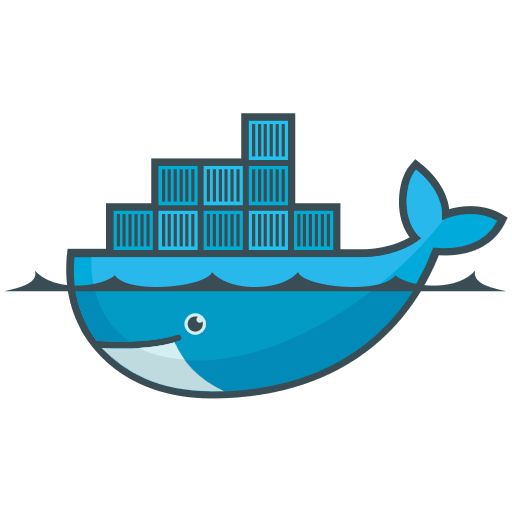
e

**Docker Diaries**

**A Beginner’s Guide to Containerization by Docker**

**Aravind U R**

****

**Container:**

A **software container** is a lightweight, portable package that includes:

* Your application (code)
* All its dependencies (libraries, binaries, runtime)
* Configuration files

This ensures your app **runs the same everywhere**, whether it’s your laptop, a server, or the cloud.

**Why Do Containers Exist?**

Let’s say you’re a developer. You write an app on your laptop, and it works perfectly.

Then you send it to a teammate, or deploy it to a server — and suddenly, it breaks. Why?

* Different OS versions
* Missing dependencies
* Conflicting libraries

💥 This is known as the **"It works on my machine"** problem.

👉 Containers solve this by **bundling your app with everything it needs**, so it works the same **everywhere**.

**Real-Life Analogy: Meal Boxes**

Imagine a chef wants to serve the same dish at different restaurants.

* Without containers: each kitchen uses different ingredients/tools — the dish varies.
* With containers: the chef sends a **sealed meal box** with exact ingredients and instructions.

The result? Same dish, same taste — anywhere.That's exactly what containers do for software.

**Key Characteristics of a Software Container:**

**1. Isolation**

Each container runs its own application with its own files, processes, and network.  
This ensures containers don’t interfere with one another — like individual rooms in a hotel.

*Analogy: Think of containers like tenants in separate apartments — each has its own kitchen, bathroom, and rules, even though they share the same building.*

**2. Portability**

A containerized application runs the same on:

* Your laptop
* A test server
* A production cloud environment

You build it once and run it anywhere!

*Analogy: Like a USB stick that carries a portable version of a program — plug it into any system and it works the same.*

**3. Lightweight**

Containers share the host machine’s operating system kernel.  
This makes them faster and more resource-efficient than virtual machines.

*Example: While a VM needs its own OS (heavy), containers only include the app and its dependencies (light).*

**4. Fast Startup**

Containers can start in seconds, unlike VMs that can take minutes to boot.  
Ideal for scaling apps quickly or running on-demand microservices.

**5. Consistency**

No more "it works on my machine" issues!  
Containers guarantee that the environment is the same across development, testing, and production.

*Result: Fewer bugs, faster testing, smoother deployments*.

**6. Scalability & Flexibility**

You can easily run multiple containers side by side on the same machine.

Containers are perfect for microservices architecture where each service runs independently.

**7. Disposable & Replaceable**

Containers can be stopped, deleted, and replaced without affecting the rest of the system. This helps in rapid updates, rolling deployments, and recovery from failures.

