ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

PRINCIPLES, MODELS AND APPLICATIONS FOR DISTRIBUTED SYSTEMS M – Module 2

Docker Hands-on

Prof. Davide Borsatti – *davide.borsatti@unibo.it*

Department of Electrical, Electronic and Information Engineering “Guglielmo Marconi”

Docker Setup

To install Docker you can follow the official guides.

If you are working with a Linux based machine you can follow the guide for your distro to install the Docker Engine [Suggested option].

Otherwise, you can install Docker Desktop for your machine [This solution works also on Windows and Mac machines].

Let’s launch our first container

Let’s run the following command:

```
$ docker run busybox echo hello world
```

- We used one of the smallest, simplest images available: busybox.
- busybox is typically used in embedded systems (phones, routers...)
- We ran a single process echo hello world

Let’s do something more complex

Let’s run the following command:

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

This is a brand new container.

- It runs a bare-bones, no-frills ubuntu system.
- `-it` is shorthand for `-i -t`
  - `-i` tells Docker to connect us to the container's stdin.
  - `-t` tells Docker that we want a pseudo-terminal.

Where’s our container?

- Can we reuse that container that we took time to customize?
  We can, but that's not the default workflow with Docker.

- What's the default workflow, then?
  Always start with a fresh container. If we need something installed in our container, build a custom image.

- Why?
  This puts a strong emphasis on automation and repeatability. Let's see why ...

Local development environments

- When we use local VMs (with e.g. VirtualBox or VMware), our workflow looks like this:
  - create VM from base template (Ubuntu, CentOS…)
  - install packages, set up environment
  - work on project
  - when done, shut down VM
  - next time we need to work on project, restart VM as we left it
  - if we need to tweak the environment, we do it live
- Over time, the VM configuration evolves, diverges.
- We don't have a clean, reliable, deterministic way to provision that environment.

Local development with Docker

- With Docker, the workflow looks like this:
  - create container image with our dev environment
  - run container with that image
  - work on project
  - when done, shut down container
  - next time we need to work on project, start a new container
  - if we need to tweak the environment, we create a new image
- We have a clear definition of our environment, and can share it reliably with others.

A non-interactive container

We will run a small custom container.

This container just displays the time every second.

```
$ docker run jpetazzo/clock
```

- This container will run forever.
- To stop it, press `^C`.
- Docker has automatically downloaded the image `jpetazzo/clock`.

Run a container in the background

Containers can be started in the background

```
$ docker run -d jpetazzo/clock
```

- We don't see the output of the container.
- But don't worry: Docker collects that output and logs it!
- Docker gives us the ID of the container.

List running containers

We check the list of running containers with `docker ps`

Docker will tells us:

- The (truncated) ID of our container.
- The image used to start the container.
- That our container has been running (Up) for a couple of minutes.
- Other information (COMMAND, PORTS, NAMES).

Other important Docker commands

`docker logs`

It will output the *entire* logs of the container.

`docker kill`

It stops the container immediately, by using the `KILL` signal.

`docker stop`

It sends a `TERM` signal, and after 10 seconds, if the container has not stopped, it sends KILL.

`docker rm`

It removes a container from memory.

Creating other images

docker commit

- Saves all the changes made to a container into a new layer.
- Creates a new image (effectively a copy of the container).

docker build **(used 99% of the time)**

- Performs a repeatable build sequence.
- This is the preferred method!

Images namespaces

There are three namespaces:

- Official images
  e.g. `ubuntu`, `busybox` ...

- User (and organizations) images
  e.g. `jpetazzo/clock`

- Self-hosted images
  e.g. `registry.example.com:5000/my-private/image`

Root Namespace

The root namespace is for official images.

They are gated by Docker Inc.

They are generally authored and maintained by third parties.

Those images include:

- Small, "swiss-army-knife" images like busybox.
- Distro images to be used as bases for your builds, like ubuntu, fedora...
- Ready-to-use components and services, like redis, postgresql...

