Cloud for Data Analytics

3 - Microservices with Docker

Juan Ángel Lorenzo del Castillo

École Internationale des Sciences du Traitement de l'Information

ING3 - Ingénierie Mathématique et Simulation Informatique

2019-2020

juanangel.lorenzodelcastillo@eisti.eu

Acknowledgements

Disclaimer

Most of the content of these slides is either taken or inspired by the official Docker documentation, from courses I took with the Docker guys, or from my own experience.

So (for the biggest part of it) I am not claiming any authorship of this course.

Table of Contents

Table of Contents

Introduction

- In the previous lesson we have seen how to manually create a Docker image.
- However, this manual process is tedious and prone to errors.
- Let us learn how to automate the build process by writing a `Dockerfile`.

Dockerfiles

- A `Dockerfile` is a is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The `docker build` command builds an image from a `Dockerfile`.

Writing a Dockerfile

1. Our `Dockerfile` must be in a **new**, **empty** directory. Create a directory to hold our `Dockerfile`:

```
$ mkdir mydockerimage
```

2. Create a `Dockerfile` file inside this directory with your favourite text editor:

```
$ cd mydockerimage
$ vim Dockerfile
```

3. Type this into your `Dockerfile`:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install figlet
```

- `FROM`: base image for our build.
- Each `RUN` will be executed by Docker during the build. RUN is used to install libraries, packages, and various files.
- In many cases, we will add the `-y` flag to `apt-get`.

Writing a Dockerfile

4. Build the image:

```
$ docker build -t figlet .
```

   - `-t`: tag to apply to the image
   - `.`: location of the build context (the directory where the `Dockerfile` is).

5. Check the new created image:

```
$ docker images
```

The Docker caching system

If we run the same build again, it will be instantaneous

- After each build step, Docker takes a snapshot of the resulting image.
- Before executing a step, Docker checks if it has already built the same sequence.
- Docker uses the exact strings defined in your Dockerfile, so:

```
RUN apt-get install progA progB
```

is different from

```
RUN apt-get install progB progA
```

- A rebuild can be forced with:

```
$ docker build --no-cache ...
```

Running the image

- The image is not different from those produced manually:

```
$ docker run -ti figlet
root@98c9f2553bfc:/# figlet hola

_           _
| |__   ___ | | __ _
| '_ \ / _ \| |/ _` |
| | | | (_) | | (_| |
|_| |_|\___/|_|\__,_|
```

Image history

- The `history` command lists all the layers composing an image.
- When an image was built with a `Dockerfile`, each line corresponds to a line of the `Dockerfile`:

```
$ docker history figlet
IMAGE          CREATED         CREATED BY                                      SIZE
93f8d2d947df   36 minutes ago  /bin/sh -c apt-get install figlet                1.008 MB
591d2039b23d   36 minutes ago  /bin/sh -c apt-get update                       40.98 MB
c73a085dc378   10 months ago   /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
<missing>      10 months ago   /bin/sh -c mkdir -p /run/systemd && echo 'doc    7 B
<missing>      10 months ago   /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/    1.895 kB
<missing>      10 months ago   /bin/sh -c rm -rf /var/lib/apt/lists/*          0 B
<missing>      10 months ago   /bin/sh -c set -xe   && echo '#!/bin/sh' > /u   745 B
<missing>      10 months ago   /bin/sh -c #(nop) ADD file:cd937b840fff16e04e   127.1 MB
```

JSON vs String syntax

Most Dockerfile arguments can be passed in two forms:

- Plain string: `RUN apt-get install figlet`
  String syntax specifies a command to be wrapped within `/bin/sh -c "..."`.
- JSON list: `RUN ["apt-get","install", "figlet"]`
  JSON syntax specifies an exact command to execute.

Change your Dockerfile...

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
```

...and build the new Dockerfile.

The JSON syntax is the preferred way. We will see later why!

Default commands

We want to execute a command by default when users run our container.

- For example, a welcome message. So we will want to execute:

```
figlet -f script hola
```

  - `-f`: to use a fancy font.
  - `hola`: message that we want to display.

