Docker & Kubernetes for Data Science

Martial Luyts

Catholic University of Leuven, Belgium

martial.luyts@kuleuven.be

KU LEUVEN

LEUVEN STATISTICS RESEARCH CENTRE

Contents

Part 0:

Introduction & outline of this class

0.1 Introduction

- Imagine you are a data scientist working on a machine learning model.
- You set it up on your laptop, but when you share it with your team, it doesnt work on their computers.

- **Question:** How can we solve this?

- **Solution:** Package everything your model needs into a **container**.
  - Source code: main scripts (e.g., .py)
  - Dependency descriptors (e.g., requirements.txt)
- **Docker** enables this possibility

0.2 Outline of the class

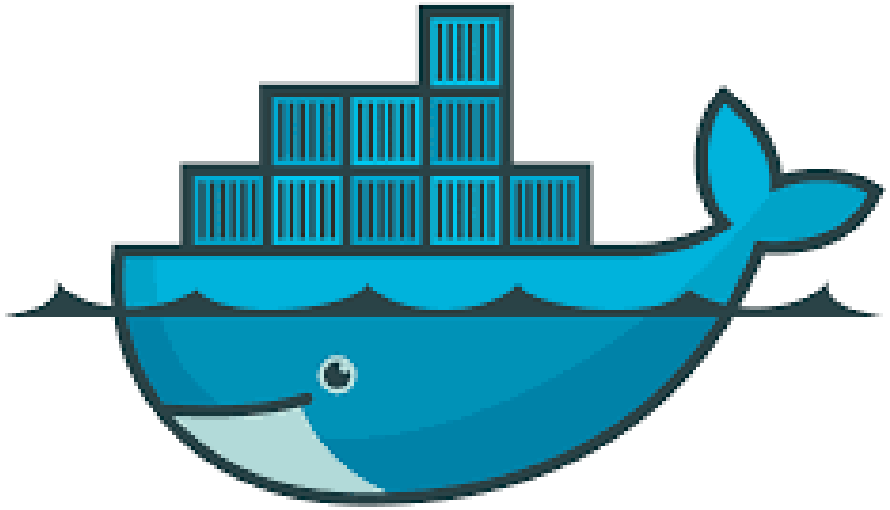
- This presentation is specifically designed for data science practitioners, who want to learn more about one of the key building blocks of MLOps
- Prior knowledge of Python is required.
- By the end, you will understand how to use Docker for data science projects, and gain some basic notion about Kubernetes, including:
  - Understanding basic Docker commands
  - Build and run a Docker image
  - Store and share data
  - Networking
  - Managing multi-container apps

Part 1:

Infrastructure of Docker

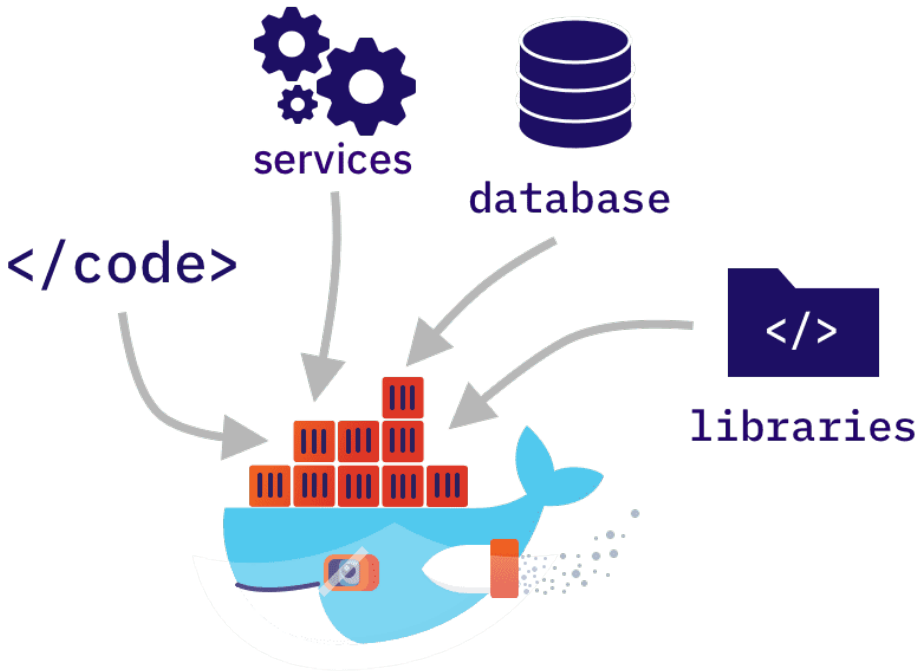
1.1 What is Docker?

- **Docker** is a container-based platform for developing, shipping, and running applications in containers.



- Containers are self-contained, lightweight environments

- Helps in eliminating "works on my machine" issues.
- Allows developers to package applications with all dependencies included in an isolated and consistent environment.



Intuitively:

- Think of a container like a lunchbox.
- You pack your meal (code and dependencies) inside it.
- No matter where you take it (Windows, Mac, Linux), you can open it and eat (run the code) without worrying about the kitchen (environment setup).

1.2 Benefits of Docker for Data Science

Several benefits of Docker for Data Science include:

- **Reproducibility:** Docker provides a consistent and reproducible environment for running applications, making it easier to replicate experiments and results
- **Portability:** Docker containers are portable across different platforms, making it easier to move applications across
- **Scalability:** Docker containers are light-weight and can be scaled easily, making it easier to manage resources efficiently
- **Collaboration:** Docker provides a standardized environment for running applications, making it easier to collaborate and share applications with others.

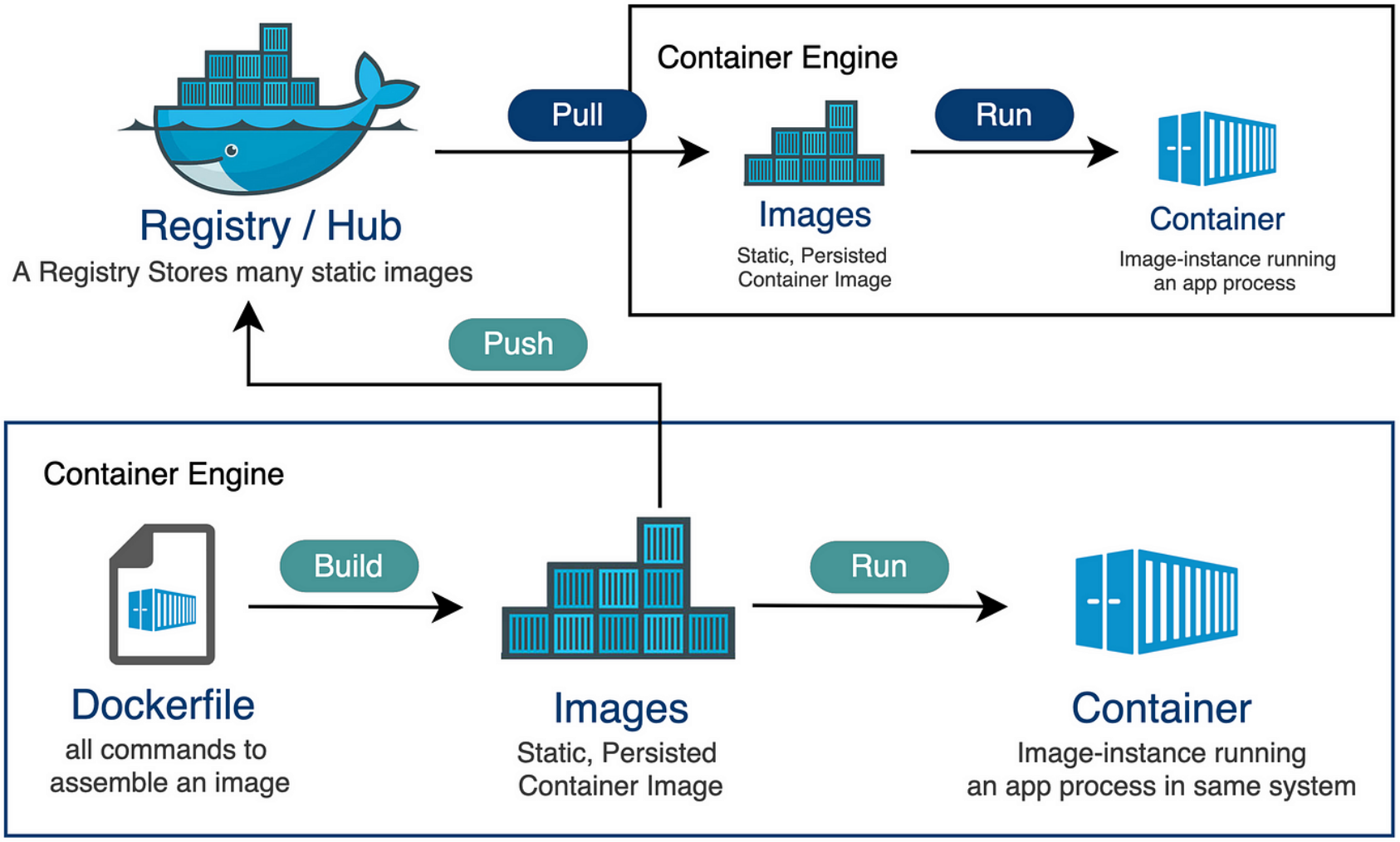

FINALLY AFTER
AN ENTIRE DAY I
SETUP MY LOCAL ENV
DOCKER!!

1.3 What is a Docker Container?

- A standardized unit of software that includes everything needed to run.
- Consists of code, runtime, system libraries, and dependencies.
- Runs the same regardless of the environment.

1.4 Key Docker Components

- **Docker Image:** Read-only template that contains the application, its dependencies, and other settings needed to run the application.
- **Docker Container:** A lightweight, standalone, and executable package that includes everything needed to run the application, including the code, libraries, and runtime.
- **Dockerfile:** A text file that contains a set of instructions for building a Docker Image
- **Docker Registry:** A repository that stores Docker images. Docker Hub is a popular public registry for storing and sharing Docker images.


Container Engine
Pull
Run
Registry / Hub
A Registry Stores many static images
Images
Static, Persisted
Container Image
Container
Image-instance running
an app process
Push
Container Engine
Build
Run
Dockerfile
all commands to
assemble an image
Images
Static, Persisted
Container Image
Container
Image-instance running
an app process in same system

1.5 Installing Docker

Docker needs to be installed on your local computer:

- Visit the official Docker website (www.docker.com/products/docker-desktop/) and download the Docker Desktop app for your operation system.
- Install Docker Desktop and follow the instructions provided by the installer.
- Open it and check whether the **Docker daemon** is running.

- To verify the installation, open the terminal and run the command

  ```
  docker --version
  ```

  This also ensures that the **Docker Command Line Interface (CLI)** can be called, i.e.,

  - Text-based tool for interacting with Docker
  - Run via terminal (with Docker commands; Section 1.6)
  - Full control over Docker functions, including advanced options
  - Universally available, works in any environment where Docker is installed

- To verify that Docker can pull images from the Docker Hub and run it, run the following test container in Docker CLI:

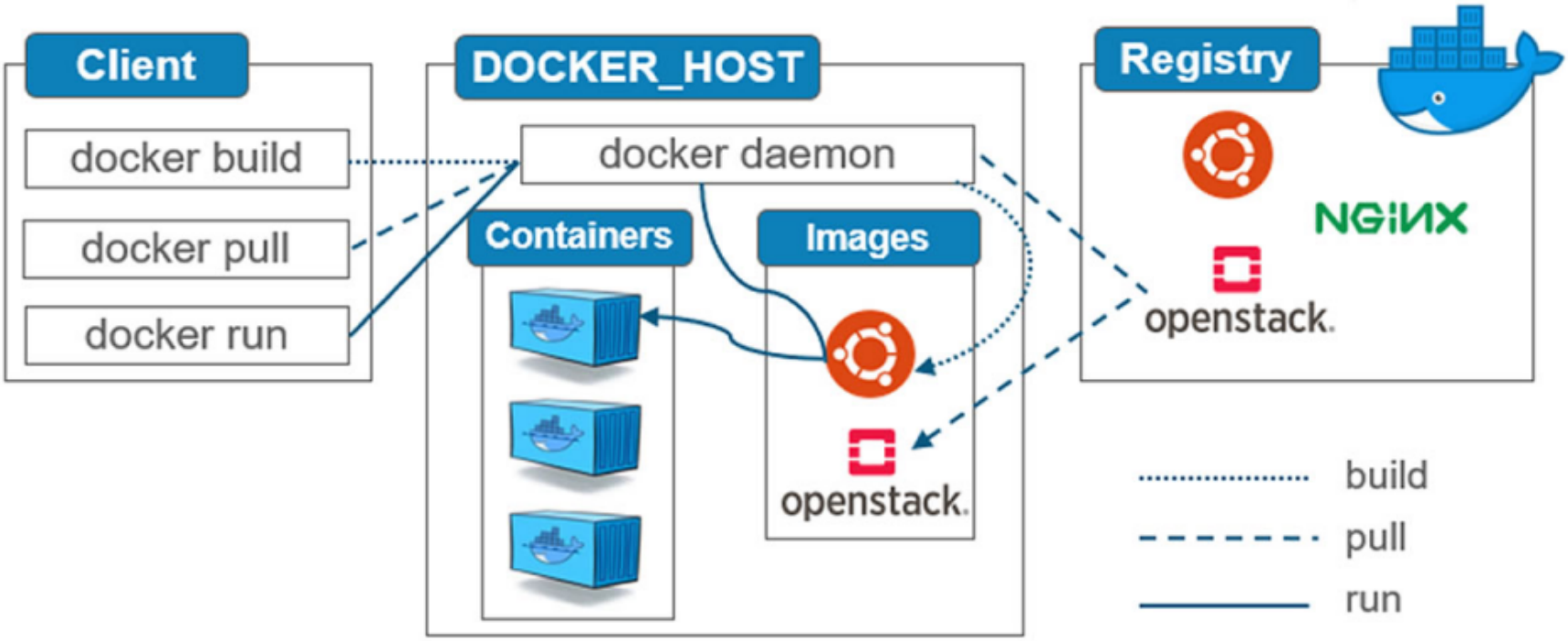
```
docker run hello-world
```

1.6 Basic Docker commands

Docker is managed primarily through a CLI, with commands that control Dockers behavior and the life cycles of containers and images.

