each worker is linked to the broker
```

[10] This Dockerfile and the remaining ones in this section, as well as the node_modules.tgz file mentioned In the Dockerfile must be downloaded from PoliformaT. They are in a ZIP called dcExample.zip in the resources folder of Seminar 4.
The “zmq-fed” image must be generated using “docker build –t zmq-fed .” in the folder where this Dockerfile is placed.

```
# container.
CMD node client tcp://a.b.c.d:8000
```

- And we generate its image from the client folder with this Docker command:

```
docker build –t client .
```

3. Create the Docker image for the **worker** component.

We need a Dockerfile similar to this:

```
FROM zmq-fed
RUN mkdir /zmq
COPY ./worker.js /zmq/worker.js
WORKDIR /zmq
# We assume that each worker is linked to the broker
# container.
CMD node worker tcp://a.b.c.d:8001
```

- And we generate its image with this command in the worker folder:

```
docker build –t worker .
```

Now, we can run these processes checking that they connect with the broker and that send and receive messages.

Note that these Dockerfiles have a static dependency resolution. This means that each time the broker is restarted and its IP changes, those Dockerfiles might need to be changed. Alternatively, you can explicitly specify the command to be run when you are using the **docker run** command.

Thus, if we start the broker component and it gets as its IP the value 172.17.0.2, we may start clients and workers in this other way (writing each command in a different terminal window):

```
docker run –it client node client tcp://172.17.0.2:8000
docker run –it worker node worker tcp://172.17.0.2:8001
```

With this, we don't need to rebuild any of these images. Indeed, those images may contain any default broker URL that may be overwritten by the command being specified when **docker run** is used.

6 Deployment in Docker Compose

URL https://docs.docker.com/compose/

Docker Compose is a tool to define and execute applications in several Docker containers.

Three steps:

1. Define each one of the application components using **a Dockerfile.**
2. Set the services that compose the application in a **docker-compose.yml** file. Thus, they run jointly.
3. Execute **docker-compose up** in the folder where the docker-compose.yml file is placed. With this, Compose initiates and executes the complete application.

It is limited to containers in the same computer, but it may be complemented with Docker Swarm (or any other orchestration software) in order to control a cluster.

It is an application that should be installed apart from Docker.

6.1 Remarkable characteristics

- We may use several isolated containers in a given computer.
- We may place the data out of the containers (using **volumes**).
- It is only necessary to rebuild those containers whose images have been modified.

The crucial element is the docker-compose command:

```
docker-compose [COMMAND] [OPTIONS] [ARGS...]
```

Typical cycle of use:

```
$ docker-compose up -d
$ Activities …
$ docker-compose stop
$ docker-compose rm –f
```

6.2 Docker Compose From the CLI

Important commands:

- **build**: (re-)Builds a service
- **kill**: Kills a container
- **logs**: Shows the output of containers
- **port**: Shows the port associated to a service
- **ps**: Lists the containers
- **pull**: Downloads an image
- **rm**: Deletes a stopped container
- **run**: Runs a command in a service
- **scale**: Number of containers to execute for a service
- **start** | **stop** | **restart**: Actions to apply (start, stop or restart) on a service
- **up**: build + start (*Roughly*)

Example of **run** command

```
$ docker-compose run web python manage.py shell
```

1. It initiates the web service.
2. It sends to the service the command **python manage.py shell** in order to run it.

3. The execution of this command in the service is detached from our shell (from where we started docker-compose)

There are two differences between **run** and **start**

- The command specified in **run** has preference on that specified in the image.
- The **run** command doesn't open any of the ports specified in the image. Thus, it avoids collisions with other instances previously initiated using **start**.

6.3 The docker-compose.yml file

https://docs.docker.com/compose/compose-file/

It is a file that follows the YAML syntax.

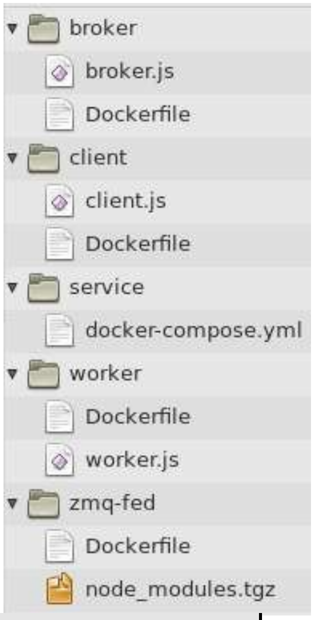
Besides the commands that are analogous to the parameters of "**docker run**", the principal commands to be applied in this configuration file are:

- **image**: Local or remote reference to an image, by name or tag
- **build**: Pathname of a directory that contains a Dockerfile
- **command**: It changes the command to be run in a container
- **links**: Links to containers of another service.
- **external_links**: Links to containers external to compose
- **ports**: Exposed ports (to be written between quotation marks)
- **expose**: Ditto, but accessible only to linked services (with links)
- **volumes**: It mounts host pathnames as container volumes

6.4 Revisiting a previous example

The example with `client`, `broker` and `worker` deployed previously (section 5.1), has manually resolved the dependency of the other components onto the `broker`: they needed its IP. This form of dependency resolution does not allow any automation. It also sets an undesirable constraint: we should start the broker before being able to configure the other components!

- If we changed the broker IP it would be necessary to reconfigure and re-deploy `client` and `worker`!?



```
▼ broker
    broker.js
    Dockerfile
▼ client
    client.js
    Dockerfile
▼ service
    docker-compose.yml
▼ worker
    Dockerfile
    worker.js
▼ zmq-fed
    Dockerfile
    node_modules.tgz
```

There must be a better solution, and that is one of the contributions of `docker-compose` and its deployment descriptor (docker-compose.yml file). We will create a folder `service` for it.

Let us assume that there is a way for publishing the broker IP, and that there is also a way to inject this information in the suitable components. This is one of the goals of the `docker-compose.yml` file:

```
version: '2'
services:
  wor:
    image: worker
    build: ../worker/
    links:
     - bro
    environment:
     - BROKER_URL=tcp://bro:8001
  cli:
    image: client
    build: ../client/
    links:
     - bro
    environment:
     - BROKER_URL=tcp://bro:8000
  bro:
    image: broker
    build: ../broker/
```

Also we need that the `Dockerfile` of `client` and `worker` can consult this information. You may note that we have used the complete broker URL to refer to the broker in the Dockerfiles of the remaining components. Thus, we must update those Dockerfiles in the following way:

In the client `Dockerfile`:

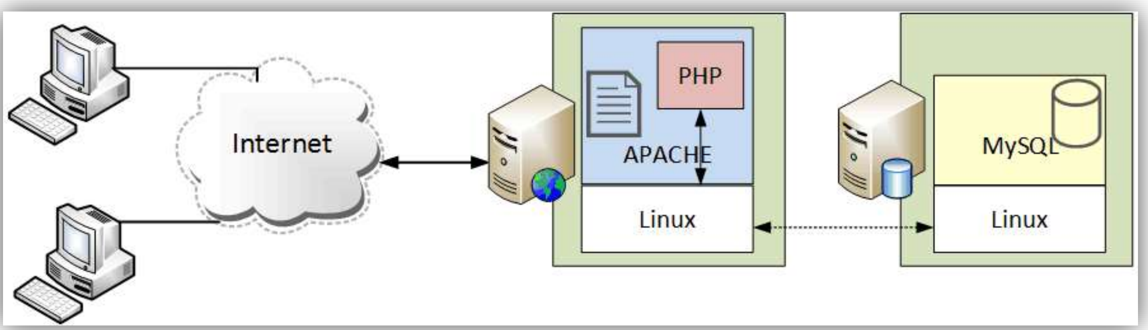
```
CMD node client $BROKER_URL
```

In the worker `Dockerfile`:

```
CMD node worker $BROKER_URL
```

6.5 Example: WordPress on Fedora

This second example shows how to deploy in an easy way a complex system. In Unit 1 we already presented Wikipedia as a system LAMP. Our next example, based on **WordPress**, belongs to that same technology (LAMP).



As in previous cases, we assume a system like that used in the labs; i.e., composed by a set of virtual machines. To test it there, we have to create a **wordpress** folder and start in it the following activities[11].
URL: https://docs.docker.com/compose/wordpress/

1. (*) We download the latest version of WordPress:

```
curl https://wordpress.org/latest.tar.gz | tar -xvzf -
```

2. (*) We create a Dockerfile with these 2 lines:

