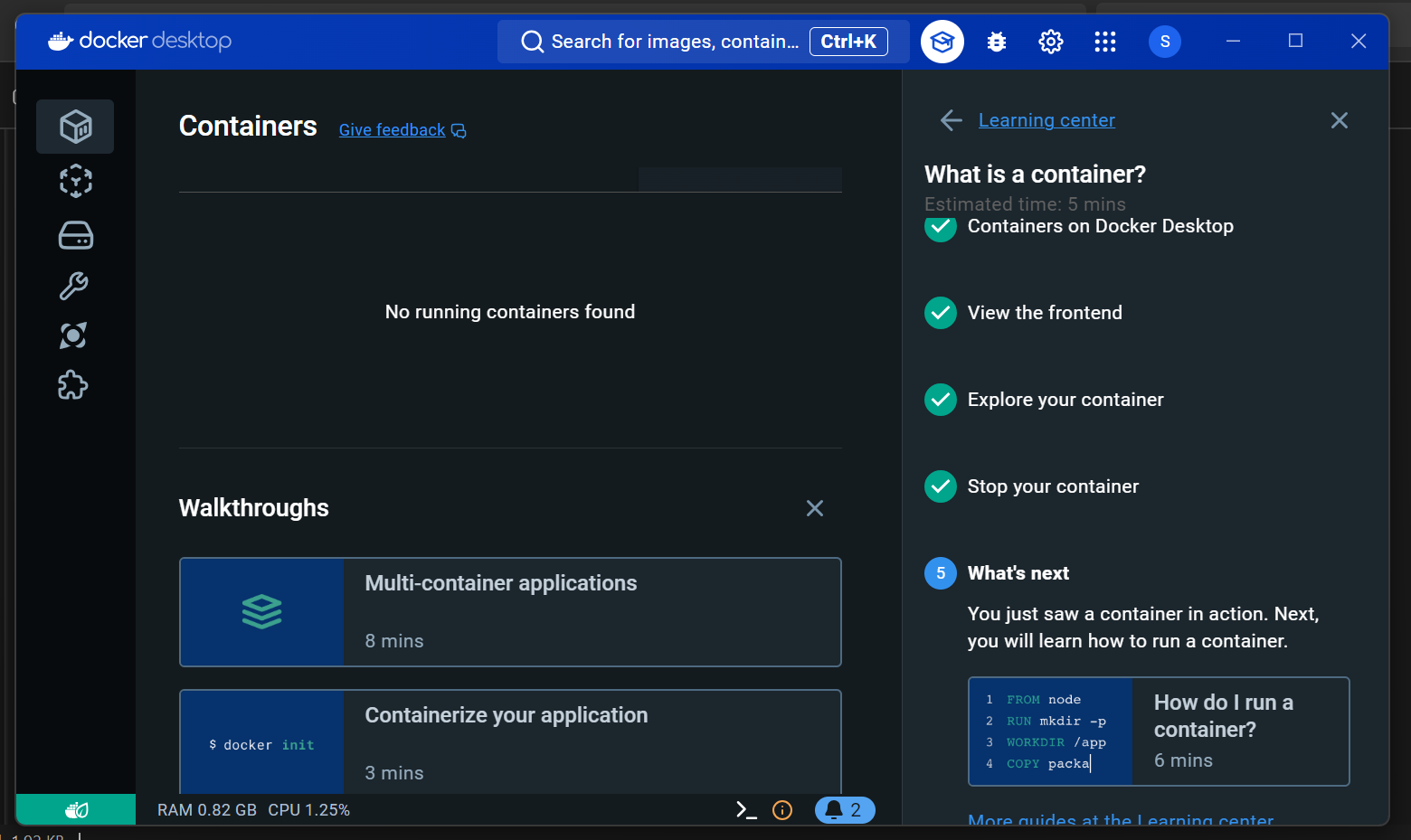
1- DOwnload Docker desktop



ON the LEft bar

1- container -> here i can see the containers which i created and if only want to see running container click on toggle.

