**Docker**

1. **Concept**

**Docker:** Docker is a platform that enables developers to package applications and their dependencies into lightweight, portable containers that can run consistently across different environments, making it easier to deploy, scale, and manage applications efficiently.

**Editions Of Docker:**

* Community Edition(free)
* Enterprise edition(paid)

Two kind of release

* Stable: tested
* Edge: comes with the latest technologies(not tested)

1. **Install Docker on Linux Machine**

Goto docs.docker.com > Docker CE version>ubuntu>Install the CE edition>Follow all the instruction given on ubuntu

After installing the docker run the command:docker version: it shows the docker client and docker server version:

**Docker Client**: The Docker Client is the command-line interface that allows users to interact with Docker by sending commands like docker run or docker build. It converts these commands into API requests and sends them to the Docker Server for processing.

**Docker Server (Docker Daemon)**: The Docker Server, also called the Docker Daemon, is a background service that receives and processes commands from the Docker Client. It manages images, containers, networks, and storage volumes, and orchestrates the lifecycle of containers, enabling deployment, scaling, and maintenance of applications in isolated environments.

After that we need to Add user to docker User group: sudo usermod -a -G docker $USER

The command is used to add your user to the Docker user group, allowing you to run Docker commands without needing sudo each time

* **sudo** gives you administrative (root) privileges to make changes to the system.
* **usermod** is the command that modifies user accounts.
* **-a** ensures that the user is **added** to the specified group (in this case, Docker) without being removed from other groups.
* **-G docker** tells the system that you want to add the user to the **docker** group, which allows permission to run Docker commands.
* **$USER** represents your current username, so the command applies to your account.

1. **Basic Concepts**

**Docker Image**

A **Docker image** is a **static file** that contains the application’s code, libraries, dependencies, and configuration files needed to run that application. It is like a **recipe** that describes how to create a container. Docker images are **immutable**, meaning they cannot be changed once created. You can, however, create new versions of an image by modifying the Dockerfile (the configuration file used to build the image) and rebuilding the image.

Docker uses images as a template to create containers. The images themselves do not run, but they provide the framework for containers to operate.

Docker images  
Lists out the docker images inside a machine

Download docker image , download docker image with specific version  
Docker pull <image\_name>  
docker pull <image\_name>:<image\_version>

Docker images are made up of layers, each representing an instruction in a Dockerfile, like installing software or copying files. Each layer is stored as a separate file system change, which Docker then combines using a *union file system* (e.g., OverlayFS or AUFS). This union file system merges all layers into a single, unified view, so when a container runs, it sees a continuous file system. Docker uses caching and *copy-on-write* techniques to optimize storage and performance: if a layer hasn’t changed, Docker reuses it instead of rebuilding, and any changes made by a container are saved in a new layer rather than modifying existing ones. This layering approach not only makes image creation and container startup faster but also saves storage by allowing multiple containers to share the same base layers securely and efficiently.

To show image layers:  
docker history <image\_name>

**Docker Image Tagging**

A Docker tag is a label applied to a Docker image, allowing you to reference specific versions of an image easily. Tags help distinguish between different builds, versions, or configurations of the same image, making it easy to manage multiple versions of a project or software component.

An image can have multiple tags pointing to the same content. For instance, you might tag a stable version as 1.0 and latest, so users can pull it using either tag.  
  
**Docker Hub**  
Docker Images can be downloaded from docker hub. Just like github. Images can be committed with changes and have multiple versions. If you don’t provide a specific version number, the client defaults to latest.

Base Image: Images that have no parent image, usually images with an OS like ubuntu ,busybox, debain.

Child Image: Images that build on the base images and add additional functionality

Official image: Images that are officially maintained and supported by the docker community.

User Images: These are created and shared by users like you and organizations. They build on base images and add additional functionality.

**Docker Container**A **Docker container** is a **runtime instance** of a Docker image. It is a lightweight, portable, and isolated environment that packages an application along with its dependencies (such as libraries and configurations) to ensure it runs consistently across different environments, sharing the host system's OS kernel while providing process isolation and resource management, making it ideal for scalable, efficient, and reproducible application deployment.