```
FROM orchardup/php5
ADD . /code
```

[11] Downloading `tsr_lab3_material.zip` from PoliformaT we may avoid the actions marked with (*)

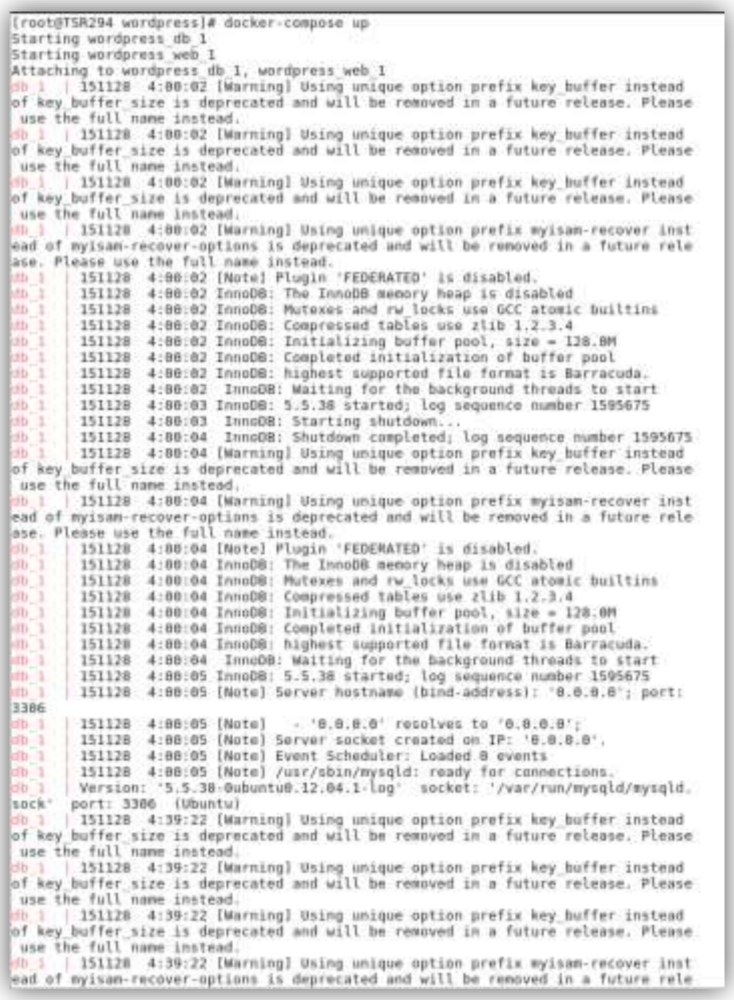
The base image (orchardup/php5) assumes an Ubuntu distribution that **will need a long time** to be downloaded. Please, do something else in the meantime.

3. (*) We write the following **docker-compose.yml** file:

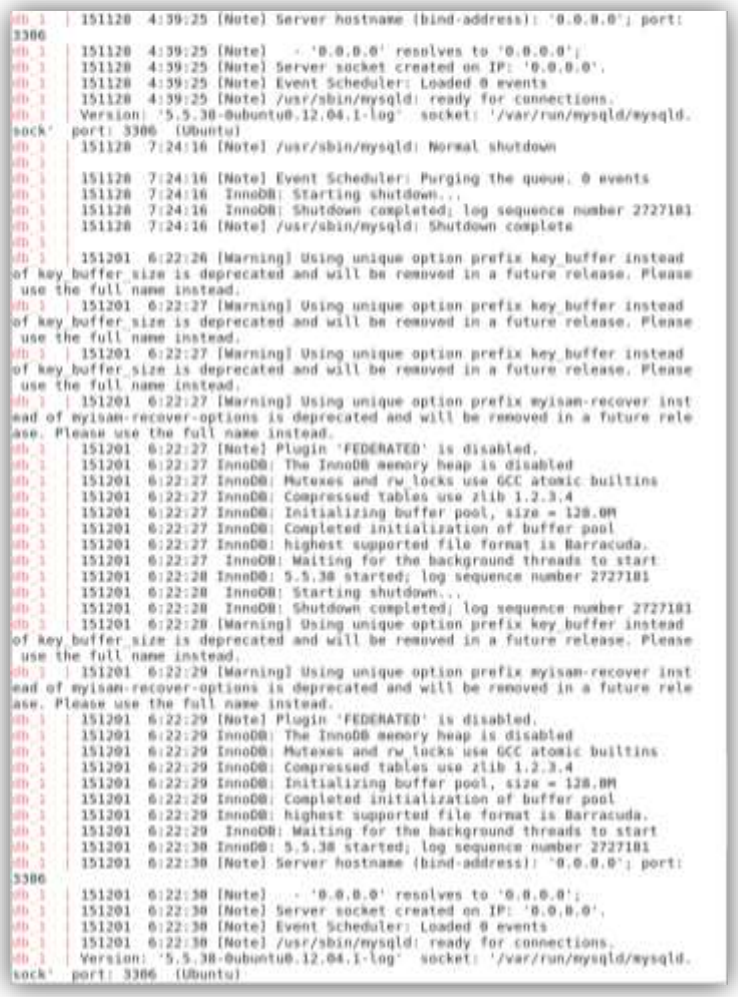
```
Web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  ports:
    - "8000:8000"
  Links:
    - db
  volumes:
    - .:/code
db:
  image: orchardup/mysql
  environment:
    MYSQL_DATABASE: wordpress
```

The annex on **YAML and docker-compose.yml** contains information on its syntax and meaning

4. (*) There are several WordPress configuration files (`wp-config`.php, `router`.php) that might be modified, but we may use those included in **tsr_lab3_material.zip**.
5. Let us start this service with `docker-compose up`
    - Please, remember to stop the service at the end!



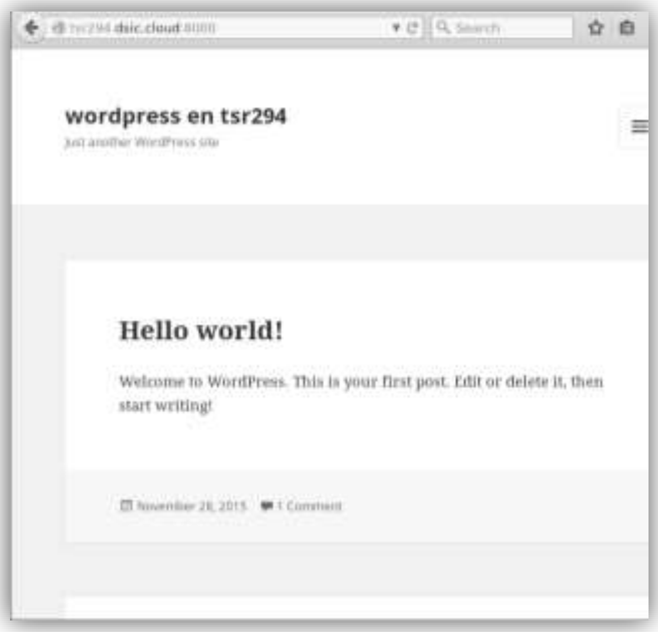
```
[root@TSR294 wordpress]# docker-compose up
Starting wordpress_db_1
Starting wordpress_web_1
Attaching to wordpress_db_1, wordpress_web_1
db_1  | 151128  4:00:02 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151128  4:00:02 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151128  4:00:02 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151128  4:00:02 [Warning] Using unique option prefix myisam-recover inst
ead of myisam-recover-options is deprecated and will be removed in a future rele
ase. Please use the full name instead.
db_1  | 151128  4:00:02 [Note] Plugin 'FEDERATED' is disabled.
db_1  | 151128  4:00:02 InnoDB: The InnoDB memory heap is disabled
db_1  | 151128  4:00:02 InnoDB: Mutexes and rw_locks use GCC atomic builtins
db_1  | 151128  4:00:02 InnoDB: Compressed tables use zlib 1.2.3.4
db_1  | 151128  4:00:02 InnoDB: Initializing buffer pool, size = 128.0M
db_1  | 151128  4:00:02 InnoDB: Completed initialization of buffer pool
db_1  | 151128  4:00:02 InnoDB: highest supported file format is Barracuda.
db_1  | 151128  4:00:02  InnoDB: Waiting for the background threads to start
db_1  | 151128  4:00:03 InnoDB: 5.5.38 started; log sequence number 1595675
db_1  | 151128  4:00:03  InnoDB: Starting shutdown...
db_1  | 151128  4:00:04  InnoDB: Shutdown completed; log sequence number 1595675
db_1  | 151128  4:00:04 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151128  4:00:04 [Warning] Using unique option prefix myisam-recover inst
ead of myisam-recover-options is deprecated and will be removed in a future rele
ase. Please use the full name instead.
db_1  | 151128  4:00:04 [Note] Plugin 'FEDERATED' is disabled.
db_1  | 151128  4:00:04 InnoDB: The InnoDB memory heap is disabled
db_1  | 151128  4:00:04 InnoDB: Mutexes and rw_locks use GCC atomic builtins
db_1  | 151128  4:00:04 InnoDB: Compressed tables use zlib 1.2.3.4
db_1  | 151128  4:00:04 InnoDB: Initializing buffer pool, size = 128.0M
db_1  | 151128  4:00:04 InnoDB: Completed initialization of buffer pool
db_1  | 151128  4:00:04 InnoDB: highest supported file format is Barracuda.
db_1  | 151128  4:00:04  InnoDB: Waiting for the background threads to start
db_1  | 151128  4:00:05 InnoDB: 5.5.38 started; log sequence number 1595675
db_1  | 151128  4:00:05 [Note] Server hostname (bind-address): '0.0.0.0'; port:
3306
db_1  | 151128  4:00:05 [Note]   - '0.0.0.0' resolves to '0.0.0.0';
db_1  | 151128  4:00:05 [Note] Server socket created on IP: '0.0.0.0'.
db_1  | 151128  4:00:05 [Note] Event Scheduler: Loaded 0 events
db_1  | 151128  4:00:05 [Note] /usr/sbin/mysqld: ready for connections.
db_1  | Version: '5.5.38-0ubuntu0.12.04.1-log'  socket: '/var/run/mysqld/mysqld.
sock'  port: 3306  (Ubuntu)
db_1  | 151128  4:39:22 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151128  4:39:22 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151128  4:39:22 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151128  4:39:22 [Warning] Using unique option prefix myisam-recover inst
ead of myisam-recover-options is deprecated and will be removed in a future rele
```