- Use `CMD` to define a default command when none is given. At the end of your Dockerfile, append the line:

```
CMD figlet -f script hola
```

  - CMD can appear at any point in the file.
  - Each CMD will replace and override the previous one.

- Build the image as before and then run it with:

```
$ docker run figlet
```

- To override CMD, we will specify a different program to run. For example, to get a bash shell:

```
$ docker run -it figlet bash
```

Default commands (II)

We want to specify the parameters (a different message), while retaining the command (figlet) and some default parameters:

- Use `ENTRYPOINT` to define a base command (and its parameters) for the container. At the end of our Dockerfile, comment out the CMD line and append the line:

```
ENTRYPOINT ["figlet", "—f", "script"]
```

  - The command line arguments will be appended to those parameters.
  - Like CMD, ENTRYPOINT can appear anywhere, and replaces the previous value.

- Build the image as before and then run it with the message of your choice:

```
$ docker run figlet Bonjour
```

Default commands (III)

We want to define a default message to show when no parameters are given to the container:

- Use `ENTRYPOINT` and `CMD` together.
- `ENTRYPOINT` to define the base command (and its parameters) for our container.
- `CMD` to specify the default parameter(s) for this command.
- Both must have JSON syntax.

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
CMD ["hello world"]
```

- If we do not specify extra command-line arguments when starting the container, the value of CMD is appended.
- Otherwise, our extra command-line arguments are used instead of CMD.

Default commands (IV)

- What happens if we want to run a shell in our container?
- We cannot use `docker run -it figlet bash` because that would just tell the container to display the word "bash".
- Use the `--entrypoint` parameter:

```
$ docker run -it --entrypoint bash figlet
```

Default commands (IV)

- What happens if we want to run a shell in our container?
- We cannot use `docker run -it figlet bash` because that would just tell the container to display the word "bash".
- Use the `--entrypoint` parameter:

```
$ docker run -it --entrypoint bash figlet
```

Default commands (IV)

- What happens if we want to run a shell in our container?
- We cannot use `docker run -it figlet bash` because that would just tell the container to display the word "bash".
- Use the `-entrypoint` parameter:

```
$ docker run -it --entrypoint bash figlet
```

Copying files during the build process

Do you remember the `docker cp` command from TP1?

- We can copy files from the build context (the Dockerfile directory) to the container that we are building.
- Let us build a container that compiles a basic “Hello world” program in C.

1. Create a `hello.c` file into a new directory with the following content:

```
int main () {
puts("Hello world!");
return 0; }
```

2. Write the following Dockerfile and put it in the new directory:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y build-essential
COPY hello.c /
RUN make hello
CMD /hello
```

   - Note `COPY` to copy the C file and `build-essential` to install a compiler.
   - Run `docker build -t hello .` in this directory.
   - Run `docker run hello`.

Copying files during the build process (II)

Note that:

- Docker will cache all steps involving `COPY`: if we do not modify the `hello.c` file, Docker will not perform any action.
- It is possible to `COPY` directories recursively.
- If we really wanted to compile C code in a compiler, we would:
  - Place it in a different directory, with the WORKDIR instruction (we will see it later).
  - Even better, use the gcc official image.

More about syntax and keywords in Dockerfiles

Dockerfile usage summary:

- Dockerfile instructions are executed in order.
- Each instruction creates a new layer in the image.
- Instructions are cached. If no changes are detected then the instruction is skipped and the cached layer used.
- The FROM instruction MUST be the first non-comment instruction.
- Lines starting with # are treated as comments.
- You can only have one CMD and one ENTRYPOINT instruction in a Dockerfile.

More about syntax and keywords in Dockerfiles

The MAINTAINER instruction:

- Tells you who wrote the Dockerfile. It is optional but recommended.

```
MAINTAINER Juan Angel Lorenzo <jlo@eisti.eu>
```

The EXPOSE instruction:

- Tells Docker what ports are to be published in this image.

```
EXPOSE 8080
EXPOSE 80 443
EXPOSE 53/tcp 53/udp
```

- All ports are private by default (not reachable from outside the container).
- The Dockerfile doesn't control if a port is publicly available.
- When you `docker run -p <port> ...`, that port becomes public (even if it was not declared with EXPOSE.) Do you remember the `-p` option used in TP1?
- When you `docker run -P ...` (without port number), all ports declared with EXPOSE become public. `-P` stands for `--publish-all`. A public port is reachable from other containers and from outside the host.

