**Classical Problem in Software Development: "It works on my machine."**

This common issue is where Docker comes in and solves our problem by **containerizing** all the dependencies along with the application in a single unit.

**What is Docker?**

Docker is a platform that helps us create **containers**.

**Container:**

A container is a bundle that includes our application along with all of its dependencies. It can be shared and run on any system without compatibility issues. It replicates the entire local development environment in a standardized way across large teams.

**Key Properties of Containers**

1. **Portable**: Containers can be shared across teams. Technically, we don’t share the containers themselves—we share **Docker images**.
2. **Lightweight**: Containers have very little overhead. They are simple to create, update, and delete.

* Docker containers and images are typically small in size.
* Each container runs in an isolated environment compared to the host machine.

**Docker Images vs. Containers**

* A **Docker Image** is an executable file that contains instructions to build a container.
* A **Docker Container** is a **running instance** of a Docker image.
* Analogy: **Image = Class**, **Container = Object**
* Multiple containers can be created from a single image.
* The image is a **static blueprint**, whereas the container is dynamic and uses actual resources.

**Key Components**

* **Docker Hub**: A repository of prebuilt Docker images.
* **Docker Daemon**: The core service behind Docker Desktop.
* **Interactive Mode (-it)**: Allows access to the container’s terminal for input and output.

**Basic Docker Commands**

bash

CopyEdit

docker pull <image> # Pull image from Docker Hub

docker run <image> # Run the image and create a container

docker run -it <image> # Access the container terminal

docker run -d <image> # Run container in detached mode (background)

docker images # List all images

docker ps # List running containers

docker ps -a # List all containers

docker start <name> # Start an existing container

docker stop <name> # Stop a container

docker rmi <image-name> # Remove an image

docker rm <container-name> # Remove a container

To delete an image, the containers created from it must be deleted first.

**Image Tags**

Tags are versions or variants of a Docker image (e.g., node:18-alpine, mysql:5.7).

**Environment Variables and Naming Containers**

* Use --env or -e to declare environment variables (e.g., for MySQL passwords).
* Use --name to give a custom name to a container.

**Docker Image Layers**

* Every Docker image consists of **layers**.
* The base layer is usually a lightweight Linux distribution (e.g., Alpine, Debian).
* On top of the base, additional layers are added.
* When a container is created, a **writable layer** (container layer) is added on top.
* All image layers are **read-only and immutable**—only the container layer can be modified.

**Networking in Docker**

* Each container gets a virtual network interface.
* **Ports in containers** are different from those on the host.
* **Port Binding** maps container ports to host ports.

bash

CopyEdit

docker network ls # List networks

docker network create <name> # Create a custom network

Containers on the same Docker network can communicate **without exposing ports** to the host.

**Troubleshooting**

bash

CopyEdit

docker logs <container> # Check container logs

docker exec -it <container> /bin/bash # Run command in a running container

**Docker Compose**

When managing multiple containers, we use **Docker Compose** to simplify orchestration.

* Define multi-container setups in a docker-compose.yml file.
* Docker creates a shared network for all services defined in the file.

bash

CopyEdit

docker compose -f compose.yml up # Start all containers

docker compose -f compose.yml down # Stop and remove all

**Dockerizing an Application**

**Dockerizing** means converting your application into a Docker container.

To do this, we create a **Dockerfile** with instructions:

**Key Dockerfile Commands**

* FROM: Specifies the base image (e.g., node, alpine, ubuntu)
* WORKDIR: Sets the working directory inside the container
* COPY: Copies files from host to image
* RUN: Executes commands while building the image
* CMD: Specifies the default command to run when a container starts (only one CMD allowed)
* EXPOSE: Declares the port on which the container listens
* ENV: Defines environment variables

**Build Docker Image**

bash

CopyEdit

docker build -t <image-name> .

**Docker Volumes (Persistent Storage)**

Docker provides **volumes** as consistent data stores for containers.

* Volumes are isolated from the container and persist even after the container is deleted.
* **Named Volumes** can be created explicitly:

bash

CopyEdit

docker volume create <volume-name>

* Volumes are **not attached to any container by default**.

**Types of Volumes**

1. **Named Volumes** – Reusable and persistent
2. **Anonymous Volumes** – Created automatically if not specified (temporary)
3. **Bind Mounts** – Directly map host directory to container

bash

CopyEdit

docker volume prune # Delete anonymous or unused volumes