```
db_1  | 151128  4:39:25 [Note] Server hostname (bind-address): '0.0.0.0'; port:
3306
db_1  | 151128  4:39:25 [Note]   - '0.0.0.0' resolves to '0.0.0.0';
db_1  | 151128  4:39:25 [Note] Server socket created on IP: '0.0.0.0'.
db_1  | 151128  4:39:25 [Note] Event Scheduler: Loaded 0 events
db_1  | 151128  4:39:25 [Note] /usr/sbin/mysqld: ready for connections.
db_1  | Version: '5.5.38-0ubuntu0.12.04.1-log'  socket: '/var/run/mysqld/mysqld.
sock'  port: 3306  (Ubuntu)
db_1  | 151128  7:24:16 [Note] /usr/sbin/mysqld: Normal shutdown
db_1  |
db_1  | 151128  7:24:16 [Note] Event Scheduler: Purging the queue. 0 events
db_1  | 151128  7:24:16  InnoDB: Starting shutdown...
db_1  | 151128  7:24:16  InnoDB: Shutdown completed; log sequence number 2727101
db_1  | 151128  7:24:16 [Note] /usr/sbin/mysqld: Shutdown complete
db_1  |
db_1  | 151201  6:22:26 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151201  6:22:27 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151201  6:22:27 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151201  6:22:27 [Warning] Using unique option prefix myisam-recover inst
ead of myisam-recover-options is deprecated and will be removed in a future rele
ase. Please use the full name instead.
db_1  | 151201  6:22:27 [Note] Plugin 'FEDERATED' is disabled.
db_1  | 151201  6:22:27 InnoDB: The InnoDB memory heap is disabled
db_1  | 151201  6:22:27 InnoDB: Mutexes and rw_locks use GCC atomic builtins
db_1  | 151201  6:22:27 InnoDB: Compressed tables use zlib 1.2.3.4
db_1  | 151201  6:22:27 InnoDB: Initializing buffer pool, size = 128.0M
db_1  | 151201  6:22:27 InnoDB: Completed initialization of buffer pool
db_1  | 151201  6:22:27 InnoDB: highest supported file format is Barracuda.
db_1  | 151201  6:22:27  InnoDB: Waiting for the background threads to start
db_1  | 151201  6:22:28 InnoDB: 5.5.38 started; log sequence number 2727101
db_1  | 151201  6:22:28  InnoDB: Starting shutdown...
db_1  | 151201  6:22:28  InnoDB: Shutdown completed; log sequence number 2727101
db_1  | 151201  6:22:28 [Warning] Using unique option prefix key_buffer instead
of key_buffer_size is deprecated and will be removed in a future release. Please
 use the full name instead.
db_1  | 151201  6:22:29 [Warning] Using unique option prefix myisam-recover inst
ead of myisam-recover-options is deprecated and will be removed in a future rele
ase. Please use the full name instead.
db_1  | 151201  6:22:29 [Note] Plugin 'FEDERATED' is disabled.
db_1  | 151201  6:22:29 InnoDB: The InnoDB memory heap is disabled
db_1  | 151201  6:22:29 InnoDB: Mutexes and rw_locks use GCC atomic builtins
db_1  | 151201  6:22:29 InnoDB: Compressed tables use zlib 1.2.3.4
db_1  | 151201  6:22:29 InnoDB: Initializing buffer pool, size = 128.0M
db_1  | 151201  6:22:29 InnoDB: Completed initialization of buffer pool
db_1  | 151201  6:22:29 InnoDB: highest supported file format is Barracuda.
db_1  | 151201  6:22:29  InnoDB: Waiting for the background threads to start
db_1  | 151201  6:22:30 InnoDB: 5.5.38 started; log sequence number 2727101
db_1  | 151201  6:22:30 [Note] Server hostname (bind-address): '0.0.0.0'; port:
3306
db_1  | 151201  6:22:30 [Note]   - '0.0.0.0' resolves to '0.0.0.0';
db_1  | 151201  6:22:30 [Note] Server socket created on IP: '0.0.0.0'.
db_1  | 151201  6:22:30 [Note] Event Scheduler: Loaded 0 events
db_1  | 151201  6:22:30 [Note] /usr/sbin/mysqld: ready for connections.
db_1  | Version: '5.5.38-0ubuntu0.12.04.1-log'  socket: '/var/run/mysqld/mysqld.
sock'  port: 3306  (Ubuntu)
```

You HAVE TO access it using a browser. Take care of the details, like the port to be used.

- The installation of Wordpress includes a last stage that is made using the browser. That is the page shown at first.



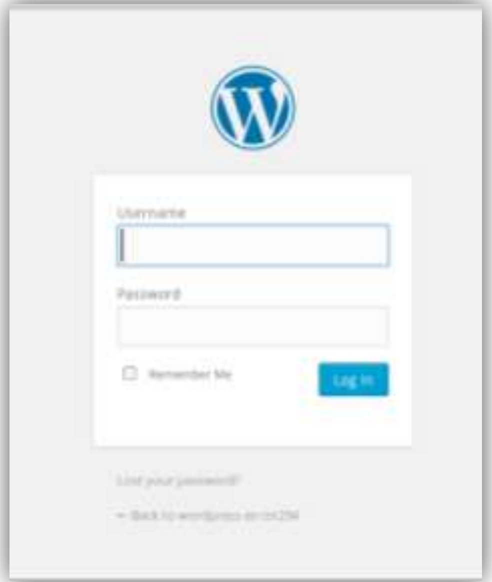
wordpress en tsr294

Just another WordPress site

Hello world!

Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

November 28, 2015   1 Comment



- The configuration creates an administrative account (usually called **admin**) to which you can access writing `/admin` at the end of the URL.

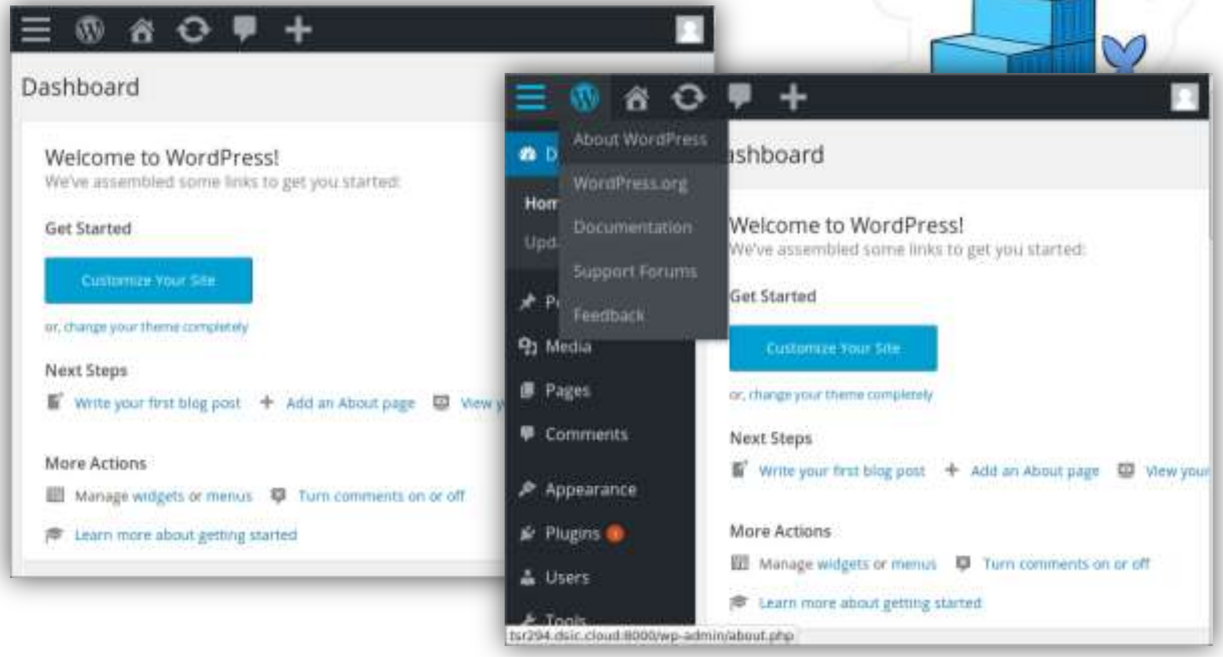
7 MULTIPLE NODES

Docker Compose is limited to deployments in a single node. However, service scalability cannot be limited in this way: it demands the aggregation of multiple nodes in order to provide a large set of resources to the applications.

- Fault tolerance, with a single node, cannot be achieved.
- The concept of a distributed application limited to one node is contradictory.

What is needed by a set of Docker nodes in order to be integrated and behave as a single node? A coordinator (or "orchestrator"; i.e. an orchestral conductor that coordinates a large set of players).

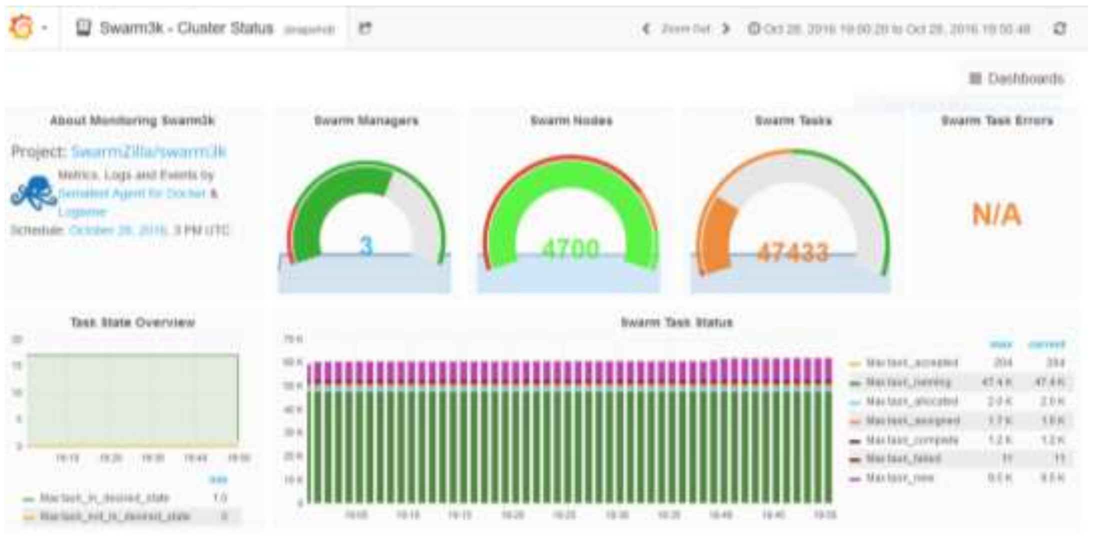
- The **orchestration** software coordinates all the nodes, offering properties to the system and functionality to the applications.



In general, the oldest tools designed with this purpose were born in the virtualisation field and in cloud systems. Some remarkable cases are Kubernetes (Google) and Apache Mesos. With the arrival of container-related technologies, these systems have also been adapted in order to interact with Docker, but there is also a native proposal called "Docker in **Swarm** mode", included in Docker version 1.12[12].