**Why Containers Required?:**

Containers package an application and all its dependencies (libraries, configurations, and binaries) into a single unit. This means the application can run consistently across different environments, such as development, testing, staging, and production, without compatibility issues. Whether it's your laptop, a colleague's machine, or a cloud server, a container ensures the app runs the same way everywhere.

Containers run applications in isolation from one another. Each container has its own file system, memory, and network resources, preventing conflicts between different applications or services running on the same system. This isolation helps avoid issues like version mismatches or dependency conflicts.

**Relationship Summary:**

* **Docker** is the tool that manages the whole process of using images and containers.
* **Docker images** are like **blueprints** or **templates** that define how an application should run, along with all its dependencies.
* **Docker containers** are the **instances** that run from these blueprints, acting as **live applications** that you interact with.

Docker Hub: **Docker Hub** is a cloud-based registry service where Docker users can share, store, and manage Docker images. It acts as a central repository that allows developers to find and pull pre-built images (such as official images for popular software like Nginx, MySQL, etc.) or push their own custom images.

1. **Start nginx webserver in Docker**
   1. Search for nginx official image in dockerhub
   2. docker pull nginx: This command downloads the official Nginx image from Docker Hub.
   3. docker run --name my-nginx-container --publish 80:80 --detach nginx  
      This command will:
      * **--name my-nginx-container**: Name the container (optional but useful for identification).
      * **--publish 80:80**: Map port 80 on the host machine (for the server) to port 80(default nginx port) inside the container. This allows external access to the web server.
      * **--detach**: Run the container in the background (detached mode). When you use the --detach option, Docker runs the container in the background, rather than blocking your terminal by displaying the container’s logs or output.
      * **nginx**: Use the official Nginx image from Docker Hub.
   4. Check whether the container is running or not:docker ps or docker container ls
   5. A white background with red text

      Description automatically generatedServer\_ip:80 to check the nginx first page.
   6. To stop : ctrl+c
2. **Docker File**

A **Dockerfile** is a script (text file) that defines the steps needed to build a **Docker image**. It contains instructions such as which **base image** to use, which files to copy, what software to install, and the commands to execute when the container starts.

A screenshot of a computer program

Description automatically generatedDocker can build images automatically by reading the instructions from a Dockerfile.

1. **Relationship Among Docker, Docker file, Docker image and docker container**

* **Dockerfile**: This is where you define the environment and configuration needed for your application (like instructions to install software and copy files).

**Example**: The Dockerfile might start with FROM ubuntu:latest and then add your app code.

* **Docker Image**: After you’ve written the Dockerfile, you run docker build to create a Docker image from it. The image is a **snapshot** that contains everything your app needs to run.

**Example**: When you run docker build -t my-app ., you get a **Docker image** tagged as my-app.

* **Docker Container**: Once you have the image, you can run it with docker run, which creates a **container**. The container is an instance of the image, running as an isolated process.

**Example**: Running docker run -d my-app will start a **container** from the my-app image.

* **Docker**: Docker is the platform that ties all this together. It provides the tools to build the image from the Docker file, run containers from the image, and manage all containers and images in an efficient way.

**Example**: You use Docker to build the image, manage containers, and pull or push images to Docker Hub (or another registry).