Some basic commands:

- `docker run`: Start a container from a specified image
  - **Example:** `docker run hello-world`
- `docker pull`: Fetches an image from a registry (like Docker Hub) without starting a container
  - **Example:** `docker pull ubuntu`
- `docker build`: Create a new image from a Dockerfile. You run it in the directory where the Dockerfile is located.
  - **Example:** `docker build -t my-image .`


Client
docker build
docker pull
docker run
DOCKER_HOST
docker daemon
Containers
Images
openstack.
Registry
NGINX
openstack.
build
pull
run

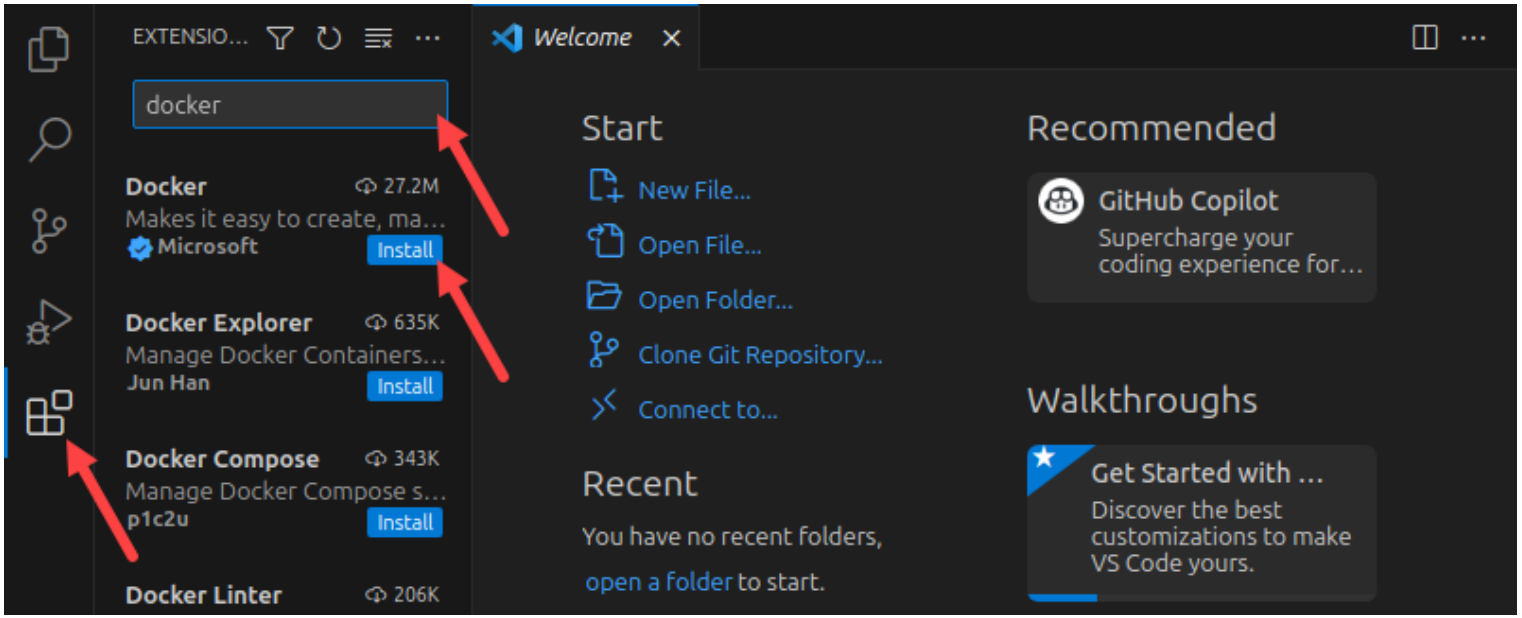
- `docker push`: Uploads an image you've created to a registry. You must be logged in and have the right to push to the repository.
- `docker images`: Lists all the images that are locally stored with the Docker engine
- `docker rmi`: Removes one or more images. Any containers using the image must be stopped and removed before the image can be removed.
- `docker ps`: Lists running containers. Using `docker ps -a` will show all containers, including stopped ones.

- `docker stop`: Stops a running container.
- `docker start`: Starts a container that has been stopped.
- `docker restart`: Restarts a container that's running or stopped.
- `docker rm`: Deletes one or more containers. The container must be stopped before it can be removed.
- `docker exec`: Runs a command in a running container.
  - **Example:** `docker exec -it container_name bash` opens a bash shell in the container.

1.7 Docker extension in VS code

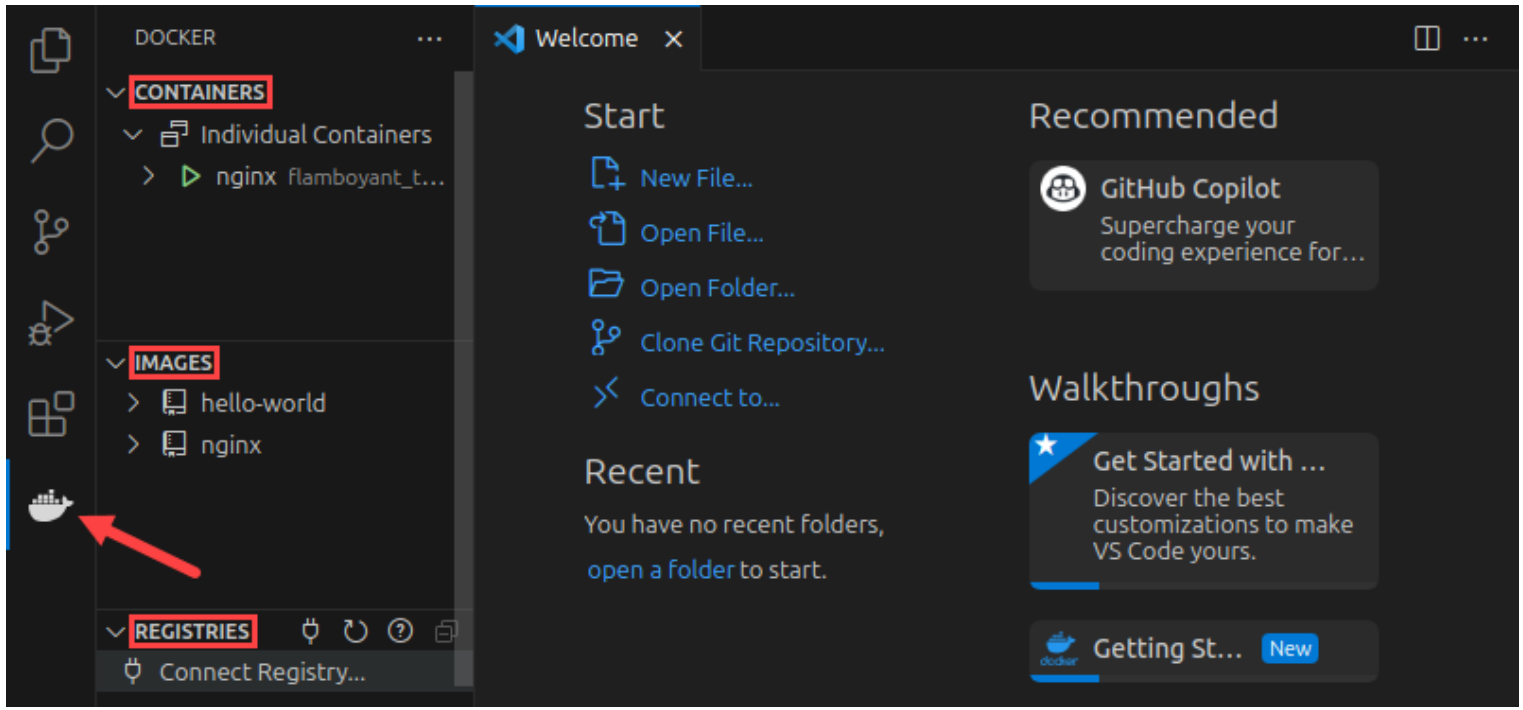
- The **Docker Extension for VS Code** brings the power of Docker into your development environment

- **Installation:**

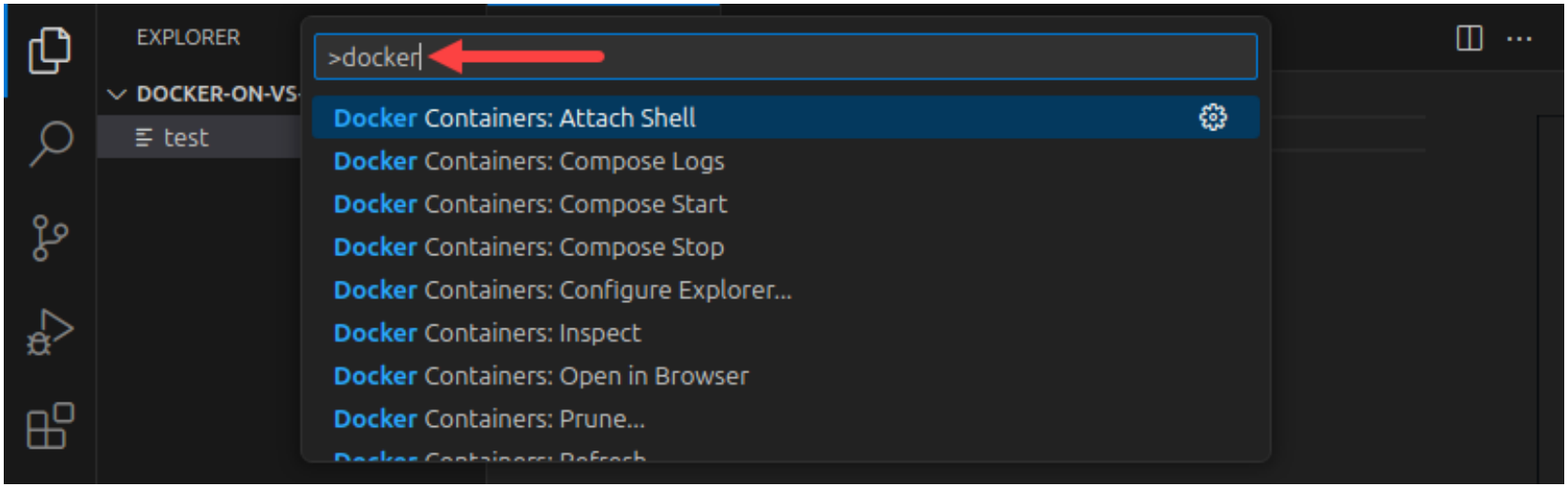


- Provides an intuitive graphical interface for working with Docker, reducing the need to switch between the terminal and your code

- The Docker Pane can be used as quick panel for visualizing all made containers, images available locally and connection with a registry (if applicable)



- Many of the most common Docker commands can be accessed into the Control Palette:
  - Open the palette by pressing **Ctrl + Shift + P**
  - Type Docker in the search bar to display the available Docker commands.

