

Docker

The docker documentation is available [here](#).

Problem Statement

The problem arises when you aren't working alone. Let's say you started working alone on a project and set up the environment. As the project progresses, you keep installing multiple tools and packages required for your project. Now, you have a new team member who needs to replicate your environment to start working on the project. How will you share your environment configuration if you weren't keeping track? Even if you did keep track and shared the details, it would be difficult for them to replicate it. There could be several reasons for this:

- They have a different OS.
- They might have prior packages that conflict with the new setup.

Even in an ideal case where the new member successfully replicates the environment, what happens when the team grows? Can you bear these hurdles for each new employee? And while deploying, you will have to set up the environment again. As traffic increases, you will need to scale up. Will you do this task repetitively for each server?

No, here comes the magic of Docker! By using docker we can **containerize** the complete configuration and share it with others so they don't have to face any hurdle while replicating environment. And they can start working on the project as they joins.

Installation of Docker CLI and Desktop

Docker Daemon Vs. Docker Desktop

- Docker daemon – It is the one that is actually responsible for doing everything.
- Docker desktop – It is just the GUI which is used to visualize the state of machine.

Understanding Images v/s Containers

- Image – This is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and configuration files. Images are read-only templates used to create containers. It could be anything from a software tool to a OS.

```
docker run -it ubuntu
```

It will start finding image of 'ubuntu' locally. If it finds the image the separate docker will get startup and run that image. Otherwise, the image will be then downloaded from hub.docker.com.

- Container – This is a runnable instance of an image. Containers are lightweight because they share the host system's kernel and use minimal resources. They are **isolated** environments where applications run, and they can be started, stopped, moved, or deleted easily.

Note: Container is like a machine/laptop and image is like a OS. A container can't access the data of other container until unless we don't want that.

Terminologies

- **Image** – A read-only template used to create containers.
- **Container** – A runnable instance of an image.

- **Dockerfile** – A text file that contains a series of instructions on how to build a Docker image.
- **Volume** – A way to persist data generated by and used by Docker containers.
- **Registry** – A storage and distribution system for Docker images.
- **Docker Hub** – A public registry that anyone can use to store and share Docker images.
- **Build** – The process of creating a Docker image from a Dockerfile.
- **Pull** – Downloading an image from a registry.
- **Push** – Uploading an image to a registry.
- **Tag** – A label applied to a Docker image to identify it.
- **Network** – A way to connect Docker containers to each other and to the outside world.
- **Orchestration** – Managing and coordinating multiple containers.

Commands

Command	Description
<code>docker run -it ubuntu</code>	Used to pull-in and spin a container with ubuntu image. -it – for interactive. --name can be used to specify container name.
<code>docker container ls</code>	Used to list only running containers.
<code>docker container ls -a</code>	Used to list all containers.
<code>docker start con_name</code>	Used to start the named container.
<code>docker stop con_name</code>	Used to stop the named container.
<code>docker exec con_name com_to_execute</code>	Used to run a command in a specified container and return the result to our terminal after disconnecting from container.
<code>docker exec -it con_name bash</code>	Now it will attach our terminal with the specified container i.e we will remain in that container. That's what -it stands for.
<code>ctrl + d</code>	To exit from a container.
<code>docker images</code>	To list images in a our computer.
<code>docker image ls</code>	To list images in a our computer.
<code>docker run -it -p 6000:1025 im_name</code>	Used to expose/map the containers port to/with our machine port so that we can run the app on our machine too. Let's say image runs a some kind of server on port 1025, so it will be accessible on port 1025 inside container and on 6000 outside the container.
<code>docker run -it -e PORT=4000 -p 6000:4000 im_name</code>	By using -e flag we can pass the environment variables in our application. Ofcourse, app should have a functionality to process that environment variable.
<code>docker build -t im_name dir/project</code>	Dir/project path that will be used for building that image.
<code>docker login</code>	Will ask for username and password to login to dockerhub. Required for pushing images to hub.
<code>docker compose up</code>	Will read docker-compose.yml from current directory and setup the requirements along with their configurations. -d flag could be used to processed setup in background (i.e. in detached mode)
<code>docker compose down</code>	Will remove the composed things.
<code>docker network inspect bridge</code>	Will give the information about this network and tell which running containers are connected with this.