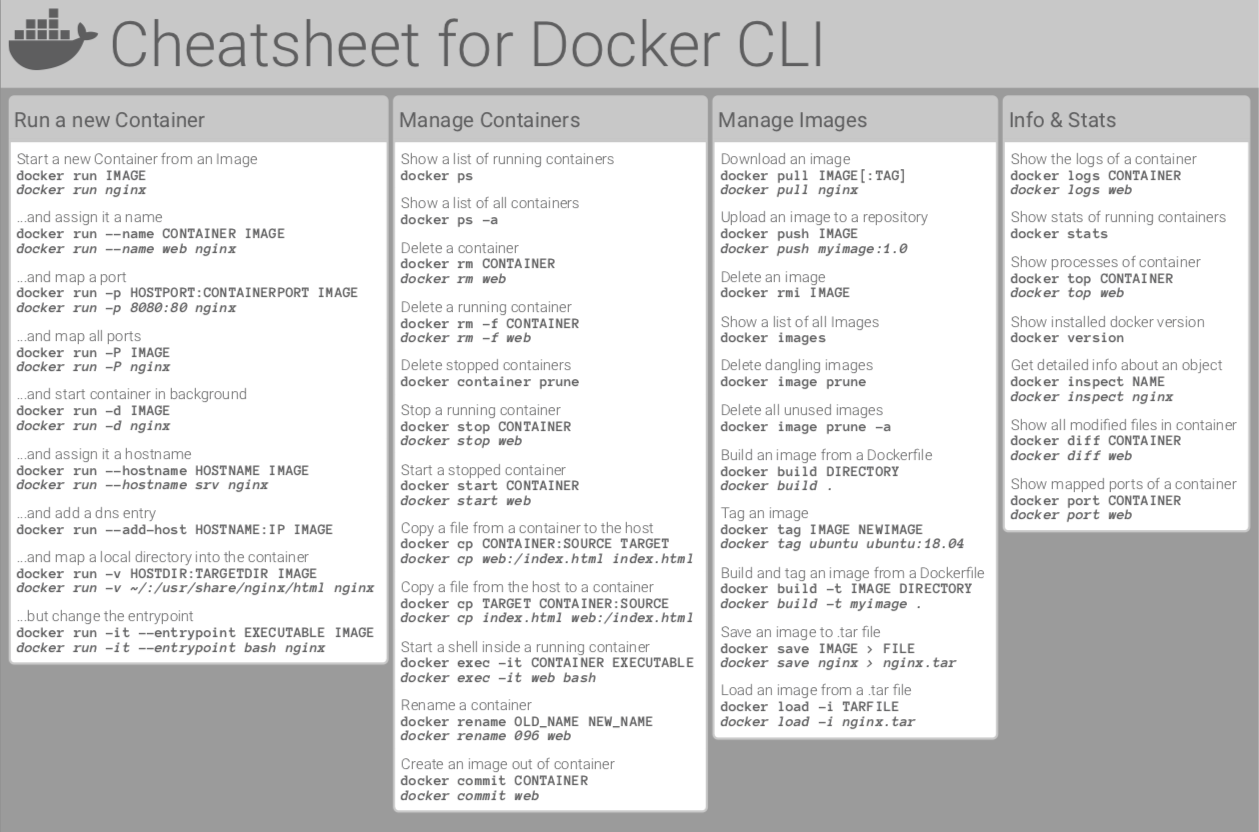
**Visualized Workflow:**

* **Write a Dockerfile** (specifies how to set up your app environment).
* **Build a Docker image** using the Dockerfile (via docker build).
* **Run a container** from the Docker image (via docker run).
* **Docker** enables and manages the entire process: building, running, and managing images and containers.

1. A screenshot of a computer program

   Description automatically generated**Docker File Instruction Commands**

* FROM: node:14 sets the base image to Node.js (version 14). This sets the *base image* for the container, providing a starting environment. Without FROM, Docker wouldn’t know what OS or runtime environment to use.
* WORKDIR: /app sets the container's working directory. WORKDIR Defines the *working directory* inside the container where commands will be executed. This keeps files organized and makes paths simpler.
* COPY & RUN: Copies package.json files and installs dependencies to save build time. By copying package.json before other files, Docker can cache dependencies, avoiding reinstallation if files other than package.json change.
* COPY: Copies all other app files to the container.
* EXPOSE: Marks port 3000 for use by the app.
* CMD: Starts the app with node app.js.



1. **Docker Internal Processing:**

Looks for the image in image cache

Then looks in the remote docker repository

Download the latest version of image

Create new container based on the image

Gives a virtual ip on a private network inside docker engine

docker container ls

shows every docker constainers with their id image …

it will also provide with own vpn and give its private ip

starts container by using CMD in imager in docker file

1. **Containers vs virtual machines**

Containers and virtual machines both have similar resource allocation and allocation benefits.

