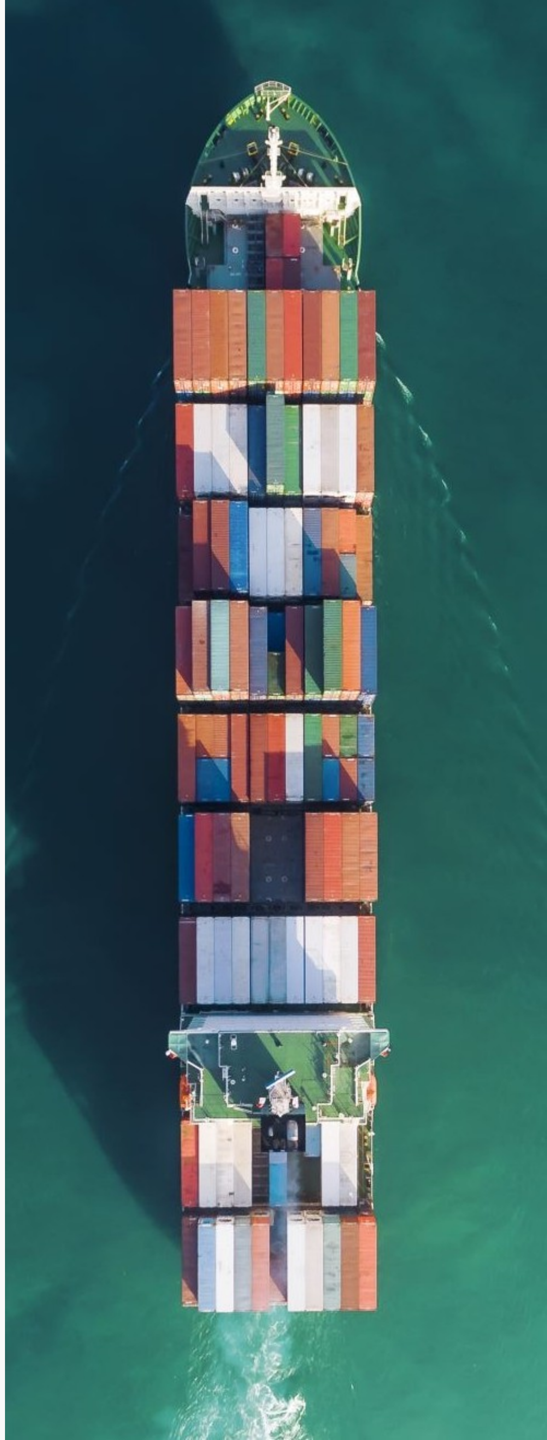
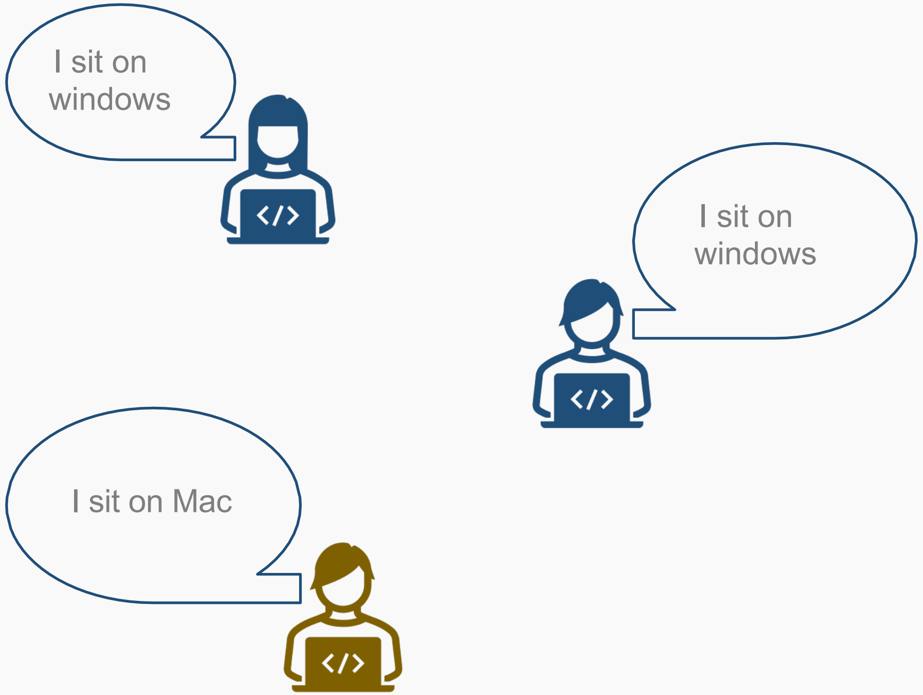
Esoon Ko

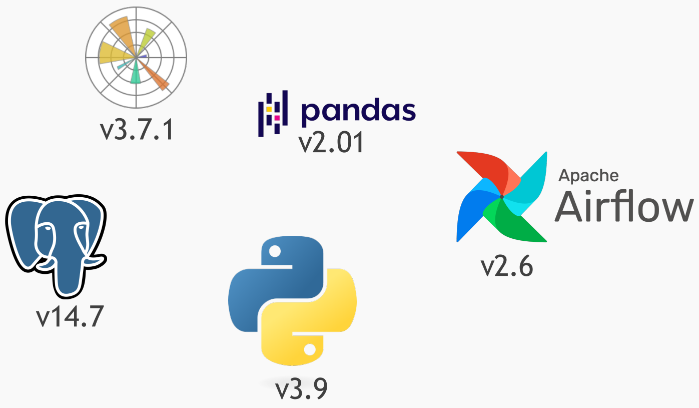
docker containers to isolate environments



