> Data Science Technology and Systems PG
>
> Week 7
>
> Ibrahim Radwan
>
> 14/9/2022

What is Docker?

Imagine the following scenario, you have created an application on device A, and you want to run it on device B. When you have tried that, you found that the application does not work on device B and this is due to the need for some dependencies, which are not installed on device B. The application's dependencies could be, for example, NumPy, Pandas and Scikit-Learn libraries or different version of Python or even a different operating system that is used in device A and not on B.

To solve this issue, we should make sure that all dependencies of the application are installed on the other machine and make the configuration and installation according to each operating system. However, to do that for multiple applications and on different machines, it's quite a complex process which requires time and effort.

Fortunately, there is a solution for this complexity that makes the process much easier. This is called *Containerization*. The main idea behind this concept is *packaging* your application with all software and dependencies it needs, and deploy this package on any other machine without a need for extra configuration.

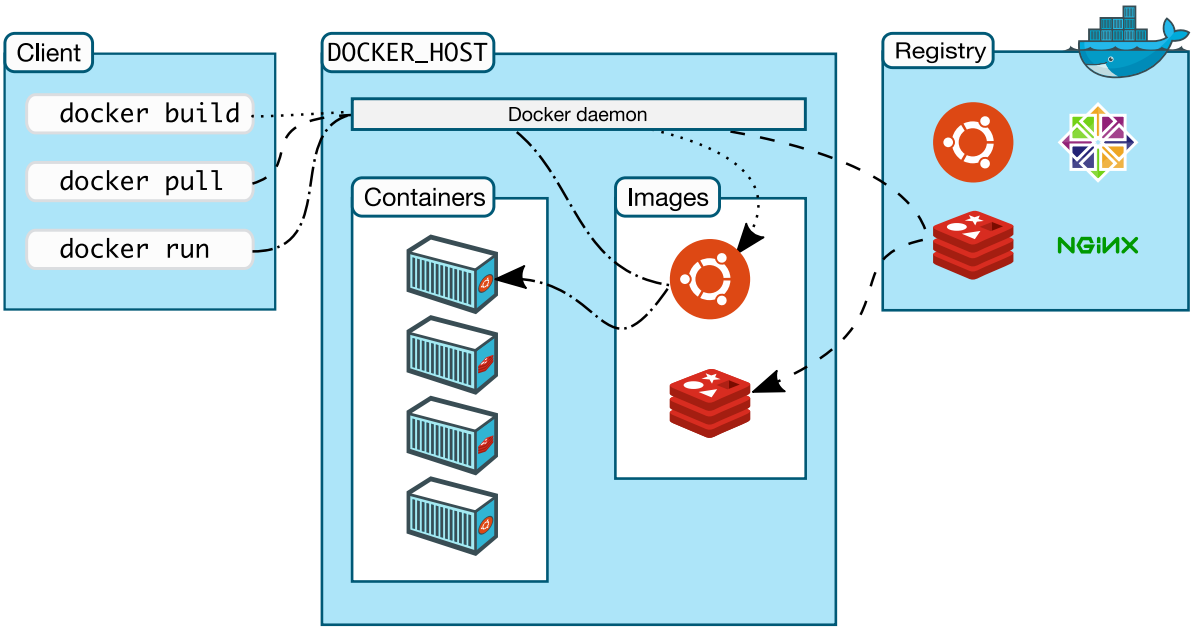
Docker is a platform that uses the containerization technology to allow developers to easily build applications, deploy them into containers which can then be run any where with different environments.

Docker Image is the executable package that contains all the dependencies and configuration, which you need to run your application in the host machine. In the above mentioned scenario, we needed to package the application with all dependencies into one package to get it working on machine B; this is the docker image.

Docker Container is a running instance of the docker image, you can think of image as python class and the container as an instance of this class, you can create as many instance of the image as you need with different names and different configurations.

Docker Architecture

Docker consists of multiple components, we will focus on the following three ones: Docker client, and Docker server, Docker Hub (Registry).



[Credit: https://docs.docker.com/get-started/overview/]

Docker Client is a command line interface (CLI), which you can use to interact with the Docker platform using commands. It's like Linux terminal that you use to interact with the underneath Linux kernels. Commands such as running an instance of the docker instance (container), delete it, pull and push an image.

Docker Server this is the docker engine, which listens for Docker's requests and manages the Docker objects such as images, containers, networks, and volumes.

Docker Hub Simply it's the repository for docker images on the cloud which allows you to download Docker images that are build by others. Anyone can create an image for his application and add it to this repository (or registry). Then, it can be publicly available and be downloaded by other people or can be privately shared with specific people only. There are many repositories for docker images such as Google Container Registry, AWS Container Registry, ... etc.