[12] Previously, there was a limited Swarm-kit that is incompatible with the current system.

- In October 2016, a scalability test was made (**swarm3k**, https://github.com/swarmzilla/swarm3k) including 4700 world-wide volunteer nodes.



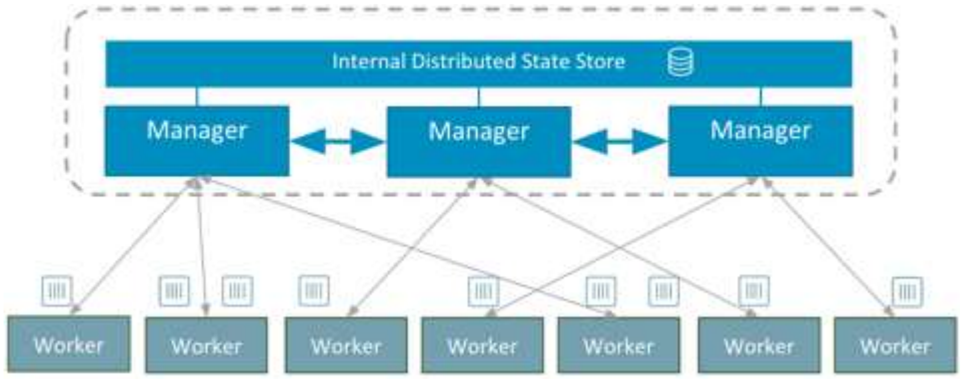
Although we won't use Swarm in the lab projects, some basic concepts about it are described in the following sections.

7.1 Concepts of Docker Swarm

A swarm is a cluster of nodes with instances of Docker (*Docker Engines*) on which to deploy services.

A node is an instance of Docker Engine that participates in the swarm.

7.1.1 Types of node



1. **Manager**. The manager sends tasks to workers, applies operations to keep the wished state, and chooses a leader for orchestration tasks.
2. **Worker**. Workers receive and execute the tasks sent by managers. Manager nodes may also behave as workers. A manager receives information on the current state of the assigned tasks.

7.1.2 Services and tasks

A **task** is the minimum scheduling unit, and is assigned by the manager to worker nodes. Once assigned to a node, it cannot migrate to another.

The **service** is the definition of the tasks to be executed in the worker nodes.

- When a service is created, it should be specified which image is needed and which must be run in the containers.

In the model of **replicated services**, the manager spreads some quantity of replicated tasks between nodes, depending on the target scaling degree.

For **global services**, the swarm executes a task of the service in each available cluster node.

7.2 Characteristics of Docker Swarm

1. **Integrated** with Docker, without requiring additional software.
2. **Decentralised** design. The distinction between manager and worker roles is set at run-time, not at the deployment time.
3. **Declarative model** of services. Docker Engine uses a declarative approach to let you define the desired state of the various services in your application.
4. **Scaling**. For each service, we may specify how many instances are required. In order to scale, the manager automates the adaptation, adding or stopping tasks according to the stated target.
5. **Target state reconciliation**. The manager monitors the state of the cluster and reconciles any existent difference with regard to the target state.
    - For instance, if it is stated that the goal is to maintain 10 replicas of a container and a node with two of these instances fails, the manager will create other two to replace them in nodes that are available.
6. **Overlay networks**. The swarm manager automatically assigns addresses to the containers on the overlay network when it initialises or updates the application.
7. **Discovery of services**. The manager assign to each service a DNS name and distributes it. Thus, load may be balanced among all the containers for that service.
8. **Load balancing**. You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.
9. **Secure by default**. Each node communicates with the others by means of TLS, with the option to establish our own certification authority.
10. **Rolling upgrades**. Service upgrading applied to the nodes is controlled by the manager and it is done following a rolling strategy (i.e., the upgrade is made in stages and in each stage only a subset of the nodes is upgraded). In case of error, the upgrade may be rolled back.

7.3 Typical steps in the use of Swarm

In the Swarm tutorial (https://docs.docker.com/engine/swarm/swarm-tutorial/), we can find an explanation on the use of this orchestrator for a minimum service, including:

1. Initialise a cluster in swarm mode
2. Add nodes to the swarm
3. Deploy services in the swarm
4. Manage the swarm when it is in operation

8 Bibliography

- www.docker.com
    - Official site of Docker
    - docs.docker.com/userguide/ (**The official documentation**)
- www.mindmeister.com/389671722/docker-ecosystem
    - Conceptual map (enormous!) on Docker/Open Container
- flux7.com/blogs/docker/ (Blog on Docker)
- www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images
    - On Dockerfile
- Advices on Dockerfile: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/
- Summary on Docker: https://github.com/wsargent/docker-cheat-sheet
- Docker in Swarm mode: https://docs.docker.com/engine/swarm/
    - Diagrams on its internal operation in http://events.linuxfoundation.org/sites/events/files/slides/containercon%20NA%20%281%29.Pdf
- The docker-compose.yml file: https://docs.docker.com/compose/compose-file/
- Poster on the Docker ecosystem: http://comp.photo777.org/wp-content/uploads/2014/09/docker-ecosystem-8.6.1.pdf
- Exhaustive list of links on Docker: https://github.com/veggiemonk/awesome-docker

9 Appendix. Dockerfile Examples

This section presents some examples that illustrate how Docker may be used for deploying multiple kinds of applications: a graphic desktop application, as LibreOffice or Firefox, a classical deployment of a web server with WordPress, and even an example of Dockerfile to deploy the code of Docker.

9.1 LibreOffice in a container

Content of the Dockerfile