Containers and VMs are functionally different because containers virtulaise the OS and VMs are virtualize the hardware.

A screenshot of a computer application

Description automatically generatedContainers are more portable.

Containers virtualize at the OS level, sharing the host OS's kernel. This means that containers package only the application and its dependencies, not the entire OS. They run isolated processes on the same OS, making them lightweight and fast.

VMs virtualize at the hardware level using a hypervisor to simulate an entire hardware system. Each VM runs its own OS along with the application and dependencies, resulting in heavier resource usage.  
  
Hypervisor is a layer of software that enables the creation and management of virtual machines (VMs) on a physical computer (host). It acts as a bridge between the hardware and the virtual machines, allowing multiple VMs to run simultaneously on a single physical host while sharing the underlying hardware resources like CPU, memory, and storage.

1. **Docker CLI Monitoring**

**Docker CLI Monitoring** refers to the use of Docker’s command-line interface (CLI) commands to monitor and manage Docker containers, images, networks, and volumes. Docker provides several CLI commands to track the performance, resource usage, and status of containers and other Docker components, which can be useful for troubleshooting, performance tuning, and ensuring applications run smoothly.

docker ps

docker ps -a

Lists all running containers, displaying essential details like container ID, image, command, status, ports, and names. Adding -a shows all containers, including stopped ones.

docker stats

Provides real-time metrics about container resource usage, including CPU, memory, network I/O, and block I/O. This command is essential for monitoring the health and performance of running containers.

docker inspect <container\_id>

Displays detailed information about a container or other Docker object, such as its configuration, IP address, environment variables, and resource limits. It's useful for examining container details and troubleshooting.

docker top <container\_id>

Displays the processes running inside a container, similar to the Unix top command, which is useful for checking application processes and diagnosing performance issues within a container.

docker system df

Shows a summary of Docker disk space usage, including images, containers, and volumes, which is useful for managing storage and cleaning up unused resources.

1. **SSH Running Container**

**SSH-ing into a running container** means using SSH (Secure Shell) to access the terminal of a Docker container, allowing you to execute commands inside the container as if you were logged directly into it.

However, **SSH is typically not needed** for Docker containers because Docker provides its own way to access a container's terminal directly using the docker exec command.

Instead of setting up SSH, you can use the docker exec command to open a shell session in a running container. For example:

docker exec -it <container\_id\_or\_name> /bin/bash

**docker exec**: Executes a command inside an existing, running container.

**-i (Interactive)**: Keeps the input stream open so you can type commands.

**-t (TTY)**: Allocates a pseudo-TTY, which makes the terminal session look and behave like a real terminal.

**/bin/bash**: Starts a Bash shell in the container (use /bin/sh if Bash isn’t available).

**Use Cases**: Useful for debugging, modifying configurations, and running commands in a running container.

1. **Docker Network**

Docker networks enable communication between containers, the host system, and external networks by providing isolated virtual networks within a Docker environment. Each container connected to a Docker network can communicate with other containers on that network, and networks can be customized to control isolation and accessibility.

Docker provides several network types:

**Bridge** (default for container-to-container communication on a single host),

**Host** (bypasses Docker's network isolation, allowing containers to share the host's networking stack),

**Overlay** (enables multi-host networking for Swarm services),

**Macvlan** (assigns a unique MAC address to each container for direct access to the physical network).

Network configurations can be managed using Docker CLI commands to control access, enhance security, and support various network topologies for containerized applications.

This is possible because of the default driver in docker

Bridge: this is default network driver of docker. All container on the same bridge can communicate each other with out port

Steps:

* + - 1. Start container to allow traffic from port on host machine

Docker container run -p <host\_port>:<docker\_port> -d image

* + - 1. Find the traffic and protocol on container

Docker port <container\_id>

This command will output the **host port**, **container port**, and **protocol**.

Example: If you run docker port my\_nginx, it might output something like 80/tcp -> 0.0.0.0:8080, showing that traffic to the container’s 80/tcp port is mapped to the host’s port 8080.

