Introducton to Data Engineering 7

Paul Blondel

UTSEUS, Shanghai

May 11th, 2021

'Our mission is to build tools of mass innovation.'

Solomon Hykes, Docker creator

Let's first understand what is and IS NOT Docker...

- You are already familiar with the concept of **Virtual Machine** (VM):
  - The hypervisor of a VM **simulate a whole computer**
  - It allocates resources from the host machine (memory, CPU)
  - Different VMs can have different OSes (Windows, Ubuntu GNU/Linux, etc)
- VMs make it **easy to change emulated physical devices** (memory, network, etc.)
- However: VMs are **very resource-intensive**
  - A **full OS is run** each time.
  - A **significant portion of the host resources** are taken

Docker also emulates systems but it IS NOT a VM hypervisor...

- It offers a **less-resource intensive solution** (Linux containers):
  - Docker **isolates spaces of the host machine's resources** that are allocated
  - Linux properties: Different spaces **can coexist independently on a single system**
- These spaces (also named **containers**) **partition the host system into sub-systems**
  - In each sub-system the naming (processes, users, network ports) is **purely local**
  - Example: you can have two web servers: one in a container A with a port 80, and one in a container B with a port 81

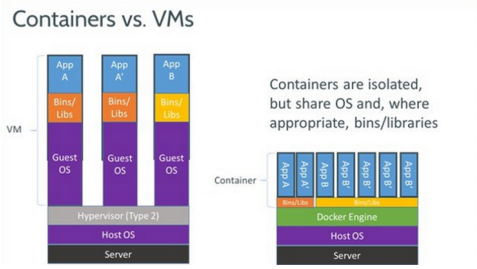

Figure: Differences between VMs and Docker containers

Thanks to Docker...

- There is **less dependency on systems** for testing applications
  - No need to worry about your host OS installation, **containers have all the necessary to run in isolation everywhere**
- Easy to **deploy applications with different versions** (one container: one version, another another version, etc.)
- **Robust to crashes** (if one application crashes th crash is contained in the container)
- **Easy to productionalize applications** (an application based on Docker will work everywhere)
- **Easy to scale** (easy to add new containers to support a bigger charge)
- **Cheaper than VM** (lighter, takes less memory, faster, etc).

The **Docker Engine** is the program allowing us to create containers and install **ready-to-use environments** (Docker images).

A little bit of vocabulary...

- The **host OS is the main operating system managing your machine** (Windows, MacOS, Linux)
- The **Docker engine is the program that manages the containers**
- A container is an **autonomous and isolated part of the host system, behaving as an independent machine**
- The **Docker client is the utility through which we transmit to the engine the commands to manage these containers**
  - It can be either from the command line (Docker CLI) or from a GUI (kitematic)

- A **Docker container needs very few resources because most system tasks are delegated to the host OS**
- Therefore: we can **multiply at low cost pseudo virtual machines** in which we could install "by hand" various software
- Fortunately, Docker offers pre-configured installations: **they are called images**
  - For example: images with pre-configured databases (Oracle, Postgresql, MySQL), Web servers (Apache, njnix), NoSQL servers (mongodb, cassandra), search engines (ElasticSearch, Solr), etc.
- When instantiating a container: **an image is placed in a container** (the same image can be placed in several containers and thus obtain a distributed system)[1]

[1] Which is very convenient for Data Engineering!

