Introduction to Docker

What is Docker?

Docker is a platform for building, running and shipping applications in a consistent manner. With Docker, if the application works in your machine, it will work in production. Without Docker, this might not happen for three different reasons:

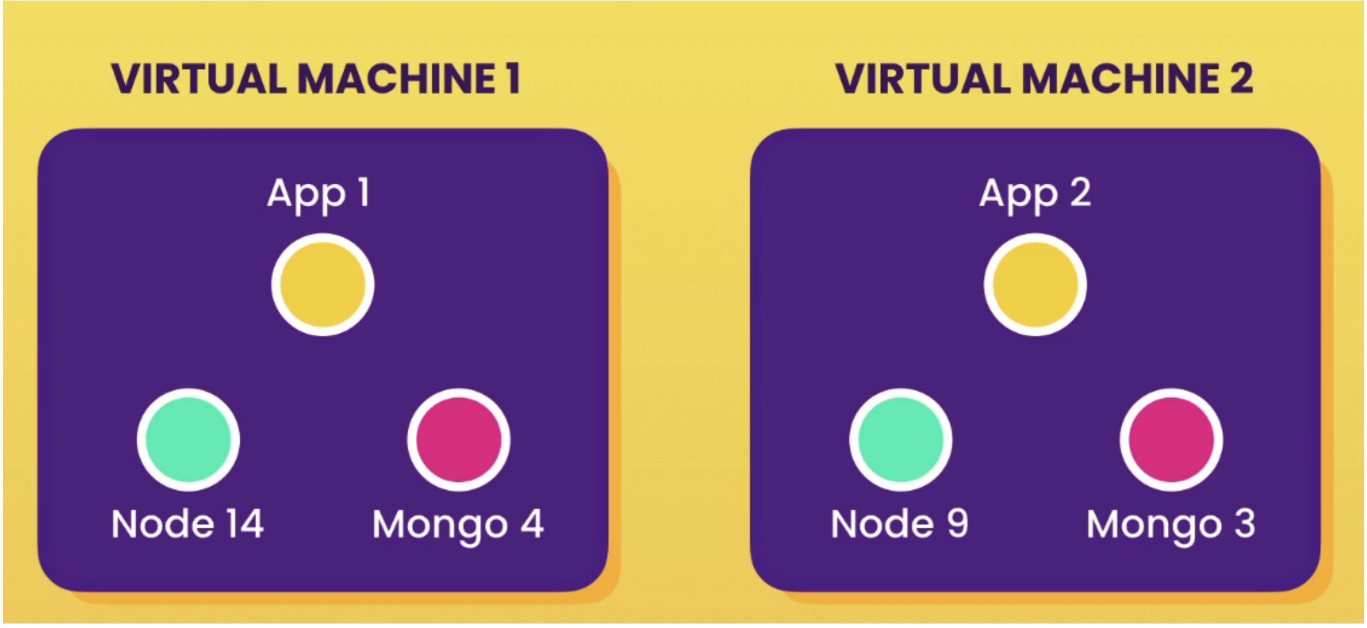
1. files are missing in the deployed version (the application is not fully deployed)
2. software versions mismatch
3. different configuration in different machines (like env variables)

What Docker does is to package the application with everything it needs to run. Therefore, if it works on the development machine, it will work on testing and production machines as well. Additionally, any new comers will not have to go through a complex local setup, but just use that Docker container.

You can run multiple Docker containers (for example, multiple applications) in you machine. Each Docker container will generate its needed environment and avoid clashes.

Virtual Machines Vs Containers

A VM is an abstraction of a machine (from its physical hardware). We can run multiple VMs in a single physical machines using a special kind of program called a *hypervisor*. There are multiple hypervisors in the market: like VirtualBox, or VMware. Each VM can run its own software (even the OS).