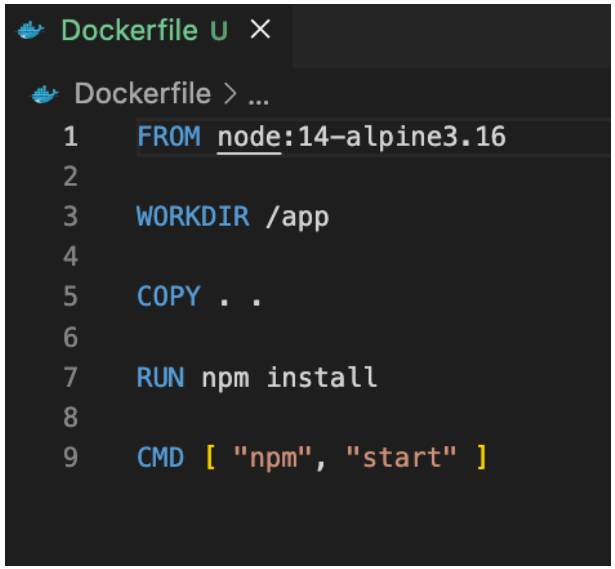


- Allows you to manage containers, images, volumes, registries and networks right from the editor
- Provides an intuitive graphical interface for working with Docker, reducing the need to switch between the terminal and your code
- Useful for developers that are building, debugging, and testing containerized applications directly inside VS Code

1.8 Build and run a Docker image

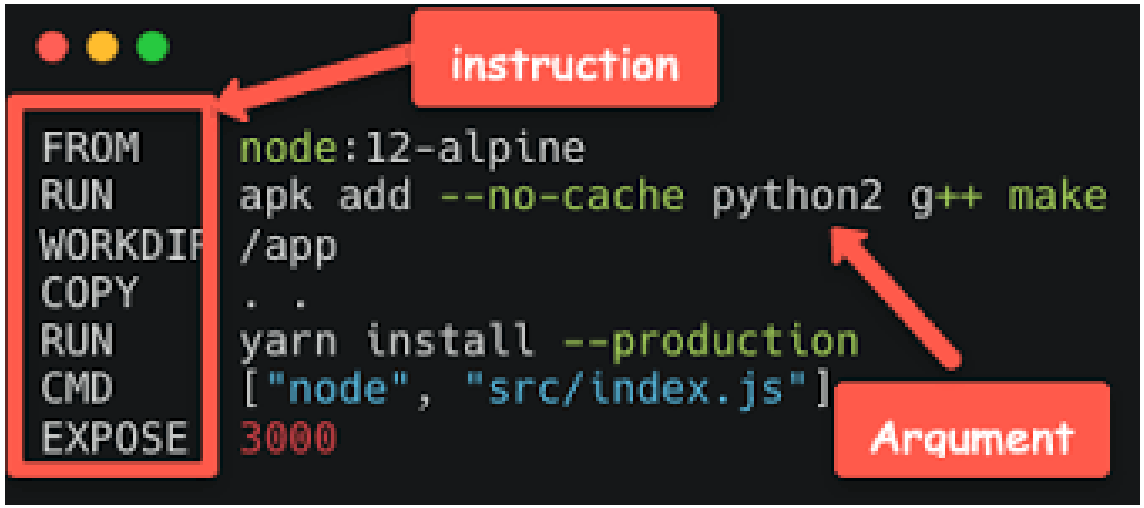
- Docker builds images by reading the instructions from a **Dockerfile**.
  - **Reminder: Dockerfile** is a text file containing instructions for building your source code.

- The default filename to use for a Dockerfile is `Dockerfile`, without a file extension. Using the default name allows you to run the docker build command without having to specify additional command flags



- To compose a Dockerfile, different types of instructions will be used, e.g., :
  - `FROM <image>`: Defines a base for your image.
  - `RUN <command>`: Executes any commands in a new layer on top of the current image and commits the result. In data science projects, this is often incorporated within `requirements.txt`
  - `WORKDIR <directory>`: Sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, and `ADD` instructions that follow it in the Dockerfile.
  - `COPY <src> <dest>`: Copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.

- `ENV <command>`: Set environment variables, if your application uses them.
- `EXPOSE <command>`: Expose a port that the application will listen on.
- `CMD <command>`: Lets you define the default program that is run once you start the container based on this image. Each Dockerfile only has one `CMD`, and only the last `CMD` instance is respected when multiple exist.



Example: Build a dockerfile for a simple data science project

Step 1: Let's say we want to install Python 3.9 with two of its libraries: `pandas` and `numpy`.

- Create a `requirements.txt` file having the following text:

```
numpy
pandas
```

- In the Dockerfile,
  - the Python 3.9 image from DockerHub will be fetched using the FROM statement;
  - The requirements.txt file is copied from our local directory to the /tmp directory using the COPY statement
  - Execute a bash command for installing via pip using the RUN statement

```
FROM python:3.9
COPY requirements.txt /tmp

RUN pip install -r /tmp/requirements.txt
```

- To create an image, we build it running the command `docker build`, where, for example, `docker-for-ds` is the image name and `1.0.0` is the version identifier.

```
docker build . -t docker-for-ds:1.0.0
```

- To check the list of images that we have created, we can run the following `docker image list` command

```
docker image list
```

```
→  docker docker image list
REPOSITORY      TAG       IMAGE ID       CREATED              SIZE
docker-for-ds   1.0.0     c1665178e0ac   About a minute ago   1.01GB
```

- We can now run the container from this image, and check whether installation of the 2 libraries took place

```
docker run <DOCKER-IMAGE-NAME>:<VERSION> <BASH-COMMAND>
```

```
→  docker docker run docker-for-ds:1.0.0 python --version
Python 3.9.16
→  docker docker run docker-for-ds:1.0.0 pip list
Package         Version
--------------- --------
numpy           1.24.1
pandas          1.5.3
pip             22.0.4
python-dateutil 2.8.2
pytz            2022.7.1
setuptools      58.1.0
six             1.16.0
wheel           0.38.4
WARNING: You are using pip version 22.0.4; however, version 22.3.1 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
```

Step 2: Make changes to the Docker image by adding a runnable Python script called `test.py` and create a directory called `documents` and populating it with a non-empty text file called `file.txt`.

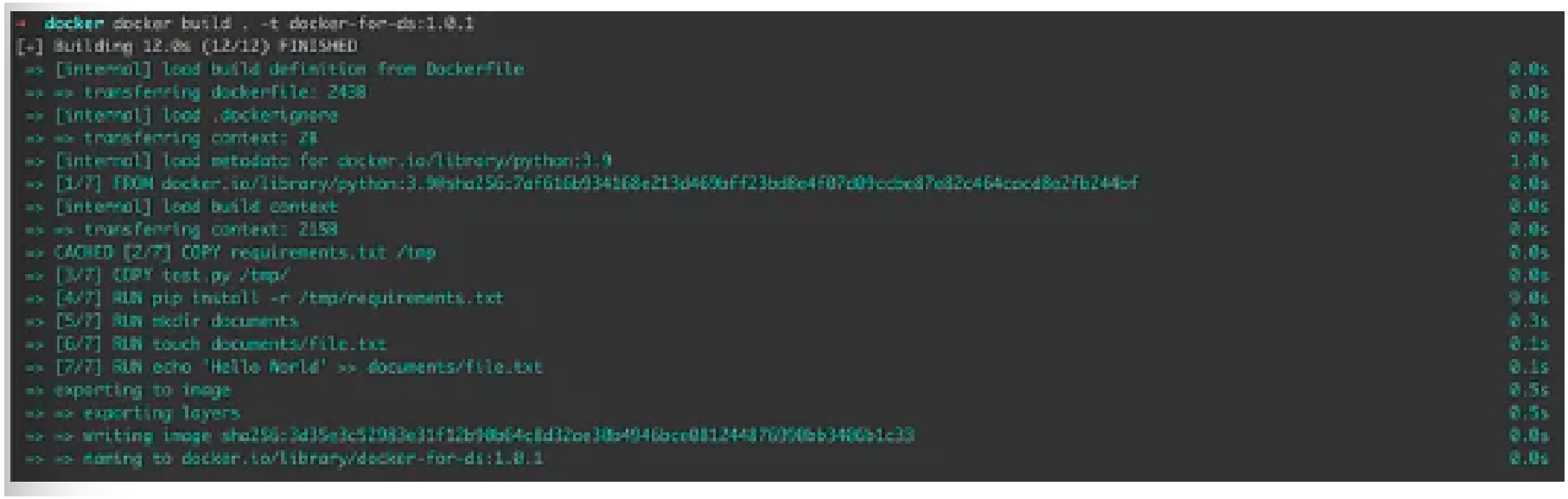
test.py:

```
import numpy as np

print("Running a simple Python Script")

list = [i for i in range(10)]
print(list, type(list), type(np.array(list)))
```

- The Dockerfile can now be adapted accordingly:

```
FROM python:3.9
COPY requirements.txt /tmp
RUN cp test.py /tmp/

RUN pip install -r /tmp/requirements.txt

RUN mkdir documents
RUN touch documents/file.txt
RUN echo 'Hello World' >> documents/file.txt
```

- Build now an updated images, under version `1.0.1`



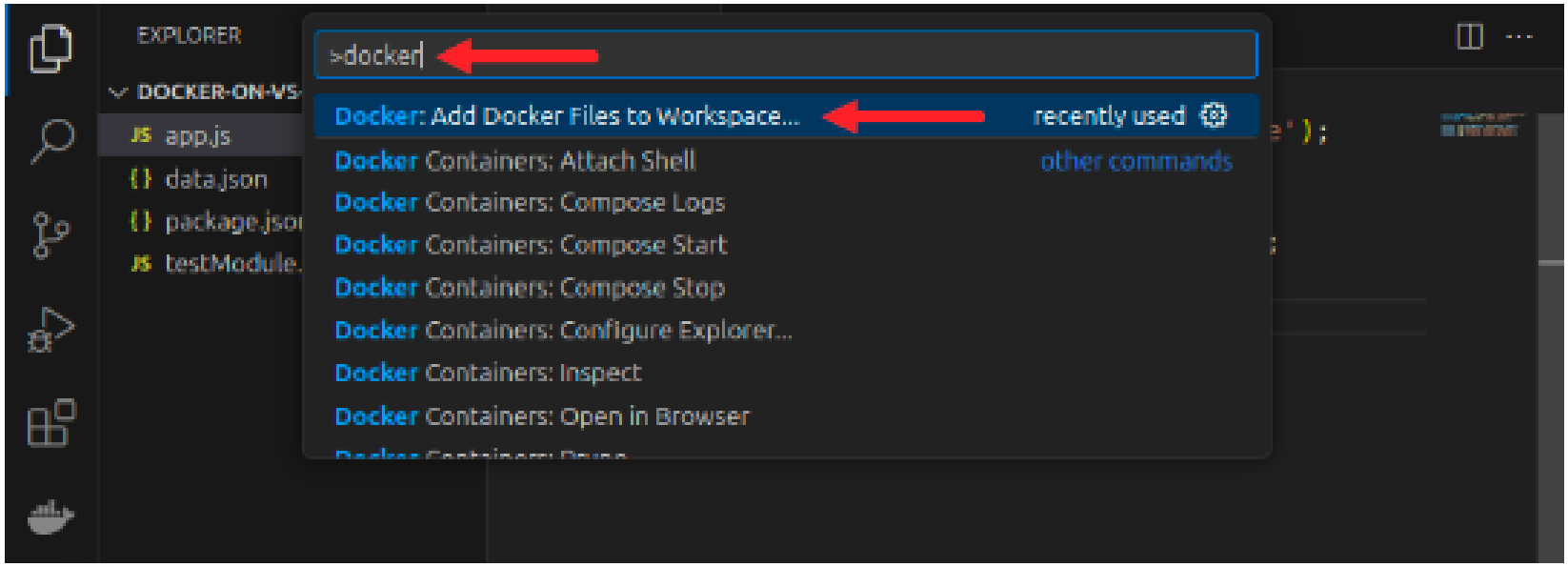
- Checking that everything works, by running the corresponding container under this image:

```
→  docker docker run docker-for-ds:1.0.1 cat /documents/file.txt
Hello World
→  docker docker run docker-for-ds:1.0.1 python /tmp/test.py
Running a simple Python Script
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'> <class 'numpy.ndarray'>
→  docker docker run docker-for-ds:1.0.1 cat /tmp/test.py
import numpy as np

print("Running a simple Python Script")

list = [i for i in range(10)]
print(list, type(list), type(np.array(list)))
```