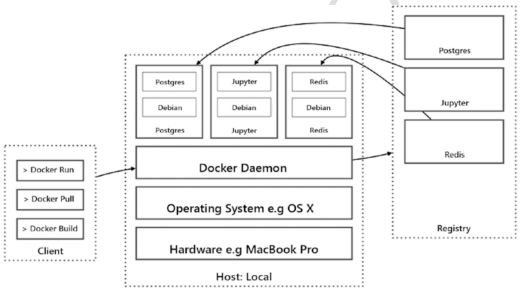

Figure: The docker ecosystem

[1]The Docker Engine is often called Docker Daemon and vice-versa

In order to understand how Docker works we have to briefly present the Docker ecosystem and how everything is related:

- The Docker client (mostly a CLI)
- The Host
- The Docker Engine (or Docker Daemon)
- Docker images
- Docker containers
- The Docker Registry
- Docker Compose (not in the diagram, will be explained later)

- The **Docker Client**: Mostly a command line interface used to **give instructions to the Docker engine** regardless of the details of the engine's implementation on your system (like a client-server architecture). The Docker client talks to the Docker engine that performs the work of containers and containerization (We will be using the Docker Client CLI)
- The **Host**: The **host is a machine on which you will run the Docker Daemon/Engine**. Locally, the host will depend upon your Docker configuration. If you are running Docker for Linux, Docker for Mac, or Docker for Windows, the host will be your system itself

- The **Docker Engine**: The Docker Engine is a persistent process managing all containers. It is running in a background service or daemon on the host. In fact, **the engine is occasionally referred to as the Docker Daemon**. The Docker Engine does the core work of Docker: **building, running, and distributing your Docker containers**. In this text, you will interact with the engine directly but will do so through the Docker Client
- The **Docker images and Docker containers**: The Docker Engine has several methods for building our own images. These include the Docker client and via a domain-specific language (DSL) known as the Dockerfile. We can also download images that other people have created. Docker containers are instances of Docker images.

- The **Docker Registry**: Docker registries hold images. These are public or private stores from which you upload or download images. For the purposes of this course, you will use the public Docker registry at Docker Hub. Docker Hub is the source of the official images of the major open source technologies you will be using including Jupyter, PostgreSQL, Redis, and MongoDB
- The **Docker Compose**: Docker Compose is a **tool for assembling applications consisting of multiple containers**. These applications are designed using a single YAML file and can be built, run, and scaled, each via a single command. Docker Compose is particularly useful for the data scientist in building standalone computational systems comprised of Jupyter and one or more databases

How to use the Docker Client to run containers?

In order to communicate with the Docker Engine, we can use a command line client program named simply docker. The simplest image is a "Hello World" image, we instantiate it with the following command:

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

The run is the command to instantiate a new image in a Docker container. Here is what you should get on the first run:

```
$ docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
4276590986f6: Pull complete
Status: Downloaded newer image for hello-world:lates

Hello from Docker!
```


The list of the available images on your system can be obtained using the following command:

```
$ docker images
```

Here is the type of output you should get now:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	4276590986f6	8 months ago	13.3kB

You can instantiate an image and give it a name using the following command:

```
$ docker run --name 'name-container' <options>
```


Let's try now to run a basic container with a minimalist Ubuntu system:

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

For more information about Docker-run:

```
$ man docker-run
```

The list of containers and their status can be obtained using:

```
$ docker ps -a
```

Example of output you could get:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
d1c2291dc9f9	mysql:latest	"docker..."	16 minutes ago	Exited (1) 9 minutes ago	3306	db
ec5215871db3	hello-world	"/hello"	19 minutes ago	Exited (0) 19 minutes ago		my test

The first field "CONTAINER ID" tells us the identifier by which we can pass instructions to the container.

Here are the most useful ones, assuming the container is d1c2291dc9f9. First of all we can stop it with the stop command:

```
$ docker stop d1c2291dc9f9
```

Stopping a container does not mean that it no longer exists, but that it is no longer active. You can restart it with the start command:

```
$ docker start id_container
```

To remove it, it is the docker rm command. To inspect the system/network configuration of a container, Docker provides the inspect command:

```
$ docker inspect id_container
```

We obtain a large JSON document. Among all the information given, the IP address of the container is particularly interesting. We get it with:

Docker provides a publishing mechanism to indicate on which port a container is listening. You simply indicate with the –publish (or -p) option how to associate a container port with a host system port. –detach (or -d) indicates that the container is launched in the background, which avoids blocking the terminal

In case your container contains a server (a http server):

```
$ docker run --name web2 --publish 81:80 --detach httpd:latest
```

Or simply:

```
$ docker run --name web2 -p 81:80  -d httpd
```

The -p option indicates that port 80 of the container is forwarded to port 81 of the host machine.