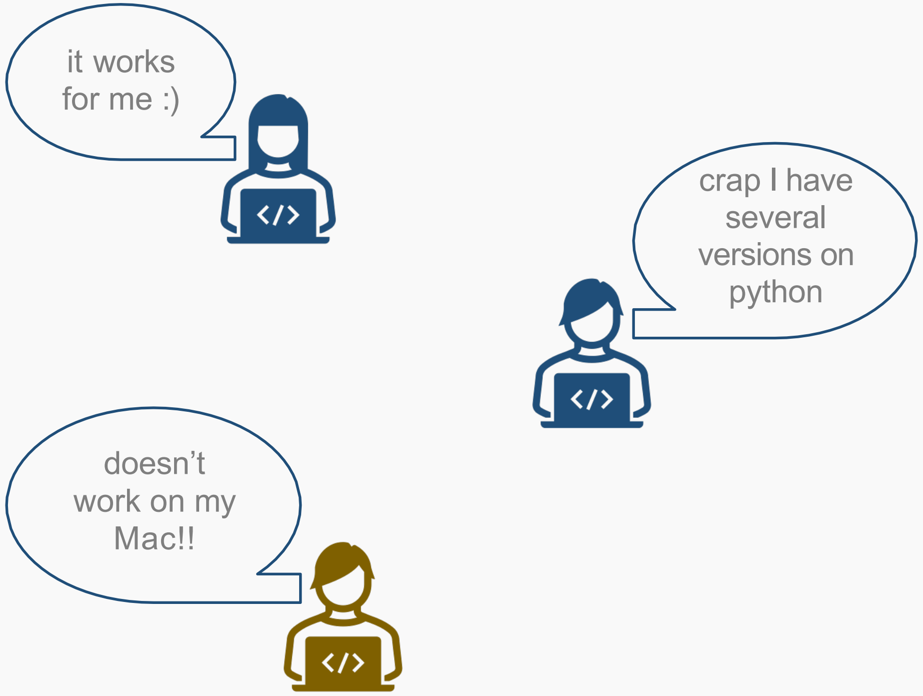
Why **Docker?**



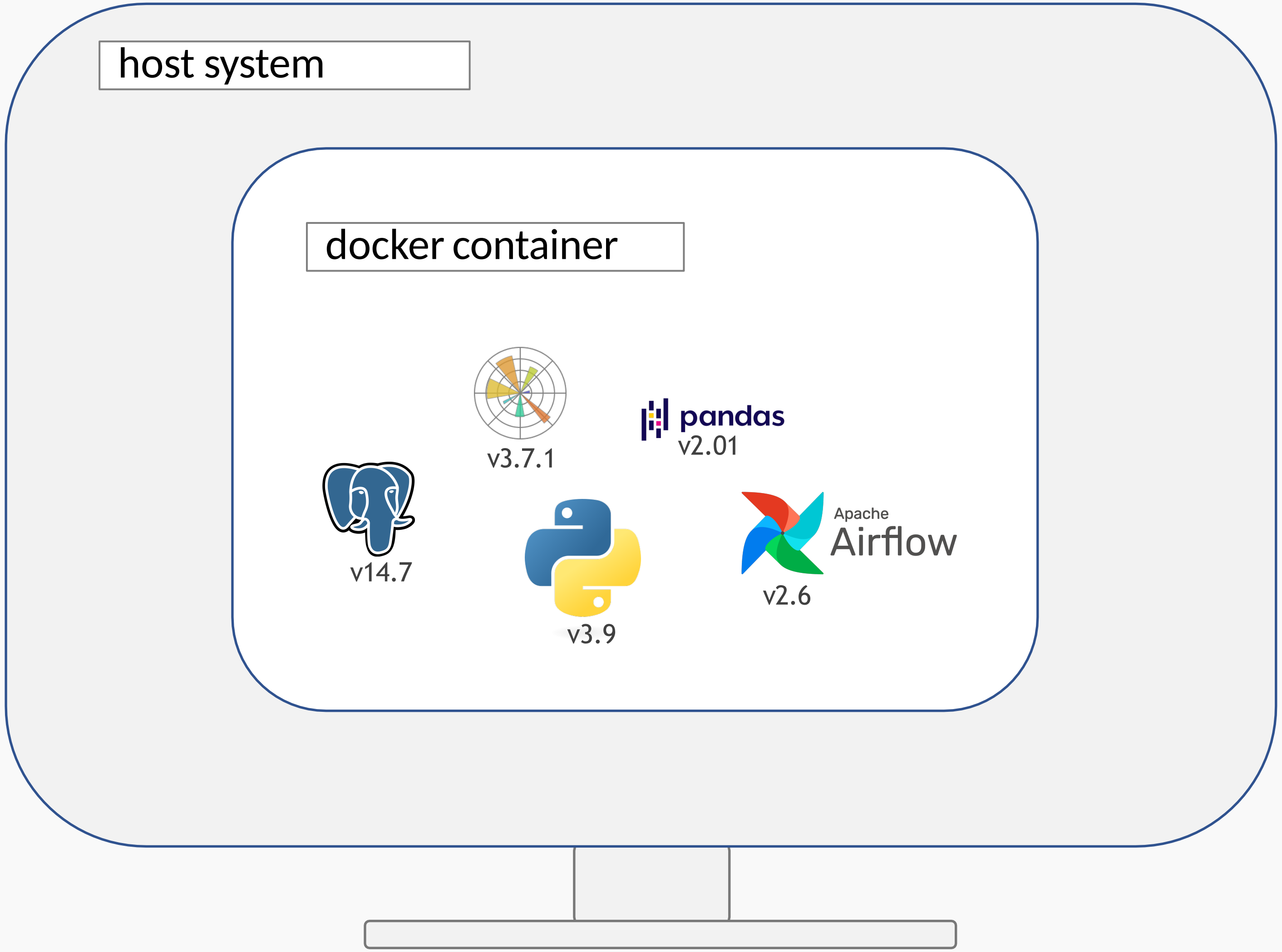
we need to setup the same **development environment**



install different softwares
and packages

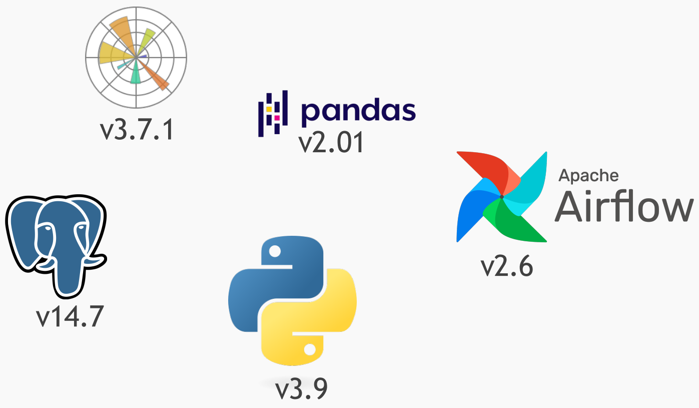


what if there was an **environment** that works for everyone?