* + - 1. Find Docker Container IP

Docker inspect<container\_id>

* The -p flag is used for **port mapping**. It binds the container’s internal port (<docker\_port>) to a port on the host machine (<host\_port>).
* host\_port: The port on the host machine that will forward traffic to the container.
* docker\_port: The internal port in the container that the application listens on.
* Example: -p 8080:80 will map port 80 in the container to port 8080 on the host machine.

docker container run -p 8080:80 -d nginx

Here, an Nginx container is created that binds the container’s port 80 to port 8080 on the host machine, allowing you to access it at <http://localhost:8080>.

**Summary:**

* **Run a Container**: docker container run -p <host\_port>:<docker\_port> -d <image>
* **View Port Mappings**: docker port <container\_id>
* **Get Container IP Address**: docker inspect <container\_id> | grep "IPAddress"

A diagram of a bridge

Description automatically generatedThis setup allows the Docker container to handle traffic from a specified port, while also enabling inspection of the traffic protocol and the container’s IP address for additional configuration or troubleshooting.

1. **Docker Network Command Management**

Docker's network management commands allow you to configure and manage network connections for containers, defining how they communicate with each other, with the host machine, and with the outside world.

Docker network ls: shows all networks

We can filter the networks

TO filter all bridge network

Docker network -f drive=bridge

To find all network IDs and Drivers

Docker network ls –format “{{.ID}}:{{.Driver}}”

docker network create <network\_name>

Creates a bridge network

docker network connect my\_bridge\_network my\_container

Connects an existing container to a specified network, allowing it to communicate with other containers on that network.You can also disconnect by replacing connect with disconnect

You can specify a type (e.g., bridge, overlay) using the --driver option:

docker network create --driver bridge my\_custom\_bridge

docker network inspect <network\_name>

Provides detailed information about a specific network, including connected containers, IP address ranges, and subnet information.

1. **Docker network DNS**

Docker Network DNS is an internal system that automatically assigns and resolves container names to their IP addresses, allowing containers to communicate using human-readable names instead of IP addresses within a shared Docker network.

In simple terms, Docker's DNS makes it easy for containers on the same network to find each other without needing to know each other's IP addresses, which might change over time. For example, if you have a database container named db and a web application container named app on the same network, the app container can simply use the name db to connect to the database instead of an IP address. Docker’s DNS service handles name-to-IP mapping automatically, so any time a container starts, stops, or is recreated, Docker updates its internal DNS records. This simplifies configuration and allows containers to work together without needing constant updates for IP changes.

Steps:

* + - 1. Create a docker network

Create a custom bridge network that will support automatic DNS resolution.

docker network create my\_custom\_network

* + - 1. Start containers on network

Run containers and connect them to this network. When connected, Docker will register each container’s name with the internal DNS.

docker run -d --name db --network my\_custom\_network mysql:latest

docker run -d --name app --network my\_custom\_network my\_web\_app\_image

In this example, mysql:latest represents a MySQL database image. Docker assigns this container an IP address within my\_custom\_network and makes it accessible by the name db.

my\_web\_app\_image is a placeholder for the web application image that would connect to the db container. Now, the app container can reference db by its name to access the database.

* + - 1. Configure the Application to Use DNS Name

In your application code (running inside app), configure it to connect to db using the name db instead of an IP. For example, if using a connection string:

DB\_HOST=db

DB\_USER=root

DB\_PASSWORD=example\_password

DB\_NAME=my\_database

* + - 1. Test the connection

To verify that app can reach db using Docker’s DNS, you can open a terminal session in the app container and ping db.