Showing current images and searching for new ones

Let's look at what images are on our host now.

```
$ docker images
```

We cannot list *all* images on a remote registry, but we can search for a specific keyword:

```
$ docker search marathon
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Downloading images

There are two ways to download images.

- Explicitly, with `docker pull`.
- Implicitly, when executing `docker run` and the image is not found locally.

Images can have **tags**:

- Tags define image versions or variants.
- `docker pull ubuntu` will refer to `ubuntu:latest`.
- The `:latest` tag is generally updated often.

When to (not) use tags – Best practices

Don't specify tags:

- When doing rapid testing and prototyping.
- When experimenting.
- When you want the latest version.

Do specify tags:

- When recording a procedure into a script.
- When going to production.
- To ensure that the same version will be used everywhere.
- To ensure repeatability later.

Let's build an image interactively

The steps:

1. Create a container (with docker run) using our base distro of choice (e.g., ubuntu).
2. Run a bunch of commands to install and set up our software in the container.
   E.g.:
   ```
   apt-get update
   RUN apt-get install figlet
   ```
3. (Optionally) review changes in the container with docker diff.
4. Turn the container into a new image with docker commit.
5. (Optionally) add tags to the image with docker tag.

Let’s build a Docker image with a Dockerfile

We will build a container image automatically, with a Dockerfile.

- A Dockerfile 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.

Dockerfile for the “figlet” image

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

- FROM indicates the base image for our build.
- Each RUN line will be executed by Docker during the build.
- Our RUN commands **must be non-interactive.** (i.e., no input can be provided to Docker during the build.)
- For this reason, usually we will add the -y flag to apt-get.

The complete list of Dockerfile instructions can be found here

Let’s build the image

We can now build the image with:

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

- `-t` indicates the tag to apply to the image.
- `.` indicates the location of the *build context*. Usually, it refers to the directory where the Dockerfile is located.

The `docker history` command lists all the layers composing an image.

For each layer, it shows its creation time, size, and creation command.

When an image was built with a `Dockerfile`, each layer corresponds to a line of the `Dockerfile`.

Why sh -c ?

- When we do RUN apt-get update, the Docker builder needs to parse the command.
- Instead of implementing its own parser, it outsources the job to the shell.
- That's why we see sh –c apt-get update in that case.
- But we can also do the parsing jobs ourselves.
- This means passing RUN a list of arguments.
- This is called the *exec syntax*.

Shell syntax vs exec syntax

Dockerfile commands that execute something can have two forms:

- plain string, or *shell syntax*:
  ```
  RUN apt-get install figlet
  ```

- JSON list, or *exec syntax*:
  ```
  RUN ["apt-get", "install", "figlet"]
  ```

We are going to change our Dockerfile to see how it affects the resulting image.

When to use exec syntax and shell syntax

- shell syntax:
  - is easier to write
  - interpolates environment variables and other shell expressions
  - creates an extra process (`/bin/sh -c ...`) to parse the string
  - requires `/bin/sh` to exist in the container
- exec syntax:
  - is harder to write (and read!)
  - passes all arguments without extra processing
  - doesn't create an extra process
  - doesn't require `/bin/sh` to exist in the container

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Defining a default command

When people run our container, we want to greet them with a nice hello message, and using a custom font.

For that, we will execute:

```
figlet -f script hello
```

- `-f script` tells figlet to use a fancy font.
- `hello` is the message that we want it to display.

Adding CMD to our Dockerfile

Our new Dockerfile will look like this:

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

- `CMD` defines a default command to run when none is given.
- It can appear at any point in the file.
- Each `CMD` will replace and override the previous one.
- As a result, while you can have multiple `CMD` lines, it is useless.

Overriding CMD

If we want to get a shell into our container (instead of running figlet), we just have to specify a different program to run:

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

- We specified `bash`.
- It replaced the value of `CMD`.

Using ENTRYPOINT

We want to be able to specify a different message on the command line, while retaining figlet and some default parameters.

We will use the ENTRYPOINT verb in Dockerfile.

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

- ENTRYPOINT defines a base command (and its parameters) for the container.
- The command line arguments are appended to those parameters.
- Like CMD, ENTRYPOINT can appear anywhere, and replaces the previous value.

Using CMD and ENTRYPOINT together

What if we want to define a default message for our container?

Then we will use ENTRYPOINT and CMD together.

- ENTRYPOINT will define the base command for our container.
- CMD will define the default parameter(s) for this command.
- They *both* **have to** use JSON syntax.

Using CMD and ENTRYPOINT together

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

- ENTRYPOINT defines a base command (and its parameters) for the container.
- If we don't 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.

Overriding ENTRYPOINT

What if we want to run a shell in our container?

We cannot just do `docker run figlet bash` because that would just tell figlet to display the word "bash.”

We use the `--entrypoint` parameter:

```
docker run -it --entrypoint bash myfiglet
```

CMD and ENTRYPOINT recap

- `docker run myimage` executes `ENTRYPOINT` + `CMD`
- `docker run myimage args` executes `ENTRYPOINT` + args (overriding `CMD`)
- `docker run --entrypoint bash myimage` executes `bash` (overriding both)

Command	ENTRYPOINT	CMD	Result
`docker run figlet`	none	none	Use values from base image (`bash`)
`docker run figlet hola`	none	none	Error (executable `hola` not found)
`docker run figlet`	`figlet -f script`	none	`figlet -f script`
`docker run figlet hola`	`figlet -f script`	none	`figlet -f script hola`
`docker run figlet`	none	`figlet -f script`	`figlet -f script`
`docker run figlet hola`	none	`figlet -f script`	Error (executable `hola` not found)
`docker run figlet`	`figlet -f script`	`hello`	`figlet -f script hello`
`docker run figlet hola`	`figlet -f script`	`hello`	`figlet -f script hola`

When to use ENTRYPOINT vs CMD

`ENTRYPOINT` is great for "containerized binaries".

Example: `docker run consul --help`

(Pretend that the docker run part isn't there!)

CMD is great for images with multiple binaries.

Example: `docker run ubuntu ifconfig`

(It makes sense to indicate *which* program we want to run!)

How can we expose our application?

Docker allows us to expose the ports of our containers by automatically configure the NAT of our machine (we can check it with the iptables tool).

We can do so in two ways:

- Publishing all ports with `-P` / `--publish-all` (the list of ports to expose is defined in the Dockerfile of the image you are running)
  all containers' ports will get a random port number (>3200o) to avoid potential collisions.
  ```
  docker run -d -P nginx
  ```
- Specifying each port of the container we want to publish with the `–p` option and how to do the port mapping with the following convention: `port-on-host:port-on-container`
  ```
  docker run -d -p 80:80 nginx
  docker run -d -p 8080:80 -p 8888:80 nginx
  ```

Reducing image size

When downloading an image, all the layers must be downloaded.

Reducing image size

When downloading an image, all the layers must be downloaded.

Dockerfile instruction	Layer size	Image size
`FROM ubuntu`	Size of base image	Size of base image
`...`	…	Sum of this layer + all previous ones
`RUN apt-get install somepackage`	Size of files added (e.g. a few MB)	Sum of this layer + all previous ones
`...`	…	Sum of this layer + all previous ones
`RUN apt-get remove somepackage`	Almost zero (just metadata)	Same as previous one

Therefore, commands like `RUN rm …` do not reduce the size of the image or free up disk space.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Reducing image size

Various techniques are available to obtain smaller images:

- collapsing layers,
- adding binaries that are built outside of the Dockerfile,
- Multi-stage builds.

Collapsing layers

We can write Dockerfiles like this:

```
FROM ubuntu
RUN apt-get update && apt-get install xxx && ... && apt-get remove xxx && ...
```

or

```
FROM ubuntu
RUN apt-get update \
&& apt-get install xxx \
&& ... \
&& apt-get remove xxx \
&& ...
```

This RUN command gives us a single layer.

The files that are added, then removed in the same layer, do not grow the layer size.

Collapsing layers: pros and cons

Pros:

- works on all versions of Docker
- doesn't require extra tools

Cons:

- not very readable
- some unnecessary files might still remain if the cleanup is not thorough
- that layer is expensive (slow to build)

Building binaries outside of the Dockerfile

This results in a Dockerfile looking like this:

```
FROM ubuntu
COPY xxx /usr/local/bin
```

Of course, this implies that the file `xxx` exists in the build context.
That file has to exist before you can run `docker build`.
For instance, it can:

- exist in the code repository,
- be created by another tool (script, Makefile...),
- be created by another container image and extracted from the image.

Building binaries outside: pros and cons

Pros:

- final image can be very small

Cons:

- requires an extra build tool
- we're back in dependency hell and "works on my machine”

Cons, if binary is added to code repository:

- breaks portability across different platforms
- grows repository size a lot if the binary is updated frequently

Multi-stage builds

Multi-stage builds allow us to have multiple stages.

Each stage is a separate image, and can copy files from previous stages.

- At any point in our Dockerfile, we can add a new `FROM` line.
- This line starts a new stage of our build.
- Each stage can access the files of the previous stages with `COPY --from=....`
- When a build is tagged (with `docker build -t ...`), only the last stage is tagged.
- Previous stages are not discarded: they will be used for caching, and can be referenced.

Multi-stage builds in practice

- Each stage is numbered, starting at **0**

- We can copy a file from a previous stage by indicating its number, e.g.:

```
COPY --from=0 /file/from/first/stage /location/in/current/stage
```

- We can also name stages, and reference these names:

```
FROM golang AS builder
RUN ...
FROM alpine
COPY --from=builder /go/bin/mylittlebinary /usr/local/bin/
```

Multi-stage build example

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

Assignment

Build a Docker image containing the REST server developed in a previous assignment.

Steps:
- Write the Dockerfile of your image with all the required packages and your code;
- Build the image;
- Run the container and verify that it's working (Did you EXPOSE the correct port?).

Requirements:
- All the requirements of your Python app can be declared inside a requirements.txt file (details here).
- You can create an account on Docker Hub, login and push your image (hints here) [Optional]