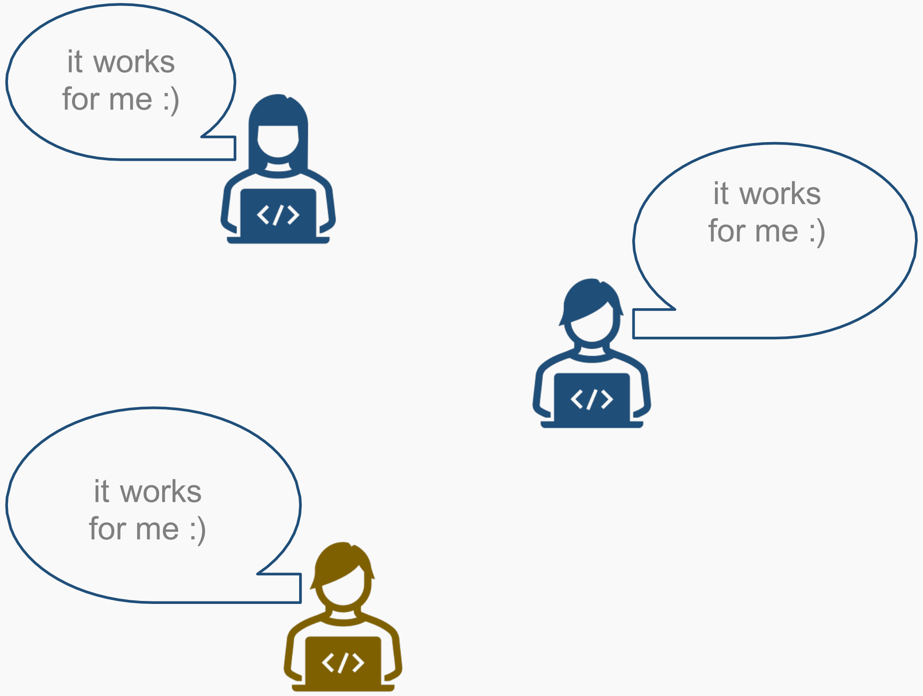


isolated environment that has all dependencies and software for an application

what if there was an **environment** that works for everyone?



install different softwares and packages



So what is a **Docker??**

Docker is a platform designed to help developers build, share, and run applications in containers.

Containers: lightweight, stand-alone, and executable software packages.

Why **Containers** when we have **VM?**
(Is there even a difference…?)

docker

Lightweight virtualization. They package an application and its dependencies into a single unit that can run consistently across different environments.

Provides process and filesystem isolation but share the host operating system’s kernel.

VM

Full virtualization technology where each VM includes a complete operating system, along with virtualized hardware resources managed by a hypervisor.

Fully isolated from each other, with each VM having its own OS kernel.

Containers/VM
Architecture

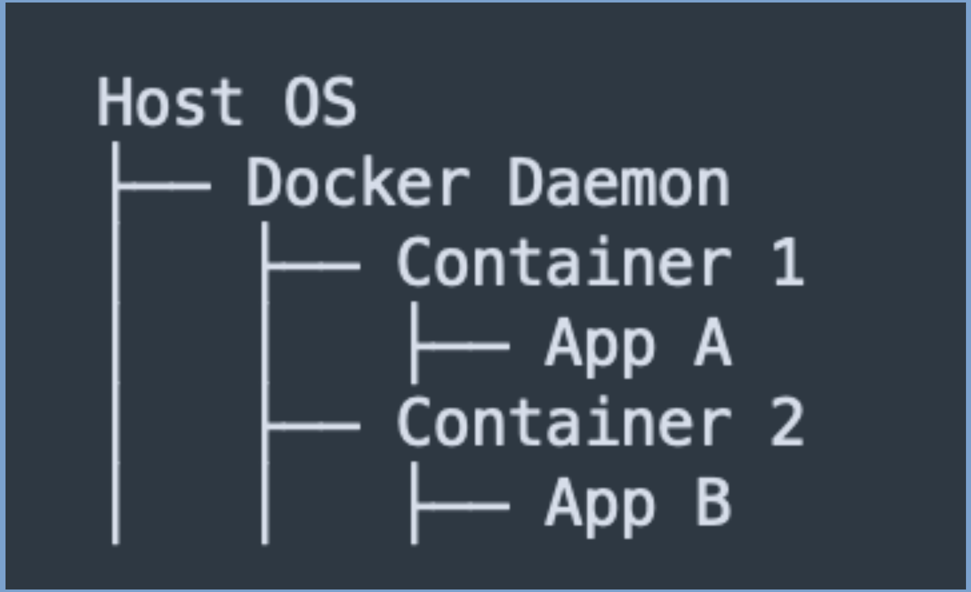
Architecture:
- Host OS: Containers share the host operating system's kernel.
- Container Engine: A container engine (e.g., Docker) manages containers.
- Container: Contains application code, runtime, libraries, and dependencies but not the OS kernel.

VM

Architecture:
- Host OS: The underlying operating system running the hypervisor.
- Hypervisor: Software (e.g., VMware, Hyper-V) that creates and manages VMs.
- Virtual Machine: Includes guest OS, virtual hardware, application code, and dependencies.

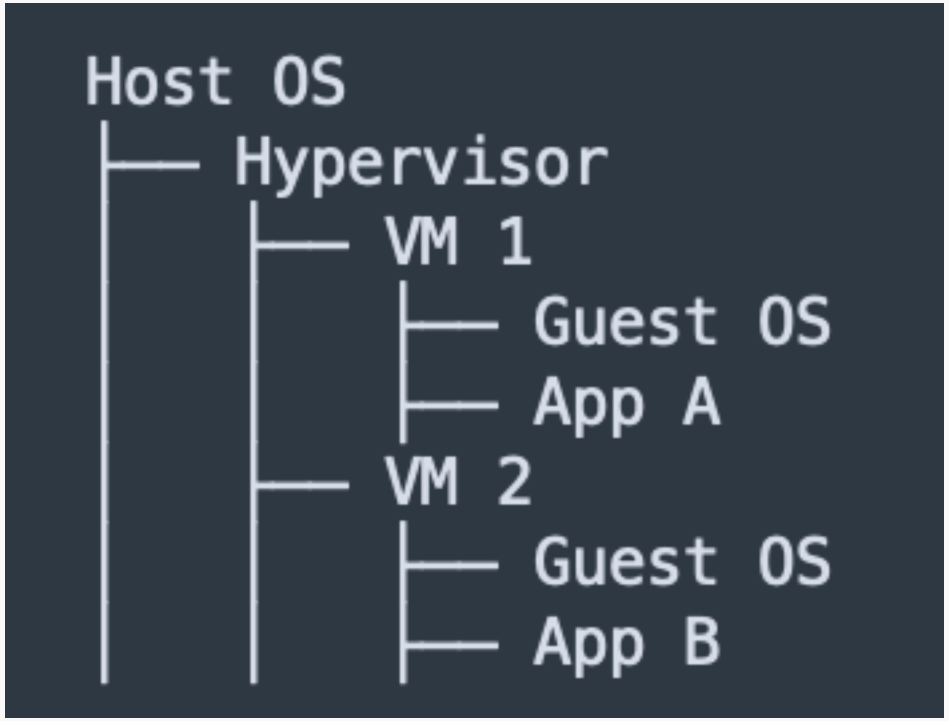
Containers/VM

Architecture



```
Host OS
├── Docker Daemon
│   ├── Container 1
│   │   ├── App A
│   ├── Container 2
│   │   ├── App B
```