docker exec -it app ping db

1. **Upload Docker Image on docker Hub**

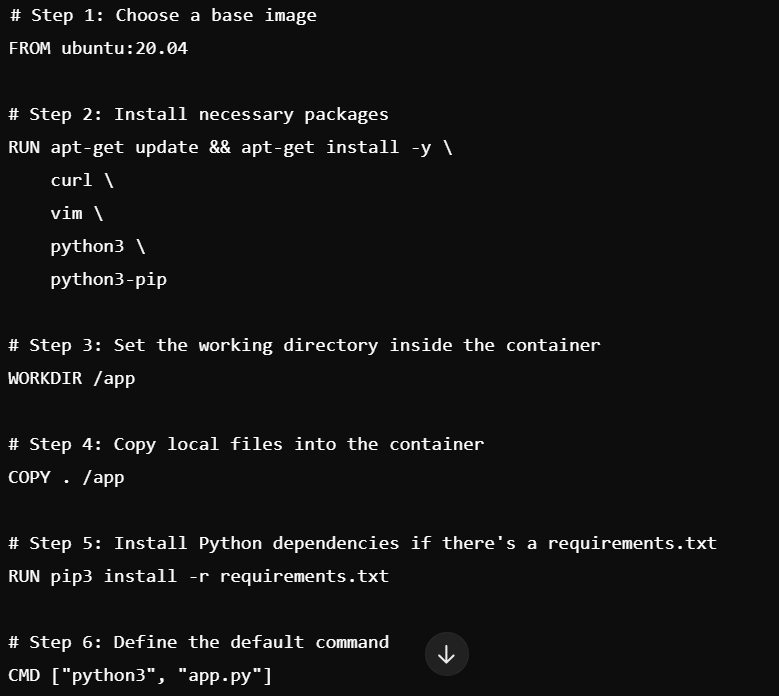
You should have the account of docker hub

Command to login:  
docker login

Push image on docker hub

Docker image push USER/image-name

1. **Create custom docker image**

First create a dockerfile inside a directory with command: echo. > Dockerfile

1. **FROM ubuntu:20.04**: Sets the base image to Ubuntu 20.04.
2. **RUN apt-get update && apt-get install -y curl vim python3 python3-pip**: Updates package lists and installs essential tools like curl, vim, Python 3, and pip inside the container.
3. **WORKDIR /app**: Defines /app as the working directory in the container.
4. **COPY . /app**: Copies the current directory's files into the /app directory in the container.
5. **RUN pip3 install -r requirements.txt**: Installs Python dependencies listed in the requirements.txt file.
6. **CMD ["python3", "app.py"]**: Specifies the default command to run the application (app.py) when the container starts.

then open your directory where docker file is located and run: docker build -t my-custom-image

The -t flag is used to tag the image with a name (my-custom-image), and the . indicates that the Dockerfile is in the current directory.

After image is built run the container by this command: docker run -d --name my-running-container my-custom-image

* + - -d runs the container in detached mode (in the background).
    - --name assigns a name (my-running-container) to the container.
    - my-custom-image is the name of the image you created.

1. **Container Persistent Data Problem**

In Docker, containers are ephemeral by default, meaning data inside a container is lost when the container is removed, restarted, or crashes, leading to a persistent data problem. This can be a major issue for applications needing long-term data storage, like databases, because containerized storage alone doesn’t survive container lifecycles. To solve this, Docker provides options like **volumes** and **bind mounts**, which allow data to persist outside the container's filesystem, linking it to the host machine. Volumes are managed by Docker, making them ideal for long-term storage, backups, and container data sharing, while bind mounts map host directories directly, offering more flexibility for development purposes.

**Docker Volume and Bind mounts**

In Docker, **volumes** and **bind mounts** are two core ways to handle data persistence by connecting a container to the host's filesystem.

**Volumes** are managed by Docker and are stored in Docker’s filesystem (typically /var/lib/docker/volumes on Linux hosts).

A volume is a storage space created and managed by Docker to save data outside of containers, so it doesn’t get erased when containers are deleted or restarted. Think of it as a special storage area just for Docker that’s reliable and easy to back up. Volumes are great for keeping important data safe over time and sharing it between multiple containers.

A bind mount directly links a specific file or directory on the host machine to a path inside the container, allowing the container to access host files directly. Unlike volumes, bind mounts are not managed by Docker; they rely on explicit host paths provided by the user. This makes bind mounts highly flexible and useful for development, as they allow real-time file synchronization, but they can introduce security risks if not managed carefully. Bind mounts are ideal for cases where data on the host needs to be directly accessible within the container, like configuration files, source code, or logs.