Problems with VMs are that:

- each VM needs its own OS
- slow to start
- resource intensive (each VM takes up a slice of the hardware)

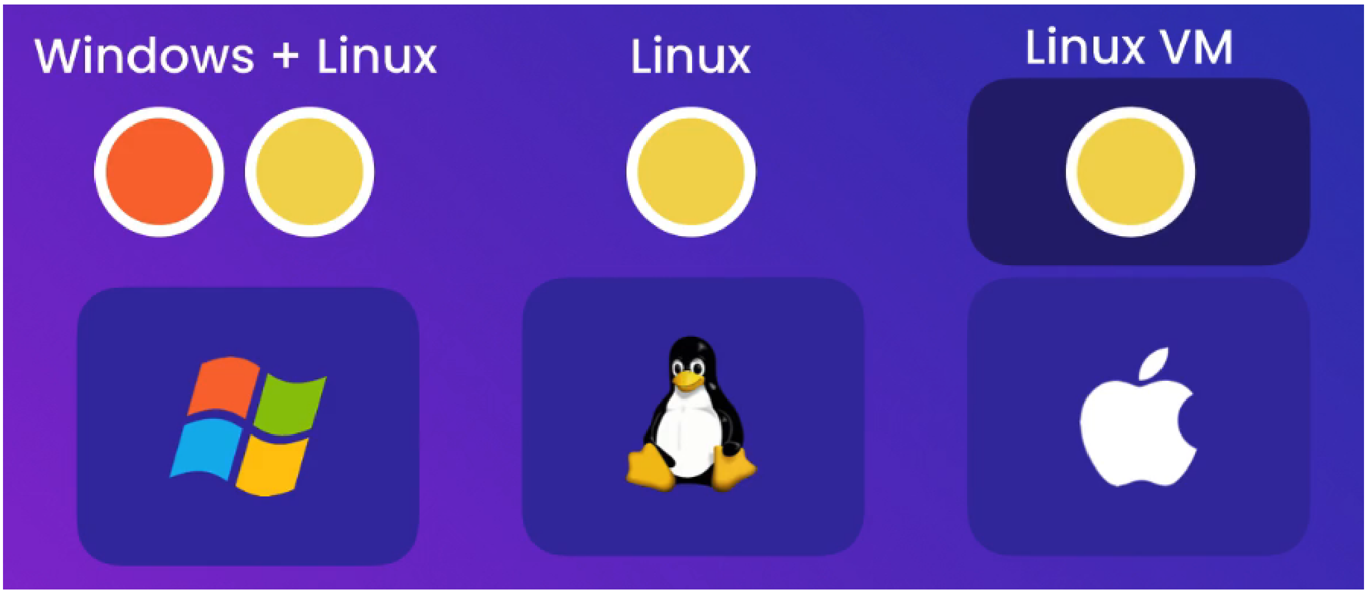
A container is an isolated environment for running an application, just as a VM. Differences are that:

- they all share the OS of the host and are therefore, more lightweight.
- containers start up very quickly.
- they use up less hardware resources.