VM



```
Host OS
├── Hypervisor
│   ├── VM 1
│   │   ├── Guest OS
│   │   ├── App A
│   ├── VM 2
│   │   ├── Guest OS
│   │   ├── App B
```

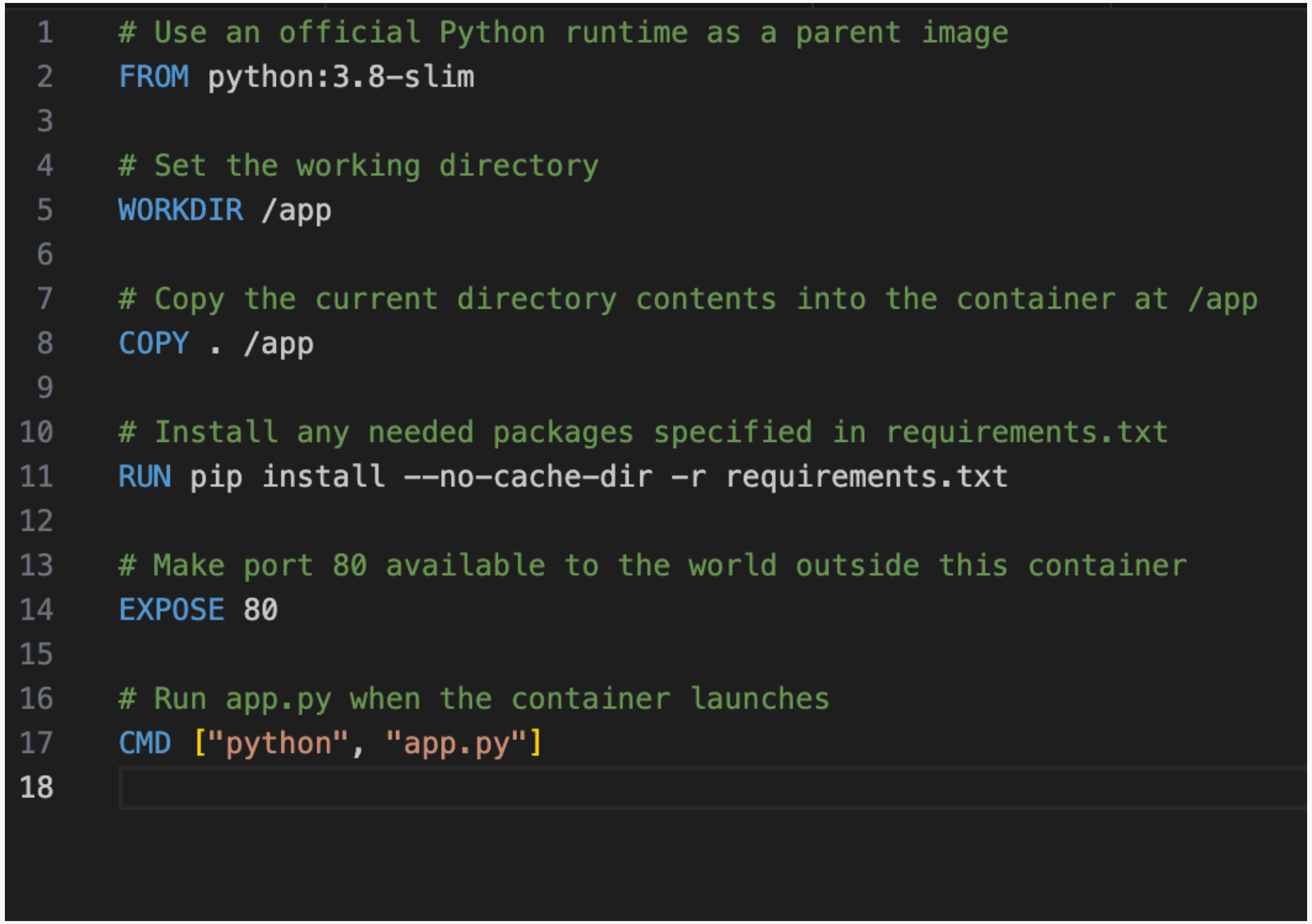
Docker **Images** and **Containers**

Docker image refers to lightweight, standalone, and executable software package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and configuration files.

Characteristics:

- Immutable
- Layered
- Base image (Ubuntu, alpine, node and so on)

Images are made through **Dockerfile**



```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Run app.py when the container launches
CMD ["python", "app.py"]