Docker in Practice

You can install Docker Desktop on your operating system. After downloading and installing you can use the terminal or your CLI to execute the Docker commands.

To check that the Docker is installed on your machine, you can open the terminal and execute `docker --version` to check the docker version. If it is installed, you will see a result similar to the following:

```
Docker version 20.10.7, build f0df350
```

Docker Hub

The Docker hub is a public repository which can be used to download Docker images from or to host your built Docker image into. To be able to do that, you need to create an account on the Docker Hub.

After creating an account, you need to link your machine to the Docker Hub using `docker login [OPTIONS] [SERVER]`, where the server is the registry or the repository that you will connect to, by default it is Docker Hub.

```
docker login -u your_user_name -p your_pass_word

Login Succeeded
```

Pulling Docker images

Once you have installed the Docker on your machine, Let's try pull a public Docker image and run it:

```
docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:7d91b69e04a9029b99f3585aaaccae2baa80bcf318f4a5d2165a9898cd2dc0a1
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

This above command tells the docker engine to go and search in the public registries for the Docker image called "hello-world". Also, it automatically checks if that image already exists in the local machine.

To check all of the already existing images on your local machine, you can try:

```
docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    d1165f221234   5 months ago   13.3kB
```

It will return the docker repository with the image name, the image tag which usually represent the version of that image (we can get an image for a specific version but if the version is not specified, by default the Docker returns the latest image), Image Id is which is a unique Id assigned for each image, and lastly we find when this image had created and its size.

Let's get another image:

```
docker pull tamermohamed/ml-microservice-kubernetes:version1.0
```

Now let us try again to check the existing Docker images:

```
docker images
REPOSITORY                                  TAG          IMAGE ID       CREATED
    SIZE
hello-world                                 latest       d1165f221234   5 months
ago    13.3kB
tamermohamed/ml-microservice-kubernetes   version1.0   40381856e6a4   15 months
ago   1.25GB
```

Running Docker images

We have images but until now none of them is running. To run any of them, we need to make an instance (*i.e.*, a Docker container) of the pulled image.

```
docker run [Image ID]
```

```
docker run  40381856e6a4

* Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 215-770-978
```

You notice that the application is running on port 80. Open the browser and copy this link in http://localhost:80. If the application works, then everything is fine, if not do not worry, we will fix it. First let's quit the terminal's process using CTRL+C (sometimes, you need to stop it from the Docker desktop), and look at the container that have been created.

```
docker ps
```

It returns all running instances (containers), but we quit the process and there is no containers running. to get all of the containers available either running or not, we use:

```
docker ps -a

CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS
                  PORTS     NAMES
71e038ea0c47   40381856e6a4   "python3 app.py"         22 minutes ago   Exited
(0) 6 minutes ago            funny_chatelet
```

You will notice the status of the container is Exited. To run the container and continue working on the terminal, even if we close the terminal, and to determine the port that we need to run on instead of the default port (80), we use:

```
docker run -p 5000:80 -d 40381856e6a4
```

This command creates and runs the container, de-attaches the process `-d`, and lets the terminal free for next commands. Also, `-p 5000:80`, means that when navigating to port 5000 on our local machine, the output of the running the container forwarded from port 80 to 5000. Now try to open the browser again http://localhost:5000 to see the result of running this container.

If you tried to run the same command again on the same port, it will fail as the port is already allocated.

```
docker run -p 5000:80 -d 40381856e6a4

82f4523b2751e7b34befe6e5a4d130939f9c45dc519089155e56578182d0d029
docker: Error response from daemon: driver failed programming external
connectivity on endpoint gallant_jang
(cd4da499bf9ae8f6559cb1ae85abae05e53e0f89d4e8989a934595de8e6ca967): Bind for
0.0.0.0:5000 failed: port is already allocated.
```

You can use another port for another instance/container of the same image:

```
docker run -p 5001:80 -d 40381856e6a4
```

Imagine if we did the same above process manually, you will need to install packages and dependencies on your machine, and if you want to run multiple instances of the application you will configure the machine multiple times to handle this. This also will be more difficult to do on multiple machines with different operating systems and with different versions of the packages. Docker solves all this problems and make it more easy.

Let's look at the running containers:

```
docker ps
CONTAINER ID   IMAGE          COMMAND            CREATED          STATUS
PORTS                                    NAMES
69a022934e3b   40381856e6a4   "python3 app.py"   9 minutes ago    Up 9 minutes
0.0.0.0:5001->80/tcp, :::5001->80/tcp    recursing_tu
2a12bf79b735   40381856e6a4   "python3 app.py"   17 minutes ago   Up 17 minutes
0.0.0.0:5000->80/tcp, :::5000->80/tcp    pedantic_pascal
```

You can give a name to a specific container as an option when we creating it:

```
docker run -p 5003:80 -d --name PredictionApp 40381856e6a4
```

Again showing the running instances:

```
docker ps
CONTAINER ID   IMAGE          COMMAND            CREATED          STATUS
PORTS                                    NAMES
7249ddb172c9   40381856e6a4   "python3 app.py"   11 minutes ago   Up 11 minutes
0.0.0.0:5003->80/tcp, :::5003->80/tcp   PredictionApp
2fe2790fc5fb   40381856e6a4   "python3 app.py"   26 minutes ago   Up 26 minutes
0.0.0.0:5002->80/tcp, :::5002->80/tcp   affectionate_bohr
69a022934e3b   40381856e6a4   "python3 app.py"   27 minutes ago   Up 27 minutes
0.0.0.0:5001->80/tcp, :::5001->80/tcp   recursing_tu
2a12bf79b735   40381856e6a4   "python3 app.py"   35 minutes ago   Up 35 minutes
0.0.0.0:5000->80/tcp, :::5000->80/tcp   pedantic_pascal
```

If you want to remove a container:

```
docker rm 2fe2790fc5fb

Error response from daemon: You cannot remove a running container
2fe2790fc5fb0fa6582052c5e9b395092d5bacf799bf320a69aef02d8226e7c1. Stop the
container before attempting removal or force remove
```

Removing a running results in an error, to fix this, you need to stop it first.

```
docker stop 2fe2790fc5fb
```

Now you can check the running containers:

```
docker ps
CONTAINER ID   IMAGE          COMMAND            CREATED      STATUS      PORTS
                                NAMES
7249ddb172c9   40381856e6a4   "python3 app.py"   2 days ago   Up 2 days
0.0.0.0:5003->80/tcp, :::5003->80/tcp   PredictionApp
69a022934e3b   40381856e6a4   "python3 app.py"   2 days ago   Up 2 days
0.0.0.0:5001->80/tcp, :::5001->80/tcp   recursing_tu
2a12bf79b735   40381856e6a4   "python3 app.py"   2 days ago   Up 2 days
0.0.0.0:5000->80/tcp, :::5000->80/tcp   pedantic_pascal
```

And to show the running and stopped containers:

```
docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED      STATUS
                  PORTS                                    NAMES
7249ddb172c9   40381856e6a4   "python3 app.py"         2 days ago   Up 2 days
                  0.0.0.0:5003->80/tcp, :::5003->80/tcp   PredictionApp
057e76530234   40381856e6a4   "PredictionApp"          2 days ago   Created
                  0.0.0.0:5003->80/tcp, :::5003->80/tcp   epic_wescoff
7f2a9a29ba0b   40381856e6a4   "PredictionApp"          2 days ago   Created
                                                          tender_hugle
2fe2790fc5fb   40381856e6a4   "python3 app.py"         2 days ago   Exited (0)
About a minute ago                                          affectionate_bohr
```

To remove a stopped container:

```
docker rm 2fe2790fc5fb
```

The container has been removed we can check using either of the following:

```
docker ps
```

```
docker ps -a
```

To remove a running container, we can enforce this using:

```
docker rm -f  69a022934e3b
```

To remove Docker images in the same way, let's show the existing images first:

```
docker images
REPOSITORY                                TAG          IMAGE ID       CREATED
    SIZE
hello-world                               latest       d1165f221234   5 months
ago    13.3kB
tamermohamed/ml-microservice-kubernetes   version1.0   40381856e6a4   15 months
ago   1.25GB
```

Then we use:

```
docker rmi d1165f221234
Error response from daemon: conflict: unable to delete d1165f221234 (must be
forced) - image is being used by stopped container 06b4e4ea661d
```

We cannot delete the image if it has a container either running or stopped, but we can force delete the image and this will remove all the containers using this image.

```
docker rmi -f d1165f221234
```

Also, the docker keeps log of what happens to any container at any given time. Note that we can use container Id or container name, but when using the container name, keep in mind that it's case sensitive.