More about syntax and keywords in Dockerfiles

The ADD instruction:

- It works almost like COPY, but has a few extra features.
- ADD can get remote files. For example, to download the `webapp.jar` file and place it in the `/var/www/` directory:

```
ADD http://www.example.com/webapp.jar /var/www/
```

- ADD will automatically unpack zip files and tar archives. For example, to unpack `assets.zip` into `/var/www/htdocs/assets`:

```
ADD assets.zip /var/www/htdocs/assets/
```

- However, ADD will not automatically unpack remote archives.

More about syntax and keywords in Dockerfiles

The WORKDIR instruction:

- The WORKDIR instruction sets the working directory for subsequent instructions. It also affects CMD and ENTRYPOINT, since it sets the working directory used when starting the container.

```
WORKDIR /src
```

The ENV instruction:

- The ENV instruction specifies environment variables that should be set in any container launched from the image. For example:

```
ENV WEBAPP_PORT 8080
```

This will result in an environment variable being created in any containers created from this image of

```
WEBAPP_PORT=8080
```

- As we saw in TP1, we can also specify environment variables in `docker run` with the `-e` option:

```
$ docker run -e WEBAPP_PORT=8000 -e WEBAPP_HOST=www.example.com ...
```

More about syntax and keywords in Dockerfiles

The USER instruction:

- It sets the user name or UID to use when running the image.
- It can be used multiple times to change back to root or to another user.

The `docker inspect` command:

- it will output detailed information about a container or an image, in JSON format.

Table of Contents

Introduction

- In TP1 we learnt how to expose some ports from a containerised service.
- However, Docker offers more possibilities to connect containers in a network.
- In this section we will see some of them.

Introduction

Why are we mapping ports ?

- We are out of IPv4 addresses.
- Containers cannot have public IPv4 addresses.
- They have private addresses.
- Services have to be exposed port by port.
- Ports have to be mapped to avoid conflicts.

Running a web server

- Example: run the Docker image `nginx`, which contains a basic web server:

```
$ docker run -d -P nginx
```

  - `-d` tells Docker to run the image in the background
  - `-P (--publish-all)` tells Docker to make this service reachable from other computers.

- To find the web server ports so that we can connect to the server, we can use `docker ps`:

```
$ docker ps
CONTAINER ID        IMAGE               ...   PORTS
c8df1b7acec6        nginx                     0.0.0.0:32768->80/tcp
```

  - The web server is running on port 80 inside the container.
  - This port is mapped to port 32768 on our host.
  - We can connect to the web server by pointing our browser to `http://localhost:32768`

- We can use as well `docker port` to find the ports exposed by the server (useful for scripting):

```
$ docker port c8df1b7acec6 80
0.0.0.0:32768
```

Manual allocation of port numbers

- We can set port numbers manually:

```
$ docker run -d -p 80:80 nginx
$ docker run -d -p 8080:80 -p 8000:80 nginx
```

  - We are running two web servers
  - The first one is exposed on port 80.
  - The second one is exposed on ports 8080 and 8000.
  - The convention is `port-on-host:port-on-container`.

Finding and setting the container's IP address

- We can use `docker inspect`. It will return a huge amount of information.
- To filter out the rest of information and get the IP address, use `--format`

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' c58785411
10.42.0.2
```

To manually set a container's IP address:

- Use `--ip`
- The IP address has to be within the subnet used for the container. For example:

```
$ docker network create --subnet 10.66.0.0/16 pubnet
$ docker run --net pubnet --ip 10.66.66.66 -d nginx
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <containerID>
```

The Container Network Model (CNM)

- The CNM adds the notion of a *network*, and a new top-level command to manipulate and see those networks: `docker network`.