<code>docker run -it --network=host im_name</code>	Now the spinned container will use host network driver.
<code>docker network create -d bridge net_name</code>	To create custom network that will use bridge network driver.
<code>docker run -it -v host_machine_path_to_folder:container_path_to_folder im_name</code>	Use to mount a specific folder of host machine to specific folder of container.

```
piyushgarg@macbook.pro > docker exec pedantic_mendel ls
bin
boot
data-1
dev
```

```
piyushgarg@macbook.pro > docker exec -it pedantic_mendel bash
root@9df3c066ad91:/# ls
```

Dockerization of Node.js Application

Dockerfile

It is basically a configuration file for an image. It has no extension so the name is case sensitive. Firstly, we have to select the base image it could be any OS or any package which is already published as a standalone image. Then if we are using a base image of OS we will have to install require libraries/packages etc. Then we have to copy our code and lastly, we have to give an entry point like what should happen when the image get started.

```
FROM ubuntu

RUN apt-get update
RUN apt-get install -y curl
RUN curl -sL https://deb.nodesource.com/setup_18.x | bash -
RUN apt-get upgrade -y
RUN apt-get install -y nodejs

COPY package.json package.json
COPY package-lock.json package-lock.json
COPY main.js main.js

RUN npm install

ENTRYPOINT [ "node", "main.js" ]
```

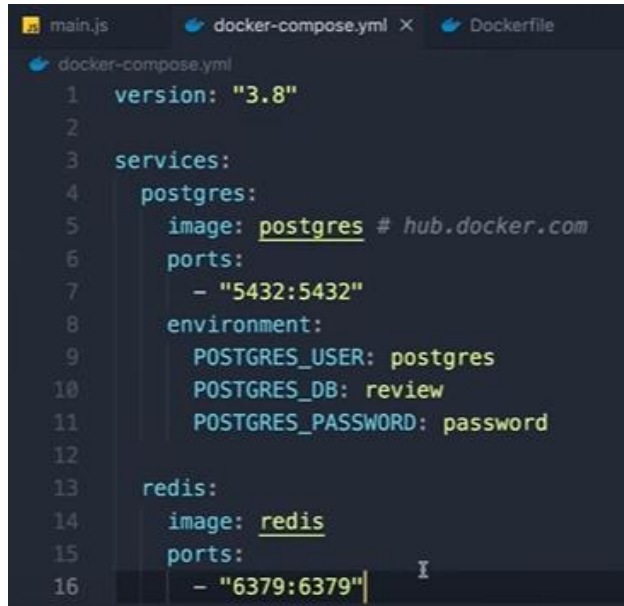
Caching Layers

Basically, docker builds the image in layers, so that they can be cached. Now, what does that means? Let's say you have done some modifications in your code so docker will not rebuild the image from start and the layers will be cached so ultimately it will save our time and our resources. The docker will only rebuild those layers which will get affected by modifications. So, we have to write the configuration file very carefully so that maximum optimization can be achieved.

Docker Compose

Now, while running containers we came across a new problem and that was – If you are working on a project which requires multiple containers to be setup and there is a configuration for each container so do you need to setup each and every container manually? Here comes the use of

docker compose. Below is a structure of a basic docker compose file which allows you to setup each container with their configurations by only using a single command.



```
1 version: "3.8"
2
3 services:
4   postgres:
5     image: postgres # hub.docker.com
6     ports:
7       - "5432:5432"
8     environment:
9       POSTGRES_USER: postgres
10      POSTGRES_DB: review
11      POSTGRES_PASSWORD: password
12
13   redis:
14     image: redis
15     ports:
16       - "6379:6379"
```

Docker Networking

The tutorial is available [here](#). Docker container can talk to outer world internet and it is possible because of docker networking. There are multiple networking drivers that enable us to communicate with internet but we will study the following drivers as they are widely used.