Docker architecture

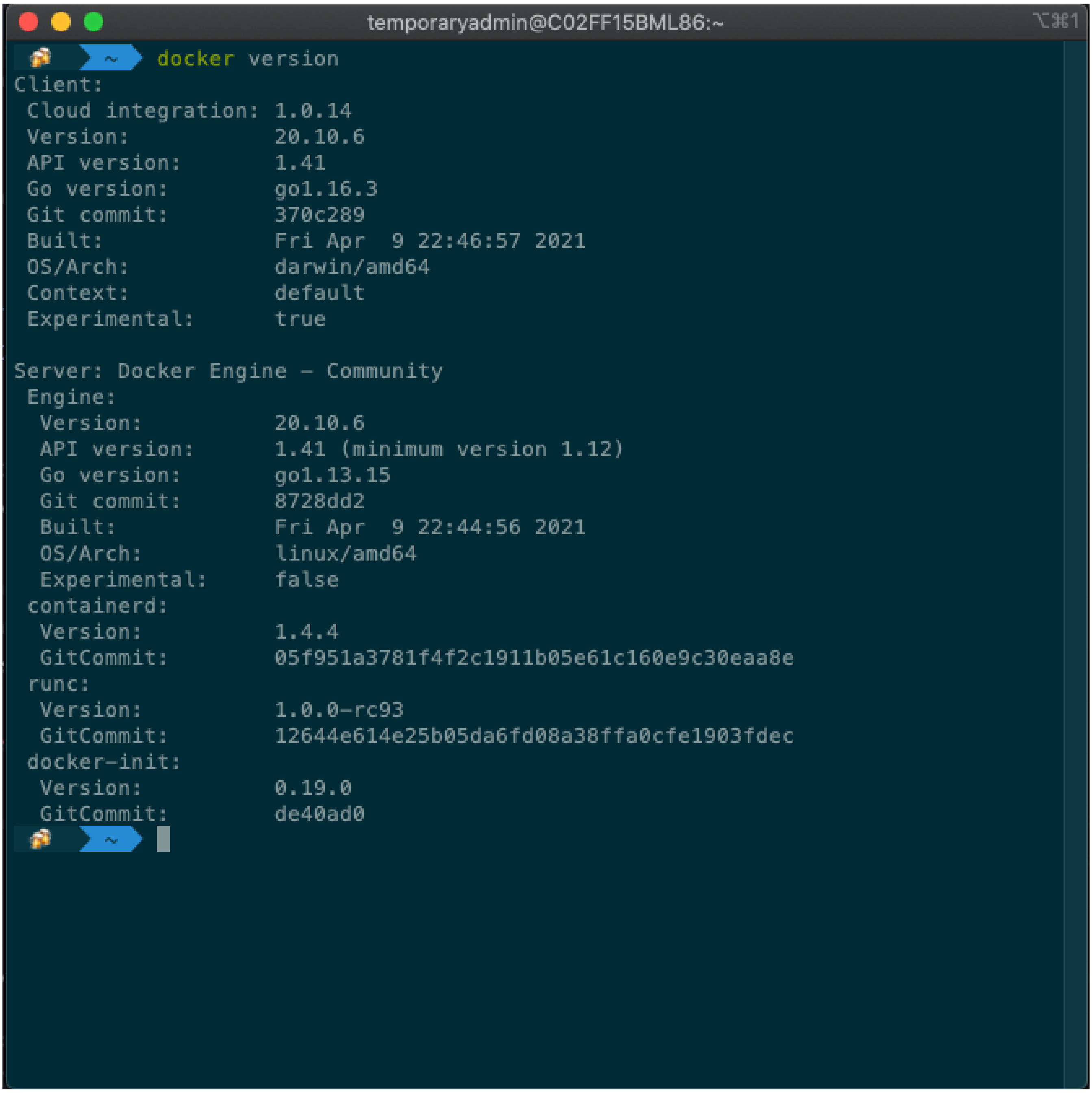
Docker uses a client-server architecture which communicate using a REST API. The server is called the **Docker Engine**. It sits on the background and takes care of building and running containers. Technically, a container is a process like any other one running in your computer.

Containers don't share the complete OS of the host, but the kernel of the OS. The kernel is the part of the OS that manages the applications and the allocation of hardware resources. Each operating system has a different kernel and can therefore only run different containers.



Starting from Windows 10, Windows is shipped with both a Windows kernel and a Linux kernel. Therefore, Windows machines can run both Windows and Linux containers. Each kernel will handle the respective ones. MacOS has a special kernel. It does not have native support for containers. Because of this, containers in MacOS need to run inside a Linux VM.