**Container vs Virtual Machine (VM)**

**Virtual Machine (VM)**

A **VM** emulates a full computer system — including its own operating system. Each VM sits on a **hypervisor** and includes:

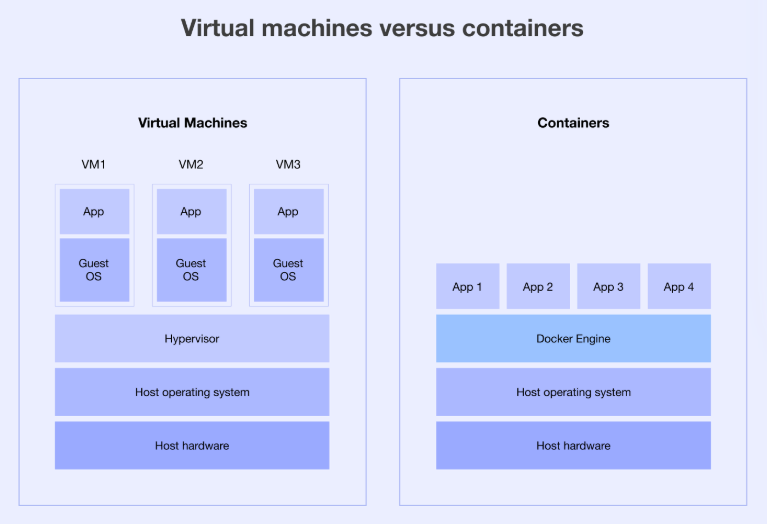
* A **guest OS**
* Application and dependencies and Virtual hardware

**Container**

A **container** only includes the app and its dependencies, but **shares the host OS kernel**. It's like a lightweight package that runs **on top of the OS**.

**Comparison Table: Containers vs VMs**

| **Feature** | **🧱 Virtual Machine** | **📦 Container** |
| --- | --- | --- |
| **Isolation** | Full OS-level isolation | Process-level isolation |
| **Boot Time** | Slow (minutes) | Fast (seconds) |
| **Size** | Heavy (GBs) | Lightweight (MBs) |
| **OS Overhead** | Needs full OS per VM | Shares host OS kernel |
| **Resource Usage** | High CPU & memory | Efficient & minimal |
| **Portability** | Portable, but depends on hypervisor setup | Highly portable, works on any OS with Docker |
| **Use Case** | Running different OSes or legacy systems | Modern apps, microservices, CI/CD pipelines |

****

**🪶 Why Are Containers Lightweight?**

Containers are often described as lightweight — but what does that really mean? Let’s understand it step by step.

**1. They Share the Host Operating System Kernel**

Unlike virtual machines that each run their own OS, containers share the same OS kernel with the host system.

* VM: Has its own full OS (adds bulk)
* Container: Just the app and its dependencies

This saves a lot of space, reduces memory use, and speeds up boot time.

**2. No Need for Virtual Hardware**

VMs simulate full hardware (CPU, RAM, disk) through a hypervisor, which adds extra overhead.

Containers run as isolated processes directly on the host OS — no need to emulate hardware.

**3. Smaller Images**

A container image usually includes:

* The application
* Runtime (like Node.js or Python)
* Required libraries and files

That’s it — no full operating system. Most container images are just a few MBs to hundreds of MBs, compared to GBs for VMs.

**4. Faster Startup**

Because they don’t boot an entire OS, containers can start in less than a second.