```
$ docker network ls
NETWORK ID          NAME                DRIVER
9430a7f363d3        bridge              bridge
d5244c7c0aae        host                host
7871bbcd7714        none                null
```

- What is in a *network*?
  - Conceptually, a network is a virtual switch.
  - It can be local (to a single Engine) or global (across multiple hosts).
  - A network has an IP subnet associated to it.
  - A network is managed by a driver (we will see it later).
  - A network can have a custom IPAM (IP allocator).
  - Containers with explicit names are discoverable via DNS.
  - A new multi-host driver, overlay, is available out of the box.
  - More drivers can be provided by plugins (OVS, VLAN...)

Creating and using networks

1. Create a network called `localnet` and verify the type of driver used:

```
$ docker network create localnet
$ docker network ls
NETWORK ID          NAME                DRIVER
...                 ...                 ...
30d798864c70        localnet            bridge
```

2. Create a container named *srv1* on this network:

```
$ docker run -ti --name srv1 --net localnet alpine sh
```

3. On a different terminal, create a second container named *srv2* on the same network:

```
$ docker run -ti --name srv2 --net localnet alpine sh
```

4. From any of the containers we can resolve and ping the other one:

```
$ ping srv1
PING srv1 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.265 ms
...
```

Note that containers can contact each other only when they are on the same network.

Networks and scope

Let us run an application that requires two containers connected together.

- The first container is a web server.
- The second container is a data store.
- We will place them both on the `localnet` network created before.
- Start the following container and check the exposed ports:

```
$ docker run --net localnet —d —P jpetazzo/trainingwheels
$ docker port e99483f9d92e
5000/tcp —> 0.0.0.0:32770
```

- Try to access the application from the host browser by pointing to `http://localhost:32770`. You will get an error: *Error -2 connecting to redis:6379. Name or service not known.*
- This happens because this container tries to resolve the name `redis`, and the `Redis` service is not running.
- Start a Redis container in the same network as the previous container. It must have the right name (`redis`) so the application can find it:

```
$ docker run --net localnet --name redis —d redis
```

Network aliases

What if we want to run multiple copies of our application?

- Since names are unique, there can be only one container named `redis` at a time.
- We can specify `--net-alias` to define network-scoped aliases, independently of the container name.
- Let us remove the redis container and create one that does not block the `redis` name:

```
$ docker rm -f redis
$ docker run --net localnet --net-alias redis -d redis
```

- The application still works.

Custom networks

When creating a network, extra options can be provided.

- `--internal` disables outbound traffic (the network will not have a default gateway).
- `--gateway` indicates which address to use for the gateway (when outbound traffic is allowed).
- `--subnet` (in CIDR notation) indicates the subnet to use.
- `--ip-range` (in CIDR notation) indicates the subnet to allocate from.
- `--aux-address` allows to specify a list of reserved addresses (which will not be allocated to containers).

It is possible to set a container's address with `--ip`.

- The IP address has to be within the subnet used for the container. For example:

```
$ docker network create --subnet 10.66.0.0/16 pubnet
$ docker run --net pubnet --ip 10.66.66.66 -d nginx
```

The different network drivers

A container can use one of the following drivers:

- bridge (default)
- none
- host
- container

The driver is selected with `docker run --net ...`

The different network drivers

The default bridge

- By default, the container gets a virtual eth0 interface. (In addition to its own private lo loopback interface.)
- That interface is provided by a veth pair.
- It is connected to the Docker bridge. (Named `docker0` by default; configurable with `--bridge`.)
- Addresses are allocated on a private, internal subnet. (Docker uses 172.17.0.0/16 by default; configurable with `--bip`.)
- Outbound traffic goes through an iptables MASQUERADE rule.
- Inbound traffic goes through an iptables DNAT rule.
- The container can have its own routes, iptables rules, etc.

The different network drivers

The null driver

- Container is started with `docker run --net none ...`
- It only gets the lo loopback interface. No eth0.
- It can't send or receive network traffic.
- Useful for isolated/untrusted workloads.

The different network drivers

The host driver

- Container is started with `docker run --net host ...`
- It sees (and can access) the network interfaces of the host.
- It can bind any address, any port (for ill and for good).
- Network traffic doesn't have to go through NAT, bridge, or veth.
- Performance = native!
- Example use cases: Performance sensitive applications (VOIP, gaming, streaming...)

The different network drivers

The container driver

- Container is started with `docker run --net container:id ...`
- It re-uses the network stack of another container.
- It shares with this other container the same interfaces, IP address(es), routes, iptables rules, etc.
- Those containers can communicate over their lo interface. (i.e. one can bind to 127.0.0.1 and the others can connect to it.)

Table of Contents

Introduction

- In TP1 we saw briefly how to have a directory from the host mounted into a container.
- We used **volumes** for that.
- In this section we will see volumes in more detail:
  - How to create containers holding volumes.
  - How to share volumes across containers.
  - How to share a host directory with one or many containers.

Declaring volumes

Volumes can be declared in two different ways:

- Within a Dockerfile, with a VOLUME instruction.

