Docker Notes

Iñaki Lakunza

May 28, 2024

Contents

1 Introduction

Docker is a platform for building, running and shipping applications in a consistent manner so that our application works in any machine.

These are some of the reasons why an application might not run in another machine:

- One or more files missing.
- Software version mismatch.
- Different configuration settings.

With Docker, we can easily package up our application with everything it needs and run it anywhere, on any machine with docker.

Moreover, we will not need to manually install all the necessary dependencies to run our application. We will be able to download and run these dependencies inside an **isolated environment called CONTAINER**.

This isolated environment allows multiple applications to use different versions of some software side by side, without messing each other.

And once we have finished with an application and will not need to use it anymore, we will be able to uninstall all its belonging dependencies in one go.

2 Virtual Machines vs Containers

A **Container** is an isolated environment for running an application.

A **Virtual Machine** is an abstraction of a machine (physical hardware).

In our real machine we can have different virtual machines at the same time, for instance one running Windows and the other one running Linux.

We are able to do this using a tool called Hypervisor, which is a tool to create and manage virtual machines. Some of the most common hypervisors are VirtualBox, VMware and Hyper-v (only Windows).

The benefit of building virtual machines is that we can run an application in isolation inside a virtual machine. So, on the same physical machine we can have different virtual machines each running a completely different application.

But, it comes with some problems: Each virtual machine needs a full-blown operating system. So, this means that virtual machines are slow to start, since the entire operating system has to be loaded just like starting our computer.

And virtual machines are resource intensive because each takes a slice of the actual physical hardware resources, like cpu, memory and disk space.

On the other hand, **Containers** give us the same kind of isolation so we can run multiple applications.

And they are more lightweight, they do not need a full operating system. In fact, all containers on a single machine share the operating system of the host.

So that means that we need to license, patch and monitor a single operating system.

Moreover, since the operating system has already started on the host, a container can start up pretty quickly.

And also these containers do not need a slice of the hardware resources on the host, so we do nt need to give them a specific number of cpu cores or a slice of memory or disk space. So on a single host we can run tens or even hundreds of containers side by side.

3 Docker Architecture

Docker uses a **client-server architecture**.

So it has a client component that talks to a server component using a restful API.

The server, also called the docker engine, sits on the background and cares about building and running docker containers.

Technically, a container is just a process like other processes running on our computer, but it is a special kind of process.

Unlike virtual machines, containers do not contain a full-blown operating system. Instead, all containers on a host share the operating system of the host.

More accurately, all containers share the **KERNEL** of the host. The kernel is the core of an operating system, it is like the engine of a car, it is the part that manages all applications and hardware resources.

Every operating system has its own kernel or engine and these kernels have different APIs, so that is why we can not run a windows application on linux. Because under the hood, this application needs to talk to the kernel of the underlying operating system.

So, on a Linux machine we can only run Linux containers, because these containers need Linux.

On a Windows machine, however, we can run both Windows and Linux containers because Windows 10 is shipper with a custom built Linux kernel. This is in addition to the Windows kernel that has always been in Windows.

On the other hand, MacOS has its own operating system, which is different from Linux and Windows kernels, and this kernel does not have native support for continuous applications. So, Docker on Mac uses a lightweight Linux virtual machine to run Linux containers.

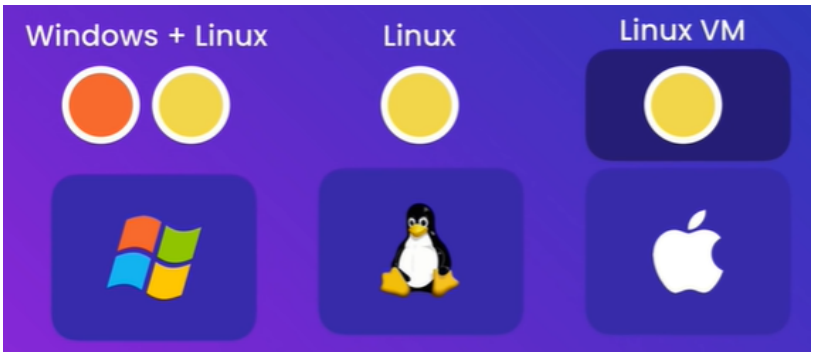

Figure 1:

4 Development Workflow

To start off, we take an application (it does not matter which kind of application it is or how it is built), and **dockerize** it. Which means we make a small change so that it can be run by docker.

This way we will get a docker file, which is a plain text file that includes instructions that docker uses to package up this application into an ***image***, which contains all the information our application needs to run, EVERYTHING.