2- image -> here i can see images

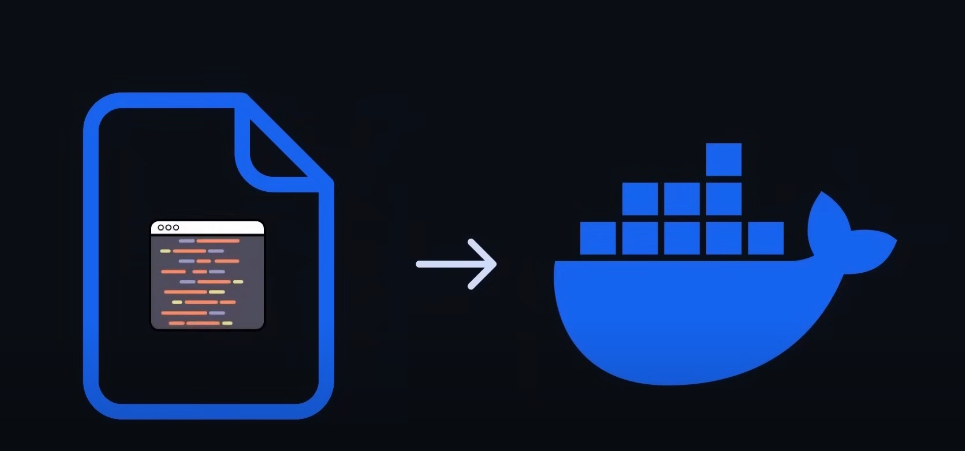
3- volume -> here we can check volume.

Then , i will go to Docker hub (remember Docker hub is default for importing images.) there you will find that so many public images are there, you can create private as well.

Now you have seen so many Images created on Docker Hub, But how we can create our own image so lets see below:



Now for creating our own Docker Image we have take start from Docker file:



Docker File is basically a set of instruction that tell the Docker that how to create Image.

**\*\*Dockerfile\*\***

A \*\*Dockerfile\*\* is a text file that contains a series of instructions used to build a Docker image. It’s essentially a script that tells Docker how to assemble the image step by step, starting from a base image, adding application code, dependencies, and configuring settings.

**\*\*Important Factors of a Dockerfile\*\*:**

1. \*\*Base Image (`FROM`)\*\*:

- \*\*What It Does\*\*: Specifies the starting point for your Docker image, usually an official image like `ubuntu`, `alpine`, or `node`.

- \*\*Why It’s Important\*\*: The base image determines the environment your application will run in. It’s the foundation upon which everything else in the Dockerfile is built.

2. \*\*Instructions\*\*:

- \*\*What They Do\*\*: Commands like `RUN`, `COPY`, `ADD`, `CMD`, `ENTRYPOINT`, `ENV`, etc., are used to build the image.

- \*\*Why They’re Important\*\*: These instructions define the environment, install dependencies, copy application files, and specify how the container should behave when it runs.

3. \*\*Layers\*\*:

- \*\*What They Are\*\*: Each instruction in a Dockerfile creates a new layer in the Docker image.

- \*\*Why They’re Important\*\*: Layers allow for efficient storage and management. If a layer hasn’t changed, Docker can reuse it, speeding up the build process.

4. \*\*Caching\*\*:

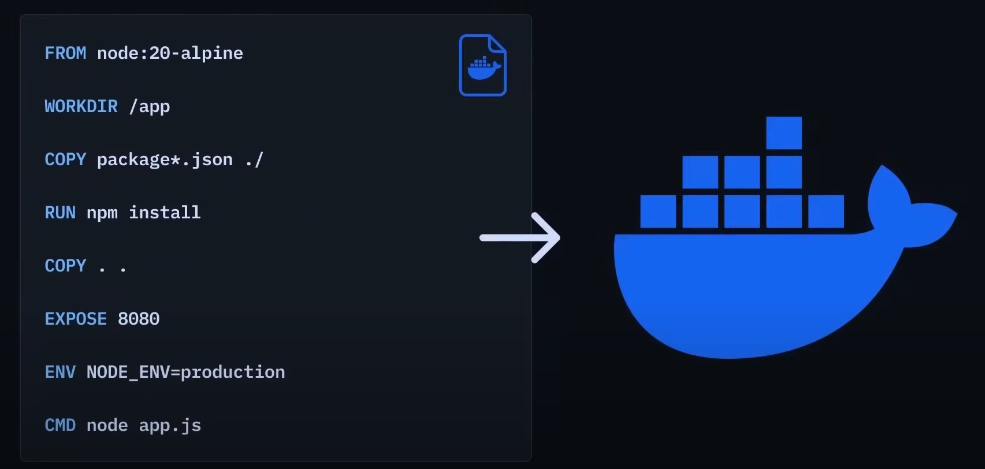
- \*\*What It Does\*\*: Docker caches layers to avoid rebuilding unchanged parts of the image.