**Steps to use Docker volumes and bind mounts to persists data in container**

* + - 1. Create a volume(optional)

Volumes can be created automatically when you run a container, but you can also create them manually for better control.

docker volume create my\_volume

* + - 1. Run container with volume:

To use the volume, specify it when running a container. Here, data written to /data inside the container will persist in my\_volume on the host.

docker run -d --name my\_container -v my\_volume:/data my\_image

The -v my\_volume:/data flag tells Docker to mount the volume my\_volume at the /data path inside the container.

* + - 1. Access the container to add some data to the volume:

docker exec -it my\_container /bin/bash

echo "Hello, Docker Volume!" > /data/hello.txt

This creates a file (hello.txt) inside /data, which is saved in my\_volume and will persist even if the container is deleted.

* + - 1. You can inspect the volume to check the data using the following command:

docker run --rm -v my\_volume:/data alpine ls /data

When you run this command, Docker starts a temporary Alpine container, mounts my\_volume, lists its contents, and then deletes the container.

* + - * + **--rm**: Automatically removes the container after it runs, keeping things clean.
        + **-v my\_volume:/data**: Mounts the Docker volume my\_volume to the /data path inside the container.
        + **alpine**: Uses the lightweight Alpine Linux image for fast startup.
        + **ls /data**: Lists files in /data, showing the contents of my\_volume.

**Using Mind Bound**

1. Choose a folder on your host system that you want to mount to the container, for example:

mkdir -p /path/to/host\_data

This directory will serve as the mount point for the container.

1. Run a Container with a Bind Mount:

Use the -v flag to specify the full path to your host directory, followed by the path where it should appear in the container.

docker run -d --name my\_container -v /path/to/host\_data:/data my\_image

Here, /path/to/host\_data on your computer is directly accessible at /data inside the container.

1. Edit data from the host:

You can add, remove, or edit files in /path/to/host\_data on your host, and those changes will appear immediately in the container’s /data directory. For example:

echo "Hello, Bind Mount!" > /path/to/host\_data/hello.txt

1. Access the container to verify the file is present in /data:

docker exec -it my\_container cat /data/hello.txt

This should output "Hello, Bind Mount!", showing the container sees changes in the mounted folder instantly.

**Bind mounts give you real-time synchronization with your host system, making them great for development, while volumes are better for production or long-term data storage managed by Docker.**

1. **Docker Compose**

Docker Compose is used to run multiple containers as a single service. Docker Compose is a tool for defining and running multi-container Docker applications using a simple configuration file.

Docker Compose simplifies the process of managing multi-container applications. Instead of running individual docker run commands for each container, Docker Compose lets you define all services, networks, and volumes needed for an application in a single docker-compose.yml file. In this file, you specify each service (a containerized component of the application), its configuration, and dependencies. Then, by running docker-compose up, Docker Compose orchestrates the creation and connection of containers, making it easy to start, stop, or scale services with a single command. This approach improves productivity by enabling consistent environments across development, testing, and production.

E.g. User can start mysql and tomcat container with one yml file without starting each saperately.

The three-step process of Docker Compose involves defining, running, and managing your application through a configuration file and Docker Compose commands:

1. **Define Services in docker-compose.yml**: In the first step, you create a docker-compose.yml file where you define the services that make up your application. Each service is described with details like the image to use, ports to expose, environment variables, dependencies, and network or volume configurations. This file essentially describes the "blueprint" for your application's infrastructure.
2. **Run docker-compose up**: Once the docker-compose.yml file is set, you can start your application with the command docker-compose up. This command reads the configuration file and pulls any necessary images, then creates and starts each defined container (service) in the correct order. Docker Compose also sets up the networking between these containers, so they can easily communicate as required.
3. **Manage and Scale with docker-compose Commands**: With the application running, Docker Compose provides a set of commands to manage and control the services. For example, docker-compose down stops and removes containers, networks, and volumes created by up. You can also scale services with docker-compose up --scale service\_name=n, which adds more instances of a specified service. This process makes it easy to start, stop, or scale your application’s components as needed.