```
FROM debian:stretch
MAINTAINER Jessie Frazelle <jess@linux.com>
RUN apt-get update && apt-get install -y \
    libreoffice \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*
ENTRYPOINT [ "libreoffice" ]
```

Invocation from the command line:

```
docker run -d \
  -v /etc/localtime:/etc/localtime:ro \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  -e DISPLAY=unix$DISPLAY \
  -v $HOME/slides:/root/slides \
  -e GDK_SCALE \
  -e GDK_DPI_SCALE \
  --name libreoffice \
  jess/libreoffice
```

Another **alternative**, without any Dockerfile (from http://linoxide.com/how-tos/20-docker-containers-desktop-user/)

```
docker run \
-v $HOME/Documents:/home/libreoffice/Documents:rw \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e uid=$(id -u) -e gid=$(id -g) \
-e DISPLAY=unix$DISPLAY --name libreoffice \
chrisdaish/libreoffice
```

9.2 Firefox in a container

Content of the Dockerfile

```
FROM debian:sid
MAINTAINER Jessie Frazelle <jess@linux.com>

RUN apt-get update && apt-get install -y \
    dirmngr \
    gnupg \
    --no-install-recommends \
    && apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 0AB215679C571D1C8325275B9BDB3D89CE49EC21 \
    && echo "deb http://ppa.launchpad.net/mozillateam/firefox-next/ubuntu wily main" >>
/etc/apt/sources.list.d/firefox.list \
    && apt-get update && apt-get install -y \
    ca-certificates \
    firefox \
    hicolor-icon-theme \
    libasound2 \
    libgl1-mesa-dri \
    libgl1-mesa-glx \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

ENV LANG en-US

COPY local.conf /etc/fonts/local.conf

ENTRYPOINT [ "/usr/bin/firefox" ]
```

local.conf file:

```
<?xml version='1.0'?>
<!DOCTYPE fontconfig SYSTEM 'fonts.dtd'>
<fontconfig>
<match target="font">
<edit mode="assign" name="rgba">
<const>rgb</const>
</edit>
</match>
<match target="font">
<edit mode="assign" name="hinting">
<bool>true</bool>
</edit>
</match>
<match target="font">
<edit mode="assign" name="hintstyle">
<const>hintslight</const>
</edit>
</match>
<match target="font">
<edit mode="assign" name="antialias">
```

```
<bool>true</bool>
</edit>
</match>
<match target="font">
<edit mode="assign" name="lcdfilter">
<const>lcddefault</const>
</edit>
</match>
<match target="font">
<edit name="embeddedbitmap" mode="assign">
<bool>false</bool>
</edit>
</match>
</fontconfig>
```

9.3 (Another) WordPress in a container

Content of the Dockerfile

```
FROM komljen/php-Apache
MAINTAINER Alen Komljen <alen.komljen@live.com>

ENV WP_PASS aeshiethooghahtu4Riebooquae6Ithe
ENV WP_USER wordpress
ENV WP_DB wordpress
ENV APP_ROOT /var/www/html

ADD http://wordpress.org/latest.tar.gz wordpress.tar.gz

RUN \
  tar xzf wordpress.tar.gz -C ${APP_ROOT} --strip-components 1 && \
  rm wordpress.tar.gz

COPY start.sh start.sh

VOLUME ["$APP_ROOT"]

RUN rm /usr/sbin/policy-rc.d
CMD ["/start.sh"]

EXPOSE 80
```

File `start.sh`

```
#!/usr/bin/env bash
#=============================================================================
#
#    AUTHOR: Alen Komljen <alen.komljen@live.com>
#
#=============================================================================
echo "Waiting for mysql:"
./tcp_wait.sh $MYSQL_PORT_3306_TCP_ADDR $MYSQL_PORT_3306_TCP_PORT
#-----------------------------------------------------------------------------
echo "Creating database:"
./create_db_mysql.sh $WP_DB $WP_USER $WP_PASS
#-----------------------------------------------------------------------------
sed -e "s/database_name_here/$WP_DB/
s/username_here/$WP_USER/
s/password_here/$WP_PASS/
s/localhost/$MYSQL_PORT_3306_TCP_ADDR/
/'AUTH_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'SECURE_AUTH_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'LOGGED_IN_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'NONCE_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'AUTH_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
```

```
/'SECURE_AUTH_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'LOGGED_IN_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'NONCE_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/"            \
$APP_ROOT/wp-config-sample.php > $APP_ROOT/wp-config.php

chown -R www-data: $APP_ROOT
#------------------------------------------------------------------------------
echo "Starting wordpress:"
source /etc/apache2/envvars
exec /usr/sbin/apache2 -D FOREGROUND
#==============================================================================
```

9.4 Metaexample: Docker From Docker

We can interpret it like "*the definite proof*". To sum up, it installs the development framework being needed for "compiling" Docker from the code available in GitHub (with a "loaned" account).

Content of the ENORMOUS Dockerfile:

```
FROM debian:jessie

# Add zfs ppa
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
E871F18B51E0147C77796AC81196BA81F6B0FC61 \
    || apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys E871F18B51E0147C77796AC81196BA81F6B0FC61
RUN echo deb http://ppa.launchpad.net/zfs-native/stable/ubuntu trusty main > /etc/apt/sources.list.d/zfs.list


# Allow replacing httpredir mirror
ARG APT_MIRROR=httpredir.debian.org
RUN sed -i s/httpredir.debian.org/$APT_MIRROR/g /etc/apt/sources.list

# Packaged dependencies
RUN apt-get update && apt-get install -y \
    apparmor \
    apt-utils \
    aufs-tools \
    automake \
    bash-completion \
    binutils-mingw-w64 \
    bsdmainutils \
    btrfs-tools \
    build-essential \
    clang \
    createrepo \
    curl \
    dpkg-sig \
    gcc-mingw-w64 \
    git \
    iptables \
    jq \
    libapparmor-dev \
    libcap-dev \
    libltdl-dev \
    libnl-3-dev \
    libprotobuf-c0-dev \
    libprotobuf-dev \
    libsqlite3-dev \
    libsystemd-journal-dev \
    libtool \
    mercurial \
    net-tools \
    pkg-config \
    protobuf-compiler \
    protobuf-c-compiler \
    python-dev \
    python-mock \
    python-pip \
    python-websocket \
    ubuntu-zfs \
    xfsprogs \
    libzfs-dev \
    tar \
    zip \
    --no-install-recommends \
    && pip install awscli==1.10.15
# Get lvm2 source for compiling statically
ENV LVM2_VERSION 2.02.103
RUN mkdir -p /usr/Local/lvm2 \
    && curl -fsSL "https://mirrors.kernel.org/sourceware/lvm2/lvm2.${LVM2_VERSION}.tgz" \
            | tar -xzC /usr/Local/lvm2 --strip-components=1
# See https://git.fedorahosted.org/cgit/lvm2.git/refs/tags for release tags

# Compile and install lvm2
RUN cd /usr/Local/lvm2 \
    && ./configure \
            --build="$(gcc -print-multiarch)" \
            --enable-static_Link \
    && make device-mapper \
    && make install_device-mapper
# See https://git.fedorahosted.org/cgit/lvm2.git/tree/install

# Configure the container for OSX cross compilation
ENV OSX_SDK MacOSX10.11.sdk
ENV OSX_CROSS_COMMIT a9317c18a3a457ca0a657f08cc4d0d43c6cf8953
RUN set -x \
    && export OSXCROSS_PATH="/osxcross" \
    && git clone https://github.com/tpoechtrager/osxcross.git $OSXCROSS_PATH \
    && ( cd $OSXCROSS_PATH && git checkout -q $OSX_CROSS_COMMIT) \
    && curl -sSL https://s3.dockerproject.org/darwin/v2/${OSX_SDK}.tar.xz –o
"${OSXCROSS_PATH}/tarballs/${OSX_SDK}.tar.xz" \
    && UNATTENDED=yes OSX_VERSION_MIN=10.6 ${OSXCROSS_PATH}/build.sh
ENV PATH /osxcross/target/bin:$PATH

# Install seccomp: the version shipped in trusty is too old
ENV SECCOMP_VERSION 2.3.1
RUN set -x \
    && export SECCOMP_PATH="$(mktemp -d)" \
    && curl -fsSL "https://github.com/seccomp/libseccomp/releases/download/v${SECCOMP_VERSION}/libseccomp-
${SECCOMP_VERSION}.tar.gz" \
            | tar -xzC "$SECCOMP_PATH" --strip-components=1 \
    && ( \
            cd "$SECCOMP_PATH" \
            && ./configure --prefix=/usr/local \
            && make \
            && make install \
            && ldconfig \
    ) \
    && rm -rf "$SECCOMP_PATH"