- \*\*Why It’s Important\*\*: This makes the build process faster and more efficient, as only the layers that have changed need to be rebuilt.

5. \*\*CMD vs. ENTRYPOINT\*\*:

- \*\*What They Do\*\*: Both `CMD` and `ENTRYPOINT` define the default command to run when a container is started.

- \*\*Why They’re Important\*\*: `ENTRYPOINT` is generally used to define the main command, while `CMD` can provide default arguments or be overridden by user inputs. Understanding their differences allows better control over container behavior.



Basically, we define some instruction in Docker file and you can also think of language in which we are writing instruction as **Docker syntax.**

**Below are some of commands:**



The `FROM` command in a Dockerfile is one of the most fundamental and crucial instructions. It specifies the base image that your Docker image will be built upon. Here's how it works:

**\*\*How the `FROM` Command Works\*\*:**

1. \*\*Specifying the Base Image\*\*:

- The `FROM` command tells Docker which base image to use as the starting point for your new image.

- Example:

```Dockerfile

FROM ubuntu:20.04

```

In this example, the base image is Ubuntu version 20.04.

2. \*\*Downloading the Base Image\*\*:

- If the specified base image is not already present on your local system, Docker will automatically pull it from the Docker Registry (usually Docker Hub).

- Docker checks if the image is available locally. If not, it downloads the image layers from the registry.

3. \*\*Foundation for Subsequent Instructions\*\*:

- The base image specified by the `FROM` command provides the environment (like the operating system, libraries, and tools) for the subsequent instructions in the Dockerfile.

- All other instructions (like `RUN`, `COPY`, `ADD`, etc.) build on top of this base image, adding layers that customize and extend the image.

4. \*\*Multiple `FROM` Commands\*\*:

- A Dockerfile can have multiple `FROM` commands if you want to create a multi-stage build. Each `FROM` command starts a new stage.

- Example:

```Dockerfile

FROM golang:1.16 AS build

# Build your application here

FROM alpine:latest

# Use the build artifacts from the previous stage

```

- In this example, the first `FROM` uses a Go language image to build an application, and the second `FROM` uses an Alpine image for the final, lightweight image that contains only the necessary artifacts.

**\*\*Key Points\*\*:**

- \*\*Primary Function\*\*: The `FROM` command sets the starting point for your Docker image, determining the base environment.

- \*\*Single Stage\*\*: In simple Dockerfiles, there's usually one `FROM` command at the top that defines the base image.

- \*\*Multi-Stage Builds\*\*: In more complex builds, multiple `FROM` commands can be used to create intermediate stages, which can optimize the final image size and build process.

**\*\*Summary\*\*:**

So basically **FROM** command ka through base Image define krtay hain which means kay jo new Image bnegi wo kis ki top pa create hogi. Ab jo base Image hoti hai that is basically consisting all thing that your app require to run such as: OS, packages , runtime etc. isi lia Base image ko define krna neccessary hai otherwise Docker will not create new image.

But agar hum koi be base image nhi define krna chahtay toh phr be base image toh deni hogi bcuz its require but wo jo base image dengay that is a empty image and that image is known as **Scratch.**

**\*\*What is `scratch`?\*\***

- \*\*Definition\*\*: `scratch` is a special, empty base image in Docker. It represents a \*\*completely minimal starting point\*\* for a Docker image, meaning it has nothing included—no operating system, no libraries, no utilities, nothing at all.

- \*\*Purpose\*\*: The `scratch` base image is typically used for creating very lightweight containers, often for simple, single-purpose applications that don't need the full operating system or standard libraries.

\*\*How Does It Work?\*\*

- \*\*No OS\*\*: When you use `scratch` as your base image, there is no operating system in the image. The only things in the image are the files you explicitly add to it, such as your application binary.

- \*\*Minimalist Approach\*\*: This is useful when you want to minimize the size of the image. For instance, if you have a statically compiled Go binary, you might use `scratch` to create an image that contains only your binary and nothing else.