Once you've installed Docker in your machine, start it and run `docker version` on the terminal.



```
 ~  docker version
Client:
 Cloud integration: 1.0.14
 Version:           20.10.6
 API version:       1.41
 Go version:        go1.16.3
 Git commit:        370c289
 Built:             Fri Apr  9 22:46:57 2021
 OS/Arch:           darwin/amd64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.6
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       8728dd2
  Built:            Fri Apr  9 22:44:56 2021
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.4.4
  GitCommit:        05f951a3781f4f2c1911b05e61c160e9c30eaa8e
 runc:
  Version:          1.0.0-rc93
  GitCommit:        12644e614e25b05da6fd08a38ffa0cfe1903fdec
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
 ~ 
```

Docker development workflow

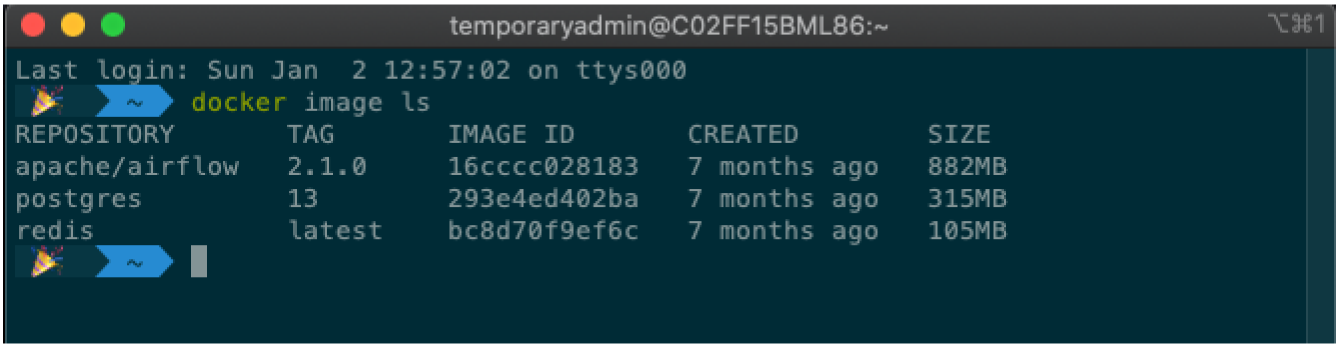
Add a `Dockerfile` to the application. We can call this *dockerizing the application*.

The `Dockerfile` is a plain text file that Docker uses to package the application into an **Image**. This image contains everything that the application needs to run. This includes a cut-down OS, a runtime environment (like Node or Python), the application files, third-party libraries, environment variables, etc. Once we have an image, we give it to Docker so that the image can be run in a Container.

Once we have that image, we can push it to a Docker registry (like DockerHub). From there we can pull it to any machine running Docker. This machine can be a testing or production server for example.

You can list the images in your computer by running

```
docker image ls
```



```
Last login: Sun Jan  2 12:57:02 on ttys000
 ~  docker image ls
REPOSITORY        TAG       IMAGE ID       CREATED        SIZE
apache/airflow    2.1.0     16cccc028183   7 months ago   882MB
postgres          13        293e4ed402ba   7 months ago   315MB
redis             latest    bc8d70f9ef6c   7 months ago   105MB
 ~
```

A basic Docker file will look something like this:

```
FROM node:alpine
COPY . /app
WORKDIR /app
CMD node app.js
```

These are just the instructions on how to build and run the image. The `FROM` keyword establishes the base image from where to start. In this case, we are setting the base image to have a `node` runtime in a Linux `alpine` distribution. The names of these base images come from the DockerHub registry. DockerHub is to Docker, what GitHub is to Git.