```
VOLUME /uploads
```

- On the command-line, with the `-v` flag for `docker run`.

```
$ docker run -d -v /uploads myapp
```

In both cases, `/uploads` (inside the container) will be a volume.

Volumes properties

1. Volumes bypass the copy-on-write system
   - The I/O performance on a volume is exactly the same as I/O performance on the Docker host.
   - When you docker commit, the content of volumes is not brought into the resulting image.
   - If a RUN instruction in a Dockerfile changes the content of a volume, those changes are not recorded neither.
   - If a container is started with the `--read-only` flag, the volume will still be writable (unless the volume is a read-only volume).

Volumes properties

2. Volumes can be shared across containers
   - You can start a container with exactly the same volumes as another one.
   - The new container will have the same volumes, in the same directories.
   - They will contain exactly the same thing, and remain in sync.
   - Under the hood, they are actually the same directories on the host anyway.
   - This is done, as we saw in TP1, using the `--volumes-from` flag for `docker run`:

```
$ docker run -it --name alpha -v /var/log ubuntu bash
root@fa087c31001d:/# echo "hola!" > /var/log/greeting
```

In another terminal, we can start another container with the same volume:

```
$ docker run --volumes-from alpha ubuntu cat /var/log/greeting
hola!
```

Check what happens if the `alpha` container is stopped before running the second container.

   - The `--volumes-from` option tells Docker to re-use all the volumes of an existing container. Useful to migrate data from a given image to a newer one.

Volumes properties

3. Volumes exist independently of containers
   - If a container is stopped, its volumes still exist and are available.
   - We can see all existing volumes and manipulate them with `docker volume`:

```
$ docker volume ls
DRIVER      VOLUME NAME
local       3b09dde06b1c237fd9ad4c52d2ec0f692380ef2705a15824b3063b98ccb00e36
local       3b65aba948ba6b0b5e5f95f2cc24ef69f7853371ec0dc8656b0a1b89eba78f48
local       415e072f9cbbcc1fd370445cf818782cac60e30ddc0f6676cfe023aa8bdf194a
local       8047d59c039367103b4e75626628161dfa2ce672d5d4783b3dc8c7adebdc3732
local       85bb6cc21f2977197b72af4e5b5fc61974991683df46e720400972fcdf68e574
```

Named volumes

- Volumes can be created without a container, then used in multiple containers.
- Volumes are not anchored to a specific path.
- We can name each container to reference them easily later.

```
$ docker volume create --name=website
website
```

- We can inspect the container with:

```
$ docker volume inspect website
```

Using named volumes

- Volumes are used with the `-v` option:

```
$ docker run -d -p 8888:80 -v website:/usr/share/nginx/html nginx
```

Check in a browser that the server is running correctly.

- We can run the same volume in another container at the same time. For example, let us run a text editor in another container to modify the content of the `website` volume:

```
$ docker run -v website:/website -w /website -it alpine vi index.html
```

Modify the content of the `index.html` page and check that it has changed on your browser. What is the purpose of the `-w` option?

Using named volumes

- We can **share a directory** between the host and a container:

```
$ docker run -d -v /path/on/the/host:/path/on/container image ...
```

  This is useful, for example, to share your source directory between your host (where the source gets edited) and the container (where it is compiled or executed).

- In the same way, we can **share a single file** between the host and a container with the `-v` option.

Some final notes on volumes

- Check with `docker volume ls` what happens when you remove a container with volumes.
- Use `docker inspect` to check if an image has volumes and which paths they have.

Table of Contents

Motivation

- We have seen how to build a single container or, at most, two containers interacting with each other.
- To do that, we have deployed directly the containers with `docker run` or we have used a `Dockerfile`.
- If we want to start a complex stack made of multiple containers, we need a different tool: **Docker Compose**.
- We will learn how to use Compose to bootstrap a development environment.

Compose overview

Workflow as a developer:

1. Build an image containing our development environment (Django, J2EE...).
2. Start a container from that image. Use a volume shared between the host and the container (`-v` flag) to mount the source code inside the container.
3. Edit the source code outside the container, using regular tools (vim, emacs, Eclipse...).
4. Test your application inside the container.
5. Repeat the last two steps until you are satisfied.
6. When done, commit+push the source code changes (because you are using version control, right?)

Compose overview

Workflow as a user:

1. Clone a code from gitHub, gitLab, etc.
2. Run docker-compose up.
3. The application is up and running.

Compose overview

How to work with Docker Compose:

- Describe a set (or stack) of containers in a YAML file called `docker-compose.yml`.
- Run `docker-compose up`.
- Compose automatically pulls images, builds containers, and starts them.
- Compose can set up links, volumes, and other Docker options for you.
- Compose can run the containers in the background, or in the foreground.
- When containers are running in the foreground, their aggregated output is shown.

Installing Docker Compose

- Check if `docker-compose` is installed:

```
$ docker-compose --version
docker-compose version 1.8.0, build unknown
```

- It can be installed with `pip install docker-compose` or, more conveniently, with `apt-get install docker-compose`.

Docker Compose in action

1. Clone the source code for the application to run. For example:

```
$ mkdir trw ; cd trw
$ git clone git://github.com/jpetazzo/trainingwheels
$ cd trainingwheels
```

2. Start the application:

```
$ docker-compose up
```

3. Verify that the application is runnning at `http://localhost:8000`
4. The application can be gracefully terminated with CTRL-C.

Docker Compose configuration file

- Check the directory `trainingwheels` that we just cloned. There is a YAML file called `docker-compose.yml`:

```
version: "2"

networks:
  localnet2:
    driver: bridge
    ipam:
      config:
        - subnet: 10.44.0.0/16

services:
  www:
    build: www
    ports:
      - 8000:5000
    user: nobody
    environment:
      DEBUG: 1
    command: python counter.py
    volumes:
      - ./www:/src
    networks:
      - localnet2

  redis:
    image: redis
    networks:
      - localnet2
```

Docker Compose configuration file (II)

- `version` is mandatory and should be "2" ("3" has already been released).
- `services` is mandatory and corresponds to the content of the version 1 format.
- `networks` is optional and can define multiple networks on which containers can be placed.
- `volumes` is optional and can define volumes to be used (and potentially shared) by the containers. It translates to one (or multiple) `-v` options.
- Each service in the YAML file must contain either `build`, or `image`.
  - `build` indicates a path containing a Dockerfile.
  - `image` indicates an image name (local, or on a registry).
- `command` indicates what to run (like CMD in a Dockerfile).
- `ports` translates to one (or multiple) `-p` options to map ports. You can specify local ports (i.e. `x:y` to expose public port `x`).
- Check the full list at
  - https://docs.docker.com/compose/compose-file/compose-file-v2/ (version 2).
  - https://docs.docker.com/compose/compose-file/ (version 3).

Docker Compose commands

In addition to `docker-compose up`, there are other useful commands and options:

- `docker-compose build`: It will execute `docker build` for all containers mentioning a build path.
- `docker-compose -d up`: To start containers in the background.
- `docker-compose ps`: To see only the status of the container in the current stack.
- `docker-compose kill`: If you have started your application in the background with Compose and want to stop it easily.
- `docker-compose rm`: To remove containers (after confirmation).
- `docker-compose down`: It will stop and remove containers.

Table of Contents

Exercise 2: Automated Build Process

Hands On!

Go to TP2