WeCloudData

Docker

Data Engineering Diploma

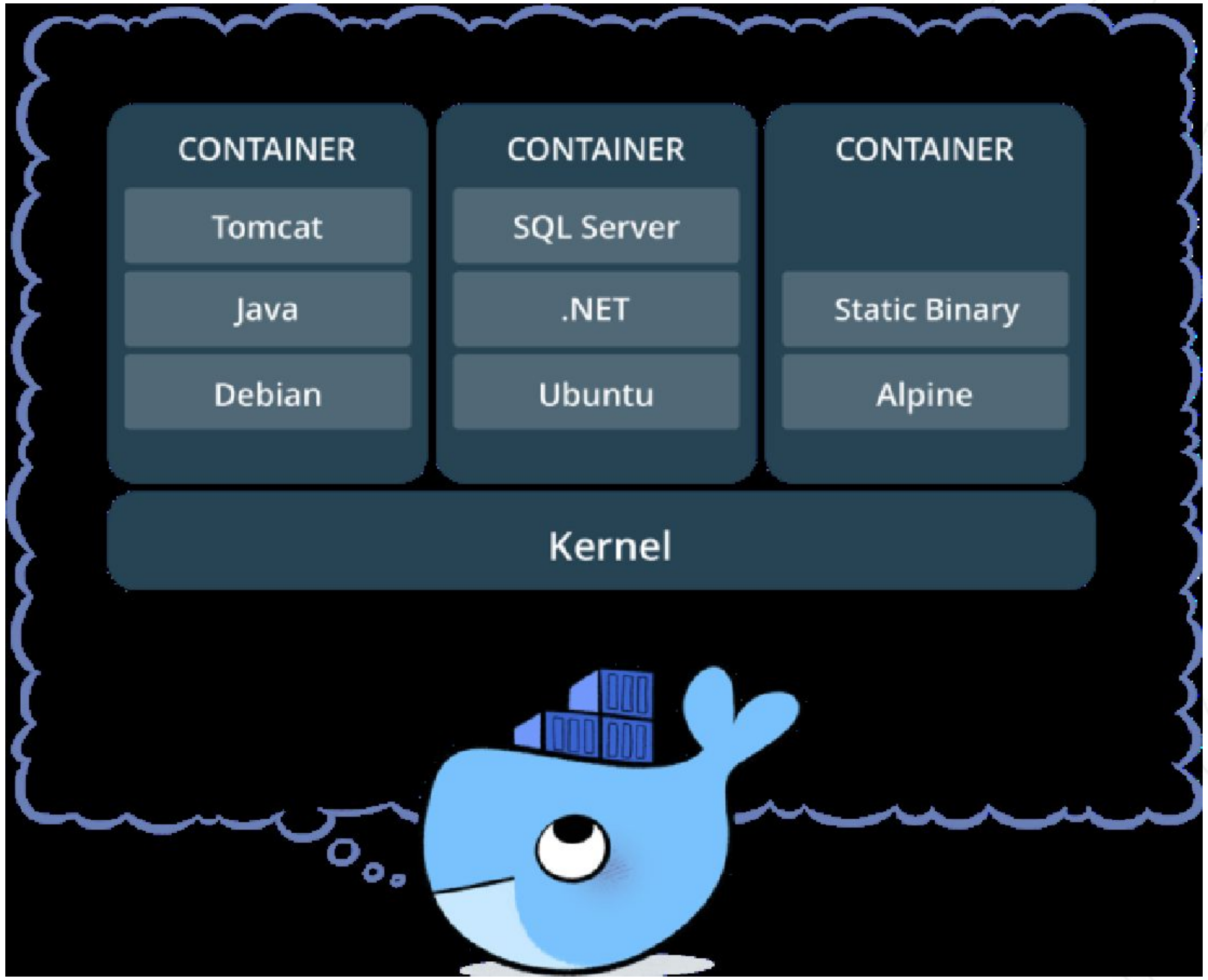
Module 1 :
Introduction to Docker

What is Docker?

