Spain

Blog Post
Author: Mario Albo

# Docker – A quick guide through docker's amazing world for devs

SWORD
UPGRADE YOUR BUSINESS

Docker is an open platform for developing, shipping, and running applications.

It was born to resolve the main problems new software projects and developers were facing, such as the inability to scale their applications and the difficulty to maintain or develop new features.

Those problems were solved before Docker was developed, but it was designed to improve those solutions and make working with applications and coding way easier.

Docker can be complex, especially if you have never worked with it.

***Before starting, it is necessary to install Docker:***

***Click this link, select your OS and follow the instructions.***

**Creating our image**

Docker delivers software in packages called images.

Those images can be found and downloaded from a repository (public like the default one, Docker Hub, or private like Nexus).

When asking Docker to use an image, it will look for it locally; if it cannot find it, Docker will search for it in the default configured repository.

When it finds it, Docker downloads the image and stores it locally.

But there is another way of getting an image, that may be more interesting rather than using pre-existing images: generate it yourself!

### Source Code

To build an image, we need the source code. We will use the following to build our image:

```
const http = require('http');

const os = require('os');

console.log("Kubia server starting...");

var handler = function(request, response) {

console.log("Received request from " + request.connection.remoteAddress);

response.writeHead(200);

response.end("You've hit " + os.hostname() + "\n");

};

var www = http.createServer(handler);

www.listen(8080);
```

apps.js

The previous program listens to the port 8080, and whenever it receives an HTTP request, it responds with the message in the body: “You’ve hit ”.

Then it logs a message on the container’s terminal “Received request from “.

### Dockerfile

The other thing necessary to build a docker image is the Dockerfile, a recipe that describes how the image is going to be built.
This is the Dockerfile used to build an image from the previous code:

```
# BASE IMAGE

FROM node:7

# BUILD INSTRUCTIONS

COPY app.js /app.js

# EXECUTABLE
```

A Dockerfile consists of 3 parts.

1. Definition of base image.

It defines which base image Docker is using (yes, every image is built on top of another image), and there are plenty of options: from base OS images like Debian6, RHEL, Ubuntu20.04, etc, to base images already modified for a more specific use, like MongoDB, Node 7, Python 3.5, etc.

Dockerfile’s directive **FROM** chooses the image, using the image’s name.

2. State of the future image.
The most common directives here are:

•**COPY**: copies a file or directory from the host’s filesystem to the image’s filesystem.
•**RUN**: executes any command it’s specified inside the image.

e. g. The command `RUN apt update -y` will update the repository index of the image, the same way executing it on your local machine would.

Keep in mind that those commands should be compatible with the OS image.

3. Starting command:

It defines the way the image will start so, typically, we place here the path to the executable or command that starts the application.

The source code and the Dockerfile are the only necessary things to generate an image.

To do so, run the following command, which tells Docker to build a new image with the desired name:

`docker build -t my-js-app ./`

This command tells Docker to build a new image, and we need to run it in the same directory where Dockerfile is.

It uses the -t flag to name it **my-js-app**. The './' (dot) defines the path where all the files necessary to create the image are.

When our image is built, we use it to launch our containerized application. There are two ways to do it:
•Through `docker run `.

•By **docker-compose**, which is a way of specifying the launch parameters and the options an image needs within a receipt.

To launch it using run, type the following command, which runs the image my-js-app:

`docker run -p8080:8080 my-js-app`

When an image is launched, it does not launch itself: it creates a container. Think about images like a class (in an object-oriented language), and about the container like an instance of that class. Multiple containers could be launched at the same time from the same image.
The flag **-p** maps the host port (left of the ':') with the container one (right of the ':').

This map allows us to access the container from the outside, through the mapped port.

Now that the application is running, it is time to test it. To do so, execute:
`curl localhost:8080`
The application should respond with a message like "You've hit ". Don't worry if the container name is composed of some random numbers, Docker assigns the hash of the container as the default name.
With docker-compose it is possible to launch docker images with the same options that docker run commands, but within code:

```
version: '3'

services:

app1:

image: my-js-app

ports:

- 8080:8080
```

docker-compose.yml

This is a simple receipt, enough to launch our application using docker images and docker-compose.

The first line defines the docker-compose version used to interpret the file. The syntax and features from one version to another vary very little, so if you are new in Docker you don't need to worry about that for now.

The **service** section specifies which images we are going to use, and how those images are launched.

In this example, we can see a defined service called **app1**, which uses the image that we built before and maps port 8080 from host to image.

There are plenty of features that Compose provides to launch the images, but we won't mention them in this post, as there is enough information about it to write a book or two.