```

Example of Dockerfile

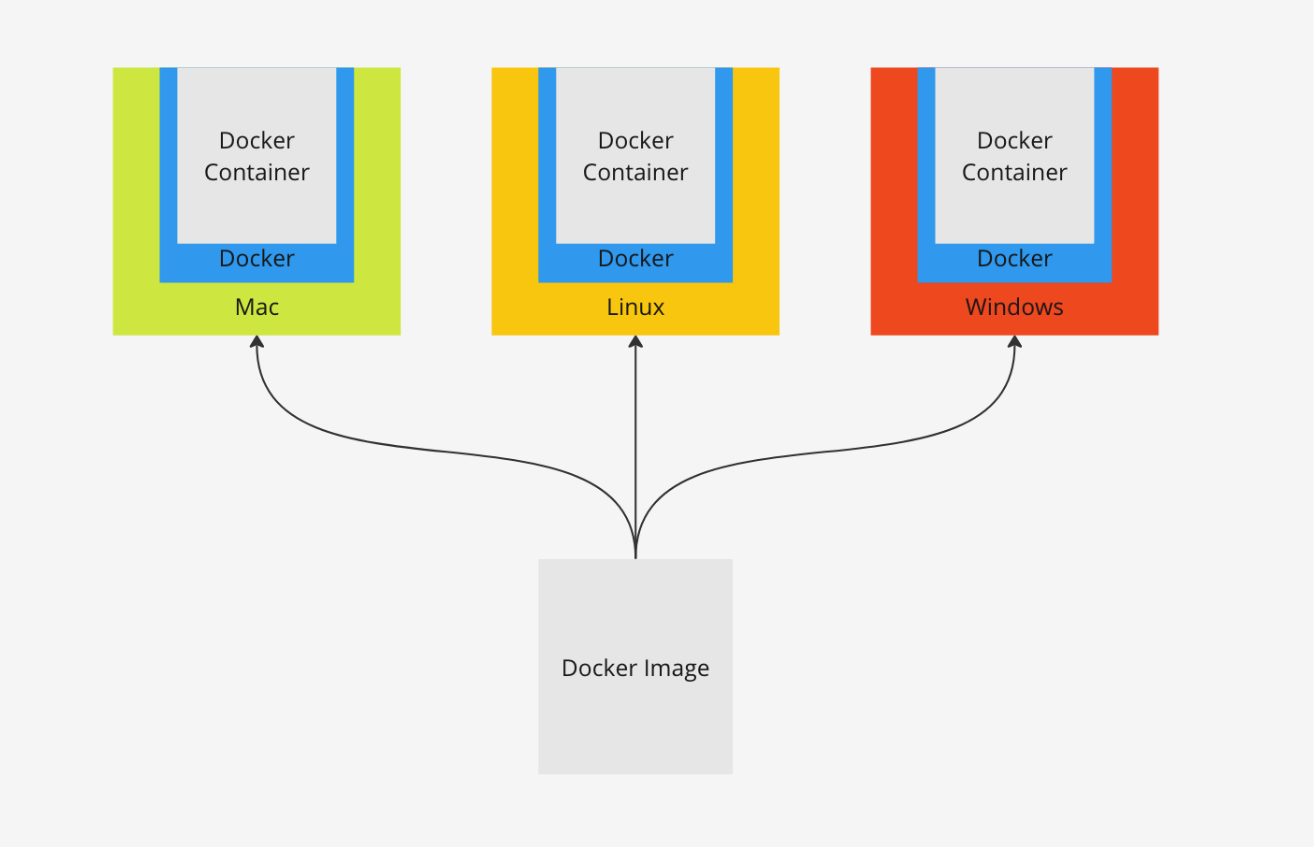
Docker **Images** and **Containers**

Docker container refers to a runnable instance of an image. It encapsulates the application and its dependencies but shares the host system's kernel.

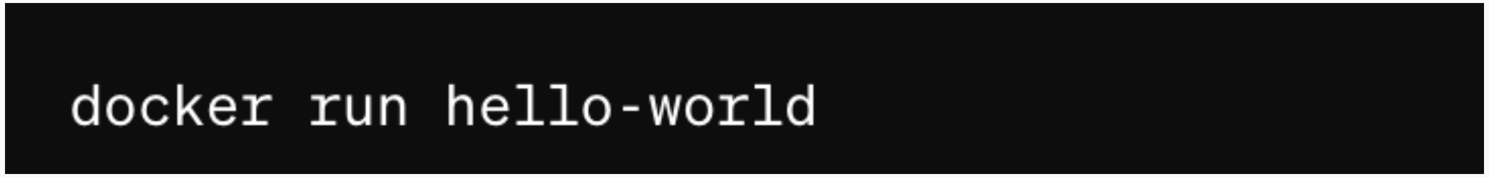
Characteristics:

- Easy to start, stop, move and delete
- Isolated (but shares kernel with host)

You can see images as blueprint for creating containers. You instantiate containers from an image and thereby you can create multiple containers that are the same no matter the environment.

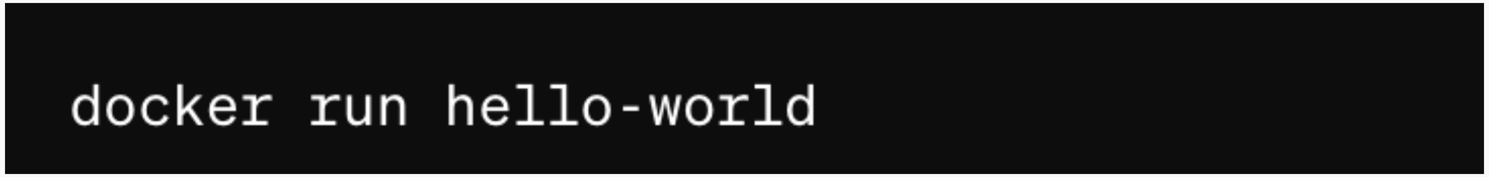


Ok, now I know about **Docker**, when can we start?

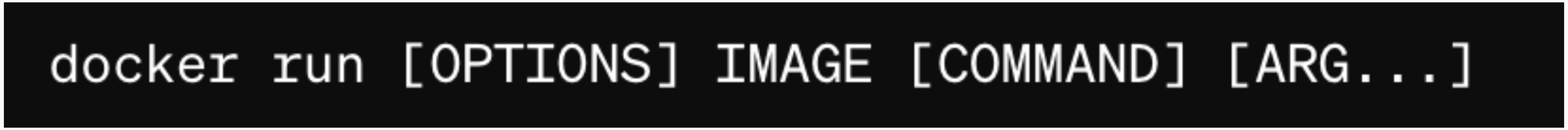


```
docker run hello-world
```

Whats going on here?

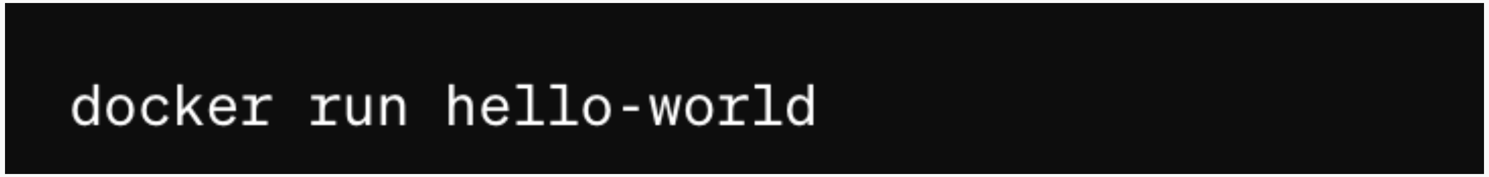


```
docker run hello-world
```

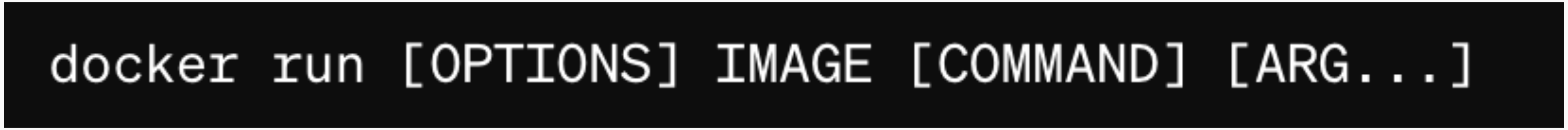


```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

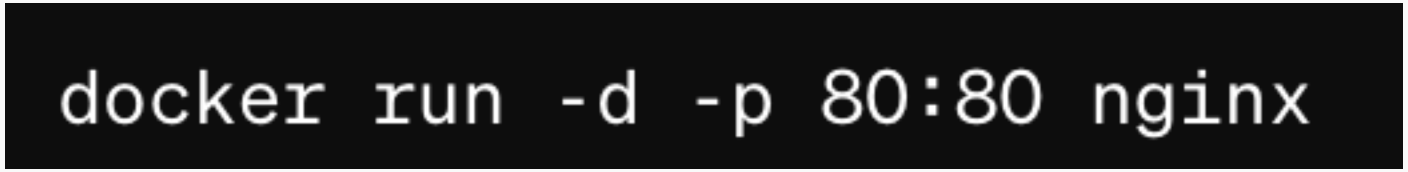
Whats going on here?