The **image** contains the following information:

- A cut-down Operating System
- A runtime environment
- Application files
- Third-party libraries
- Environment variables

Once we have an image, we tell docker to start a container using that image, so, a container is just a process, but it is a special kind of process because it has its own file system, which is provided by the image.

So, **instead of directly launching the application and running it inside a typical process, we tell Docker to run it inside a container, an isolated environment**.

Once we have the image, we can push it to a docker registry like **docker hub** (same as Git and GitHub). And then we can put this image on any machine running docker.

This new machine has the same image we have on our development machine, which contains a specific version of our application with everything it needs, so we can start it on our development machine just by telling docker to start a container using the image.

So, with Docker we do not have to maintain long complex release documents that have to be precisely followed. All the instructions for building an image of an application are written in a docker file and with that we can package up our application into an image, and run it virtually anywhere.

5 Docker in Action

The first step will usually be to open the terminal and create a directory and open it in Visual Studio Code:

```
mkdir hello_docker
cd hello_docker
code .
```

In this directory we can add any files that we want.

And, if we would like to run it in another machine without using Docker, we would have to start the operating system, install all the needed applications, copy app files and run them.

But, with Docker we can just write these instructions inside a Docker file and let Docker package up our application.

To do this, we have to add another file to our project, called **Dockerfile**, with that exact name and without any extensions. And we will write there the necessary instructions for packaging our applications.

Typically we start from a **base image**. This base image has a bunch of files, and we are going to take those files and add additional files to it, kind of like inheritance in programming.

Lots of base images are published in DockerHub, and we will find lots of official images.

The images are built on top of different distributions, which are used for different purposes.

So, first we will kind of import our base image and then we are going to copy all the files in the current directory, into the app directory.

Then we are going to use the command instruction to execute the commands. Here we have to remember that the file is inside the app directory, so we have to prefix it with the directory name.

Alternatively, we can set the current working directory using the *WORKDIR* command, and so the below instructions will assume that we are currently inside the specified directory.

So, this instructions clearly document our deployment process.

```
FROM python:3
COPY . /app
WORKDIR /app
CMD python3 test.py
```

Now, we have to go to the terminal and tell docker to package our application. For that, we have to give our image a **tag**, a tag to be identified, we do this by using *-t*. And then we have to specify where Docker can find a Docker file, so, if we are currently inside our app directory and our Dockerfile is right there, we can just use a . to reference the current directory:

```
docker build —t hello_docker .
```

We will see that any file will not be added to our project, this is because the image is not stored here, and in fact an image is not a single file, how Docker stores it is very complex, but we do not

have to worry about it.

If we want to see the images stored in our computer, we can go to the terminal and type one of the two following options:

```
docker image ls
docker images
```

Each image has its own unique identifier, and we can see when the image was created and its size. We can see that the image is quite big since it contains all what is needed to run it.

Now we can run our image in any computer with Docker. For that, we just have to type the image tag, and it does not matter in which directory we are in, since the image contains all the necessary information to run our application:

```
docker run hello_docker
```

We can then publish this image in DockerHub so anyone can use it, so then we can go to another machine and just pull and run the image.

To do pull the image we just have to specify its path, for instance:

```
docker pull codewithmosh/hello—docker
```

Then we can just find it using *docker images* and just run it using its tag, or we can just run it using the path we used for downloading it, that easy.

So we can just dockerize any application.

6 Linux Command Line

Docker has its foundations built on Linux concepts, so it is easier for solving issues if we work and understand Linux. Moreover, most online tutorials use Linux.

6.1 Linux Distributions

Linux is open source, and people have created different implementations and adaptations called distributions, such as Ubuntu, Debian, Alpine, ...

This distributions support pretty much the same set of command, but sometimes there might be small differences.

We will work with **Ubuntu**.

6.2 Running Linux

Instead of using the following line to pull the base image:

```
docker pull ubuntu
```

We will use a shortcut:

```
docker run ubuntu
```

If we have this image locally, docker is going to start a container with this image, otherwise it is going to pull this image behind the scene and then start a container.

When we use it we will see how it will stop. The reason is that since we did not interact with the container it stopped.

We can use the following command to see the running processes, and so we will see that it is not running:

```
docker ps
```

And if we use the following line we will see the images we used and when, so we can see we used the Ubuntu image:

```
docker ps -a
```

In order to use the Ubuntu image we have to run it **interactively**, for this purpose we use the *-it* line:

```
docker run -it ubuntu
```

Now a shell will appear. A shell is a program which takes our commands and passes them to the operating system for execution.