Install

* 1. **install Docker Compose**: Docker Compose is usually included with Docker Desktop (for Windows and macOS). For Linux, you may need to install it separately.
  2. A screen shot of a computer program

     Description automatically generated**Create the docker-compose.yml File inside your project directory**: This YAML file defines all your application services and their configurations. For example, here’s a simple setup for a web application with a Python Flask server and Redis for caching:
* version: Specifies the version of the Compose file format.
* services: Each service represents a container. Here, web is the Flask server, and redis is the Redis caching service.
* image: Defines the Docker image to use (e.g., python:3.8 for the Python server).
* ports: Maps host to container ports (5000:5000 exposes port 5000).
* volumes: Mounts your code directory into the container for live changes.

1. **Run mysql and wordpress via docker compose**
   1. Install docker and docker compose
   2. Create a directory to contain docker compose file and configuration

mkdir wordpress-mysql-docker

cd wordpress-mysql-docker

* 1. Create the docker-compose.yml file. Inside it:

**version**: Specifies the version of the Docker Compose file format. Here, 3.7 is used, which is a commonly supported version for most modern Docker Compose setups.

**wordpress**: This is the name of the WordPress service (container).

**image**: wordpress:latest tells Docker to use the official WordPress image from Docker Hub. The latest tag means that it will always use the most recent version of WordPress.

**container\_name**: Specifies a custom name for the container (wordpress).

**ports**: The ports mapping exposes the WordPress web server (running on port 80 inside the container) to your host machine on port 8080. So, you can access WordPress at http://localhost:8080 on your browser."8080:80" means: map port 8080 on your local machine to port 80 on the WordPress container.

**environment**: This section defines environment variables that WordPress will use to connect to the MySQL database.

1. WORDPRESS\_DB\_HOST: The hostname and port of the MySQL service. The value mysql:3306 means it will connect to a service named mysql on port 3306.
2. WORDPRESS\_DB\_NAME: The name of the WordPress database that WordPress will use (wordpress in this case).
3. WORDPRESS\_DB\_USER: The username for connecting to the MySQL database. Here, it’s set to root (the MySQL root user).
4. WORDPRESS\_DB\_PASSWORD: The password for the root user to connect to the MySQL database. In this case, it’s set to example.

**environment variables** are key-value pairs that provide configuration settings to a container at runtime. These variables help configure how a containerized application (like WordPress or MySQL) behaves by passing necessary information, such as passwords, database names, or API keys.

**volumes**: This defines the volume used to persist WordPress data.

wordpress\_data:/var/www/html: This means the directory /var/www/html inside the WordPress container will be mapped to a persistent volume called wordpress\_data. This ensures that the WordPress site data (like themes, plugins, uploads) are not lost when the container is stopped or removed.

**depends\_on**: This ensures that the MySQL container is started before the WordPress container. It ensures that WordPress doesn't attempt to connect to the database until MySQL is ready.

**mysql**: This is the name of the MySQL service (container).

**image**: mysql:5.7 tells Docker to use the official MySQL image version 5.7 from Docker Hub.

**container\_name**: Specifies a custom name for the MySQL container (mysql).

**environment**: Defines the environment variables for MySQL

1. MYSQL\_ROOT\_PASSWORD: The root password for MySQL. In this case, it's set to example. This password is used to authenticate as the root user when connecting to MySQL.
2. MYSQL\_DATABASE: Defines the name of the default database to create on initialization. Here, the wordpress database is created so WordPress can use it to store data.

mysql\_data:/var/lib/mysql: This maps the directory /var/lib/mysql inside the MySQL container (which stores MySQL's data) to a persistent volume called mysql\_data. This ensures that MySQL's data (tables, settings, etc.) is not lost when the container is stopped or removed.

* 1. Run the docker compose

Docker-compose up

* 1. Access the wordpress

http://localhost:8080