```
docker run hello-world
```



```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```



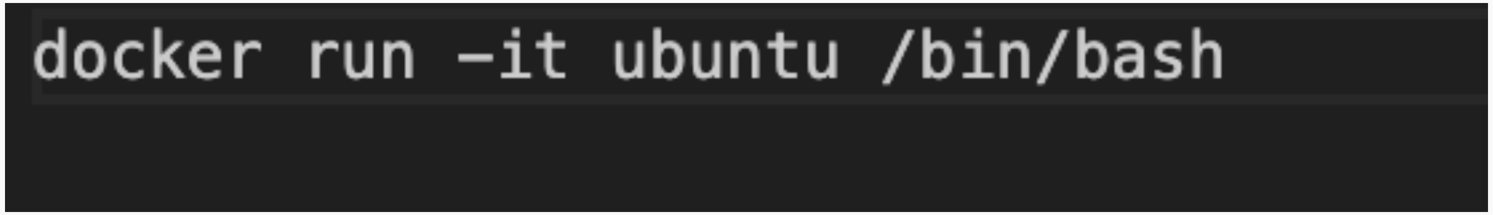
```
docker run -d -p 80:80 nginx
```

Other **Docker** commands

- Listing running containers: docker ps
- Stopping a running container: docker stop CONTAINER_ID
- Removing a stopped container: docker rm CONTAINER_ID
- Removing multiple containers: docker rm $(docker ps -a -q)
- Listing images: docker images
- Removing an image: docker rmi IMAGE_ID
- Pulling an image from Docker Hub: docker pull IMAGE_NAME

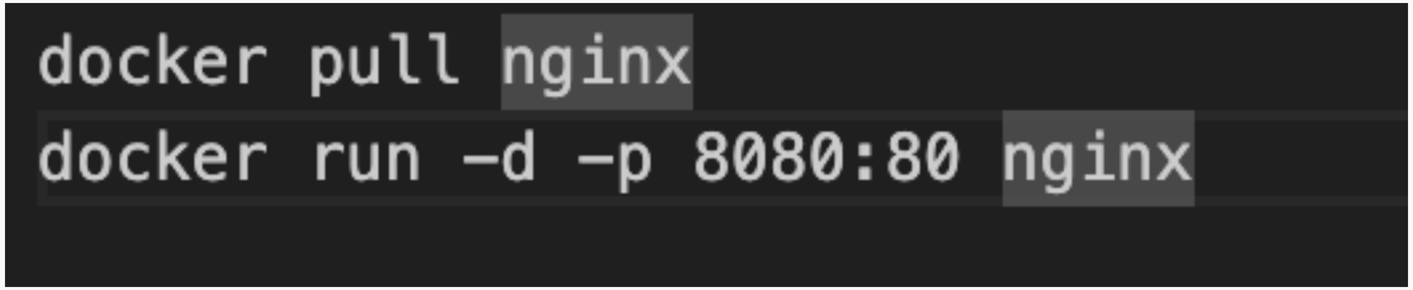
Using Docker for more

Using Docker as Ubuntu Bash



```
docker run -it ubuntu /bin/bash
```

Using Docker as nginx server



```
docker pull nginx
docker run -d -p 8080:80 nginx
```

More here: https://docs.docker.com/reference/cli/docker/
Or run —help

Lets build our first **Dockerfile!**

Dockerfile: File with instructions to build our image

Use a Base Image:
Start with an official Python base image from Docker Hub.

Set Working Directory:
Create and set the working directory inside the container.

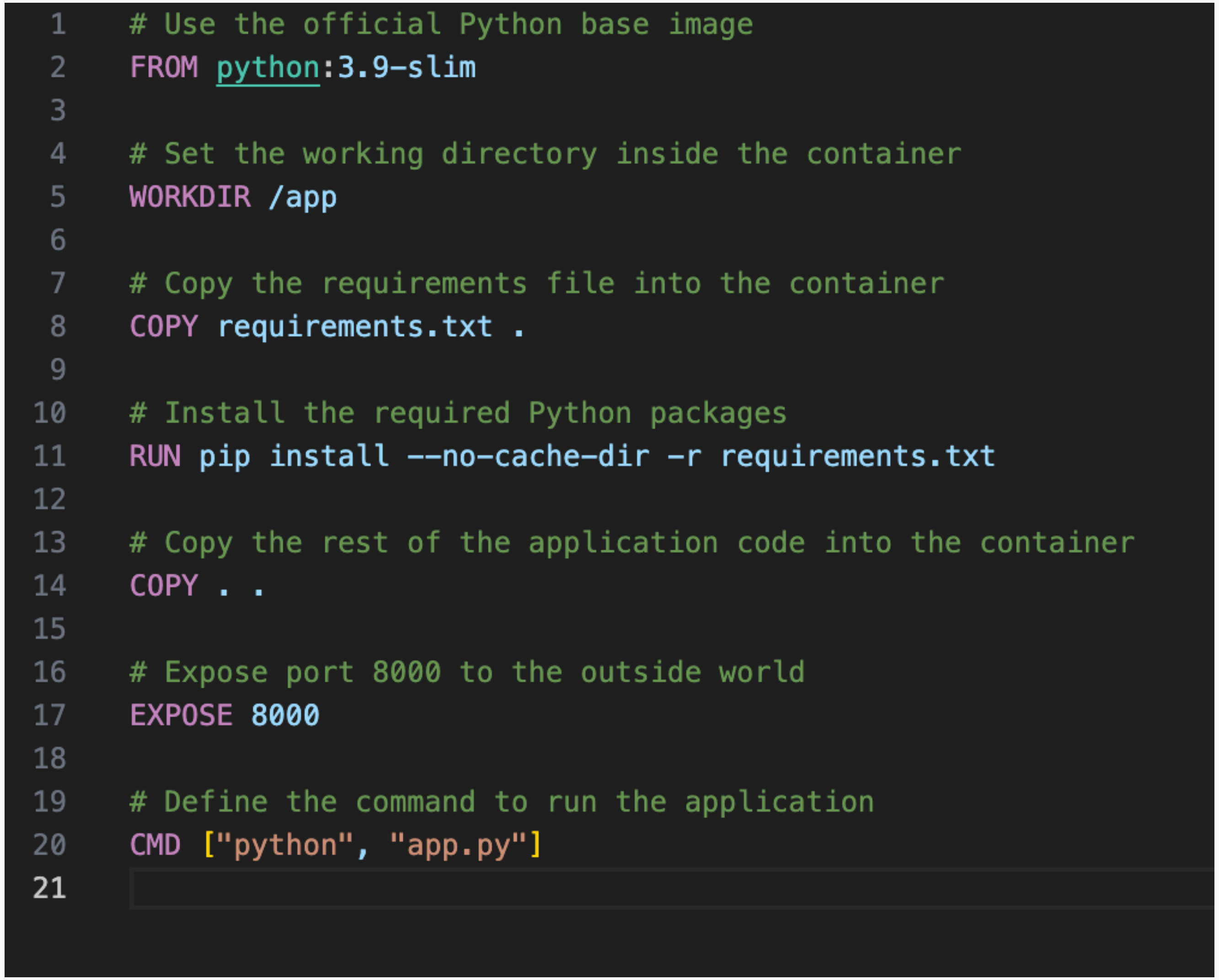
Copy Application Files:
Copy the application code from the host to the container.