\*\*Example\*\*:

```Dockerfile

FROM scratch

COPY myapp /myapp

CMD ["/myapp"]

```

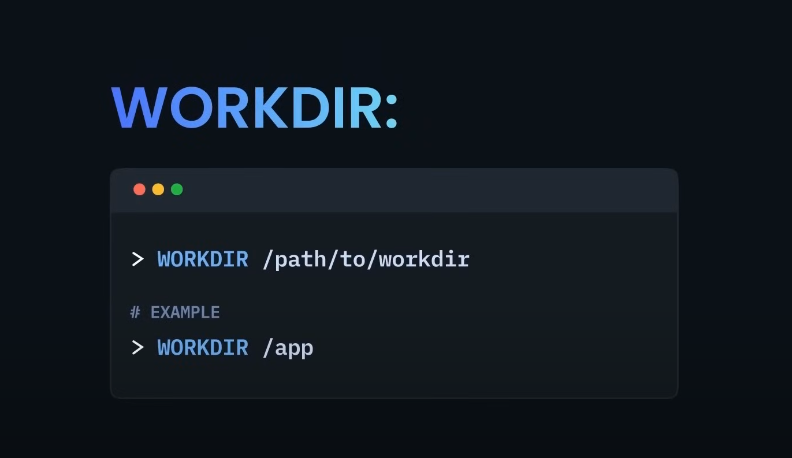
In this example:

- The `scratch` image doesn't include an OS or any utilities.

- The only file in the container will be the `myapp` binary that you copied in.

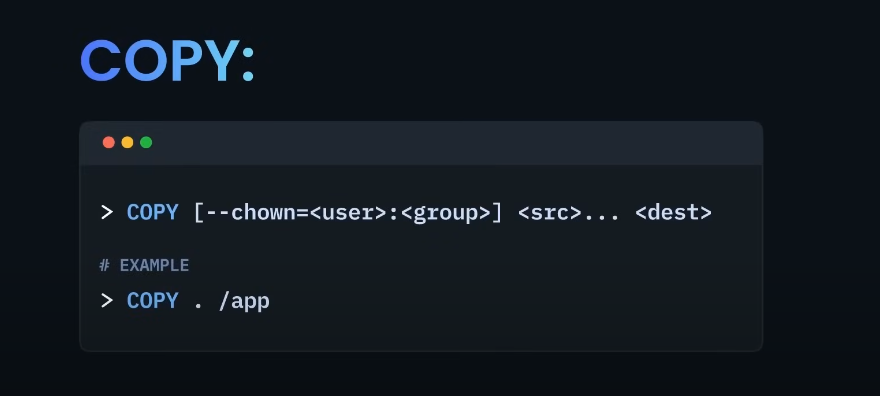
\*\*Interaction with the Host OS\*\*:

- When a container based on `scratch` runs, it relies directly on the host system’s kernel. There is no user-space operating system in the container itself; it’s as bare-bones as possible.



WORKDIR (working directory) -> It specify that in which directory/folder the all instruction/work will execute.

WORKDIR in a Dockerfile sets the working directory for all subsequent commands. It ensures that all operations occur in the correct directory, simplifies path management, and automatically creates the specified directory if it doesn’t exist. This feature helps keep Dockerfiles clean, organized, and efficient.



COPY -> this command is use to copy all the files, directories to the Working Directory which you will specify.

The COPY command in a Dockerfile is used to copy files and directories from your local filesystem (the build context) into the Docker image. It’s one of the primary instructions used to bundle your application’s code, configuration files, or any other necessary assets into the Docker image.

**Basic Usage:**

Dockerfile

**COPY <source> <destination>**

<source>: The path to the file or directory on your local system that you want to copy into the image.

<destination>: The path inside the Docker image where you want the file or directory to be copied.

**Key Features:**

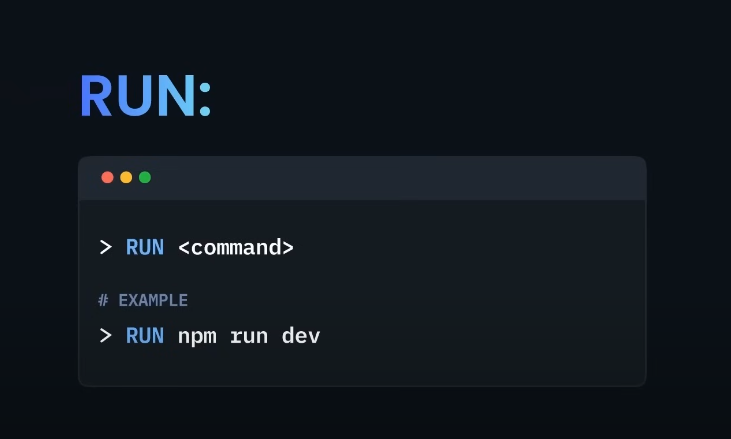
Simple File Transfer: Copies files/directories without additional processing.