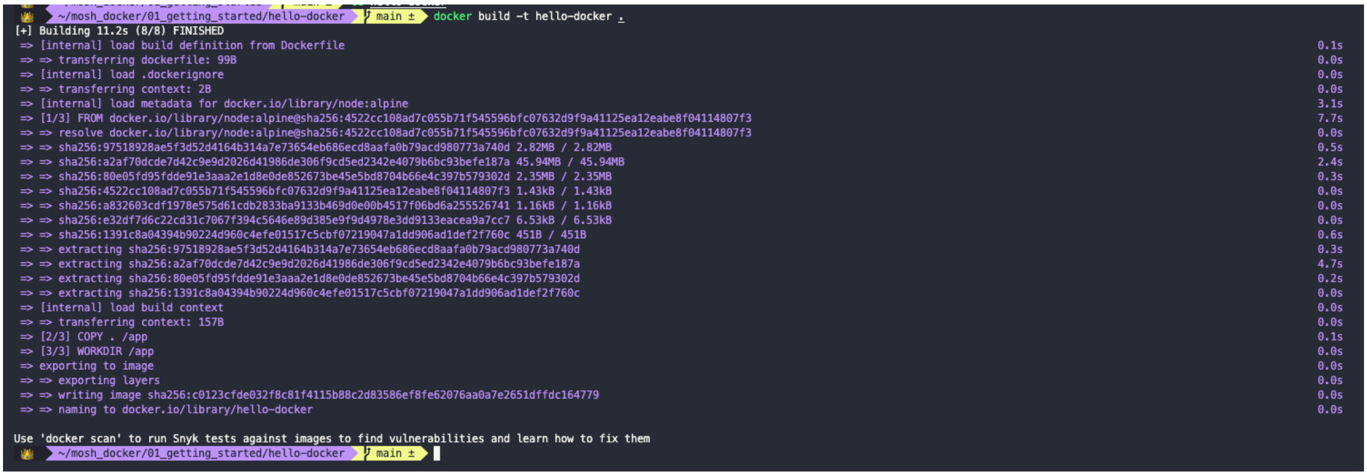
The `COPY` step is telling Docker that once the base image is pulled, it needs to copy into the container all the application files. We could list directories or even files, but here we are telling it to copy all by passing `.` as the first argument. All these files need to be copied into the `/app` directory in the image. Docker will take care of creating this directory.

Next we set the working directory with the `WORKDIR` command. In this case, we are setting it to be the `/app` directory that we created previously.

Lastly, we need to specify which commands to run using the `CMD` instruction. In this case, we are telling it to run `node app.js` so that it starts our application.

Now that we have a Dockerfile, we need to tell Docker to build the image. To do that, we use the `docker build` command. We need to give our image a tag using the `-t` flag. Finally we need to tell Docker where it can find the Dockerfile. If we are running the command from our application directory and the Dockerfile is in the root, we can just pass a `.` to the command.