The shell will have a combination of letters and numbers. The first part represents the currently logged in user, so by default we may be logged in as the *root* user, which has the highest privileges.

Then we will have a sign and afterwards the name of the machine will appear.

Then we will have : and afterwards we will see where we are in the file system: if we have an /, it means that we are in the root directory.

And, if we afterwards have a # it will mean that we have the highest privileges (this will happen if we are logged as the root user).

If instead we would be logged as a normal user then we would see a dollar sign ($).

Here we can work normally, executing normal linux commands, like the following:

```
echo hello
whoami
echo $0  (see location in the shell program)
```

We have to remember that in Linux we use a forward slash (/) and a backslash in Windows (\) to separate files and directories.

The other difference is that Linux is a case sensitive operating system.

6.3 Managing Packages

Some of the most well known package managers are npm, yarn, pip, ...

In Ubuntu there is another package manager called **apt** (Advanced Package Tool).

If we execute it, we will see that this command has a bunch of sub-commands, such as list, search, show, install, ...

apt-get is a very used package manager too.

If we want to install *nano*, if we use the following line we will get an error:

```
apt install nano
```

The error will happen because in Linux we have a package database, which contains lots of packages, but not all these packages are installed.

We can see the packages using the following line:

```
apt list
```

We will see that some of them are installed, but not all of them.

And, when we use the *apt install nano* line, this command looks at the package database and it can't find a package called *nano*.

So, we have to update the package database:

```
apt update
```

And so now we can install *nano*.

So, the most important thing to remember is that before installing any package, we should always run *apt update* to update our package database, and then install the package.

So now we can just type *nano* and use the text editor normally.

If we want to clear the terminal we can just press ctrl+l.

Now, if we want to uninstall a package, we just have to run the following line:

```
apt remove nano
```

6.4 Linux File System

In Linux, just like in Windows, our files and directories are organized in a tree, in a hierarchical structure.

We will have the root directory in top of our hierarchy (/), and below that we have a bunch of standard directories, such as bin (binaries and programs), boot (files related to booting), dev (devices), ...

So, **in Linux everything is a file**, including devices, directories, network socket pipes, ...

We also have etc (etidable text configuration), home (home directories for users are stored here), root (the home directory of the root user, only the root user can access this directory), lib (keeping library files like software library dependencies), var (variables, for files that are updated frequently), proc (running processes).

6.5 Navigating the File System

- **pwd**: Print working directory, to see where we are in the file system
- **ls**: List, for listing the files in the current directory.
- **ls -l**: Here we will see all the files with all the information: who owns it, the size, the creation date, ...
- **cd**: Change directory, for moving through the directories. If we want to go to our home directory we can just write *cd*

6.6 Manipulating Files and Directories

In order to create a directory we can just use **mkdir** followed by the name we want to give it.

We can also rename directories, for that we have to use the move command: **mv**. With it, we can move our files or directories or we can rename them. For instance:

```
mv test docker
```

For creating a file we can use **touch**, for instance:

```
touch hello.txt
```

So now we have a new empty file.

And we can create a bunch of files in one go, just by writting different names after *touch*.

If we want to remove one or more files we can use the **rm** command, followed by the name or names.

Or we can also use a pattern: if we want to remove, for instance, all the files that start with *file* (if we have file1.txt, file2.txt, file3.txt, ...), we can simply use *rm file**

If we want to remove our directory we have to use **rm -r**, the -r represents *recursive*, since we want to remove this directory and all of its content recursively.

6.7 Editing and Viewing Files

nano is a basic text editor for Linux.

If we are using the docker image we downloaded, we do not have nano there, so as we have seen, we have to install it.

To launch it we can just type *nano*, or we can give it a filename, so to start editing it:

```
nano file1.txt
```

To see the content of this file we have a bunch of options.

We can use **cat** (concatenate), we can use it to concatenate different files, but we can also use it for seeing the content of a file.

But, if our file is quite large, it is better to use **more**. If we use it this way we will see the file page after page, below it will appear how much we have seen, and we can move to the next page using the space. Or we can use the enter to scroll one line at a time.

The only problem with **more** is that we just can scroll down, not up.

So to scroll up we have to use a different command called **less**. With it we can just press the up and down arrows to move through the file.

6.8 Redirection

The standard input represents the keyboard and the standard output represents the screen. But we can always change the source of the input or the output. This is called **redirection**.

But we can use the **redirection operator (>)** to redirect the input or the output.

For instance, we can write

```
cat file1.txt > file2.txt
```

So the content of file1.txt will be written in file2.txt