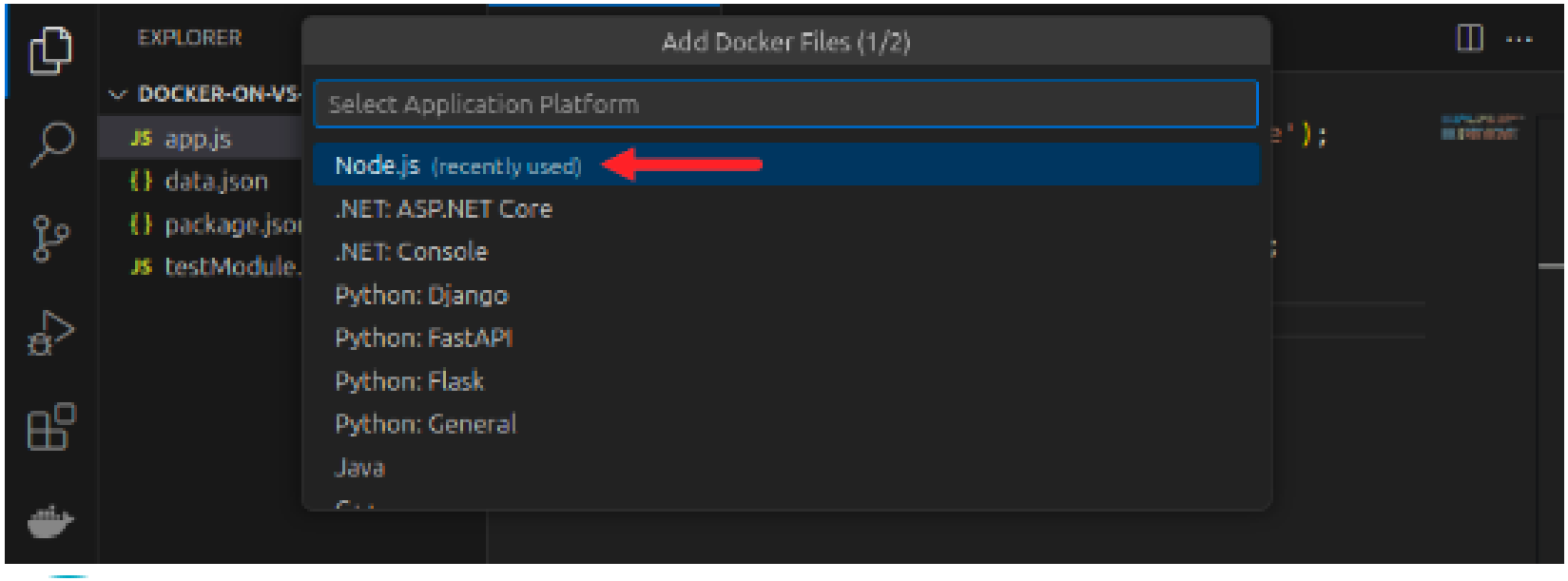
- Until now, a Dockerfile is composed manually with known instructions, and an image is build throughout Docker CLI.
  - **Limitations:** Time consuming and higher possibility for errors, resulting in debugging.

- Alternatively, a Dockerfile and corresponding image can also be created using VS Code, with the Docker extension.
  - **Advantages:** Point-and-click strategy with pre-build statements, resulting in less error making and faster composition.

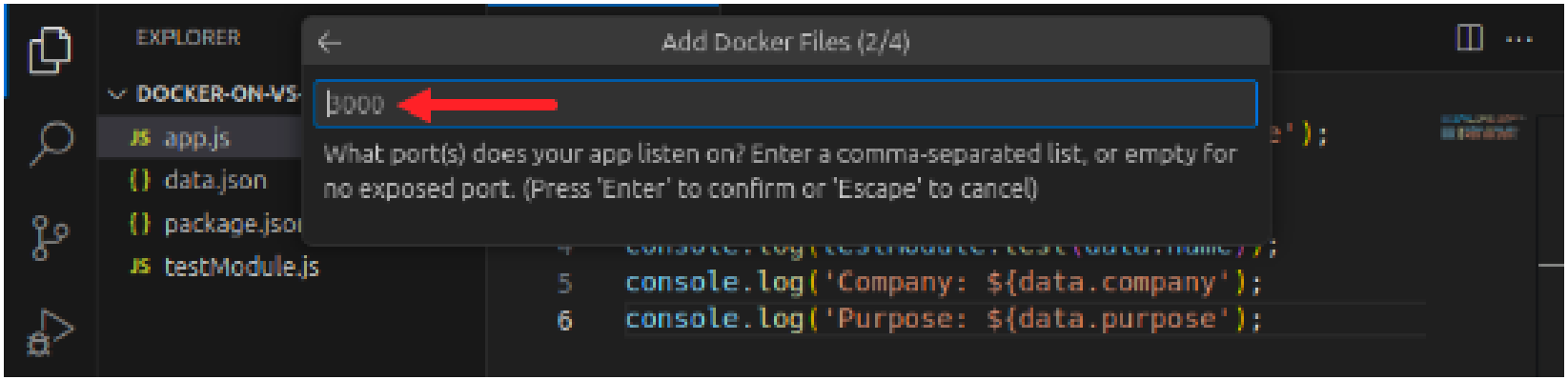
- **Steps:**
  - Press Ctrl + Shift + P to open the Control Palette.
  - Select `Docker: Add Docker Files to Workspace...` from the search results list.



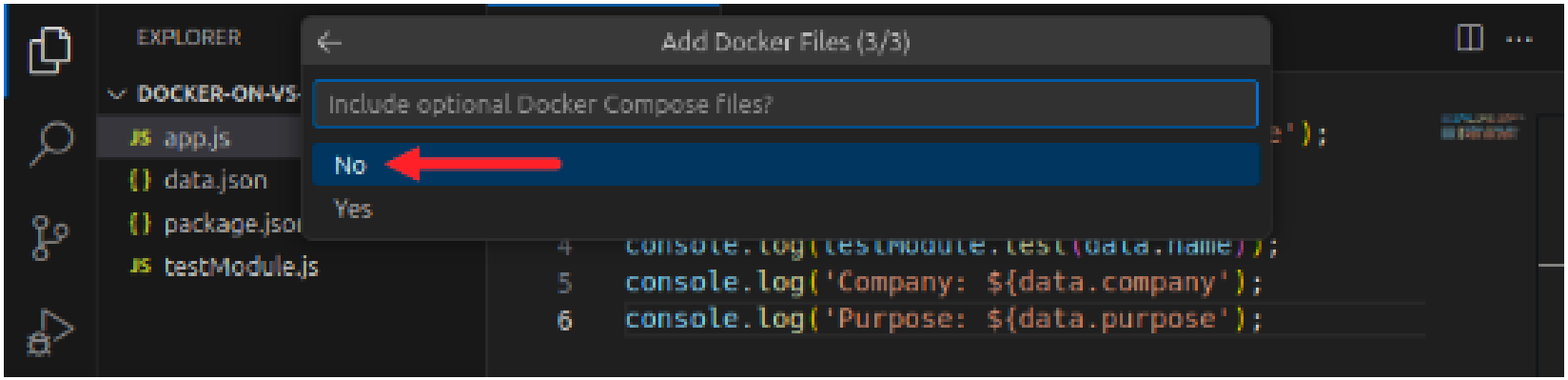
- Choose the platform you used to develop the app, e.g., Python, Node.js, etc.



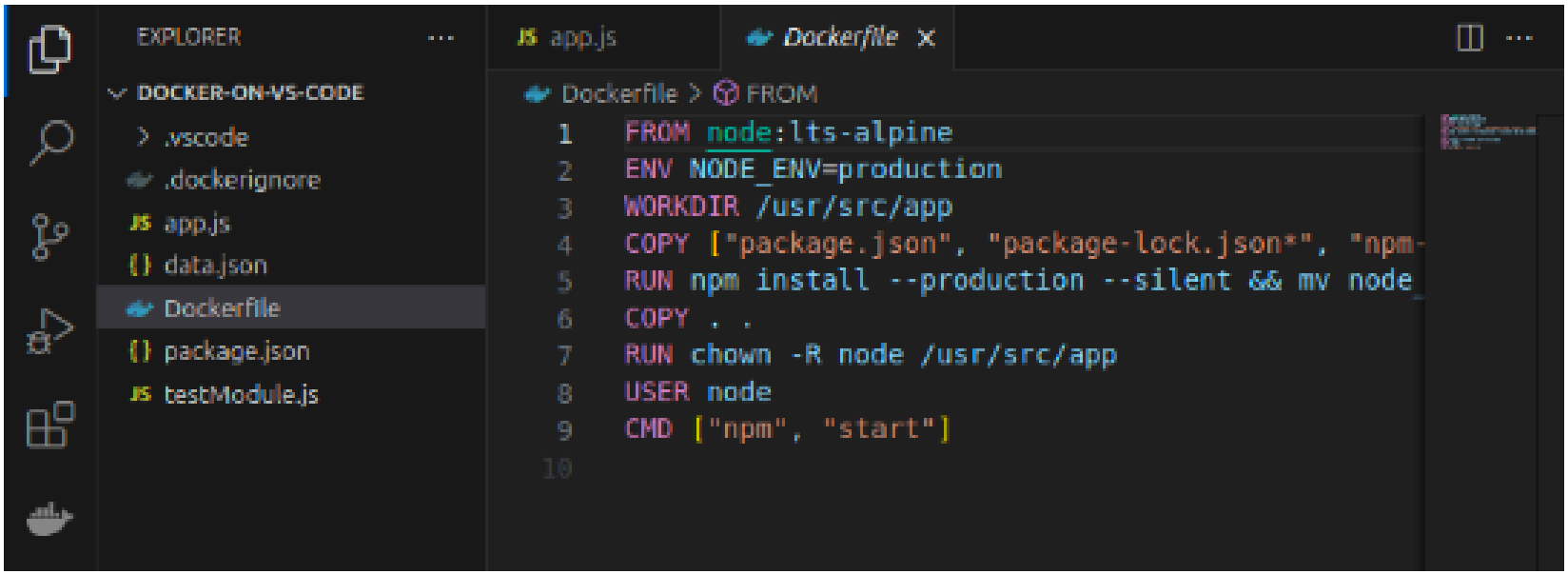
- Type the number of the port your application listens on. Leave the field empty if the app does not expose a port.



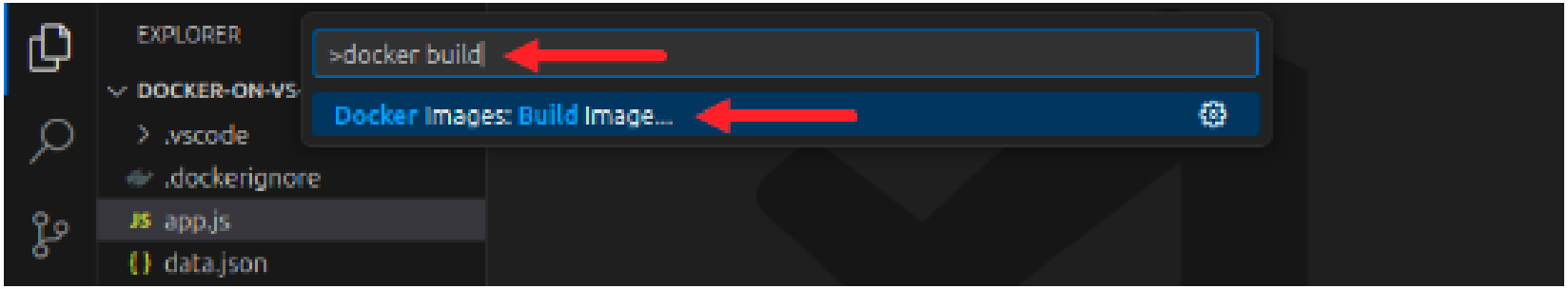
- Choose whether to include Docker Compose files (see later).