Install Dependencies:
Install the Python packages required by the application.

Expose a Port:
Expose the port the application will run on.

Run the Application:
Specify the command to run the application

Lets build our first **Dockerfile!**



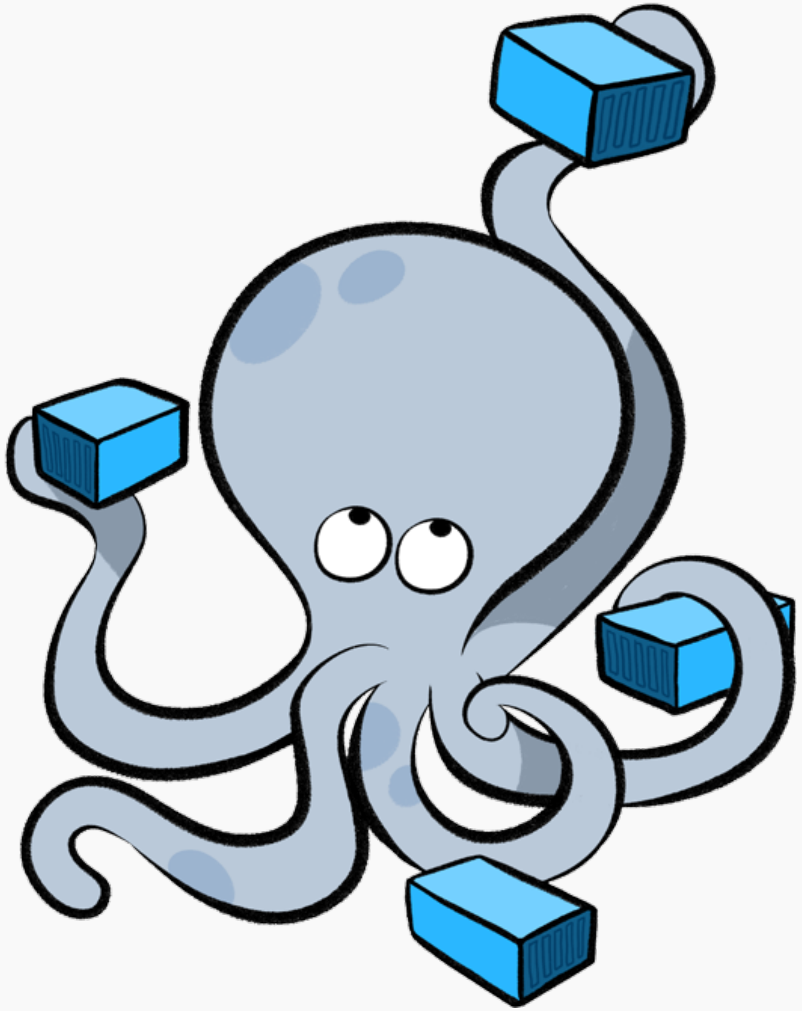
```dockerfile
# Use the official Python base image
FROM python:3.9-slim

# Set the working directory inside the container
WORKDIR /app

# Copy the requirements file into the container
COPY requirements.txt .

# Install the required Python packages
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code into the container
COPY . .

# Expose port 8000 to the outside world
EXPOSE 8000

# Define the command to run the application
CMD ["python", "app.py"]

```

Now I have mastered **Dockerfile!**

Slow down buckaroo

What if we want to manage several docker containers at the same time?