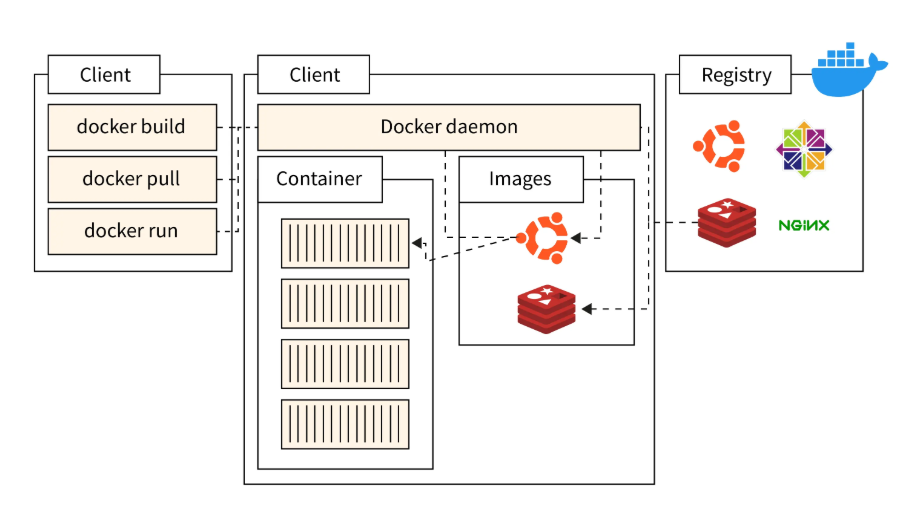
**Docker:**

Docker is a containerization platform that provides easy way to containerize your applications, which means, using Docker you can build container images, run the images to create containers and also push these containers to container registries such as DockerHub, Quay.io, AWS ECR and so on.

In simple words, ***“you can understand as containerization is a concept or technology and Docker Implements Containerization”.***

**Docker Architecture**

Docker architecture follows a **client-server** model. It consists of multiple components that work together to build, ship, and run containerized applications.



Docker is made up of several integrated components that work together to build, ship, and run containerized applications.

**Components of Docker Architecture**

**1. Docker Client:**

The Docker Client is the command-line tool you use to interact with the Docker Engine.

Example command:

***docker run hello-world***

**How it works**:

* Sends commands to the Docker Daemon using the REST API.
* Can be installed on the same host as the daemon or on a remote machine.

**Analogy**: Like a **remote control** you use to operate a smart TV (the daemon).

**2. Docker Daemon (dockerd)**

* **What it is**: The **background service** responsible for managing:
  + Containers
  + Images
  + Volumes
  + Networks
* **How it works**:
  + Listens for Docker API requests.
  + Can communicate with other Docker daemons for managing distributed containers.
* **Key Feature**: It is the **heart** of Docker — does the actual heavy lifting.

**3. Docker Engine**

* **What it is**: The **complete package** that installs:
  + Docker Daemon
  + Docker Client
  + Docker CLI tools
* **Platform**: Available on Windows, macOS, and Linux.
* **Note**: This is what you download when you install Docker Desktop.

**4. Docker Images**

* **What it is**: A **read-only blueprint** of the application.

Often, an image is based on another image, with some additional customisation. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

* **Contents**:
  + Code
  + Libraries
  + Environment variables
  + Dependencies
* **Created Using**: Dockerfile
* **Important**: Images are **layered**, making them space-efficient.

Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualisation technologies.

**5. Docker Containers**

* **What it is**: A **running instance** of a Docker image.
* **Key Features**:
  + Isolated environment
  + Fast startup
  + Uses system resources efficiently
* **Analogy**: Think of the **image as a recipe** and the **container as the prepared dish**.
* **Ephemeral**: Can be started, stopped, deleted, or recreated easily.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

**6. Docker Registries**

* **What it is**: A **storage and distribution system** for Docker images.
* **Popular Options**:
  + Docker Hub (public)
  + Amazon ECR, GitHub Container Registry (private/public)
* **Commands**:

*docker push myimage*

*docker pull ubuntu*

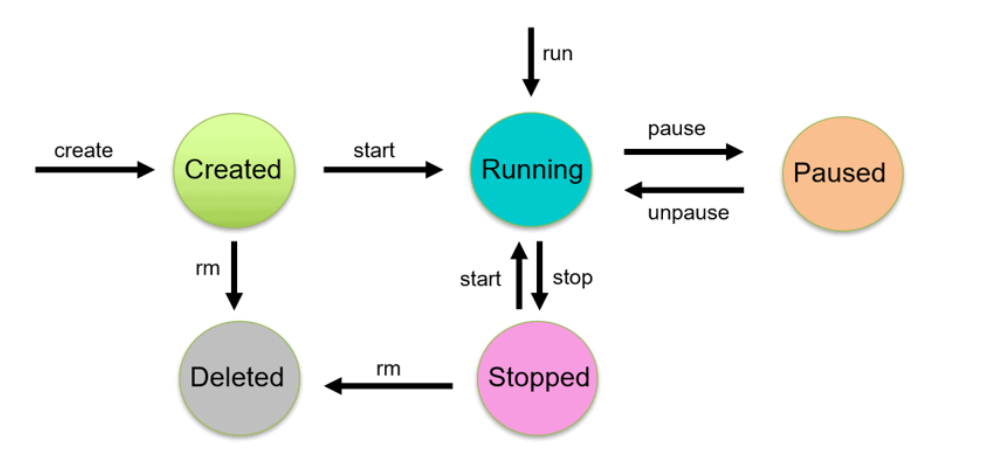
**7. Docker REST API**

* **What it is**: The **communication bridge** between the Docker Client and Docker Daemon.
* **Format**: HTTP over Unix socket or TCP.
* **Used For**:
  + Automating Docker operations
  + Integrating Docker into CI/CD pipelines
* **Example**: GET /containers/json lists all running containers.

**8. Docker Desktop**

Docker Desktop is a simple-to-install application for Mac or Windows that allows you to create and share containerized applications and Microservices. Docker Desktop includes Dockerd, Docker Client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

**Docker lifecycle:**



|  |
| --- |
| **1. Create** |

|  |
| --- |
| A container is created from an image using the ***docker create*** or ***docker run*** command. At this point, it’s not running yet. |

***docker create --name <name-of-container> <docker-image-name>***

**2. Start and Run**

The container begins execution using ***docker start <container\_id>***. It moves to a running state.

In the running state, the docker container starts executing the commands mentioned in the image. To run a docker container, use the docker run command.

***docker run <container-id>***

or

***docker run <container-name>***

The docker run command is basically a shortcut that does **three things** behind the scenes (if needed):

1. **Pull** the image (if it doesn't exist locally)
2. **Create** a container from that image
3. **Start** the container

**4. Pause and Unpause**

In the paused state, the current executing command in the docker container is paused. Use the docker pause command to pause a running container.

***docker pause container <container-id or container-name>***

In the unpaused state, the paused container resumes executing the commands once it is unpaused.

***docker unpause <container-id or container-name>***

**6. Stop and Restart**

Gracefully stops a running container with ***docker stop <container\_id>.*** It allows clean shutdown.

Restarts a stopped container using ***docker restart.***

***7.* Kill**

Forcefully terminates the container immediately with ***docker kill***. No graceful shutdown.

**8. Remove (Delete)**

Removes the container from the system entirely.

***docker rm <container\_id>***

**Diff. b/w Docker Image and Docker Container**

| **Docker Image** | **Docker Container** |
| --- | --- |
| It is a blueprint of the Container. | It is an instance of the Image. |
| Image is a logical entity. | The container is a real-world entity. |
| Images are created only once. | Containers are created any number of times using an image. |
| Images are immutable. One cannot attach volumes and networks. | Containers change only if the old image is deleted and a new one is used to build the container. One can attach volumes, networks, etc. |
| Images do not require computing resources to work. | Containers require computing resources to run as they run with a Docker Virtual Machine. |
| To make a docker image, you have to write a script in a Dockerfile. | To make a container from an image, you have to run the “docker run <image>” command |
| Docker Images are used to package up applications and pre-configured server environments. | Containers use server information and a file system provided by an image in order to operate. |

**Docker Commands**

**Docker Version, info & Help Commands**

* Check Docker Version:

***docker –version***

* Get Full Docker Info:

***docker -info***

Displays details like:

*-Number of containers*

*-Running/stopped containers*

*-Docker root directory*

*-Storage driver*

*-Kernel version*

**Docker Help Commands**

| **Command** | **Description** |
| --- | --- |
| ***docker help*** | Lists all top-level Docker commands |
| ***docker <command> --help*** | Shows help for a specific command |

**Docker Image Commands**

* ***docker pull <image>***

Downloads an image from Docker Hub.

e.g. **docker pull nginx**

it pulls the nginx image from Docker Hub.