- Every Docker Image is **defined as a stack of layers** each defining **fundamental, stateless changes** to the image:
  - The **first layer might be a minimalist operating system image** (Debian, Ubuntu, etc. Docker image)
  - The **next layer is the installation of dependencies necessary for your application to run** and all the way up to the source code of your application
- The file named "**Dockerfile**" permits to **define the layers of the image and thus build the image**:
  - The file uses a domain-specific language to tell the Docker daemon how to sequentially build a Docker image
  - All the instructions contained in Dockerfile are read and executed one by one to build the image

When writing a Dockerfile one should follows the following best practices:

- **Containers should be ephemeral or stateless** in that they can be re-instantiated with a minimum of set up and configuration
- **Avoid installing unnecessary packages**
- Each container **should have only one purpose** (ex: a database, a web server, etc.)
- The **number of layers should be minimized**

About statelessness:

- An **application may run more than one container at a time**
  - All these containers **should be stateless**
- If info is to be persisted beyond the termination of the container it **should be written to a stateful backing service such as a database** (like MongoDB or Postgres) or a key-value store (like Redis)
- Indeed: We **should be able to shut down a container and remove it from our system**, then **start an identical container using the same image**, and **run command with no effect to our work**

The first thing you have to do is to **create a file named "Dockerfile"** then define in it the image you are going to use as a base, thanks to the **FROM** statement. In our case, we will use a **Ubuntu base image**[2].

```
FROM ubuntu
```

The **FROM** statement can only be used once in a Dockerfile!
You can add the name and coordinate of the maintainer of the Dockerfile with:

```
MAINTAINER Firstname Lastname (mail@mymail.com)
```

[2] To find the list of the docker images you can use as base images you can go here: *https : //hub.docker.com/*

Then you can use the **RUN** instruction to execute command lines in your container, for example updating the Ubuntu system and installing mongodb:

```
RUN apt-get update
RUN apt-get install mongodb
```

Notes

apt-get is a Ubuntu/Debian package manager, you can use it to search, install, update or remove software. The package manager your can use depend on the base image you use (for instance, of your base image is archlinux, you have to use pacman)

The **ENV** instruction permits to set or update environment variables (like PATH, etc).

Then use the **ADD** statement to copy or upload files to the image. In our case, we use it to add the sources of our local application in the /app/ folder of the image.

```
ADD . /PATH_IN_CONTAINER/
```

Then use the **WORKDIR** command to change the current directory. The command is equivalent to a cd command on the command line. All the commands that follow will be executed from the defined directory.

```
WORKDIR /PATH_IN_CONTAINER
```

Now that the source code and dependencies are present in your container, we need to tell our image some final information.

```
EXHIBIT 27017
VOLUME /PATH_IN_CONTAINER_TO_SOME_REPOSITORY
```

The **EXPOSE** statement is used to indicate the port on which your application is listening. The **VOLUME** statement is used to indicate which directory you want to share with your host.

We will conclude with the instruction that must always be present, and place it in the last line for better understanding: **CMD**. It allows our container to know which command it should execute when it starts.

```
CMD sudo systemctl start mongodb
```


Put everything together, the Dockerfile file looks like:

```
FROM ubuntu
MAINTAINER Firstname Lastname (mail@mymail.com)
RUN apt-get update
RUN apt-get install mongodb
ADD . /app/
WORKDIR /app
EXPOSE 27017
VOLUME /app/logs
CMD  sudo systemctl start mongodb
```


You can now create your first own Docker image!

Type this command in the repository containing the Dockerfile file:

```
$ docker build -t my-mongodb-image .
```

The -t argument allows you to give a name to your Docker image. This makes it easier to find your image later. The . is the directory where the Dockerfile is located. In our case, at the root of our project.