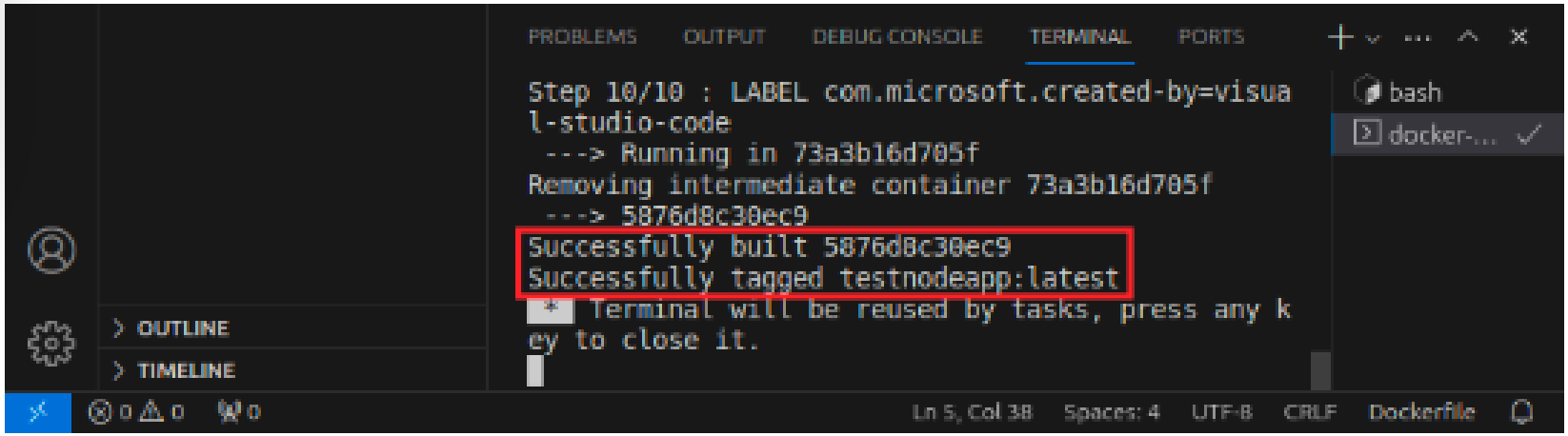


- VS Code checks the files' syntax and creates the necessary Dockerfile

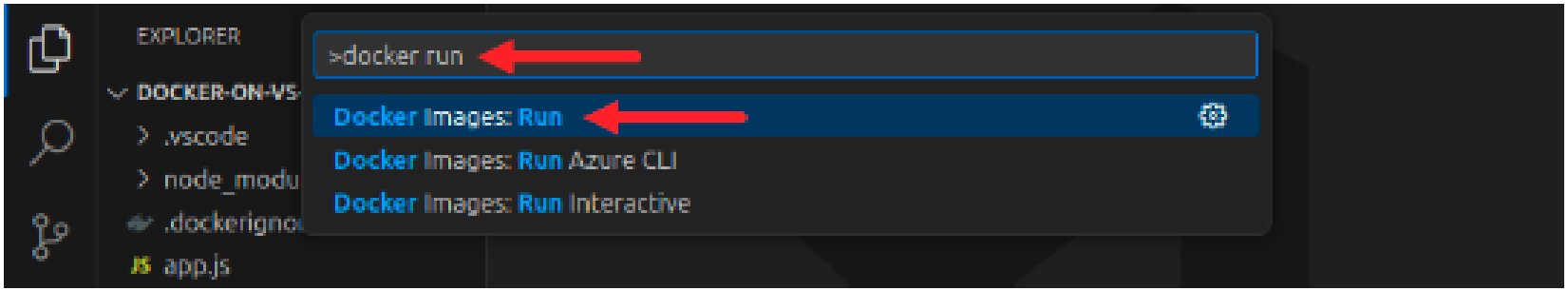


- To build an image for this Dockerfile, search for and select the `Docker Images: Build Image...` command.

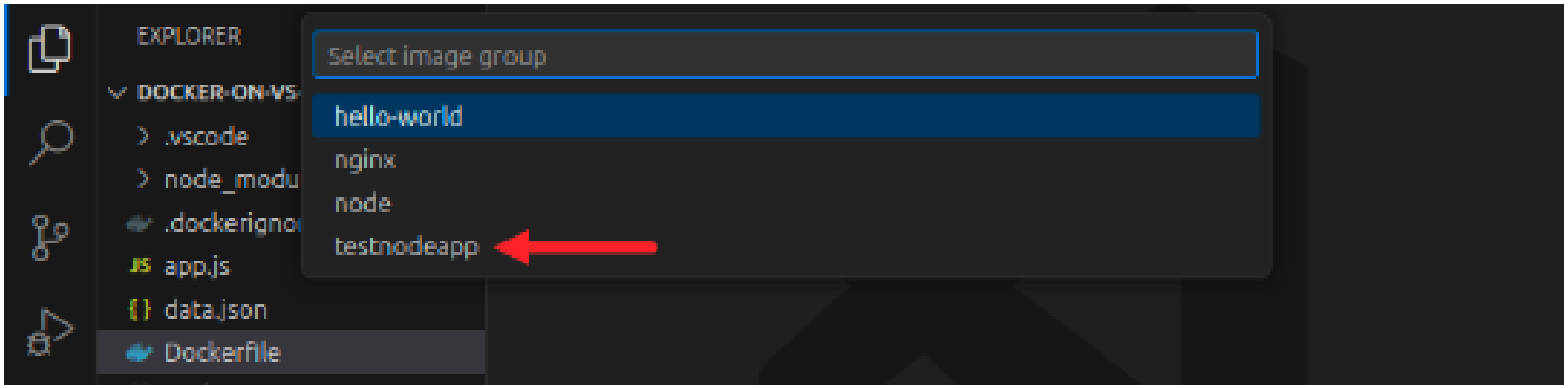




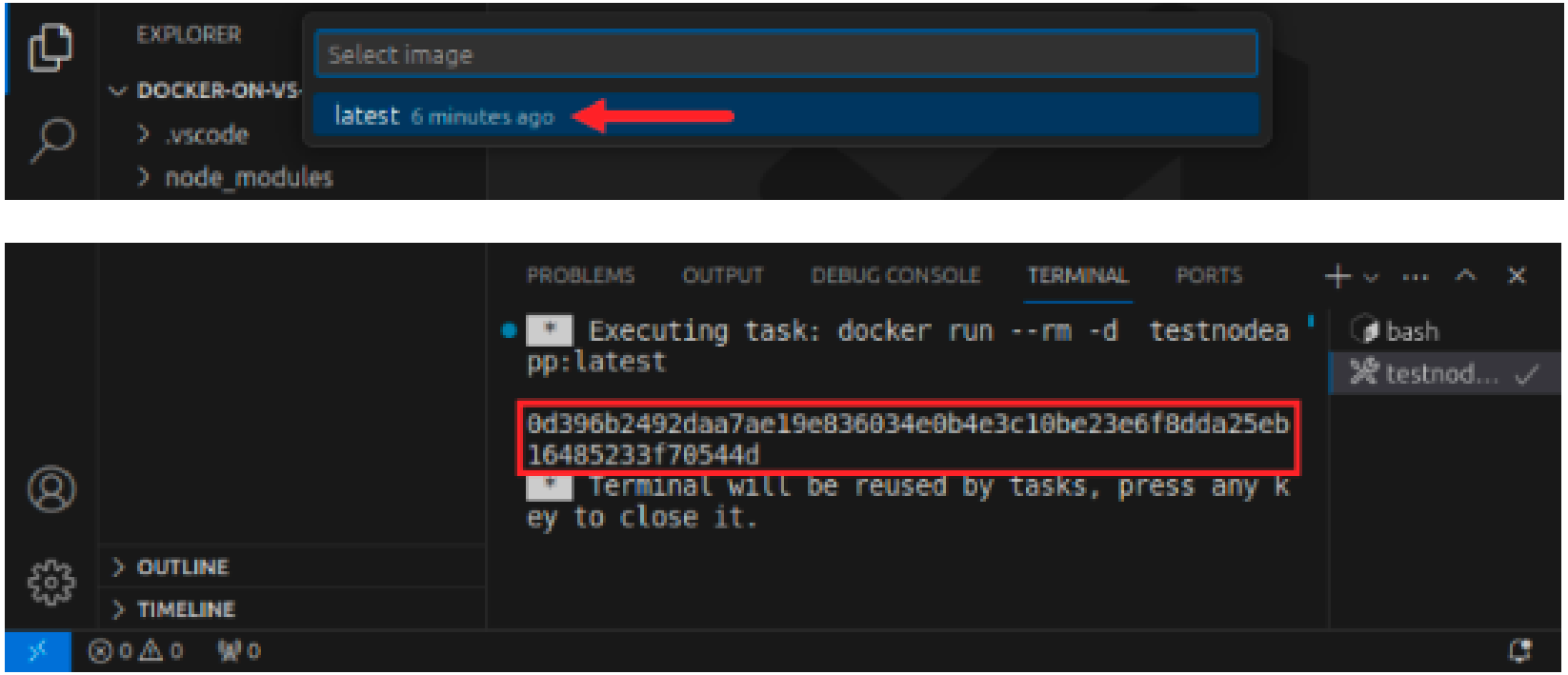
- To run a container based on the image, search for and select the `Docker Images: Run` command in the Control Palette



- Select the image to use for the container.

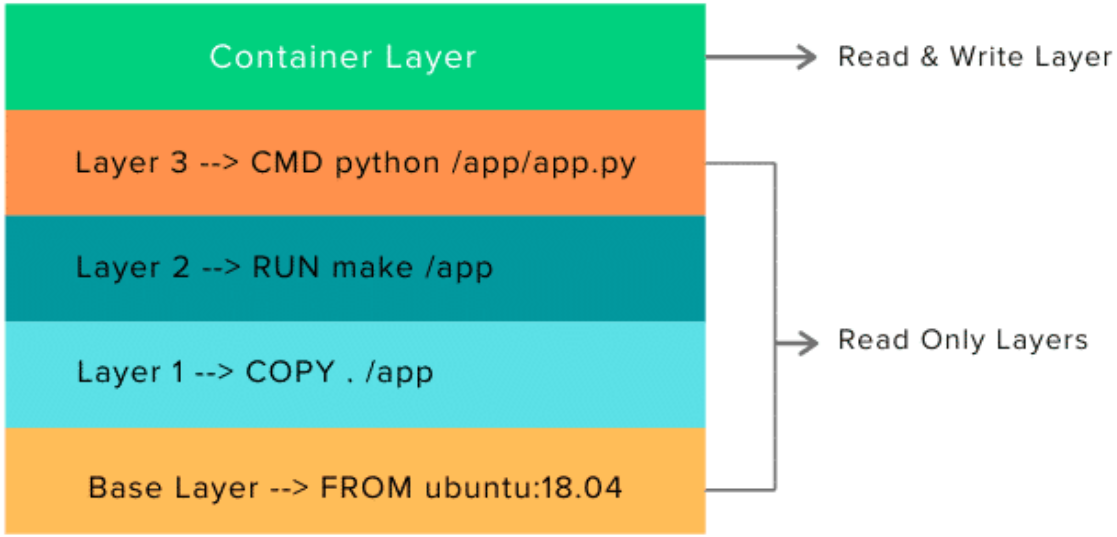


- Select the image tag.

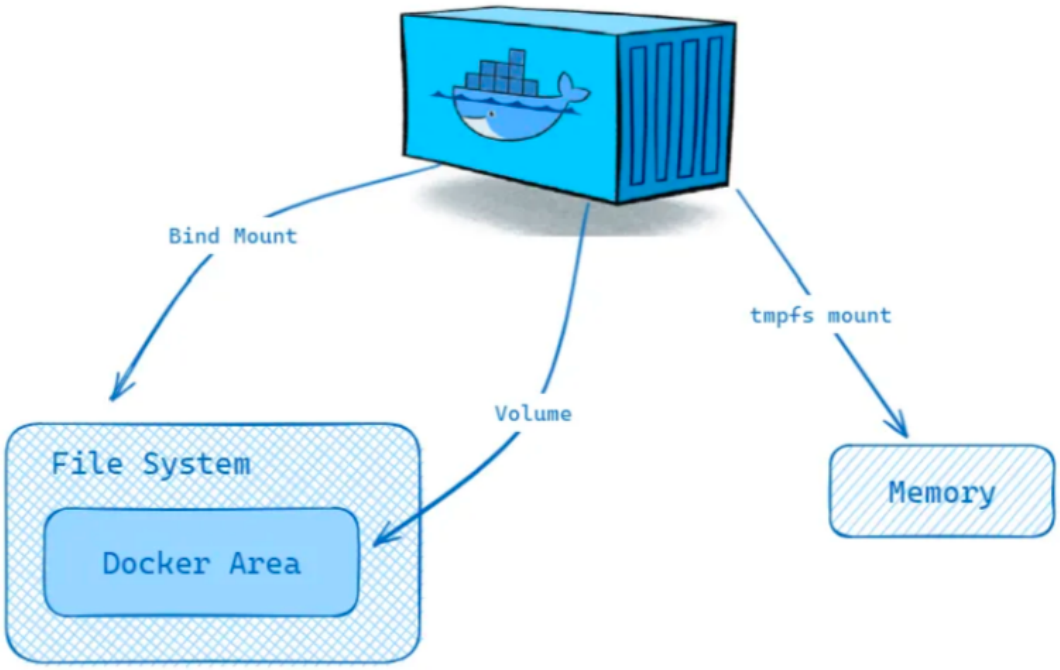


1.9 Storing and sharing data

- By default, all files created inside a container are stored on a **writable container layer**.
  - The **writable layer** is unique per container.



- **Problem 1:** Data written to the container layer doesn't persist *when the container is destroyed*.
- **Problem 2:** You can't easily extract the data from the writeable layer to the host, or to another container.
- **Solution:** Store data outside the writable layer of the container
- In **Docker**, different types of storage mounts exist, i.e.,
  - **Volumes**
  - **Bind mounts**
  - **tmpfs mounts**



- Each of them work differently and has different use cases.
- In what follows, we will discuss them, and explore different use cases and examples.

Volumes

- **Volumes** are persistent data stores for containers, created and managed by Docker.

- **Syntax for creating a volume in Docker CLI:**

```
docker volume create <volume-name>
```

Example:

```
docker volume create my-vol
```

- When you create a volume, it is stored within a directory on the Docker host.

- But in order to interact with the data in the volume, you must mount the volume into a container.