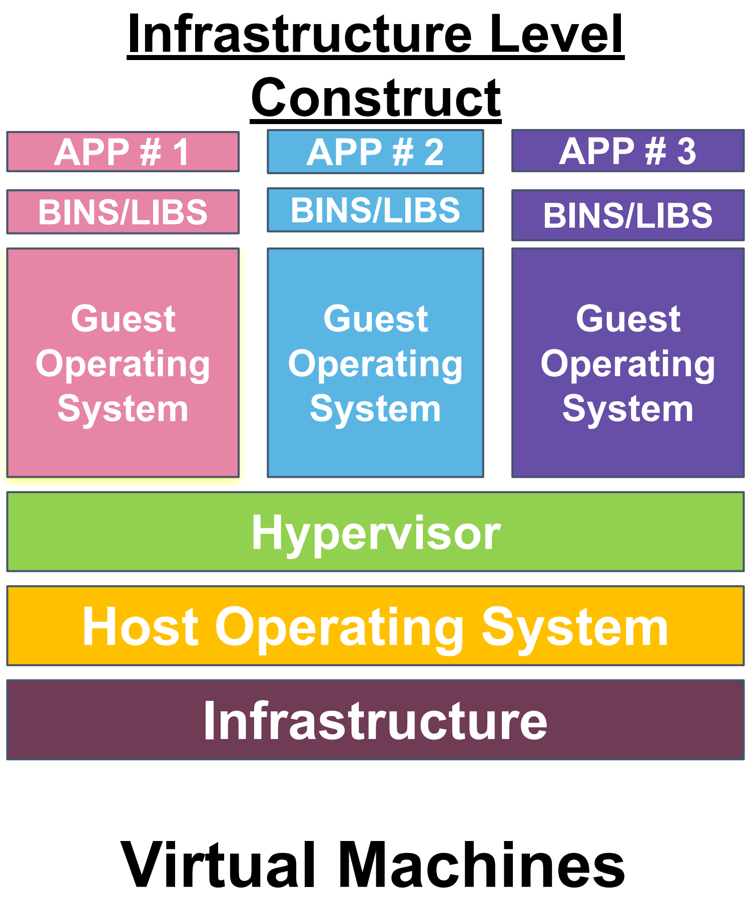
Introduction to Docker

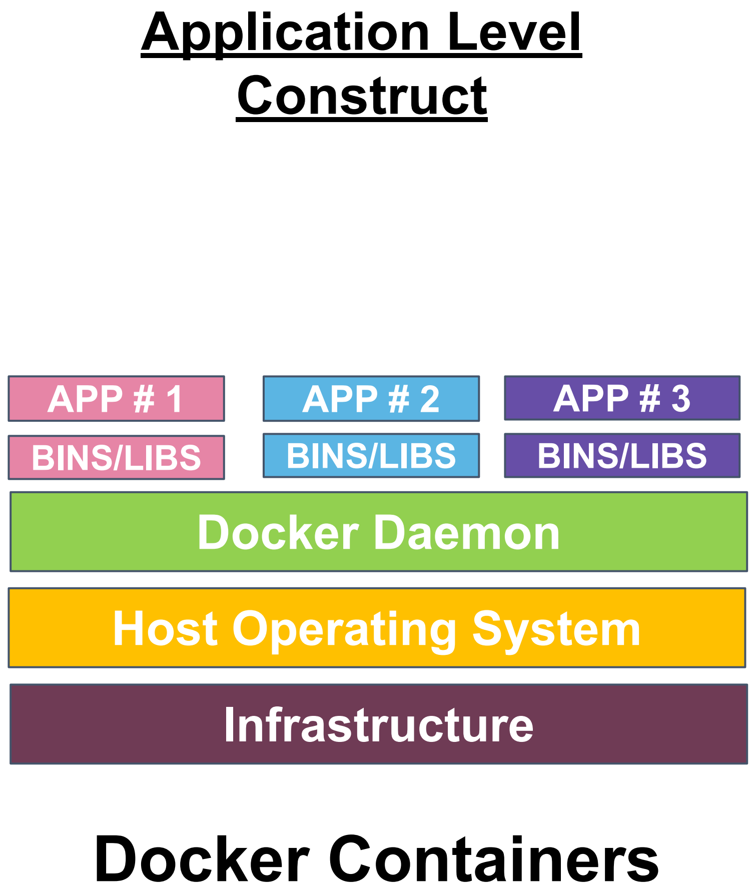
Docker is an open platform for developers and system administrators to build, ship and run **containerized** applications.

Containerization is a form of virtualization where applications run in isolated user spaces, called containers, while using the same shared operating system (OS). A container is essentially a fully packaged and portable computing environment.



Virtual Machine vs. Container





Virtual Machine vs. Container

Getting started with Docker

Virtual Machines

- Virtual machines are an abstraction of physical hardware turning one server into many server.
- Multiple VMs can run on single machine through Hypervisor
- Each VM has full copy of the OS
- Takes minutes to start

Containers

- Think of containers as isolated processes
- Containers are abstraction at the application layer that packages the code and dependencies together
- Each container runs in isolation with other containers;
- multiple containers can run on same machine sharing OS kernel with other containers
- Container can be up and running in milliseconds

Why Docker?

Introduction to Docker

Why we use Docker?

- In a Data Engineering project, we use docker to install various applications and tools, such as Apache Airbyte, Apache Spark Cluster, MySql, etc.
- We use docker to install several tools we daily use, like Jupyter notebook, Zeppelin, etc.

Why use Docker instead of installation directly?

- Installation with Docker is cleaner (easy install and uninstall)
- Installation with Docker is quicker (Everything we need is in a single image)
- Docker is flexible and extendable, we can deploy same docker to many servers at the same time.

Docker Editions

Introduction to Docker

Docker Community Edition

- Free; community-supported product for delivering a container solution
- Intended for: Software development and test

Docker Enterprise Edition

- Subscription based commercially supported products for delivering a secure software supply chain
- Certified on specific platforms
- Intended for: Production Deployments and Enterprise Customers

Setting up Docker on Ubuntu

Introduction to Docker

- Update APT

```
sudo apt-get update
```

- Uninstall any old versions of docker, if you have any. [Optional]

```
sudo apt-get remove docker docker-engine docker.io
```

- Download and install the Docker

```
sudo apt install docker.io
```

- Start and enable the Docker.

```
sudo systemctl start docker
sudo systemctl enable docker
```

Setting up Docker on Ubuntu

- To be able to run the docker with root privileges without using ‘sudo’, add the user to the docker group. [Read here]

```
sudo usermod -aG docker $USER
```