Now you can launch your container with the docker run command like this:

```
$ docker run -d -p 27017:27017 my-mongodb-image
```


A full application requires the assembly of several containers.

For example: for a dynamic website you may need a database (MySQL), a web-server (Apache) and a scripting language interpreter (PHP). In that case you need three containers: the **application is then the assembly of these three containers** together.

With Docker compose we can create a multi-container architecture that we can call "stack". This Stack will be :

- Autonomous (because it is ready to be "set up" everywhere, whatever the target platform)
- Pre-parameterized (everything is in the docker-compose.yml file)
- Isolated (all services are not necessarily publicly accessible, but are accessible by your other applications)
- Easily manageable with the CLI (docker-compose up/down/start/stop)

In order to define a multi-container Docker application we have to write a docker-compose.yml file. ".yml" is the extension of a text file formatted in YAML (formatted explicited below) :

The syntax of the YAML stream is relatively simple and efficient. It has been established to be as readable as possible by humans:

- Comments are indicated by the hash sign (#) and extend over the whole line
- List elements are denoted by a dash (-) followed by a space, with one element per line.
- Arrays are of the form "key : value", with one pair per line.
- Scalars can be surrounded by double quotes (") or single quotes (')
- etc.

First thing to do when writing a docker-compose.yml file is to **write the version of Docker Compose** you use:

```
version: '3'
```

This instructs Docker Compose that we're using version 3 of the tool. The next line will instruct Docker Compose that what follows will be **the services to deploy**. That is defined with:

```
services:
```


Let's imagine we want to build an application with a web server and a MongoDB database. Our first section to define will be the web portion of the stack. We can do this with two simple lines:

```
web:
    image: http
```

What the above does is instruct Docker Compose to deploy a container using the official http Docker image. We're not doing anything special for this container. It's just basic.

Our next section will define a database. This one gets a bit more complicated, as we must configure the necessary parameters for the database to function. We begin by defining the section with:

```
db:
     image:mongodb
```


Now we define both the external and internal ports to use for the database. For this we'll make use of the default MongoDB ports and define them:

```
ports:
     - "27017-27019:27017-27019"
```

Why do we write the port two time? The **first port is the port on the host machine** and the **second the port in the container** (we can map the default port of the server to different ports of the host machine and thus have several mongodb servers in multiple containers).

Finally, we configure the database environment. The environment will be the configuration options for the database (passwords, users, database name):

```
environment:
     -MONGO_INITDB_DATABASE=mongo_db_name
     -MONGO_INITDB_ROOT_USERNAME=mongo_db_username
     -MONGO_INITDB_ROOT_PASSWORD=mongo_db_password
```


Putting everything together gives:

```
version: '3'
services:
  web:
    image: http
    container_name :"WEB Server"
    ports :
    - "80:80"
  db:
    image: mongo
    container_name :"Mongo DB"
    ports:
      - "27017-27019:27017-27019"
    environment:
      - MONGO_INITDB_DATABASE=mongo_db_name
      - MONGO_INITDB_ROOT_USERNAME=mongo_db_username
      - MONGO_INITDB_ROOT_PASSWORD=mongo_db_password
```


Now that you have created your docker-compose.yml file you can run the application by typing:

```
$ docker-compose up
```

This must be typed inside the repository containing the docker-compose.yml[3] file. You can also run your application in detached mode using -d:

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

Later you can stop it using:

```
$ docker-compose stop
```

You can bring everything down, removing the containers entirely, with the down command:

```
$ docker-compose down
```

[3] The complete and offical documentation about Docker Compose: https://docs.docker.com/compose/

That's all for today!

Please install the following software on your own system for the next session (not your VM!):

- Docker
- Docker Compose
- For Windows users: git bash (https://gitforwindows.org/)
- You may also install: Kitematic (optional: we will not use it)