Answer: **Docker Compose**



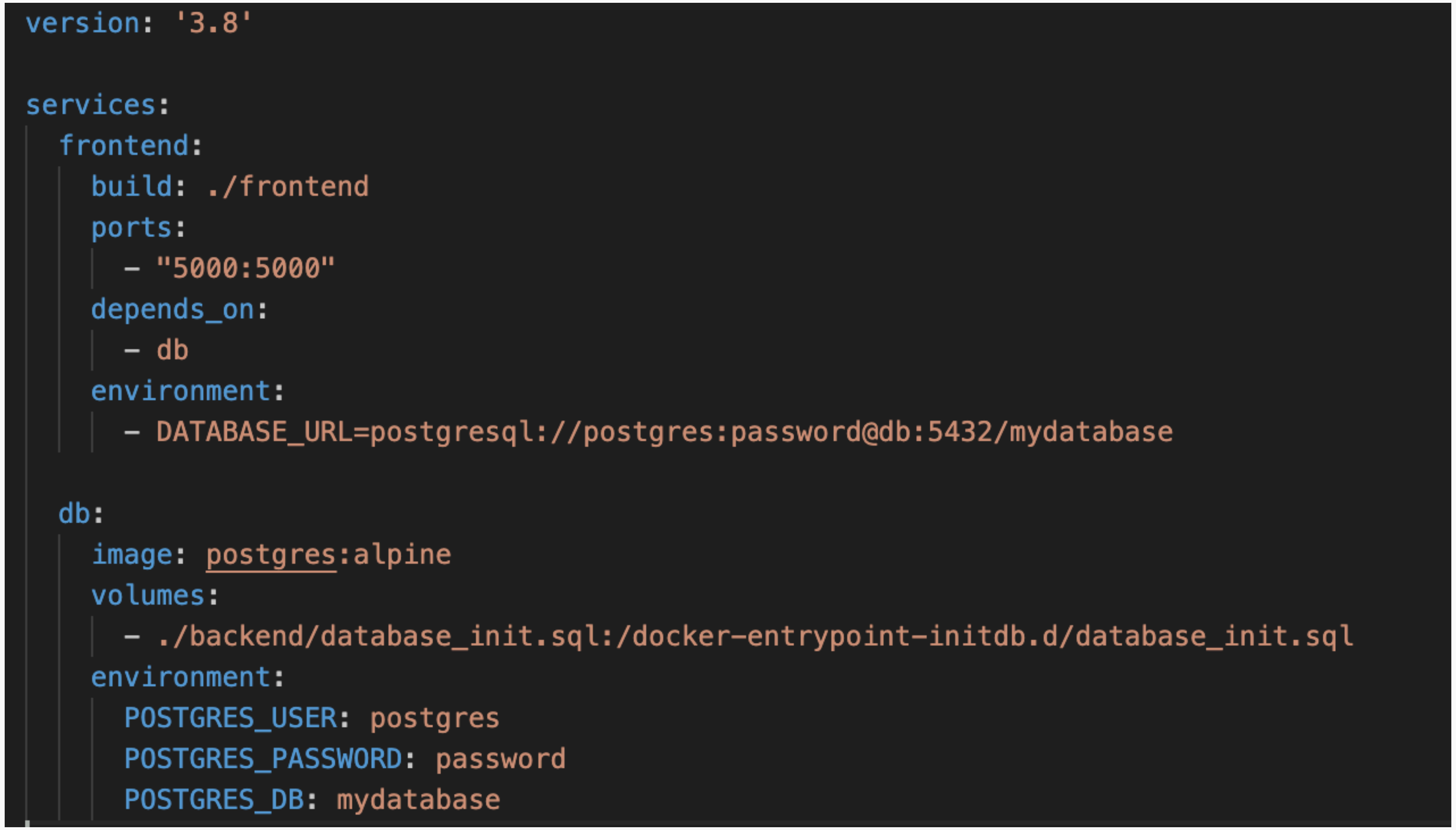
Docker Compose

A tool for defining and running multi-container Docker applications. It allows you to use a YAML file to configure your application's services and their dependencies, then with a single command, you can create and start all the services defined in your configuration.

Done through a YAML file!

Allows for:

- Defining Services
- Linking Services
- Simple Configuration
- Lifecycle Management
- Environment Variables
- Scaling



```yaml
version: '3.8'

services:
  frontend:
    build: ./frontend
    ports:
      - "5000:5000"
    depends_on:
      - db
    environment:
      - DATABASE_URL=postgresql://postgres:password@db:5432/mydatabase

  db:
    image: postgres:alpine
    volumes:
      - ./backend/database_init.sql:/docker-entrypoint-initdb.d/database_init.sql
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
```

Volume and Networking

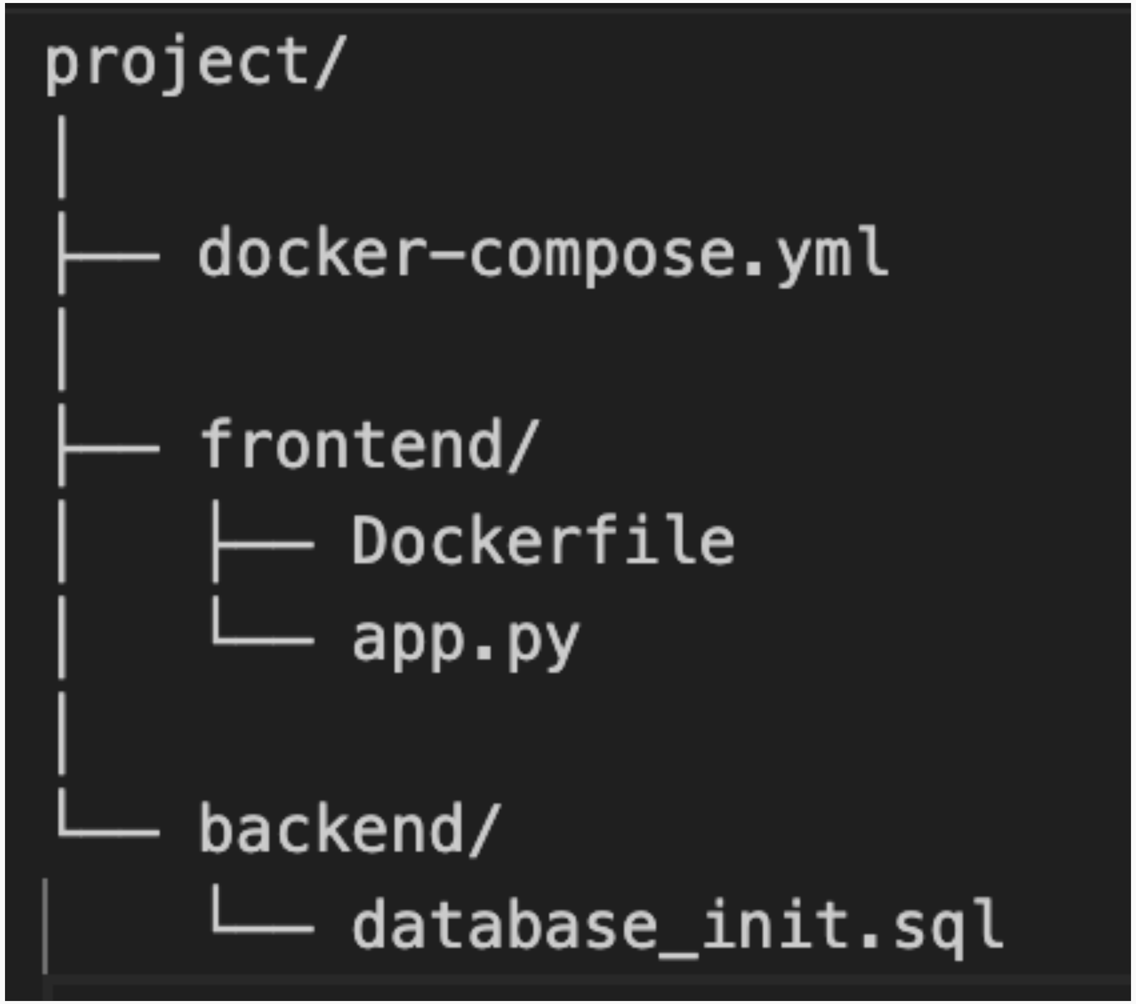
Docker storage that allows for:

- Persistent storage: Volumes are used to persist data generated by and used by Docker containers.
- Decoupling storage: Volumes allow you to decouple the storage lifecycle from the container lifecycle.

Networking allows for containers to communicate with each other

Docker Compose

- Lets get coding!



```
project/
│
├── docker-compose.yml
│
├── frontend/
│   ├── Dockerfile
│   └── app.py
│
└── backend/
    └── database_init.sql
```

Docker

- Thats all there is to it!