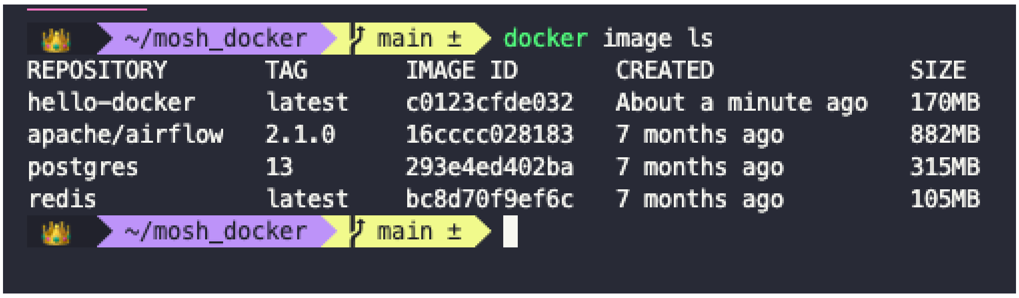
```
docker build -t <image_name> .
```



```
👑 ~/mosh_docker/01_getting_started/hello-docker  main ± docker build -t hello-docker .
[+] Building 11.2s (8/8) FINISHED
 => [internal] load build definition from Dockerfile                                                                   0.1s
 => => transferring dockerfile: 99B                                                                                    0.0s
 => [internal] load .dockerignore                                                                                      0.0s
 => => transferring context: 2B                                                                                        0.0s
 => [internal] load metadata for docker.io/library/node:alpine                                                         3.1s
 => [1/3] FROM docker.io/library/node:alpine@sha256:4522cc108ad7c055b71f545596bfc07632d9f9a41125ea12eabe8f04114807f3   7.7s
 => => resolve docker.io/library/node:alpine@sha256:4522cc108ad7c055b71f545596bfc07632d9f9a41125ea12eabe8f04114807f3   0.0s
 => => sha256:97518928ae5f3d52d4164b314a7e73654eb686ecd8aafa0b79acd980773a740d 2.82MB / 2.82MB                         0.5s
 => => sha256:a2af70dcde7d42c9e9d2026d41986de306f9cd5ed2342e4079b6bc93befe187a 45.94MB / 45.94MB                       2.4s
 => => sha256:80e05fd95fdde91e3aaa2e1d8e0de852673be45e5bd8704b66e4c397b579302d 2.35MB / 2.35MB                         0.3s
 => => sha256:4522cc108ad7c055b71f545596bfc07632d9f9a41125ea12eabe8f04114807f3 1.43kB / 1.43kB                         0.0s
 => => sha256:a832603cdf1978e575d61cdb2833ba9133b469d0e00b4517f06bd6a255526741 1.16kB / 1.16kB                         0.0s
 => => sha256:e32df7d6c22cd31c7067f394c5646e89d385e9f9d4978e3dd9133eacea9a7cc7 6.53kB / 6.53kB                         0.0s
 => => sha256:1391c8a04394b90224d960c4efe01517c5cbf07219047a1dd906ad1def2f760c 451B / 451B                             0.6s
 => => extracting sha256:97518928ae5f3d52d4164b314a7e73654eb686ecd8aafa0b79acd980773a740d                              0.3s
 => => extracting sha256:a2af70dcde7d42c9e9d2026d41986de306f9cd5ed2342e4079b6bc93befe187a                              4.7s
 => => extracting sha256:80e05fd95fdde91e3aaa2e1d8e0de852673be45e5bd8704b66e4c397b579302d                              0.2s
 => => extracting sha256:1391c8a04394b90224d960c4efe01517c5cbf07219047a1dd906ad1def2f760c                              0.0s
 => [internal] load build context                                                                                      0.0s
 => => transferring context: 157B                                                                                      0.0s
 => [2/3] COPY . /app                                                                                                  0.1s
 => [3/3] WORKDIR /app                                                                                                 0.0s
 => exporting to image                                                                                                 0.0s
 => => exporting layers                                                                                                0.0s
 => => writing image sha256:c0123cfde032f8c81f4115b88c2d83586ef8fe62076aa0a7e2651dffdc164779                           0.0s
 => => naming to docker.io/library/hello-docker                                                                        0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
👑 ~/mosh_docker/01_getting_started/hello-docker  main ±
```

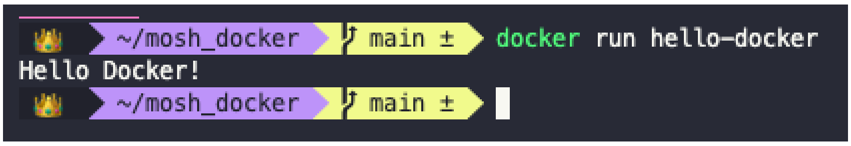
If we now re-run `docker image ls` we'll see our new image there.



```
👑 ~/mosh_docker  main ± docker image ls
REPOSITORY       TAG       IMAGE ID       CREATED              SIZE
hello-docker     latest    c0123cfde032   About a minute ago   170MB
apache/airflow   2.1.0     16cccc028183   7 months ago         882MB
postgres         13        293e4ed402ba   7 months ago         315MB
redis            latest    bc8d70f9ef6c   7 months ago         105MB
👑 ~/mosh_docker  main ±
```

Now we can run the application from any directory in our computer with the `docker run` command plus the image name.

```
docker run <image_name>
```



```
👑 ~/mosh_docker  main ± docker run hello-docker
Hello Docker!
👑 ~/mosh_docker  main ±
```

If we publish our images to a registry (like DockerHub) we can then pull them from other computers by running

```
docker pull author/image-name
```