  Syntax:

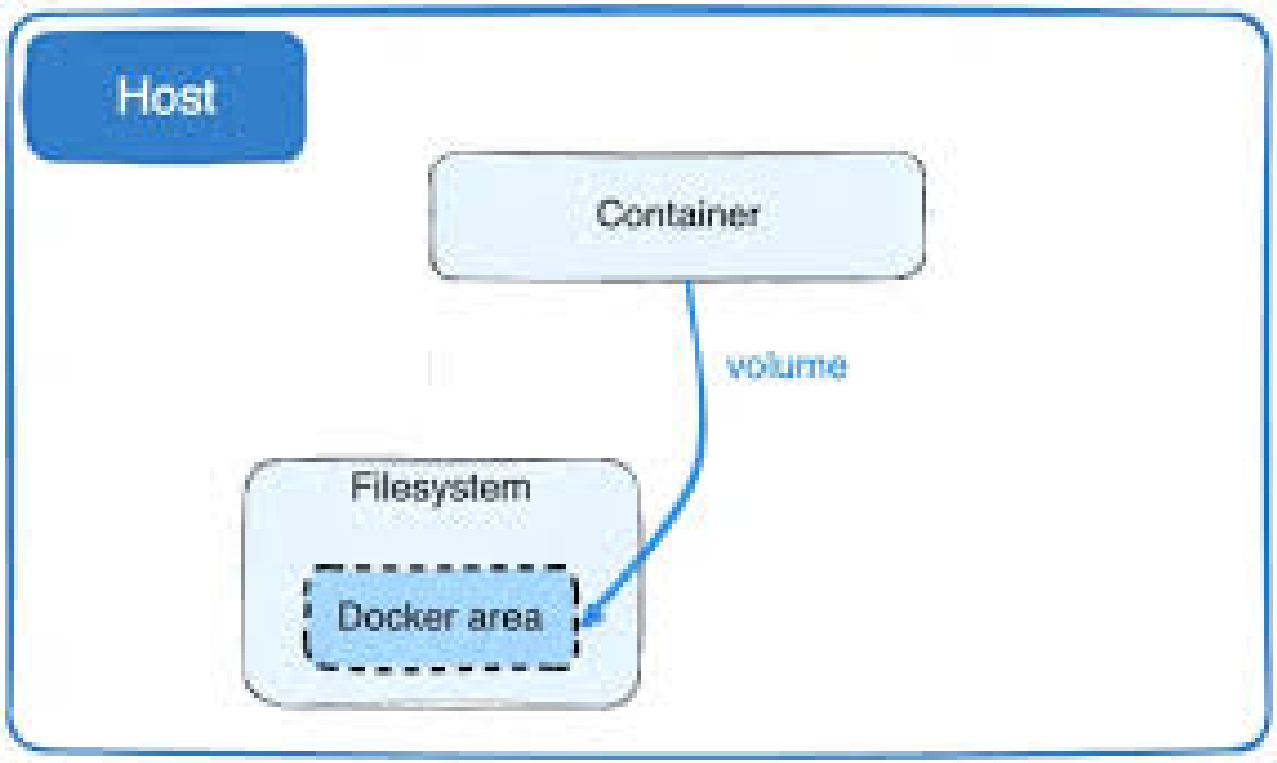
  ```
  docker run -d -v <volume-name>:<mount-path> <image-id>
  ```

  Example:

  ```
  docker run -d -v my-vol:/app/data my-image
  ```

- These volumes retain data even after the containers using them are removed.

- **Volume** is managed by Docker, and are isolated from the core functionality of the host machine.



- To manage volumes, **Docker CLI** can be used, e.g.,
  - **List volumes**: `docker volume ls`

```
local               my-vol
```

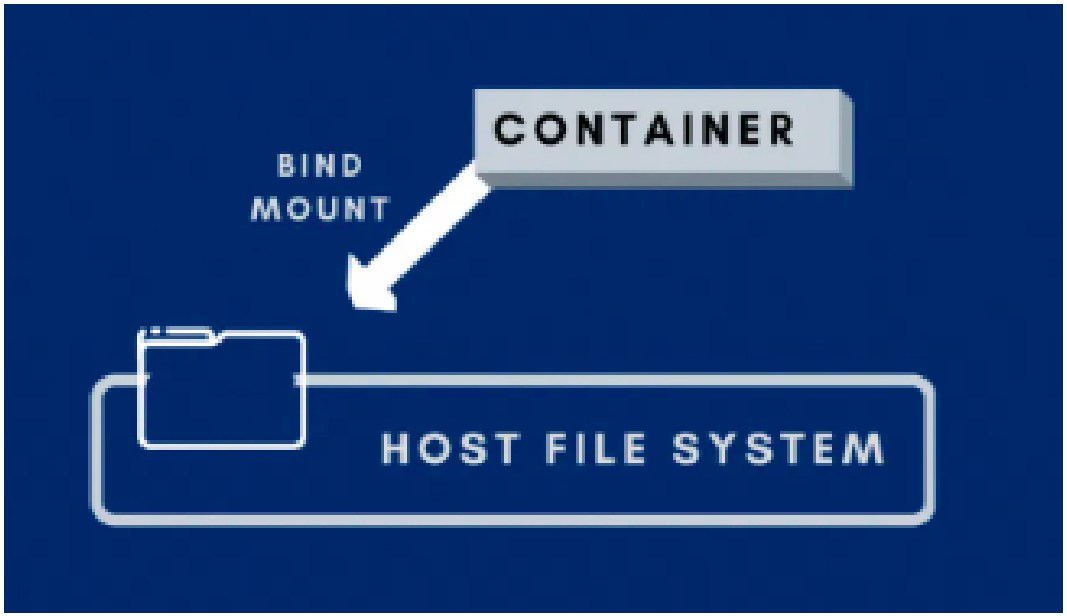
  - **Inspect a volume**: `docker volume inspect my-vol`

```
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

  - **Remove a volume**: `docker volume rm my-vol`

Bind mounts

- **Bind mounts** create a direct link *between a host system path and a container*, allowing access to files or directories stored anywhere on the host file system.



- Since they aren't isolated by Docker, both non-Docker processes on the host and container processes can modify the mounted files simultaneously.

- **Usage:** When you need to be able to access files from both the container and the host.

- Interesting when developing code locally and testing it inside the container

- **Syntax:**

```
docker run -v <host-path>:<container-path> <image-id>
```

Example:

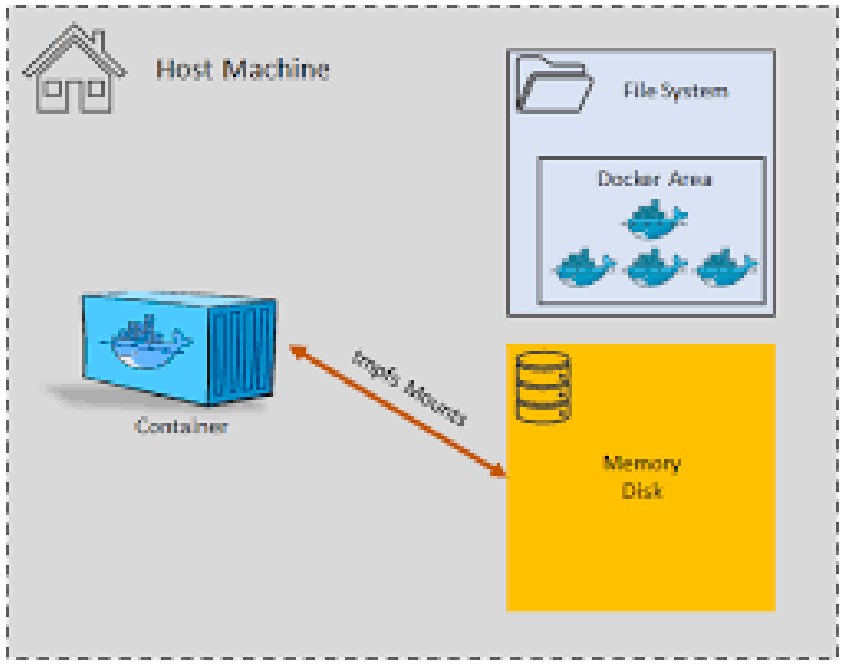
```
docker run -v ${PWD}:/app/data my-image
```

  - ${PWD} means "Print Working Directory". Alternatively, you can also write the full path directly.

  - Mount it inside the container at /app/data

tmpfs mounts

- A **tmpfs mount** stores files directly in the host machine's memory, ensuring the data is not written to disk.



- This storage is ephemeral, i.e., the data is lost when the container is stopped or restarted, or when the host is rebooted.

- **tmpfs mounts** do not persist data either on the Docker host or within the container's filesystem.

- These mounts are suitable for
  - Scenarios requiring temporary, in-memory storage, such as catching intermediate data
  - handling sensitive information like credentials
  - Fast testing, since memory is faster than disk

- **Syntax:**

```
docker run --tmpfs <container-path> <image-id>
```

Example:

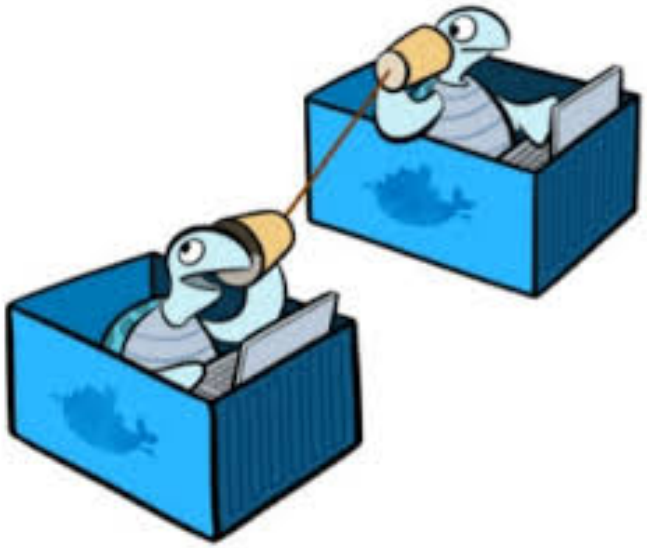
```
docker run --tmpfs /app/logs my-image
```

  - Mount /app/logs inside the container using tmpfs
  - Everything written to /app/logs is stored in RAM
  - Disappears when the container stops

1.10 Networking

- Each Docker container can be seen as an app/service running inside its own "box", e.g.,
  - Box 1 might have your data processing script
  - Box 2 might have a database
  - Box 3 might be a Jupyter notebook interface
- As result, each box is isolated, and operates independently from each other

- **Question:** How can we connect containers with each other, and to the outside world?
  - Your script might need to get data from the database
  - Your Jupyter notebook might want to send commands to the script
  - The script might need to download data from the internet
- **Answer: Docker networking**



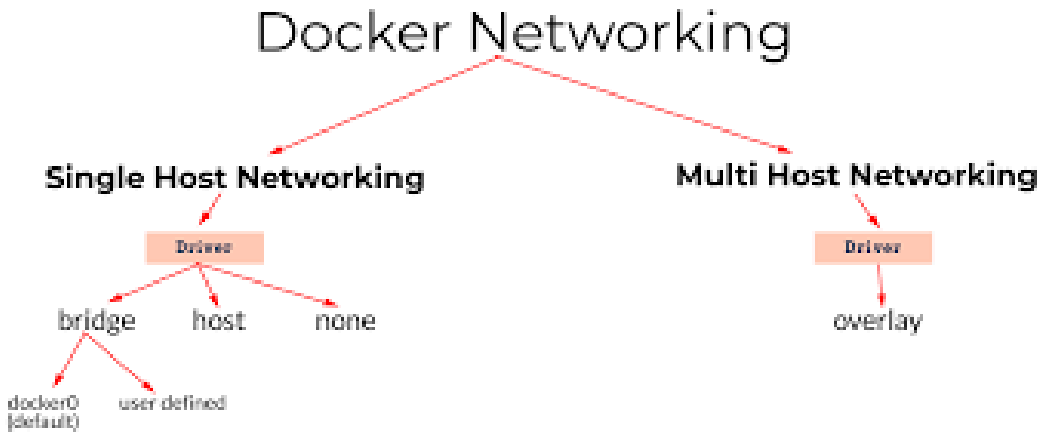
- **Docker networking** allows precise control over container communication
- It can be seen as a **system of invisible cables and routers** that:
  - Connect containers with each other

    → Your script can find the database
  - Connect containers to the internet

    → They can download stuff
  - Control what connections are allowed

    → Things don't accidentally talk to the wrong services
- Without networking, containers are just **silent boxes**!

- Different types of Docker networks exist:
  - **Bridge network** (default and most common)
  - **Host network** (fast but no isolation)
  - **None network** (totally isolated)
  - **Overlay network**



- In what follows, we will discuss them in simple terms!

Think of Docker containers as rooms in a house (your Docker host), and the system of hallways and doors as Docker networking.

Bridge network

- A **bridge network** enables the connection between rooms within a house by means of bridges
- A bridge is like a private hallway connecting these rooms.
- All the rooms can talk to each other if they are on the same hallway (i.e., the same bridge network)
- By default, only rooms connected to the same hallway can talk to each other, and not to the outside world.
- **Question:** Is there a way to access several rooms (containers) from the outside?

- **Answer:** Yes, by **exposing/publishing a port**
  - If you install a doorbell (publish a Docker port, open to the world), accessible at the front entrance by outside persons, Docker knows which room (container) to send the message to (port inside the container)
  - **Syntax:**

    `docker run --p <host-port>:<container-port> <image-id>`

    Example:

    ```
    docker run --p 8888:80 my-image
    ```

    - **Port 8888** on your computer (Docker host) is open to the world
    - Any request to **localhost:8888** get forwarded to **port 80** inside the container
  - The container is still on the bridge network, but now we have created a "window"

- Docker uses bridge networking *by default*
  - When installing Docker, a predefined default bridge network called "`bridge`" is created

```
$ docker network list
NETWORK ID          NAME                DRIVER              SCOPE
6f23ed77734d        bridge              bridge              local
3bbca0d09738        host                host                local
a67ca10d6dfd        none                null                local
```

  - When you don't specify a specific network to a container, e.g.,

```
docker run hello-world
```

    Docker automatically connects the container to this bridge network

- Alternatively, you can also explicitly create a custom bridge network, e.g., with name "`my_bridge`":

```
docker network create my_bridge
```

- To use this custom bridge network, and share it among different containers, e.g., Jupyter notebook and a Postgres database, the following syntax can be used:

```
docker run -d --name X1 --network my_bridge jupyter/base-notebook
docker run -d --name X2 --network my_bridge postgres
```

- The Jupyter container can now talk to Postgres using the name "`X2`"

Host network

- In the **host network**, the containers share the house's address directly
- Thus, in contrary to bridge networking, a container doesn't got its own hallway/room, but it lives right in the main part of the house
- It uses the host's network directly, meaning no isolation, same IP and ports as the host.