- To check the docker version

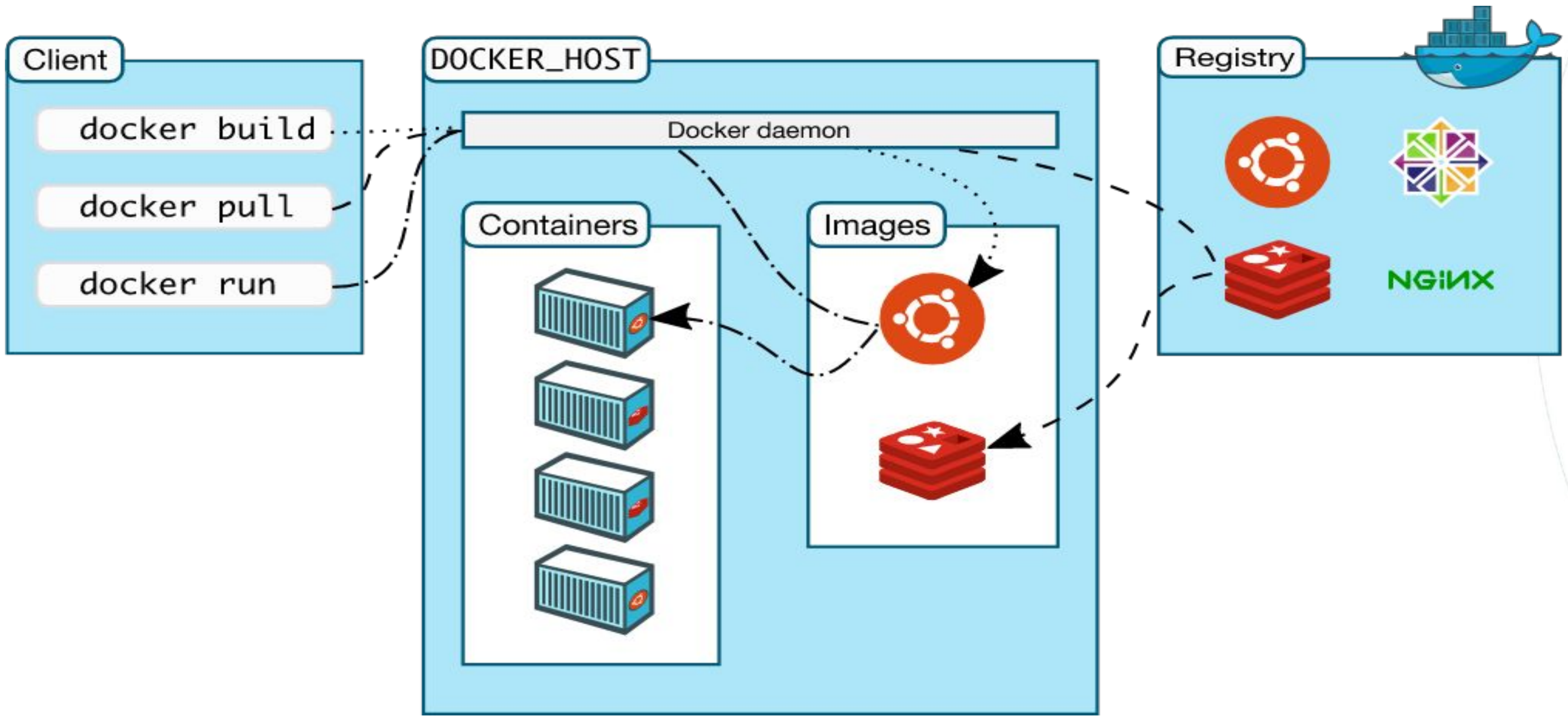
```
docker --version
```

Docker installation Guide on Mac: https://docs.docker.com/desktop/mac/install/

Module 2 :
Docker Architecture

Docker Architecture

Docker Architecture



Source: https://docs.docker.com/engine/docker-overview/

Docker Architecture: Docker Host

Docker Architecture

- Runs docker daemon which listens to and performs actions requested by Docker Client.
- Manages Docker objects such as images, containers, networks and volumes.
- The Docker client and daemon can run on same systems or can connect docker client to a remote docker daemon.

(***A Daemon*** *allows client to start communications with a host server. The daemon does this by handling and routing incoming connection requests.*)

Source: https://docs.docker.com/engine/docker-overview/

Docker Architecture: Docker Client

Docker Architecture

- The interface through which users interact with the docker using the docker commands.
- 2 basic ways to interact with Docker:
  - Docker CLI
  - Docker APIs
- Docker Client can communicate with more than one daemon.

Source: https://docs.docker.com/engine/docker-overview/

Docker Architecture: Docker Registry

Docker Architecture

- The place to store docker images and make them available to others.
- Docker Hub is a public registry that anyone can use .
- *docker pull* and *docker push* are the commands used to pull and push the images from the docker registry.

Source: https://docs.docker.com/engine/docker-overview/

Module 3 :
Getting started with Docker

Docker Hub Signup

Getting started with Docker

1. Make sure you have installed docker on your computer.

```
docker  - -version  # run this command in your terminal
```

1. Go to the Docker Hub, register an account.
   https://hub.docker.com/

1. After sign up in docker hub, go to your terminal, to login in local with command:

```
docker  login -u <your account name>
```

   After entering the above command, input your password in the next line, and "enter"

```
password: <your password>
```

1. Login successfully.

Try in system

Running first Docker Container

Getting started with Docker

1. Go to the “hello-world” image in Docker Hub to take look at the page .
   https://hub.docker.com/_/hello-world

1. Copy and paste the below command to download the **image.**

```
docker  pull hello-world  # run this command in your terminal
```

1. Check if the image has been downloaded into you image list.

```
docker images   # this is the command to check your image list
```

1. In the docker list, know the docker image structure, similar to this:

```
REPOSITORY        TAG          IMAGE ID        CREATED          SIZE

hello-world       latest       feb5d9fea6a5    8 months ago     13.3kB
```

image name:can be used to create a container

Tag: usually tab with version

image unique id: can be used to create container

Try in system

Running first Docker Container

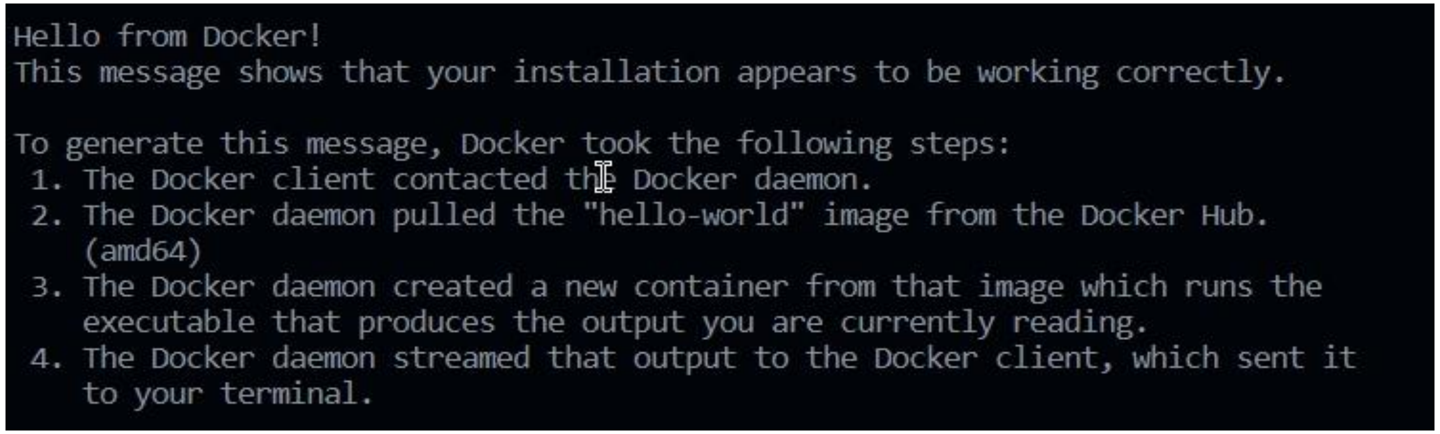
5. Create a Docker container instance. you can use the **image name** to create:

```
docker  run hello-world
```

or
run with the docker image id

```
docker  run feb5d9fea6a5
```

6. Check the terminal, you will see the following words.



```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

Try in system

Running first Docker Container

7. Let’s check the docker container of hello-world by:

```
docker  ps -a
```

you will see the container id, image id, etc.

8. Then, let’s check if the container is actively running by:

```
docker ps
```

We did not see any container run, this is because hello-world image from docker has no extra service running inside the container. So after print out the words in the terminal, the container stopped automatically.

Try in system

Running Docker Container

1. Let’s run a more complex container. Install a Jupyter Notebook via docker.First, go to https://hub.docker.com/r/jupyter/datascience-notebook . Download the image.

```
docker pull jupyter/datascience-notebook
```

In this case, we need to be aware of 2 factors:

1) We may already have a jupyter installed on our computer locally using port 8888. In order to install the jupyter notebook from docker, we need to use another port to avoid conflict. Here we use 10000 (you can use any).
2) If we have a existing notebook file, how can we put in the container? How can we process these notebooks in the container just as easy as we work on local computer? We need a concept: **Volume**. We will use volume to mount local folder with folder in Jupyter.

Try in system

Running Docker Container

2. We need to set port and volume when run docker.

```
docker run -it --rm -p 10000:8888 -v "${PWD}":/home/jovyan/work jupyter/datascience-notebook:latest
```

1. docker run: is the main command.
2. -it: means create a shell in the container, so that you can input command in here, the command will run in the container’s shell.
3. -rm: is the option to tell docker automatically cleans up the container and removes the file system when the container exits.
4. - p: to set port. The first port number 10000 is the port number we want on our local computer. The second number is the original application (Jupyter) port number in the docker.
5. -v: is the option to set volume. "${PWD}" indicates where you want to put your notebook in your local Linux system; “home/jovyan/work” is the mounted directory in docker;
6. jupyter/datascience-notebook:latest: words before ‘:’ is the image name, and after ‘:’ is the image tag.

Try in system

Running Docker Container

3. After this, you can check if the container is running by

```
localhost:10000
```

4. Go to the container inside:

```
docker exec -it <container id> bash
```

5. Jump out of the container:

```
exit
```

Try in system

Running Docker Container

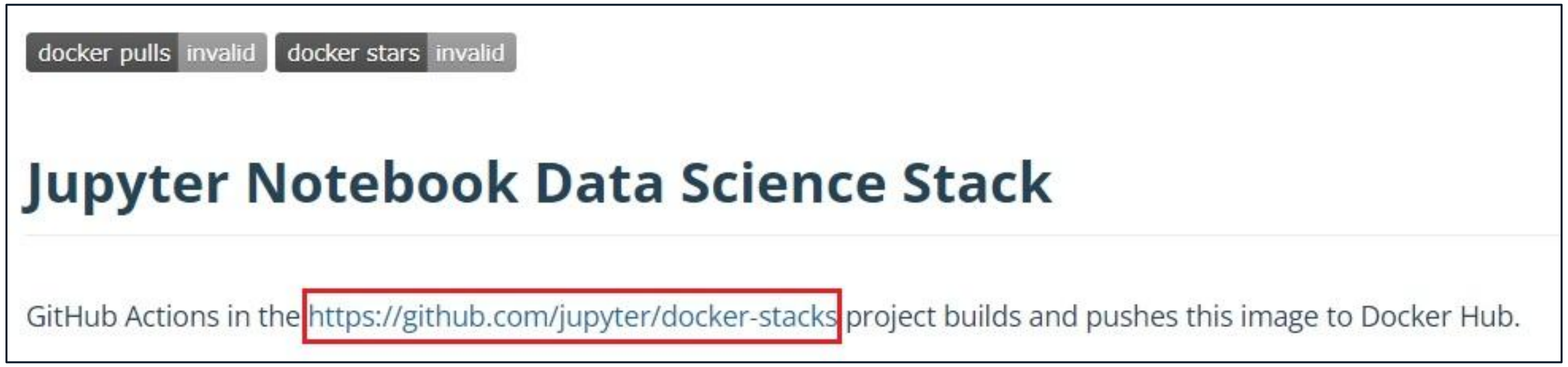
7. Stop the container.
   a. Get the container id:
   ```
   docker  ps
   ```
   b. Stop the container
   ```
   docker  stop <container id>
   ```
   c. Next Step:
   - Option 1: Remove the container if you don’t use it any more.
   ```
   docker  rm  <container id>
   ```
   - Option 2: Start it when you next time going to use it.
   ```
   docker  start  <container id>
   ```

Try in system

Running Docker Container

How do we know which directory inside the docker is what I need to mount? How can we know more information about the image?

1. Let’s go to the Jupyter container page in Docker Hub to find the solution. You will find the github page of the image. Let’s go to this page:



docker pulls invalid docker stars invalid

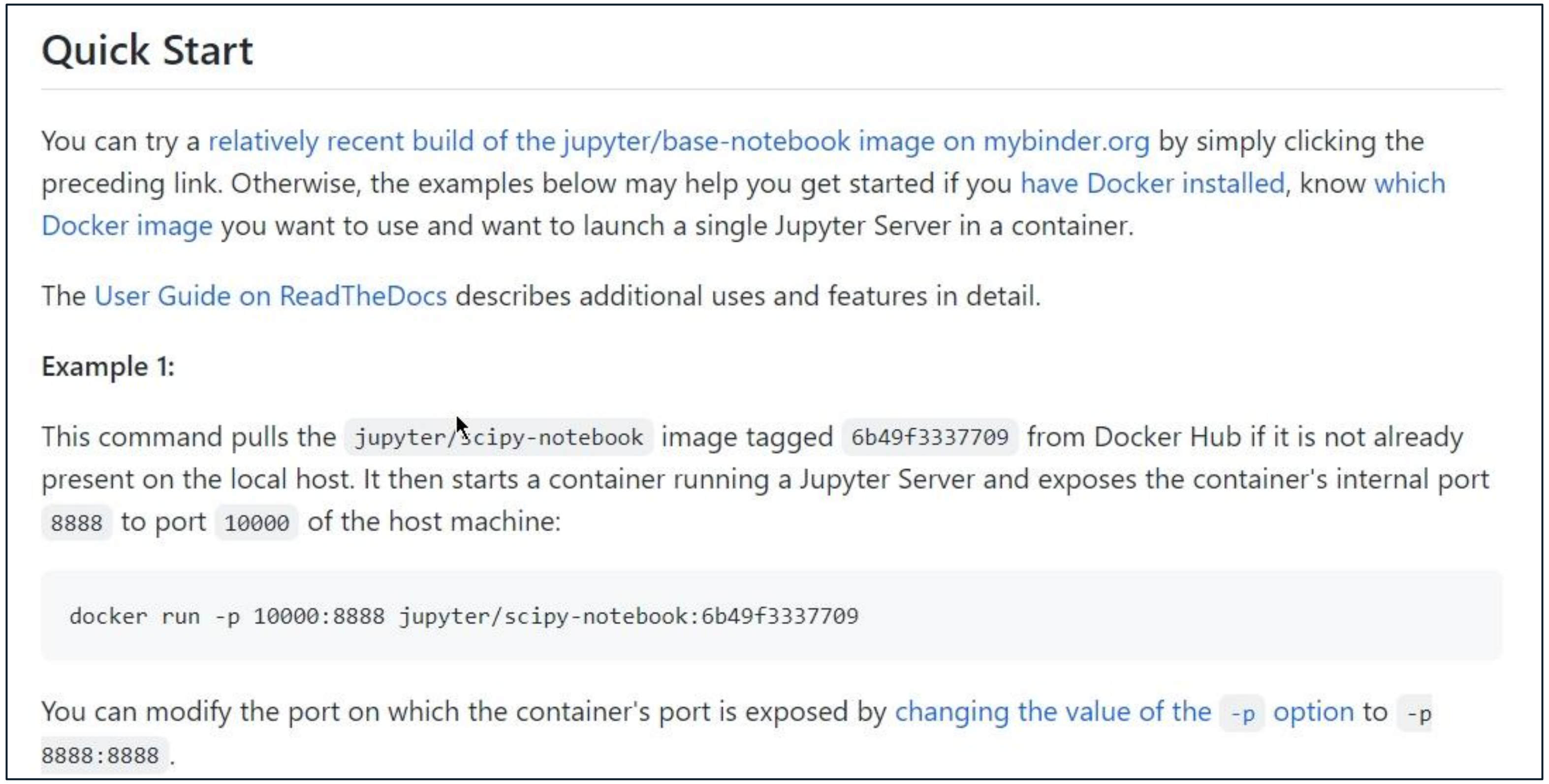
Jupyter Notebook Data Science Stack

GitHub Actions in the https://github.com/jupyter/docker-stacks project builds and pushes this image to Docker Hub.

Running Docker Container

How do we know which directory inside the docker is what I need to mount? How can we know more information about the image?

2. Let's go to the Github page of the image. You will find the instruction about the image:



Quick Start

You can try a relatively recent build of the jupyter/base-notebook image on mybinder.org by simply clicking the preceding link. Otherwise, the examples below may help you get started if you have Docker installed, know which Docker image you want to use and want to launch a single Jupyter Server in a container.

The User Guide on ReadTheDocs describes additional uses and features in detail.

Example 1:

This command pulls the `jupyter/scipy-notebook` image tagged `6b49f3337709` from Docker Hub if it is not already present on the local host. It then starts a container running a Jupyter Server and exposes the container's internal port `8888` to port `10000` of the host machine:

```
docker run -p 10000:8888 jupyter/scipy-notebook:6b49f3337709
```

You can modify the port on which the container's port is exposed by changing the value of the `-p` option to `-p 8888:8888`.

WeCloudData

Running Docker Container

docker run options:

- -p: Set ports.
- -v: Bind mount a volume.
- -i: Keep STDIN open even if not attached
- -t: Allocate a pseudo-TTY.
- -d: Run container in background and print container ID.
- -u: Username or UID (format: <name|uid>[:<group|gid>])
- -e: Set environment variables.
- - -rm: Automatically remove the container when it exits
- - -name: Assign a name to the container

Try in system

Running Docker Container

All commands for docker:

https://docs.docker.com/engine/reference/commandline

Module 4 :
Interact with Docker

Interact with Docker

Docker commands:

- docker stop: Stop one or more running containers.
- docker kill: Kill one or more running containers. Comparing stop, kill leads to unsafe exit.
- docker rm: Remove one or more containers.
- docker rmi: Remove one or more images.
- docker start: Start one or more stopped containers.
- docker restart: Restart one or more containers.
- docker exec: Run a command in a running container.
  - Format: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
  - Example: ***docker exec -d ubuntu_bash touch /tmp/execWorks***
- docker cp: Copy files/folders between a container and the local filesystem.
- docker images: List docker images.
- docker ps: List running containers.
- docker ps -a: List all containers.
- docker logs: Fetch the logs of a container.
- docker rename: Rename a container.
- docker volume: Manage volumes.
- docker login: Log in to a Docker registry.

Module 5 :
Docker Volumes

Persistent Data Problem

- Containers are usually immutable and ephemeral
  - Immutable: a container won't be modified during its life: no updates, no patches, no configuration changes
  - Containers can be restarted, stopped or replaced easily
  - Redeploy a new container, if configuration needs to change

- Persistent Data problem: What about the unique data produced by application?

- Two solutions:
  - Volumes and Bind Mounts

Docker Volumes: What and How?

- Volumes: Make special location outside of container UFS
- ***VOLUME*** command in Dockerfile
- Also override with ***docker run -v /path/in/container***
- Bypasses Union File System and stores in alt location on host
- Include its own management commands under ***docker volume***
- Connect to none, one or multiple containers at once
- Not subject to ***commit***, ***save***, or ***export*** commands
- By default, they only have a unique ID, but you can make it “named volume” by assigning name to it.

Try in system

Docker Volumes: What and How?

- Bind Mounts: link container path to host path
- Maps a host file or directory to a container file or directory
- Basically, just two locations pointing to the same file(s)
- Again, skips UFS, and host files overwrite any in container
- Can't use in Dockerfile, must be at container run
- ***... run -v /Users/bret/stuff:/path/container*** (mac/linux)
- ***... run -v //c/Users/bret/stuff:/path/container***(windows)

Module 6 :
Docker Exercises

Exercise 1: Multiple Containers

- Run a nginx, a mysql, and a httpd (apache) server
- Run all of them --detach (or -d), name them with --name
- Nginx should listen on 80:80, httpd on 8080:80, mysql on 3306:3306
- When running mysql, use the --env option (or -e) to pass in MYSQL_RANDOM_ROOT_PASSWORD=yes
- Use docker container logs on mysql to find the random password it created on startup
- Clean it up all with docker container stop and docker container rm (both the commands can accept multiple names and IDs)
- Use docker container ls to ensure everything is correct before and after clean up.
- Refer docs.docker.com and --help for any help

Try in system

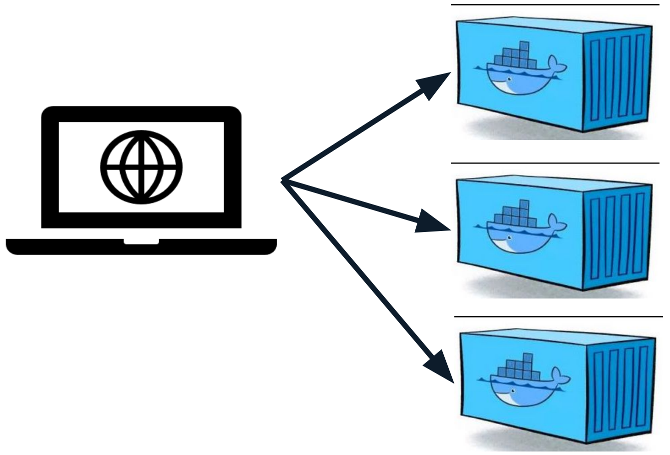
Module 7 :
Docker Compose

What is Docker Compose?

Docker Compose

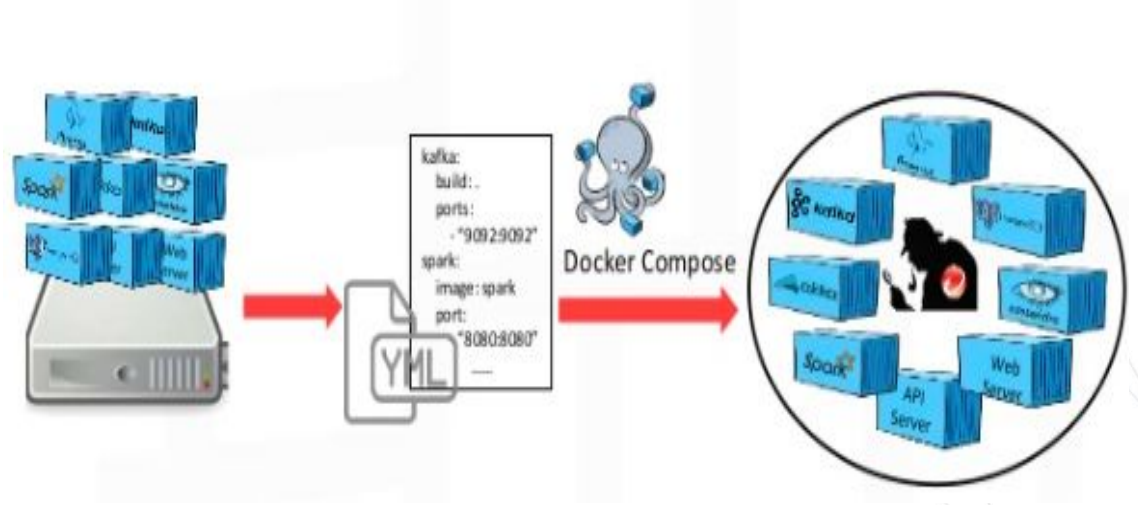
Without Compose

- At a time, build one container and manually connect containers together for application.
- Care must be taken with Dependencies and the order of starting containers.



With Compose

- Define multi-container app in .yml file and spin the application with single command.
- Handles application dependencies