* ***docker images***

Lists all images available locally.

* ***docker rmi <image>***

Removes an image from local machine.

* ***docker build -t <name>***

Builds a new image from a Dockerfile in the current directory.

e.g. **docker build -t myapp .**

it builds image with name “myapp” in current directory.

**Docker Container Commands**

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| **docker run** | Creates and starts a new container | docker run hello-world |
| **docker ps** | Lists running containers | docker ps |
| **docker ps -a** | Lists all (including stopped) containers | docker ps -a |
| **docker stop** | Stops a running container | docker stop <container\_id> |
| **docker start** | Starts an existing stopped container | docker start <container\_id> |
| **docker restart** | Restarts a container | docker restart <container\_id> |
| **docker rm** | Removes a stopped container | docker rm <container\_id> |

**Docker Container troubleshooting commands**

* **-it Flag in Docker** - Interactive Terminal

To run an image and get an interactive terminal inside the container.

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

You get a shell (/bin/bash) inside the Ubuntu container.

* **docker exec** — Run a Command in a Running Container

***docker exec -it <container\_id\_or\_name> <command>***

***e.g. docker exec -it myapp /bin/bash***

Opens a terminal into the running container named myapp, Helps for;

--Checking files inside the container

--Debugging running services

--Installing missing packages (temporarily).

* **docker logs** — View Logs of a Container

***docker logs <container\_id\_or\_name>***

Helps for checking:

--App startup issues

--Runtime errors

-- Output from console.log or print

* **docker inspect** — View All Configuration Details

***docker inspect <container\_id\_or\_name>***

Gives low-level system information as JSON:

--IP address

--Environment variables

--Mounts

--Network settings

**Docker Container Management Commands**

| **Command** | **Description** |
| --- | --- |
| **docker ps** | Lists running containers |
| **docker ps -a** | Lists all containers (including stopped) |
| **docker stop <container\_id>** | Stops a running container |
| **docker start <container\_id>** | Starts a stopped container |
| **docker restart <container\_id>** | Restarts a container |
| **docker rm <container\_id>** | Removes a container |

**Docker Clean Up Commands**

| **Command** | **Description** |
| --- | --- |
| **docker system prune** | Removes unused containers, networks, images |
| **docker container prune** | Removes all stopped containers |
| **docker image prune** | Removes dangling (unused) images |

**Docker Compose**

**Docker Compose** is a tool for **defining and running multi-container Docker applications** using a single YAML file (docker-compose.yaml).

💡 **Docker Compose** lets you define and run multi-container applications with a single command. It simplifies managing interconnected services—like frontend, backend, and database—by ensuring consistent, repeatable setups without manually linking containers or remembering complex docker run flags.

**Why Use Docker Compose?**

| **Traditional Way** | **With Compose** |
| --- | --- |
| docker run for each container | One command: docker-compose up |
| Manual networking | Auto networking |
| Manual volume linking | Easy volume setup |
| Hard to manage env vars | .env support built-in |

Most real-world apps rely on multiple services, like an app container needing a separate database container. These services require configurations like volumes, environment variables, and ports. With **docker-compose up**, you can launch the entire stack in one command, ensuring consistent setups across environments.

**Basic Compose Commands**

| **Command** | **What it Does** |
| --- | --- |
| ***docker-compose up*** | Starts all services |
| ***docker-compose up -d*** | Starts in detached mode |
| ***docker-compose down*** | Stops and removes containers, network, volumes |
| ***docker-compose ps*** | Lists all running services |
| ***docker-compose logs*** | Shows logs from all services |

**Benefits of Docker Compose**

| **Benefit** | **Description** |
| --- | --- |
| 🔄Reusability | Easy to share with teams |
| 📂 Project-Based | Isolated setups per project |
| 🔗 Networking | All services auto-connected on the same network |
| 🛠 Environment Variables | Supports .env file for secure configs |
| 🔁 Restart Policies | Auto restart on failure or reboot |

**We will build a real-world Docker application and provide a line-by-line explanation of the Compose file in our next project.**

**Docker Network**

Think of Docker networking as virtual wiring that connects your containers, allowing them to communicate with each other or the outside world.