# Install Go
# IMPORTANT: If the version of Go is updated, the Windows to Linux CI machines
#            will need updating, to avoid errors. Ping #docker-maintainers on IRC
#            with a heads-up.
ENV GO_VERSION 1.7.1
RUN curl -fsSL "https://storage.googleapis.com/golang/go${GO_VERSION}.linux-amd64.tar.gz" \
    | tar -xzC /usr/Local

ENV PATH /go/bin:/usr/Local/go/bin:$PATH
ENV GOPATH /go:/go/src/github.com/docker/docker/vendor

# Compile Go for cross compilation
ENV DOCKER_CROSSPLATFORMS \
    linux/386 linux/arm \
    darwin/amd64 \
    freebsd/amd64 freebsd/386 freebsd/arm \
    windows/amd64 windows/386

# Dependency for golint
ENV GO_TOOLS_COMMIT 823804e1ae08dbb14eb807afc7db9993bc9e3cc3
RUN git clone https://github.com/golang/tools.git /go/src/golang.org/x/tools \
    && (cd /go/src/golang.org/x/tools && git checkout -q $GO_TOOLS_COMMIT)

# Grab Go's lint tool
ENV GO_LINT_COMMIT 32a87160691b3c96046c0c678fe57c5bef761456
RUN git clone https://github.com/golang/lint.git /go/src/github.com/golang/lint \
    && (cd /go/src/github.com/golang/lint && git checkout -q $GO_LINT_COMMIT) \
    && go install -v github.com/golang/lint/golint

# Install CRIU for checkpoint/restore support
ENV CRIU_VERSION 2.2
RUN mkdir -p /usr/src/criu \
    && curl -sSL https://github.com/xemul/criu/archive/v${CRIU_VERSION}.tar.gz | tar -v -C /usr/src/criu/ -xz --
strip-components=1 \
    && cd /usr/src/criu \
    && make \
    && make install-criu

# Install two versions of the registry. The first is an older version that
# only supports schema1 manifests. The second is To newer version that supports
# both. This allows integration-cli Tests to cover push/pull with both schema1
# and schema2 manifests.
ENV REGISTRY_COMMIT_SCHEMA1 ec87e9b6971d831f0eff752ddb54fb64693e51cd
ENV REGISTRY_COMMIT 47a064d4195a9b56133891bbb13620c3ac83a827
RUN set -x \
    && export GOPATH="$(mktemp -d)" \
    && git clone https://github.com/docker/distribution.git "$GOPATH/src/github.com/docker/distribution" \
    && (cd "$GOPATH/src/github.com/docker/distribution" && git checkout -q "$REGISTRY_COMMIT") \
    && GOPATH="$GOPATH/src/github.com/docker/distribution/godeps/_workspace:$GOPATH" \
            go build -o /usr/local/bin/registry-v2 github.com/docker/distribution/cmd/registry \
    && (cd "$GOPATH/src/github.com/docker/distribution" && git checkout -q "$REGISTRY_COMMIT_SCHEMA1") \
    && GOPATH="$GOPATH/src/github.com/docker/distribution/godeps/_workspace:$GOPATH" \
            go build -o /usr/local/bin/registry-v2-schema1 github.com/docker/distribution/cmd/registry \
    && rm -rf "$GOPATH"

# Install notary and notary-server
ENV NOTARY_VERSION v0.3.0
RUN set -x \
    && export GOPATH="$(mktemp -d)" \
    && git clone https://github.com/docker/notary.git "$GOPATH/src/github.com/docker/notary" \
    && (cd "$GOPATH/src/github.com/docker/notary" && git checkout -q "$NOTARY_VERSION") \
    && GOPATH="$GOPATH/src/github.com/docker/notary/vendor:$GOPATH" \
            go build -o /usr/local/bin/notary-server github.com/docker/notary/cmd/notary-server \
    && GOPATH="$GOPATH/src/github.com/docker/notary/vendor:$GOPATH" \
            go build -o /usr/local/bin/notary github.com/docker/notary/cmd/notary \
    && rm -rf "$GOPATH"

# Get the "docker-py" source so we can run their integration tests
ENV DOCKER_PY_COMMIT e2655f658408f9ad1f62abdef3eb6ed43c0cf324
RUN git clone https://github.com/docker/docker-py.git /docker-py \
    && cd /docker-py \
    && git checkout -q $DOCKER_PY_COMMIT \
    && pip install -r Test-requirements.txt

# Set user.email under crosbymichael's in-container merge commits go smoothly
```

```
RUN git config --global user.email 'docker-dummy@example.com'

# Add an unprivileged user to be used for tests which need it
RUN groupadd -r docker
RUN useradd --create-home --gid docker unprivilegeduser

VOLUME /var/lib/docker
WORKDIR /go/src/github.com/docker/docker
ENV DOCKER_BUILDTAGS apparmor pkcs11 seccomp selinux

# Let us use a .bashrc file
RUN ln -sfv $PWD/.bashrc ~/.bashrc
# Add integration helps to bashrc
RUN echo "source $PWD/hack/make/.integration-test-helpers" >> /etc/bash.bashrc

# Register Docker's bash completion.
RUN ln -sv $PWD/contrib/completion/bash/docker /etc/bash_completion.d/docker

# Get useful and necessary Hub images under we can "docker load" locally instead of pulling
COPY contrib/download-frozen-image-v2.sh /go/src/github.com/docker/docker/contrib/
RUN ./contrib/download-frozen-image-v2.sh /docker-frozen-images \
    buildpack-deps:jessie@sha256:25785f89240fbcdd8a74bdaf30dd5599a9523882c6dfc567f2e9ef7cf6f79db6 \
    busybox:latest@sha256:e4f93f6ed15a0cdd342f5aae387886fba0ab98af0a102da6276eaf24d6e6ade0 \
    debian:jessie@sha256:f968f10b4b523737e253a97eac59b0d1420b5c19b69928d35801a6373ffe330e \
    hello-world:latest@sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
# See also "hack/make/.ensure-frozen-images" (which needs to be updated any time this list is)

# Install tomlv, runc, containerd and grimes
# Please edit hack/dockerfile/install-binaries.sh to update them.
COPY hack/dockerfile/install-binaries.sh /tmp/install-binaries.sh
RUN /tmp/install-binaries.sh tomlv runc containerd grimes

# Wrap all commands in the "docker-in-docker" script to allow nested containers
ENTRYPOINT ["hack/dind"]

# Upload docker source
COPY . /go/src/github.com/docker/docker
```

Invocation from the command line

1. Assemble the entire development framework. The first time is very slow.

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

2. To test with comfort, mount our source in an interactive container:

```
docker run -v `pwd`:/go/src/github.com/docker/docker --privileged -i -t docker bash
```

3. We run the tests

```
docker run --privileged docker hack/make.sh test-unit test-integration-cli test-docker-py
```

4. We publish the result again in GitHub:

```
docker run --privileged \
 -e AWS_S3_BUCKET=baz \
 -e AWS_ACCESS_KEY=foo \
 -e AWS_SECRET_KEY=bar \
 -e GPG_PASSPHRASE=gloubiboulga \
 docker hack/release.sh
```

10 Appendix. docker-compose.yml

This annex is not exhaustive, and `docker-compose` is actively **in evolution**, so that the original documentation is a mandatory source of information and reference.

The `docker-compose.yml` file specifies the characteristics for the deployment by means of `docker-compose` of a distributed application. Its content is a sequence of text lines that respect a YAML (https://docs.docker.com/compose/yml/) syntax.

Each one of the services defined in `docker-compose.yml` has to specify exactly one image. The rest of keys are optional, and are analogous to their counterparts in the Docker **run** command. The commands in the Dockerfile do not need to be repeated in `docker-compose.yml`.

10.1 Brief introduction to YAML

YAML (*YAML Ain't Markup Language*) is a data serialisation language that describes structures with properties (trees of objects) by means of text files with a "yml" extension. These files are easier to read than JSON (see **Wikipedia** https://en.wikipedia.org/wiki/yaml).

To write files of this type, we can employ any text editor, although some like Visual Studio Code parse this language and provide some help.
To have an actual parser, we may use these commands:

- `npm install js-yaml`, that will install a local parser
- `https://github.com/nodeca/js-yaml`, that is an on-line service

In any case, for our aims, we should validate files checking whether they respect not only YAML in general, but the concrete case of Docker Compose (on-line validation in https://lorry.io/)

The syntax rules in YAML files are not complicated, but it is worth noting that...:

- Tabs cannot be used. Use blanks, instead.
- Properties and lists must be indented using at least one blank.
- Capital letters are significant in both keywords and property identifiers.

A minimum example

```
- login: ceronin23
  pass: kjh_47!
  dni: 5566557
- login: jascua
  pass: Cko05Hs
```

This example is only a sequence of keys that could be nested if necessary. A lot of parsers could treat the value 5566557 as a number. It is not mandatory the usage of quotation marks in strings.
Symbols & and * define and refer to aliases, respectively. Thus, in...