```
kafka:
  build: .
  ports:
    - "9092:9092"
spark:
  image: spark
  port:
    "8080:8080"
  ---
```

Image Courtesy: DockerCon

What is Docker Compose?

Docker Compose

- Docker Compose allows you to accomplish the following tasks with a single command:
  - Build Docker Images
  - Launch Containerized Applications as Services
  - Launch full systems of services
  - Manage the state of individual services in a system
  - View logs for the collection of containers making a service
- Compose works in local environments:
  - Testing/ Prototyping/Learning CI workflows.

Use Case: Docker Compose?

Docker Compose

- **Microservices:** Modern apps are made up of smaller services. These services interact together to deliver results.
- **Example:** Consider an app with following services:
  - Web Front-end
  - Catalog
  - Back-End database
- In general, **each container should do one thing and do it well.** Few reasons can be:
  - Easy scaling of APIs and front-ends differently than databases
  - Separate containers allow versioning, and these versions can be updated in isolation
  - Allows you to use managed service for database in production
  - Running multiple processes in one container will introduce additional complexity in container start-up/shutdown as then it would require process manager to manage the multiple processes.

Use Case: Docker Compose?

Docker Compose

- To run and deploy all these services so that they work smoothly together can be hard:
  - One way: writing scripts to put everything together and running several docker commands,
  - More efficient way: Docker Compose
- Docker Compose lets you describe an entire app in a single declarative configuration file.
  - The application service can then be deployed using a single command
- Once the app is deployed, its entire lifecycle can be managed with a simple set of commands.
- You can even store and manage the configuration file in a version control system!
- Read more about Docker Compose: https://docs.docker.com/compose/.

Install Docker Compose

- Linux (ubuntu)
  a. Go through docker documentations:
     - https://docs.docker.com/compose/install/linux/#install-the-plugin-manually
- Mac
  a. Docker for Mac has contain Docker-Compose already.

The docker-compose File

- Docker Compose uses YAML files to define multi-service applications.
- The default name for the Compose YAML file is ***docker-compose.yml***.
  - ***-f*** flag can be used to specify custom filenames.

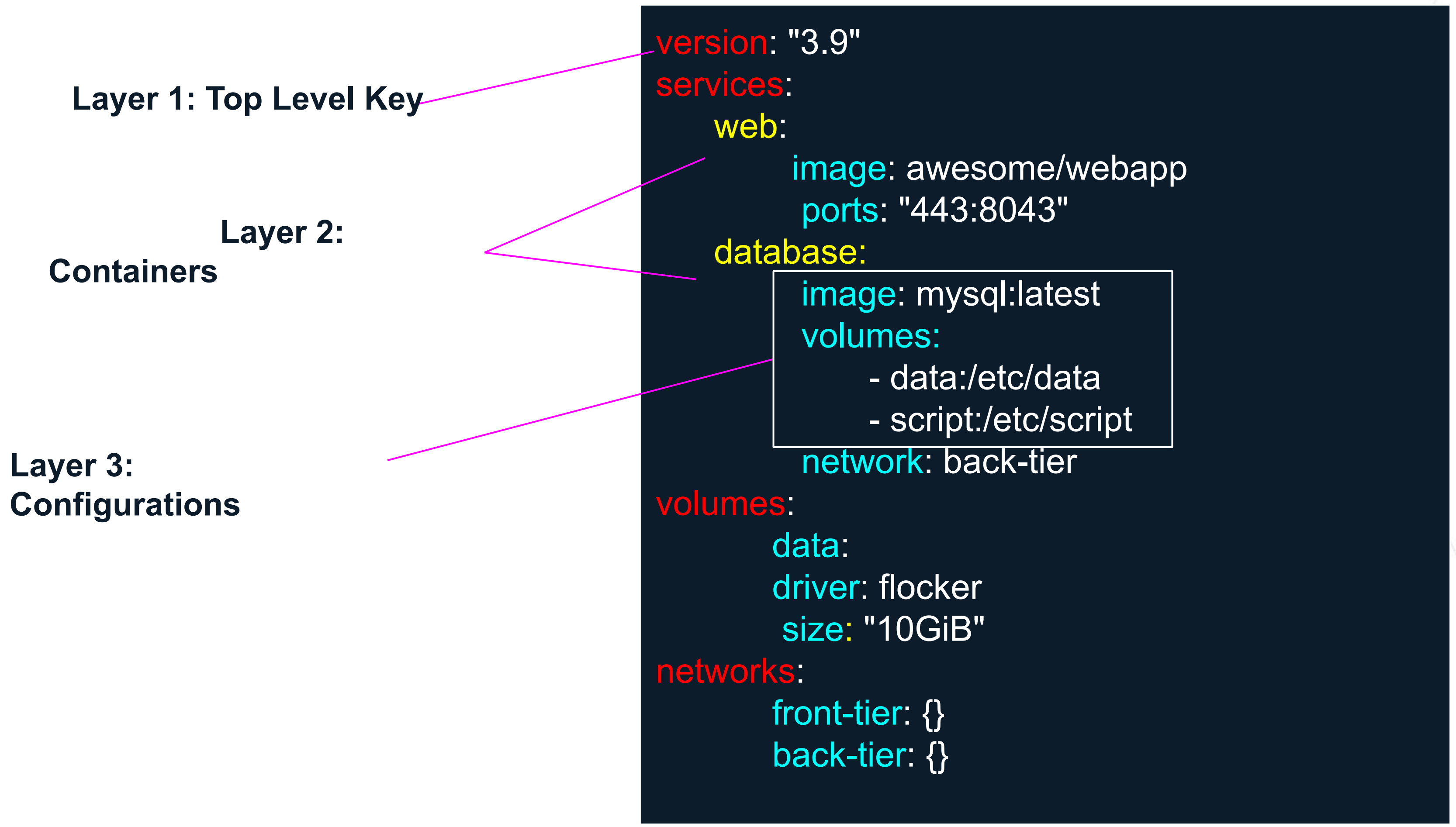
```
docker compose -f docker-compose-specificname.yml
```

- Docker Compose consists of multiple layers that are split using tab stops or spaces. There are four main things almost every Compose-File should have which include:
  - The **version** of the compose file
  - The **services** which will be built
  - All used **volumes**
  - The **networks** which connect the different services
- The core aspects of the Compose file are it allows to manage and create a network of containers.

The YAML File

Docker Compose

- A sample ***docker-compose.yml*** file looks like this:



```
version: "3.9"
services:
    web:
          image: awesome/webapp
           ports: "443:8043"
    database:
           image: mysql:latest
           volumes:
                - data:/etc/data
                - script:/etc/script
           network: back-tier
volumes:
         data:
         driver: flocker
          size: "10GiB"
networks:
         front-tier: {}
         back-tier: {}
```

Layer 1: Top Level Keys

- **version (Mandatory)**:
  - The first thing to add to the header of the file.
  - Specifies the version of the **Compose file format** we are using. Different versions have different formats.
  - It does not define the version of Docker Compose or the Docker Engine.
  - There are several versions of the Compose file format – 1, 2, 2.x, and 3.x. We usually use version 3.+, because it is compatible with 1 and 2.
- **services (Mandatory):**
  - Here we define the different application services.
  - Compose will deploy each of these services as its own container.
- **networks(Optional):**
  - Tells Docker to create new networks.
  - By default, Compose creates bridge network
  - These are single-host networks that can only connect containers on the same host.
  - Use driver property to specify different network types.
- **volumes (Optional)**:
  - This is where we specify the volumes to be created for the containers.
  - same way as how you use docker volume create.

Layer 1: networks

Networks define the communication rules between containers, and between containers and the host system.

- By default Compose sets up a single network for each container. Each container is automatically joining the default network which makes them reachable by both other containers on the network, and discoverable by the hostname defined in the Compose file.
- Instead of only using the default network you can also specify your own networks within the top-level networks key, allowing to create more complex topologies and specifying network drivers and options.