**Docker Network types**

* **Bridge (default):**

Used for containers on a single host. Each container gets its own IP on a virtual bridge network.

**Bridge** networks are the most suitable option for the majority of scenarios you’ll encounter. Containers in the network can communicate with each other using their own IP addresses and DNS names. They also have access to your host’s network, so they can reach the internet and your LAN.

* **Host:**

The container shares the host’s network stack. No isolation.

**Host** networks are best when you want to bind ports directly to your host’s interfaces and aren’t concerned about network isolation. They allow containerized apps to function similarly to network services running directly on your host.

* **None:**

No networking. Container is completely isolated.

* **Overlay:**

Used for multi-host communication in Swarm mode.

**Overlay** networks are required when containers on different Docker hosts need to communicate directly with each other. These networks let you set up your own distributed environments for high availability.

* **Macvlan:**

Assigns MAC address to container. Makes it appear as a physical device on the network.

networks are useful in situations where containers must appear as a physical device on your host’s network, such as when they run an application that monitors network traffic.

**Docker Network Commands**

* ***docker network ls*** : List networks
* ***docker network inspect bridge*** : Inspect a network
* ***docker network create my\_custom\_net :*** Create a custom network
* ***docker run -d --name web --network my\_custom\_net nginx :*** Run container in a custom network

**Why Use a Custom Network?**

1. Container Name as Hostname: db instead of an IP.
2. Better Isolation: Only connected containers can talk.
3. Required in Compose: Docker Compose automatically creates a network and connects services.

**Dockerfile**

A **Dockerfile** is a text file that contains a series of instructions to build a **Docker image**. Each instruction in the file creates a layer in the image. When you run a container from the image, it executes from this setup.

It defines:

* The **base image** (like Ubuntu, Node.js, Python, etc.)
* Files to be copied
* Software to be installed
* Default commands to run
* Environment setup

**Main Dockerfile Instructions**

**1. FROM**

**Purpose:**

Specifies the **base image** to build from.

**Why:**

Every image needs to start from something, like Ubuntu, Alpine, Python, Node, etc.

**Example:**

***FROM python:3.11-slim***

This starts from a lightweight Python 3.11 image.

**2. WORKDIR**

**Purpose:**

Sets the **working directory** inside the container. All subsequent paths are relative to this.

**Why:**

Keeps paths clean, avoids long absolute paths.

**Example:**

***WORKDIR /app***

Now COPY and RUN will happen in /app.

**3. COPY**

**Purpose:**

Copies files/folders from your local machine (build context) into the image.

**Why:**

You need your code, dependencies, or configs inside the container.

**Example:**

***COPY requirements.txt .***

***COPY . .***

**4. RUN**

**Purpose:**

Runs a shell command **at build time** to set up the image.

**Why:**

Used to install software, set permissions, etc.

**Example:**

***RUN pip install --no-cache-dir -r requirements.txt***

Note: Each RUN creates a new image layer.

**5. CMD**

**Purpose:**

Sets the **default command** to run when the container starts.

**Why:**

Gives a default runtime behaviour, but can be overridden.

**Example:**

***CMD ["python", "app.py"]***

Or shell-style:

***CMD python app.py***

**6. ENTRYPOINT**

**Purpose:**

Defines the **main executable** that runs when the container starts.

**Why:**

Locks the command; arguments can be passed via CMD or docker run.

**Example:**

***ENTRYPOINT ["python", "app.py"]***

***CMD ["--debug"]***

At runtime: python app.py --debug

**7. EXPOSE**

**Purpose:**

Informs Docker (and users) that the container listens on a specific port.