```
defaults: &defaults
  adapter:  postgres
  host:     localhost

development:
  database: myapp_development
  <<: *defaults

Test:
  database: myapp_Test
  <<: *defaults
```

…& identifies a key (“`defaults`”), << means “*place where the key will be included*”, and * makes the inclusion. The expanded version of the previous example is:

```
defaults:
  adapter:  postgres
  host:     localhost

development:
  database: myapp_development
  adapter:  postgres
  host:     localhost

Test:
  database: myapp_test
  adapter:  postgres
  host:     localhost
```

Flow style

There is an alternative syntax in YAML called “*flow style*”, in which it avoids indentation using brackets and braces, for vectors and keys respectively. With this characteristic, JSON may be considered a subset of YAML.

```
---
# Vectors
colors:
  - Network
  - blue
# With flow style...
colors: [red, blue]

# Keys
- name: Xavier
  age: 24
# With flow style...
- {name: Xavier, age: 24}
```

10.2 image

Identifier of the (local or remote) image. Compose will pull it, if that image isn’t in the local repository.

```
image: ubuntu
image: orchardup/postgresql
image: a4bc65fd
```

10.3 build

Route to a directory with the Dockerfile. If it is a relative pathname, its origin will be the folder where the YAML file is placed. Compose will build and name an image using that Dockerfile.

```
build: ./dir
build:
  context: ./dir
  dockerfile: Dockerfile-alternate
  args:
    buildno: 1
```

10.3.1 dockerfile

Alternative Dockerfile to build the image.

10.4 command

Provides a default command.

```
command: bundle exec thin -p 3000
```

10.5 links

It sets links with containers of another service. It specifies the name of the service and the alias for the `link` (SERVICE:ALIAS), or only the name if no different alias is needed.

```
web:
  links:
 - db
 - db:database
 - redis
```

Links express dependences between services and they may condition the order in which services are started.

10.6 external_links

Links to external containers to this docker-compose.yml, or even external to Compose, especially for those that provide shared services. The semantic of this command is similar to **`links`** when it is used for specifying the name of the container and the link alias (CONTAINER:ALIAS).

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

10.7 ports

Exposed ports, either stating both (COMPUTER:CONTAINER), or stating only that of the container (using a random port from the host in that case).

Note: Quotation marks are recommended for enclosing the **ports** values.

```
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
```

10.8 expose

It exposes ports without publishing them in the host, so that they are only accessible from other services linked through the docker-compose.yml file.

```
expose:
 - "3000"
 - "8000"
```

10.9 volumes

It mounts host pathnames as volumes in the container, optionally specifying a pathname in the host (HOST-pathname:CONTAINER-pathname), or an access mode (HOST-pathname:CONTAINER-pathname:mode).

```
volumes:
  # Just specify a path and let the engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to this Compose file
  - ./cache:/tmp/cache

  # Named volume
  - datavolume:/var/lib/mysql

  # User-relative path, with access mode specification
  - ~/configs:/etc/configs/:ro
```

10.10 environment

It adds environment variables.

```
environment:
  RACK_ENV: development
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SESSION_SECRET
```

10.11 labels

It adds metadata to containers by means of `labels` of Docker. It can choose between array and dictionary.

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

10.12 container_name

It sets a name for the container.

```
container_name: my-web-container
```

Since container names need to be unique, we cannot scale a container whose name has been assigned using **container_name** (it would generate an error).

10.13 dns

DMS servers to be used.

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

10.14 Analogous to docker run

working_dir, entrypoint, user, hostname, domainname, mac_address, mem_limit, memswap_limit, privileged, restart, stdin_open, tty, cpu_shares, cpuset, read_only, volume_driver

Each command has only one parameter, as in its **`docker run`** counterpart.

```
cpu_shares: 73
cpuset: 0,1

working_dir: /code
entrypoint: /code/entrypoint.sh
user: postgresql

hostname: foo
domainname: foo.com

mac_address: 02:42:ac:11:65:43

mem_limit: 1000000000
memswap_limit: 2000000000
privileged: true

restart: always

stdin_open: true
tty: true
read_only: true

volume_driver: mydriver
```

11 Appendix. Exercises

11.1 Small test

1. Let us deploy a web component web using a Docker container. The source code of the component is in folder **/usr/aplicacion/components/web.** In that directory there is also a Dockerfile to create the component image. The Dockerfile uses the **EXPOSE 8000** command to state that the server should be accessible through port 8000 of the container.

Please specify that command sequence needed in that deployment if we assume that we are in folder **/usr/aplicacion/components,** and we want to access to the component through port 80 of the host.

a)

```
$ cd web
$ docker build
$ docker run -p 80:8000 web_image
```

b)

```
$ cd web
$ docker build -t web_image .
$ docker run -p 8000:80 image
```

c)

```
$ cd web
$ docker build -t web_image .
$ docker run -port 8000 web_image
```

d)

```
$ cd web
$ docker build -t web_image .
$ docker run -p 80:8000 web_image
```

2. **Let us assume that the web component of the previous question is running in** a container with identifier **"ef5jf678rlg….".** The latter has been created from image **web_image.** ***We have made a modification in the code of the web component and we want to create a new **web_image** image**.**

   Let us assume that we are in folder **usr/aplicacion/components/web** (remember that the component Dockerfile is in that folder).
   Please choose a valid sequence of commands to launch again the modified component ensuring that the previous component is deleted:

a)

```
$ docker rm –f ef5j
$ docker build –t web_image .
```

b)

```
$ docker rmi -f web_image
$ docker build -t web_image .
$ docker run …
```

c)

```
$ docker rm –f ef5j
$ docker rmi -f web_image
$ docker build –t web_image .
$ docker run …
```

d)

```
$ docker rm -f web_image
$ docker build -t web_image .
$ docker run …
```

3. The web component image presented in the previous questions contains node, the binary library of ZeroMQ and also npm. However it does not include the zmq *binding* for node. Let us extend the image including the zmq binding for node.

Please choose which of the following sentences is correct:

a) The unique way to do it is extending the Dockerfile of the component and creating again its image with "**docker build** …"

b) We can run the container in an interactive way with "**docker run -i -t …"**and install zmq with "**npm install zmq"** directly in that running container. Later, we may obtain the new image using "**docker commit …"** and specifying the container ID as one of its arguments.

c) We can run the container in an interactive way with "**docker run -i -t ….." a**nd install zmq with "**npm install zmq"** directly in that running container. The image is updated automatically.

d) If we use a Dockerfile to obtain the new image, it must contain the command: **"CMD npm install zmq".**

4. Now we add to the web component a database component. We use docker-compose to deploy the new service. The file **docker-compose.yml** is in folder **/usr/aplicacion/service.** Let us assume that we are in that folder of the host.

We should:

1. Launch the service with all its components.
2. Check that the service is running.
3. Stop all service components.
4. Remove all service containers.

Please choose which of the following sentences is true:

a) This sequence implements all the specified actions:

```
$ docker-compose up
$ docker ps -a
$ docker-compose stop
$ docker-compose rm -f
```

b) This sequence implements all the specified actions:

```
$ docker-compose up
$ docker ps -a
$ docker stop
$ docker rm -f
```

c) In order to delete the containers associated to a service we should remove, individually, each one of them.

d) In order to deploy with **"docker-compose up"** we need to create, previously, all the component images.

11.1.1 Solutions

1. d)
2. c)
3. **b)**
4. **a)**

11.2 TcpProxy For WordPress

The example of WordPress based on Fedora from Section 6.5 describes, configures and deploys a **WordPress** service.

Once this service is deployed, we should extend it, including a pertinent new component: myTCPProxy.js from Lab project 1. Let us assume the following **Lab3myTcpProxyA.js**

Code (Lab3myTcpProxyA.js)

```
var net = require('net');

var LOCAL_PORT = 80;
var LOCAL_IP = '127.0.0.1';
if (process.argv.length != 4) {
  console.error("The WordPress port and IP should be given as arguments!!");
  process.exit(1);
}
var REMOTE_PORT = process.argv[2];
var REMOTE_IP = process.argv[3];

var server = net.createServer(function (socket) {
  socket.on('data', function (msg) {
    var serviceSocket = new net.Socket();
    serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
      serviceSocket.write(msg);
    });
    serviceSocket.on('data', function (data) {
      socket.write(data);
    });