```
docker logs PredictionApp

 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 268-526-508
172.17.0.1 - - [31/Aug/2021 03:53:29] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [31/Aug/2021 03:53:29] "GET /favicon.ico HTTP/1.1" 404 -
172.17.0.1 - - [31/Aug/2021 17:24:50] "GET / HTTP/1.1" 200 -
```

Creating a Docker Image

In the previous section, we investigated how to pull a Docker image and how to create/run a container from that image. Now let's discuss how to build a new image for our application and how to share it on the Docker Hub.

To make a Docker image, we create a dockerfile, which is nothing but a text file that contains the instructions for creating the image. For example:

```
FROM python:3.8
LABEL Maintainer="University of Canberra"
COPY . /prediction
WORKDIR /prediction
RUN pip install -r requirements.txt
CMD [ "python" ,"prediction.py"]
```

Let's navigate through the above instructions line by line:

`FROM [--platform=<platform>] <image> [AS <name>]`: This command specifies the base image `<image>`, which the new image is built from, it is a god practice to pull a base image from the Docker hub to use it as your base image. You can give a name to the built image using `AS`. The `platform` could be for example `linux/amd64`. In the above example, we use a Python image, where the version of the Python is 3.8.

`LABEL <label_name>="<label_value>"`: This is a label that you use in your image to describe your image. You can add multiple labels.

```
LABEL Maintainer="University of Canberra"
LABEL CreatedOn="2021"
```

`COPY ["<src>"... "<dest>"]`: This instruction is used to copy the files that are on the local machine to the docker image. In our case, we need to copy the code and/or data that we need to run our application from our local machine to the image, so that we copy files form the current directory on local machine to the prediction directory.

`WORKDIR`: This is the working directory which the docker docker will use to find files that we need to run the commands on.

`RUN ["executable", "param1",...,"paramN" ]`: This instruction is used to run commands, such as installing the required packages that our code will need when executed. We already are using a python image as our base, but our code needs some dependencies to run, so you need to install them. Here we add a requirement file that includes all the packages that are required to run the image on any other machine.

`CMD ["executable", "param1",...,"paramN" ]`: This executes a command to run our application when the container starts.

Building a Docker image

We need to understand that the image is built as layers on top of each other, where every instruction in the `dockerfile` creates a layer over the previous one. For instance, `FROM` creates a layer refers to the base image, The `COPY` will create another layer on top of the `FROM` layer to copy the files and so on.

Let's create a Docker image for the code of the use case, which we have done in week 4, which builds a logistic regression on the "WeatherAUS" data. First we will modify the code to save the built model to be shipped with the code.

```
from joblib import dump, load
dump(logreg, 'model.joblib')
```

Then, let us add the dockerfile in the code directory and add another `requirements.txt` which contains all of the necessary libraries and their versions needed to run the code. If we plan to ship the code, logistic regression model and the test data into the docker image as our prediction service. To achieve that, we need to create a `.dockerignore` file which contains the files that will not be shipped with the Docker image (the same as in the .gitignore file). For example let's ignore the `.ipynb` files, and the training data. You can add the below lines into the `.dockerignore` file.

```
*.ipynb
data/*.*
!data/weather_test.csv
```

Now, we are ready to build the Docker image using:

`docker build . -t dockerhub_username/prediction`

To refer to the current directory we use `.` as we are already in it. We can use other locations or even using URL as this is the path which all subsequent executions are relative to this directory. For example, to copy, the files should be relative to this directory/path.
The Docker image has been built and you can check that either from the Docker desktop app or via:

```
docker images

REPOSITORY                                  TAG          IMAGE ID       CREATED
         SIZE
ibrahimradwan/predictionv2                  latest       b36fb096148e   29
seconds ago   1.44GB
```

To run a container (an instance) of this image, we can use:

```
docker run --name predictionapplication image_id

         Date Location  MinTemp  ...  RainToday  RISK_MM  RainTomorrow
0  2008-12-01   Albury     13.4  ...         No      0.0            No
-02   Albury      7.4  ...         No      0.0            No
```

It will print the results of the prediction code. Also, we can get the result using `docker logs image_id` command. The Docker image has been built and a container of this image has been tested. The next step is to try to ship this onto the Docker Hub.

Pushing a Docker image

Until now we still have the image on our local machine , however if we want it to be used by others, we need to push it to the Docker Hub, which is the common public repository for docker images. We can do that using:

```
docker push [OPTIONS] NAME[:TAG]
```

Remember that we have used the user name of the Docker hub when building our image. This helps when publish the image to the repository, otherwise the push process will fail.

```
docker push dockerhub_username/predictionv2
```

After performing the above command, check the repositories on your Docker Hub account, you will find the image are pushed there.

If you want to delete the local image and get the image from docker hub:

`docker rmi - image_id`

`docker pull dockerhub_username>/prediction`

```
docker images

REPOSITORY                                TAG          IMAGE ID       CREATED
     SIZE
ibrahimradwan/predictionv2                   latest       b36fb096148e   48
minutes ago   1.44GB
```

Now, we have learned how to create a new image, upload it to a public repository and again download it to run on our local machine.