**Why:** Helps with documentation and tools like docker run -P, but doesn’t publish the port.

**Example:**

***EXPOSE 5000***

**8. ENV**

**Purpose:**

Sets an **environment variable** inside the container.

**Why:**

Useful for config (e.g. API keys, mode settings).

**Example:**

***ENV APP\_ENV=production***

You can use it later in RUN, CMD, or your app.

**Comparison Table**

| **Feature** | **ENTRYPOINT** | **CMD** |
| --- | --- | --- |
| Purpose | Fixed command to run | Default args or fallback |
| Overridable? | No(unless with --entrypoint) | Yes (with command-line args) |
| Best for | Containers with single main process | Simple scripts or config |
| Syntax | Exec form preferred | Exec or shell form |

**Tip: Use both for Flexibility**

Using both ENTRYPOINT and CMD gives you:

* A **fixed executable**
* With **flexible defaults**

**Sample Dockerfile (Node.js Example)**

# 1. Use the official Node.js image as the base

**FROM node:18**

# 2. Set the working directory inside the container

**WORKDIR /app**

# 3. Copy package.json and package-lock.json for dependency installation

**COPY package\*.json ./**

# 4. Install app dependencies

**RUN npm install**

# 5. Copy all files from current dir to container's working dir

**COPY . .**

# 6. Expose port 3000 to outside world

**EXPOSE 3000**

# Define ENTRYPOINT and CMD

**ENTRYPOINT ["node"]**

**CMD ["server.js"]**

**Line-by-Line Explanation**

* **FROM node:18**

Chooses the base image (Node.js v18). All builds start from a base image.

* **WORKDIR /app**

Sets the working directory inside the container. Future commands will run from here.

* **COPY package\*.json ./**

Copies dependency files to the container. The \* allows both package.json and package-lock.json.

* **RUN npm install**

Installs the app dependencies using npm.

* **COPY . .**

Copies all files from your local project into the container.

* **EXPOSE 3000**

Informs Docker that the app will run on port 3000.

* **ENTRYPOINT ["node"]**

**CMD ["server.js"]**

ENTRYPOINT :Sets the main command to run — in this case node

Specifies the **default argument** passed to ENTRYPOINT.

***ENTRYPOINT ensures Node is always used***

***CMD allows you to override the script if needed***

**Benefits of Dockerfile**

The following are the benefits of Dockerfile:

* **Consistency and Reproducibility:**

Dockerfile ensures that environment setups and dependencies are consistently facilitated across different setups minimizing the host environment dependent issues.

* **Version Control:**

Dockerfiles can be used for versioning along with your source code, It helps in tracking the changes and rollbacks.

* **Automation:**

It provides the automation with the process of building, configuring, and deploying the applications with reducing the manual intervention and errors.

**Multi-Stage Docker Build**

A multi-stage build is a Dockerfile strategy where you use multiple FROM instructions to build your app in one stage, and copy only the necessary files into a final, minimal image.

**Why Use It?**

* Keeps your final image **clean and small**
* Removes unnecessary build tools, configs, and cache
* Useful when using compilers, package managers, etc.

**How It Works**

You typically:

* Use the **first stage** to compile or build your application.
* Use the **second stage** to copy the built artifacts (binaries, compiled code) into a fresh image.

**Example: Multi-Stage Dockerfile**

**# Stage 1: Build the application**

FROM node:18 AS builder

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build **# for frontend apps or TS -> JS**

**# Stage 2: Run only necessary files**

FROM node:18-alpine

WORKDIR /app

COPY --from=builder /app/dist ./dist

COPY --from=builder /app/node\_modules ./node\_modules

EXPOSE 3000

CMD ["node", "dist/server.js"]

**builder:** Uses full Node.js image to install dependencies and build the app.Only the compiled code and dependencies are in the final image.

**final:** Uses **Alpine Linux** version of Node.js (much smaller). Copies only the build output (dist) and node\_modules. No devDependencies, no source files, no build tools.