  });
}).listen(LOCAL_PORT, LOCAL_IP);
console.log("TCP server accepting connection on port: " + LOCAL_PORT);
```

But this code is not enough for integrating the new component in a WordPress service deployment. The additional component has to forward the service requests. Thus, it behaves as a receiver of requests and forwards them to the port and IP of the WordPress web component. To this end, you should **complete the following tasks**:

1. Create a **Dockerfile** to deploy this component. It should forward the requests to the IP and port of WordPress. As a reference, you may take as a base this Dockerfile. It was used for building the **worker** component in the client-broker-worker system:

```
FROM tsr/fedora-node-devel
COPY ./worker.js worker.js
# We assume that each worker is linked to the broker
# container. Such broker container should have the
# 'BROKER' name and should be linked to this worker.
CMD node worker $BROKER_PORT
```

2. Write a **docker-compose.yml** that includes this component.

11.2.1 Solution

It is worth noting that the distributed application has to keep the same external interfaces, so that port 8000, previously associated to the web component, is now linked to the TCP proxy. This does not force any change on the web component port, but on its visibility (e.g., by means of expose) specified in the docker-compose.yml file.

Dockerfile

It has a general part analogous to that of the previous example (worker) on which some adjustments are needed:

a) The NodeJS program to be executed: it is necessary to copy that file previously.
b) Expose the service port to be used by this TCP proxy.
c) Use the suitable data from the **web** component (port and IP) in the program invocation.

```
FROM tsr/fedora-node-devel
COPY ./Lab3myTcpProxy.js Lab3myTcpProxy.js
EXPOSE 80
CMD node Lab3myTcpProxy $WEB_PORT $WEB_ADDR
```

It is important to use the suitable command to create a component from this Dockerfile, since this sets the component name. To this end, in the folder that contains both the Dockerfile and the Lab3myTcpProxy.js file, we use:

```
docker build -t proxy .
```

`docker-compose.yml`

We need a section for the component proxy, standing out:

a) Association with the component that we have created (`image` command)
b) Access to the information created for the web component in its deployment (value web for the `links` command)
c) A command to use a port (8000) from the host as if it were port 80 (variable LOCAL_PORT) in the container: value “8000:80” for the **`ports`** command.

And also we need “to hide” port 8000 of the web component to avoid collisions with that of the proxy, and/or that it could be directly accessed without going through the proxy.

a) Remove the `ports` command from the web section.
b) Specify value “8000” for the `expose` command in the web section (this allows its internal usage)

```
proxy:
  image: proxy
  links:
    - Web
  ports:
    - "8000:80"
  environment:
    - WEB_PORT=8000
    - WEB_ADDR=web
Web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  expose:
    - "8000"
  links:
    - db
  volumes:
    - .:/code
db:
  image: orchardup/mysql
  environment:
    MYSQL_DATABASE: wordpress
```

If these previous steps have been followed, and we go to the folder of this docker-compose.yml file, then the service could be started using:

```
docker-compose up
```

11.3 Publisher/Subscriber

This exercise discusses a distributed application based on NodeJS+ZMQ and its execution in containers in a virtual machine of the DSIC labs.

Let us use the PUB/SUB pattern to intercommunicate the components. There is a single publisher (`pubextra.js`) that sends at regular intervals a message with a subject, whereas an indeterminate number of subscribers (`subextra.js`) share the same code.

An example of invocation of `pubextra.js` is…

```
node pubextra.js tcp://*:9999 2 sport science society
```

…and, in this example, this means that…:

- The input endpoint of the service is `tcp://*:9999` (first argument)
- The interval between message publications is **2** seconds (second argument). After each sending, the publisher prints the name of the broadcast subject to the screen.
- The subjects (rest of arguments) between which it is necessary *to commute* are **`sport`** (first message), **`science`** (second) and **`society`** (third). After the last subject, it has to go back to the first. There is not any default number of subjects.

This is a possible implementation of **pubextra.js** that respects its requirements:

```
// pubextra.js
var zmq = require('zmq')
var publisher = zmq.socket('pub')

// Check how many arguments have been received.
```

```
if (process.argv.length < 5) {
  console.error("Format is 'node pubextra URL secs topics+'");
  console.error("Example: 'node pubextra tcp://*:9999 2 sport science society'");
  process.exit(1);
}
// Get the connection URL.
var url = process.argv[2];
// Get period
var period = process.argv[3];
// Get topics
var topics = process.argv.slice(4, process.argv.length);
var i=0;
var count=0;
publisher.bind(url, function(err) {
  if(err) console.log(err);
  else console.log('Listening on '+url+' ...');
})

setInterval(function() {
  ++count;
  publisher.send(topics[i]+' msg '+count);
  console.log('Sent '+topics[i]+' msg '+count);
  i=(i+1)%topics.length;
  if (count>100) process.exit();
},period*1000);
```

In the case of subscribers, their invocation of `subextra.js` is:

```
node subextra.js tcp://localhost:9999 sport
```

That, <u>in this example</u>, it means that…:

- It is connected to the publisher in `tcp://`**`localhost:9999`** (first argument)
- It wishes to receive messages only for the **`sport`** subject (second argument). After each reception, the subscriber should print the received message.

This is an implementation of **subextra.js**:

```
// subextra.js
var zmq = require('zmq')
var subscriber = zmq.socket('sub')

// Check how many arguments have been received.
if (process.argv.length != 4) {
  console.error("Format is 'node subextra URL topic'");
  console.error("Example: 'node subextra localhost:9999 sport'");
  process.exit(1);
}
// Get the connection URL.
var url = process.argv[2];
// Get topic
var topic = process.argv[3]
subscriber.on('message', function(data) {
  console.log('Received '+data);
})

subscriber.connect(url);
subscriber.subscribe(topic);
```

A possible use of these components can consist of an instance of the publisher (initiate it at the end) and three subscriber instances. It is recommended to run each one in a different window:

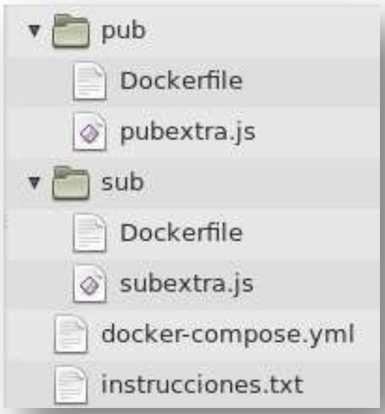
```
node subextra.js tcp://localhost:9999 fashion
node subextra.js tcp://localhost:9999 health
node subextra.js tcp://localhost:9999 leisure
node pubextra.js tcp://*:9999 1 fashion health business leisure
```

The goal in this exercise is to deploy these components in containers in our lab virtual machine.
To this end, you should write a publisher `Dockerfile`, and another one shared by subscribers.
The basic requirements *include* these instructions:

```
FROM fedora
RUN dnf -y -q install nodejs zeromq-devel python gcc-c++ make
RUN npm install zmq
```

But each component will have to add its specific requirements to be deployed.

- Subscribers will always request subject `leisure` (unknown data for the publisher).
- The publisher will always listen to `tcp://*:8888` (unknown data for subscribers).



You should write the `Dockerfile` for each one, the `docker-compose.yml` that relates these components, and scale the resulting service to 4 subscribers.

To complete this exercise, it is convenient to create a **pub** folder that contains all the elements for publisher deployment, a **sub** folder for that related to subscribers, and the `docker-compose.yml` file should be placed in the folder that holds the pub and sub folders.

It is a good idea to create an `instructions.txt` file stating with accuracy the sequence of necessary commands to build the components and to deploy, as already said, a service composed of 1 publisher and 4 subscribers.

11.3.1 Solution

Instructions.txt

Let as assume that `pubextra.js` is in `pub`, `subextra.js` is in `sub`, and that our current folder holds both of those folders.

```
#!/bin/sh
for i in pub sub
do
  cd $i
  docker build -t ${i}extra .
  cd ..
done
docker-compose up -d
docker-compose scale pub=1 sub=4
```

docker-compose.yml

```
version: '2'
services:

  sub:
    image: subextra
    links:
     - pub
    environment:

     - PUB_URL=tcp://pub:888


  pub:
    image: pubextra
```

pub/Dockerfile

```
FROM fedora
RUN dnf -y -q install nodejs zeromq-devel python gcc-c++ make
RUN npm install zmq
COPY ./pubextra.js pubextra.js
EXPOSE 8888
CMD node pubextra.js *:8888 1 fashion health sport leisure
```

sub/Dockerfile

```
FROM fedora
RUN dnf -y -q install nodejs zeromq-devel python gcc-c++ make
RUN npm install zmq
COPY ./subextra.js subextra.js
# We assume that each subscriber is linked to the publisher
# container.
CMD node subextra.js $PUB_URL leisure
```