```
networks:
  frontend:
  backend:
    driver: custom-driver
    driver_opts:
      foo: "1"
```

Layer 1: volumes

Volumes are Docker's centralized way of persisting data which is generated and used by Docker containers. They are completely managed by Docker and can be used to share data between containers and the Host system.There are multiple types of volumes:

- **Normal Volume**: Define a specific path and let the Engine create a volume for it.

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql
```

- **Path mapping**: Define absolute path mapping of your volumes by defining the path on the host system and mapping it to a container destination.

```
volumes:
  - /opt/data:/var/lib/mysql
```

- **Named volume**: similar do the other volumes but has it's own specific name that makes it easier to use on multiple containers. That's why it's often used to share data between multiple containers and services.

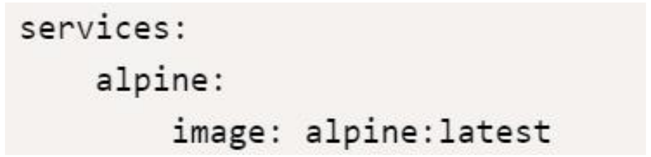
```
volumes:
  - datavolume:/var/lib/mysql
```

Layer 2: Containers

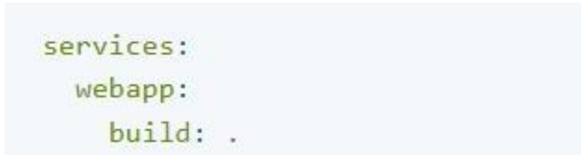
This layer contain all the containers you are going to use in the Docker Compose. You can define the container name based on your own requirements.

Key Configurations:

- image & build
  - use keyword image when use a preexisting image that is available on Docker Hub.
  - use keyword build when building images using a Dockerfile.

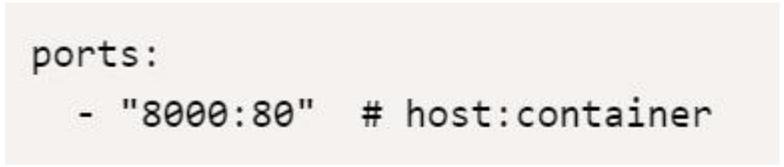


```
services:
    alpine:
        image: alpine:latest
```



```
services:
  webapp:
    build: .
```

- ports:
  - In ports, we define which port we want to expose and the host port it should be exposed to.



```
ports:
  - "8000:80"  # host:container
```

Layer 2: Containers

Key Configurations:

- volumes
  - volumes defines the path of the host system mapping to the path of the container.

```
volumes:
  - /opt/data:/var/lib/mysql
```

- depends_on:
  - depends_on in Docker are used to make sure that a specific service is available before the dependent container starts.

```
    depends_on:
      - db
      - redis
```

- environment:
  - Set environment variables in a container.

```
web:
  environment:
    - NODE_ENV=production
```

Layer 2: Containers

Key Configurations:

- command
  - Execute actions once the container is started and act as a replacement for the CMD action in your Dockerfile.

```
ports:
    - '3000:3000'
command:
    - 'npm run start'
```

- restart:
  - By default, Docker will not restart containers after a host system reboot.Docker provides a restart policy for your containers:
    - **no**: The default value.
    - **on-failure[:max-retries]**: Restart only if the container exits with a failure (non-zero exit status). One can limit the number of restart retries the Docker daemon attempts.
    - **always**: Always restart the container regardless of the exit status.
    - **unless-stopped**: Always restart the container regardless of its exit status, but do not start it on daemon startup if the container has been put to a stopped state before.

The YAML File

Docker Compose file summary

- ***version***:
  - The first thing to add to the header of the file.
  - Specifies the version of the Compose file format we are using.
  - It does not define the version of Docker Compose or the Docker Engine.
  - There are several versions of the Compose file format – 1, 2, 2.x, and 3.x.
- ***services:***
  - Here we define the different application services.
  - Compose will deploy each of these services as its own container.

The YAML File

Docker Compose file summary

- ***networks:***
  - Tells Docker to create new networks.
  - By default, Compose creates bridge network
  - These are single-host networks that can only connect containers on the same host.
  - Use driver property to specify different network types.
- ***volumes***:
  - This is where we specify the volumes to be created for the containers.
  - It is an optional key
  - You use the volume mapping in a way that is like how you use docker volume create.
  - Services can reference volumes in each service’s volumes configuration key.

Docker Compose CLI
Docker Compose

Commands	Description
docker-compose build	Reads your ***docker-compose.yml*** and builds a custom image.
docker-compose up	To deploy a Compose app. It expects the Compose file to be called **docker-compose.yml** or **docker-compose.yaml**, but you can specify a custom filename with the **-f** flag.
docker-compose down	To stop and delete a running Compose app. It deletes containers and networks, but not volumes and images.
docker-compose stop	To stop all the containers in Compose app without deleting them. Changes made to Compose app since stopping it will not appear in the restarted app. The app will have to be re-deployed to get the changes.
docker-compose restart	To restart the stopped Compose app

Docker Compose CLI

Docker Compose

Commands	Description
docker-compose ps	To list each container in the Compose app. It shows current state, the command each one is running, and network ports.
docker-compose rm	To delete the Compose app that has been stopped. It will delete containers and networks, but it will not delete volumes and images.

useful link: https://gabrieltanner.org/blog/docker-compose/

System Demo

- Build a basic compose file for a Drupal content management system website.
- Use Docker Hub to look for images: The drupal image along with the postgres image
- Use 8080 ports to expose Drupal
- Be sure to set POSTGRES_PASSWORD for postgres
- Walk though Drupal setup via browser
- Challenge: Use volumes to store Drupal unique data