**Distroless Image**

A **distroless image** means an image that does **not include an operating system** (like Ubuntu or Alpine).  
It contains **only** the app and the runtime it needs (e.g., Node, Java, Python).

**Benefits:**

The main benefit of using *Distroless* is enhanced security. This heightened security is achieved through several attributes inherent to *Distroless*:

* **Reduced Attack Surface:**

By excluding unnecessary tools, binaries, and shell, Distroless images offer a smaller surface for potential attacks. There’s simply less in the image that can be exploited.

* **Minimized Vulnerabilities:**

With fewer components in the image, there are fewer potential points of failure. This can reduce the number of vulnerabilities and the frequency of required patches.

* **No Shell:**

*Distroless* images don’t contain a shell. This means if an attacker manages to get into the container, they won’t have a shell to execute further malicious commands, making it more challenging to move laterally or escalate privileges.

* **Clearer Dependency Management:**

By including only what’s necessary to run the application, its clearer what dependencies are present, making it easier to manage and update them. This clarity ensures that security patches are more straightforward to track and apply.

**Docker Volume**

A Docker volume is a persistent storage mechanism used by Docker outside the container’s filesystem. It helps preserve data, even when containers are stopped, removed, or rebuilt.

* Docker volumes help you **persist and share data** outside of container lifecycles.
* They’re crucial for **databases, uploads, config files**, and more.
* Managed and isolated = **secure and portable**.

**Docker Volume Commands**

Volumes can be created and managed using the docker volume command. You can create a new volume using the following command:

***docker volume create <volume\_name>***

Once a volume is created, you can mount it to a container using the ***-v or --mount*** option when running a docker run command.

For example:

***docker run -it -v <volume\_name>:/data <image\_name> /bin/bash***

This command will mount the volume <volume\_name> to the /data directory in the container. Any data written to the /data directory inside the container will be persisted in the volume on the host file system.

**Example with Docker Compose**

***version: '3.8'***

***services:***

***db:***

***image: mongo***

***volumes:***

***- mongo-data:/data/db***

***app:***

***build: .***

***ports:***

***- "3000:3000"***

***depends\_on:***

***- db***

***volumes:***

***mongo-data:***

**Explanation:**

* volumes: section defines a named volume mongo-data.
* It is attached to the MongoDB container at /data/db (where Mongo stores data).
* Data will persist even if db container is stopped or deleted.

**Named Volumes and Bind Mounts**

There are two primary types of volumes: Named Volumes and Bind Mounts.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Named Volume** | **Bind Mount (Mount Volume)** |
| **Definition** | Managed by Docker; stored in Docker’s internal path | Maps a **host system directory** to the container |
| **Location** | /var/lib/docker/volumes/... (Linux) | Anywhere on your host filesystem (/home/...) |
| **Creation** | Created by Docker or explicitly by you | You provide the exact path on your system |
| **Use Case** | Great for **persistent data** (e.g. DB) | Ideal for **local dev**, sharing files |
| **Docker-managed?** | Yes | No – You manage the files on host |
| **Portability** | Very portable (Docker will handle it) | Tied to host filesystem path |
| **Backup Friendly?** | Easy to backup using docker volume commands | You need to manually back up host folders |

**Named Volume**

Create:

***docker volume create mydata***

Usage:

***docker run -d -v mydata:/app/data myimage***

* mydata is the **volume name**
* /app/data is the **path inside the container**

Docker handles everything – even creating the volume directory in the backend.

Default Storage location:

***/var/lib/docker/volumes/mydata/\_data/***

**Bind Mount (Mount Volume)**

Usage:

***docker run -d -v /home/user/app:/app myimage***

* Left side /home/user/app is a real folder on your host system.
* Right side /app is the path inside the container.

This gives you real-time access to host files inside your container.

Changes inside the container will affect your local filesystem directly.

Use Case:

* Local development
* Sharing logs or configuration files
* Editing source code in IDE and watching it update in the contain

-End-