```
$ docker network list
NETWORK ID      NAME      DRIVER    SCOPE
6f23ed77734d    bridge    bridge    local
3bbca0d09738    host      host      local
a67ca10d6dfd    none      null      local
```

- **Syntax:**

```
docker run -d --name <name-id> --network host <image-id>
```

Example:

```
docker run -d --name postgres --network host postgres
```

None network

- In a **none network**, every container is locked in a soundproof, sealed room in the house

- The container has no network access at all

- Can't talk to the host, internet, or other containers

```
$ docker network list
NETWORK ID          NAME                DRIVER              SCOPE
6f23ed77734d        bridge              bridge              local
3bbca0d09738        host                host                local
a67ca10d6dfd        none                null                local
```

- **Syntax:**

```
docker run -d --name <name-id> --network none <image-id>
```

Example:

```
docker run -d --name postgres --network none postgres
```

Overlay network

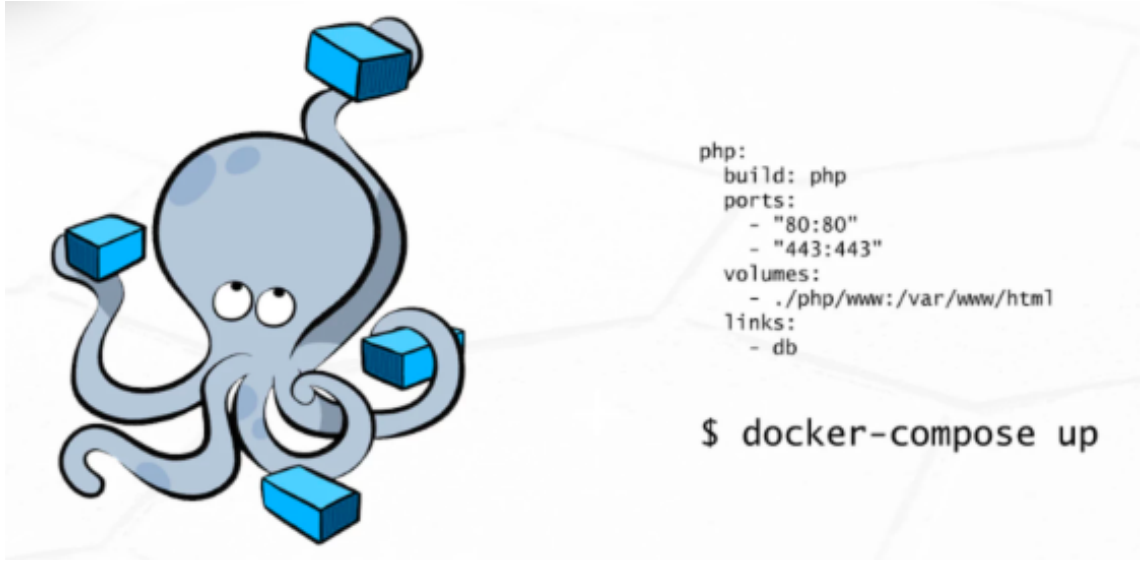
- In an **overlay network**, rooms can come from different houses (hosts), but have a private communication line
- Used in Docker Swarm, i.e., a tool for managing a cluster of Docker nodes (hosts).
- It is like building a private telecom network so rooms in different houses can talk as if they are in the same house
- Link for more information:
  $https://docs.docker.com/engine/network/drivers/overlay/$

1.11 Managing multi-containers

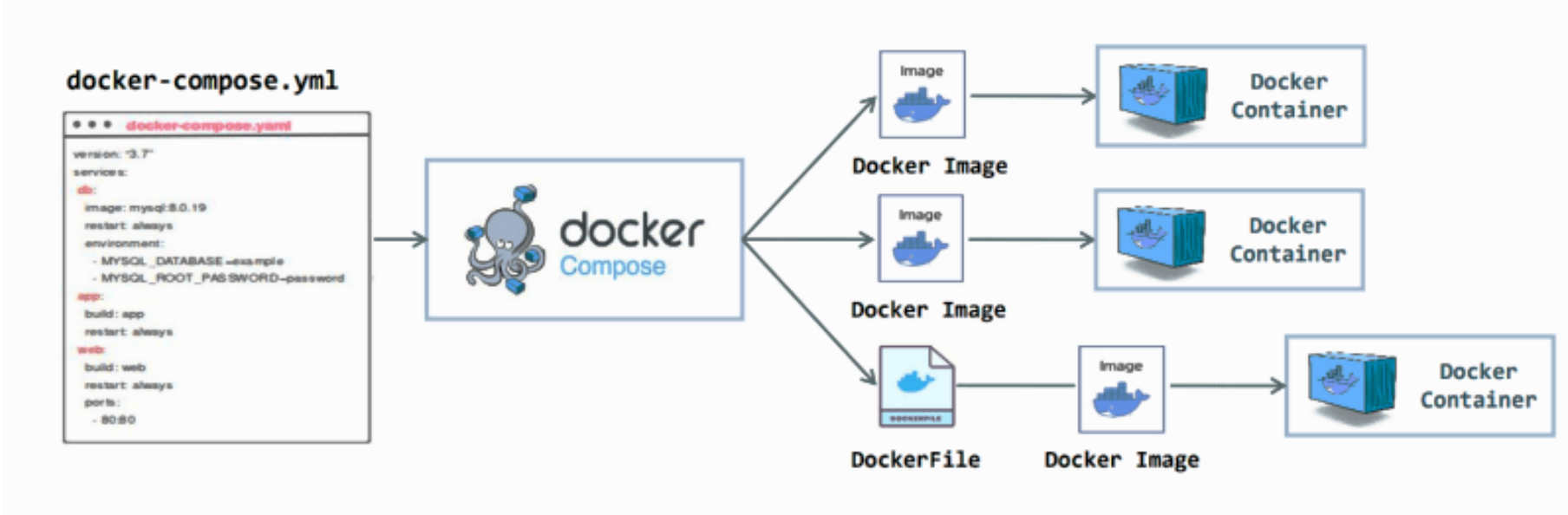
- When your project starts to grow with different containers (for example, Jupyter notebook + a database + a backend), you don't want to start each container manually with long `docker run` commands.

- **Question:** Is there a method that launch all containers with just one command?

- **Solution: Docker compose**



- **Docker compose** is a tool that lets you define your whole setup in a simple "compose" file (`docker-compose.yml`), and then launch everything with **just one command**.



- Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (`docker-compose up`).

- To stop them, you can use the following command:

```
docker-compose down
```

- **Docker compose** has different benefits, including:
  - **Reproducibility**: It works the same every time, everywhere
  - **Collaboration**: Your teammates can start working immediately
  - **Scaling your project**: You can go from one container to many services, or from your laptop to the cloud, without rewriting your project

- In what follows, we will setup a simple docker compose file for a single data science container project, and extend it with an additional database container.

Project 1: Setting up 1 container in Docker Compose

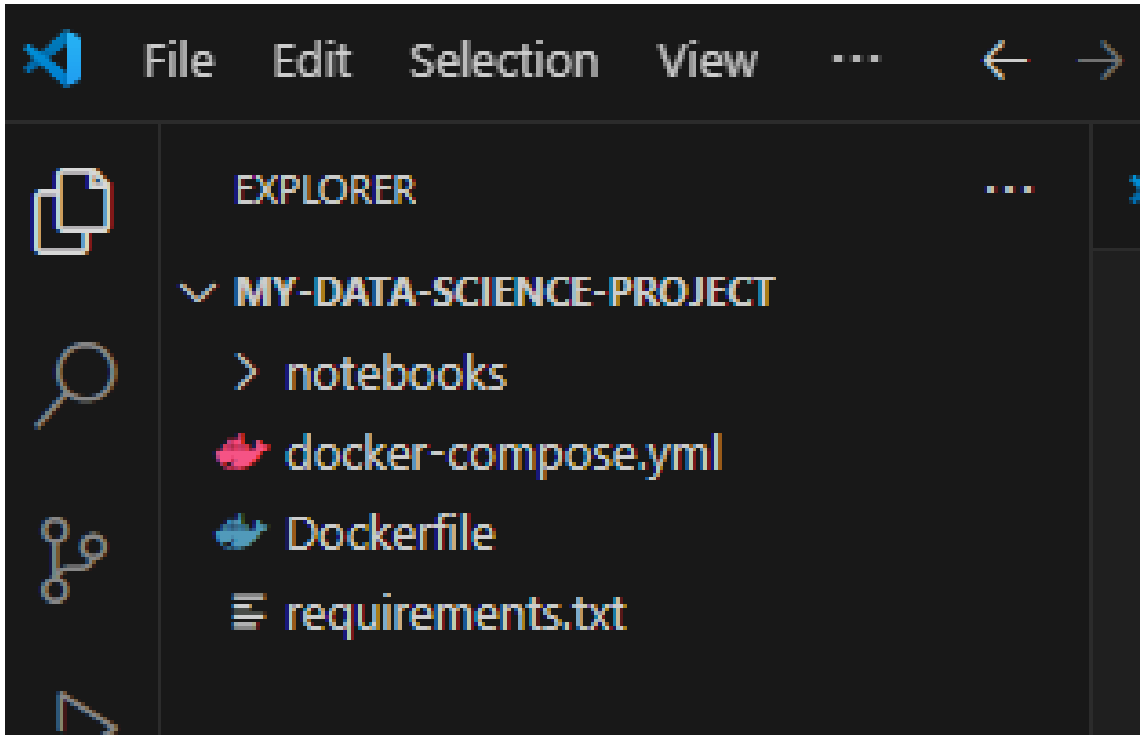
- Here, we would like to setup a container in Docker compose that
  - Run Jupyter Notebooks

    → `jupyter/scipy-notebook` image from Dockerhub
  - Install extra libraries like `xgboost` and `seaborn`

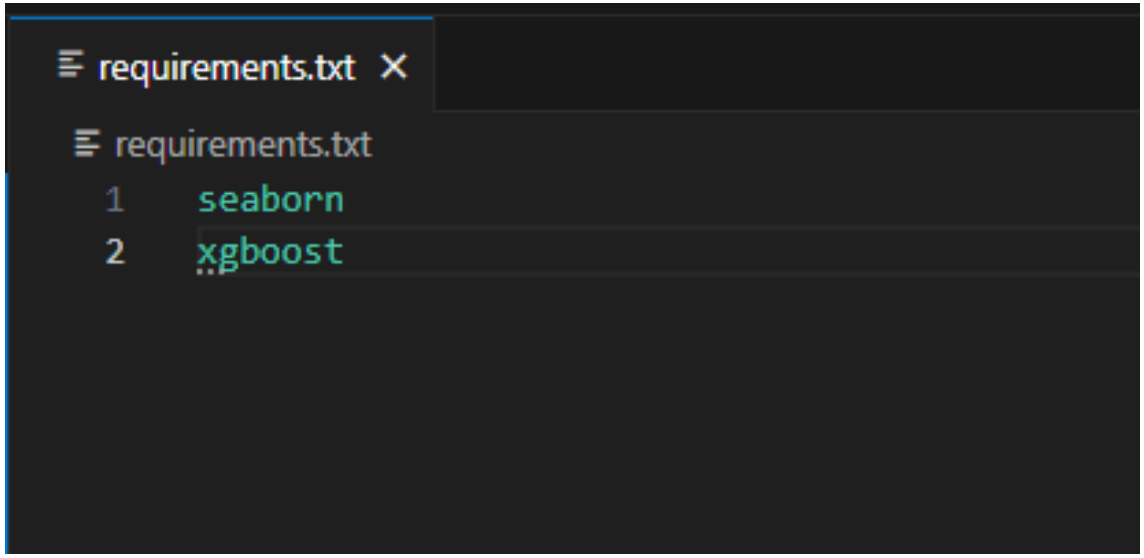
    → Customize the container with Dockerfile
  - Save them locally under `notebooks`

    → Volumes

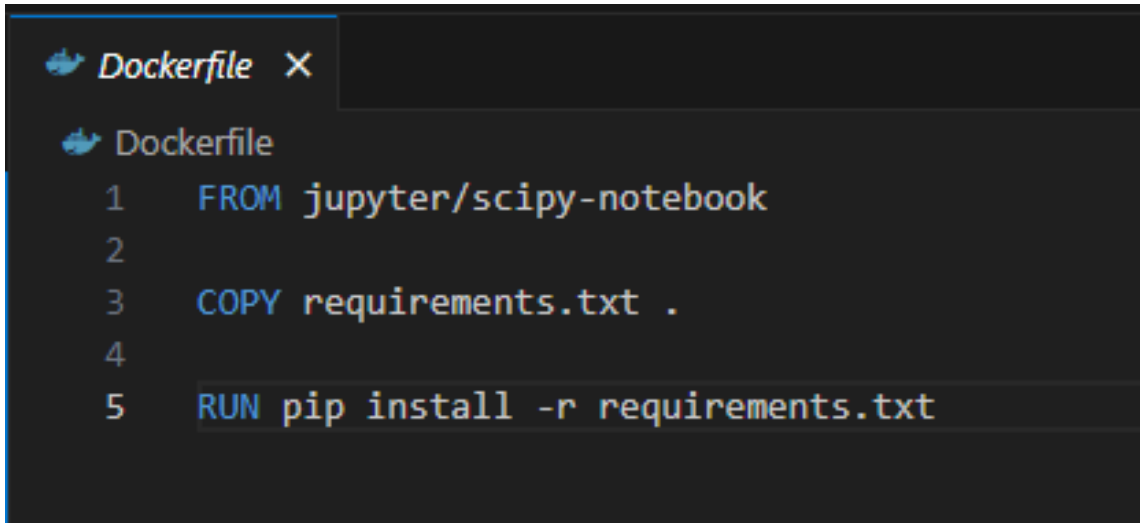
- Folder structure:



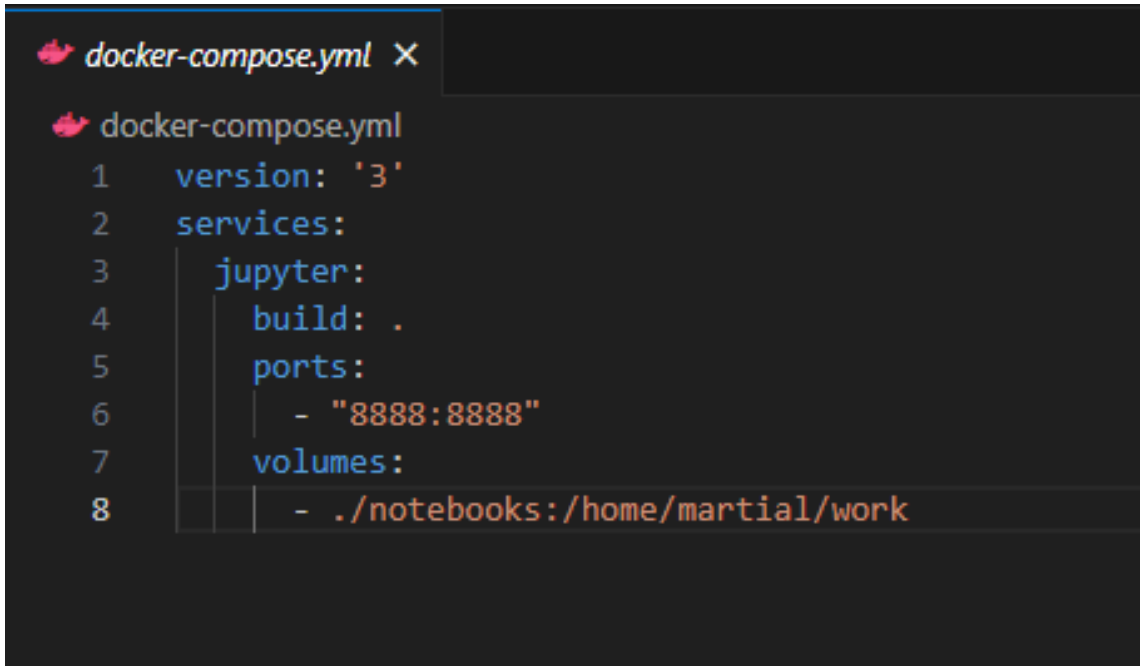
- In the `requirements.txt` file, we are going to include extra libraries



- To create a customized image, i.e., running Jupyter notebooks with extra libraries, we are constructing a Dockerfile as follows:
  - Start with the official `jupyter/scipy-notebook` image from Dockerhub
  - Copy the `requirements.txt` file into the container
  - Install the extra libraries



- We will now define the container within `docker-compose.yml` that runs the following project:
  - Build the image using the Dockerfile
  - Open Jupyter on `port 8888` (so you can visit it in your browser)
  - Use volumes to save notebooks locally in the notebooks folder on your computer



- To use it, run the following command in your terminal:

  ```
  docker-compose up
  ```

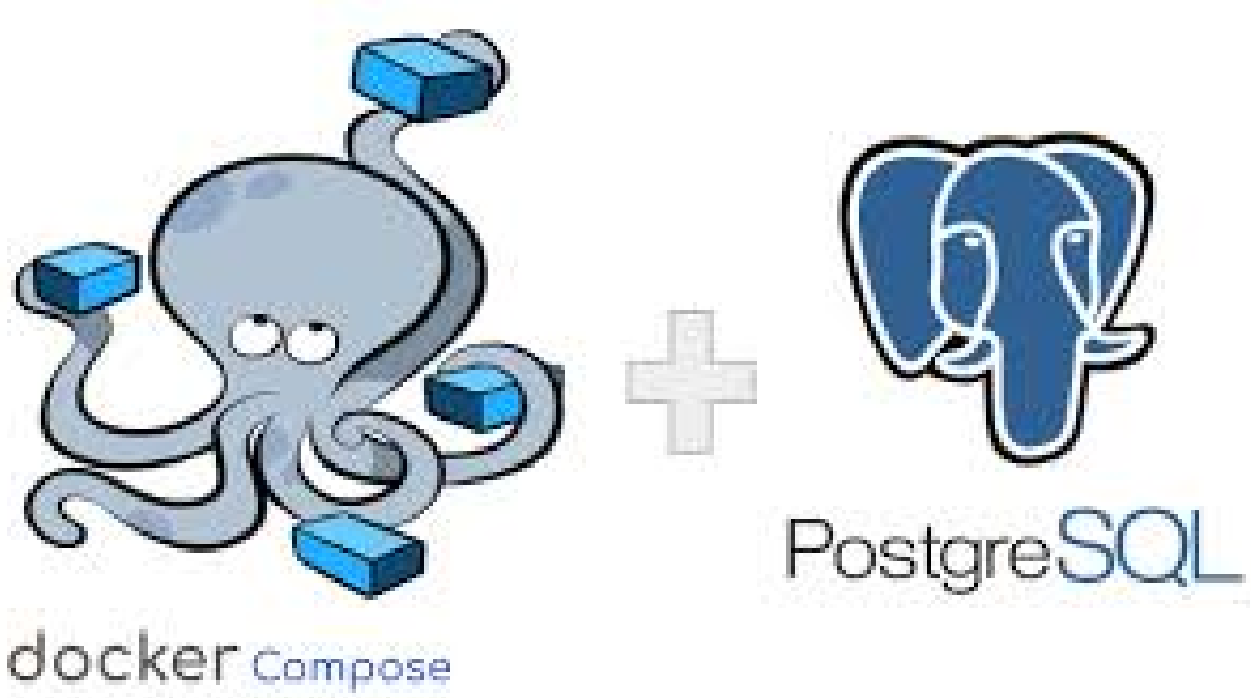
  You will see logs in the terminal, and it will show a Jupyter Notebook URL with a token. Open that in your browser!

- When you are done, close it as follows:

  ```
  docker-compose down
  ```

Project 2: Setting up 2 containers in Docker Compose

- Extend Project 1 by adding the option to store data in a powerful, open source object-relational database, with the PostgreSQL database container



- We extend `docker-compose.yml` by adding the official PostgreSQL database image

docker-compose.yml

```yaml
version: '3'
services:
  jupyter:
    build: .
    ports:
      - "8888:8888"
    volumes:
      - ./notebooks:/home/martial/work
    depends_on:
      - db

  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
    ports:
      - "5432:5432"
```

- In Docker compose, the `depends_on` field is used to express the dependencies between the services (images) and specify the order in which these services should be started and stopped.
- Here, the `jupyter service` depends on the `db service`
- When using the `docker-compose up` command, Docker compose will start the `db service` first, then start the `jupyter service`

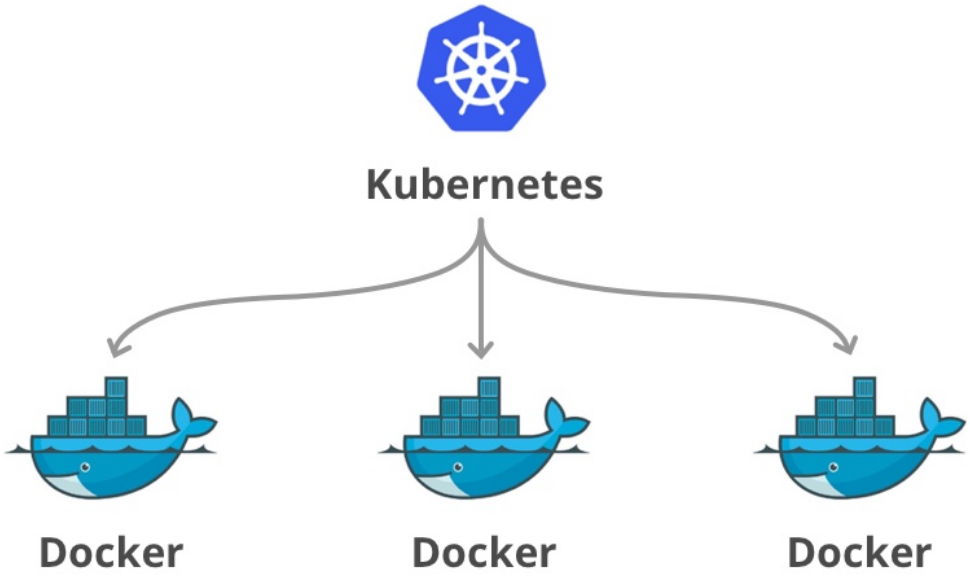
- Link for more information: $https://docs.docker.com/compose/$

Part 2:

Introduction to Kubernetes

2.1 What is Kubernetes?

- **Reminder: Docker compose** is a tool that lets you define and run multi-container Docker applications on 1 host using a **single YAML file**
- **Kubernetes** takes that idea and scales it up to many machines
- In other words, it is a system to manage many containers, under different hosts.



- **Question:** Why is this useful?

- Imagine you don't have just one container, but hundreds, and you wish to:
  - Launch them across many computers
  - Restart them if they crash
  - Scale them up or down
  - Manage updates smoothly

- **Problem:** Handling all these things manually is hard!

- **Kubernetes** automates this, and can be seen as the shipping logistic system, which decides when, where, and how containers run across a fleet of machines.

- In other words, while **Docker compose** helps you *build and test apps locally*, **Kubernetes** helps you *run and manage them at scale in production*.

- **Remark:** Setting up Kubernetes is hard, and involves much practice!

2.2 Comparison with Docker

Feature	**Docker Compose**	**Kubernetes**
Scale	Single machine	Multi-machine, production-ready
Complexity	Simple, easy to set up	More complex, but powerful
Use case	Local dev, testing	Production, large-scale systems
Configuration	One YAML file	Many YAML files (pods, services, etc.)
Self-healing	No automatic recovery	Automatically restarts crashed containers
Ecosystem	Basic networking and volumes	Full orchestration (load balancing, scaling)

Table 0.1: Comparison between Docker Compose and Kubernetes