Bridge

While spinning a container if we don't specify any networking driver, by default this driver will be used. It's same as its name suggests, basically a bridge will be established between host machine and by using these drivers an IP address will be assigned to that container that will be used to communicate with outer world.

Host

Now again as name suggests, container using host network driver will be directly connected to the host machine network. So the basic advantage of using this driver is we will don't have to expose the internal ports of our container to access anything we can directly access that port on our machine as both are on the same network.

None

If we specify the network driver to **none** then the container will not use any network so it will not be able to communicate outside.

Custom

Custom networks can be made to make the communication process easier. You can make your network by using the bridge driver then bind any container with your custom driver and boom you will now easily communicate with all of your containers.

Volume Mounting

Docker containers have their own memory that's why they are isolated. So, when we delete the container, their memory also gets deleted and it might be not suitable for some cases. So, volume mounting is a solution for this problem. We can mount a particular directory/volume of our host machine to our container directory/volume. And we can also create custom volume in the docker

itself and spin the container by using that volume so when we will delete the container our data will be still there in that volume.

Efficient Caching in Layers

```
RUN apt-get install -y curl
RUN curl -sL https://deb.nodesource.com/setup_18.x | bash -
RUN apt-get upgrade -y
RUN apt-get install -y nodejs

COPY package.json package.json
COPY package-lock.json package-lock.json

RUN npm install

COPY main.js main.js

ENTRYPOINT [ "node", "main.js" ]
```

This is the optimized version, because every layer will be rebuild after the layer that got some changes. Let's say if we change version to 16 then that layer needs to be rebuild definitely but the layers after that layer will also be rebuild. So, the order matters! Now we have moved the npm install line up because it should only installed again when there is a change in package or package-lock file not the code.

Now still much optimization is there. Should we copy our code like this? For bigger projects it won't suitable. So, **COPY .** can be used to copy all the code at once. But it will copy all the code so to ignore some specific files we can make a **.dockerignore** file for that just like **.gitignore**.

```
.dockerignore
1 node_modules/
```

Still there is an issue! Should we copy our code in the root directory? No, we can copy it in a specific folder which will be good practice.

```
COPY package.json /app/package.json
COPY package-lock.json /app/package-lock.json

RUN cd app && npm install

COPY main.js /app/main.js

ENTRYPOINT [ "node", "app/main.js" ]
```

Should we write /app everywhere? No, you can set the working directory and then you will not have to write like this.

```
WORKDIR /app

COPY package.json package.json
COPY package-lock.json package-lock.json

RUN npm install

COPY main.js main.js

ENTRYPOINT [ "node", "main.js" ]
```

Docker Multi-Stage Builds

```
FROM ubuntu as build

RUN apt-get update
RUN apt-get install -y curl
RUN curl -sL https://deb.nodesource.com/setup_18.x | bash -
RUN apt-get upgrade -y
RUN apt-get install -y nodejs
RUN apt-get install typescript

WORKDIR /app

COPY package.json package.json
COPY package-lock.json package-lock.json

RUN npm install
RUN tsc -p . # build

FROM node as runner

WORKDIR app/

COPY --from=build app/ .

ENTRYPOINT [ "node", "main.js" ]
```

It is efficient when we have a package that will be only required while building whereas there is no need of that package while running. So, why do we include it in our final image? It will just use more resources. So, it's better to use multi stage builds there. We are assuming there that a typescript package is needed only while building a package so in the final stage we are just copying everything from the first stage because now our code is compiled and we don't need that package. So, the image will be built from the final stage. In my case, take an example of potree converter once the file is converted do you need that converter? No, it's better to use multistage there.

Deploying Docker Containers on AWS ECS & ECR

The tutorial is available [here](#).