Multiple Files: Supports copying multiple files/directories using wildcards.

Directory Preservation: Keeps the directory structure intact.

Build Context: Only accesses files within the build context.

Efficient Layering: Creates new image layers, benefiting from Docker’s caching if files haven't changed.



RUN -> This command is use to execute instructions/command in the shell during image build.

**RUN Command in Dockerfile**

The RUN command in a Dockerfile is used to execute commands inside the Docker image during the image build process. It’s one of the most commonly used instructions and is crucial for installing software, configuring the environment, and setting up the application within the Docker image.

**How It Works:**

Execution During Build: The RUN command executes the specified command(s) in a new layer on top of the current image and commits the results. Each RUN command creates a new layer in the image.

Purpose: It’s typically used to install packages, set environment variables, or configure the system in the way that the application requires.



Expose -> Yeh basic docker ko btata haka iss image say jo container create hoga wo srf iss specific port pa run hoga.But yeh port ko publish nhi krta mtlb kay yeh srf btata hai document krta hai kay bhai agr apko iss image say container build krkay usko run krwana hai toh iss port pay run hoga. you can define more than one port as well.

**\*\*EXPOSE Command in Dockerfile\*\***

The `EXPOSE` command in a Dockerfile is used to \*\*indicate\*\* which port(s) the container will listen on at runtime. It’s a way of \*\*documenting\*\* or \*\*declaring\*\* the network ports that the containerized application will use, but it doesn’t actually publish the ports to the host system on its own.

**\*\*Important Features of EXPOSE\*\*:**

1. \*\*Documentation\*\*:

- \*\*What It Does\*\*: `EXPOSE` serves as documentation within the Dockerfile to inform users and developers which ports the container is expected to be listening on.

- \*\*Why It’s Important\*\*: Helps others understand the intended network setup of the container, making it easier to work with and troubleshoot.

2. \*\*Internal Use\*\*:

- \*\*What It Does\*\*: It tells Docker that the container will listen on the specified port(s), which can be useful for linking containers together or when using Docker’s networking features.

- \*\*Why It’s Important\*\*: It’s used internally by Docker for automatic service discovery and container communication, especially in multi-container environments.

3. \*\*No Port Publishing\*\*:

- \*\*What It Does Not Do\*\*: `EXPOSE` by itself does not make the port accessible from the host or outside the container. To actually expose the port to the host system, you must use the `-p` or `-P` flag when running the container (e.g., `docker run -p 8080:8080`).

- \*\*Why It’s Important\*\*: Users must explicitly publish the ports they want accessible, giving them control over which ports are exposed to the outside world.

4. \*\*Supports Multiple Ports\*\*:

- \*\*What It Does\*\*: You can use multiple `EXPOSE` instructions to specify that the container listens on more than one port.

- \*\*Why It’s Important\*\*: Useful for applications that use multiple network ports, like web servers and databases that might need to listen on HTTP, HTTPS, and other ports.

**\*\*Example\*\*:**

```Dockerfile

FROM nginx:alpine

EXPOSE 80

EXPOSE 443

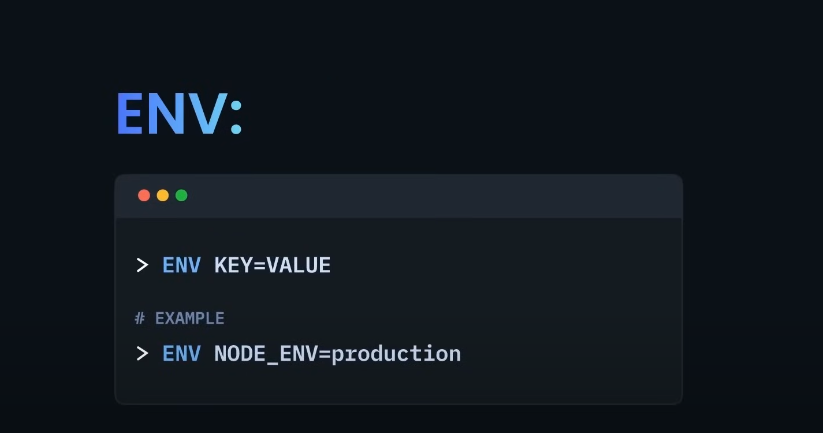
```

In this example:

- The `EXPOSE 80` and `EXPOSE 443` commands indicate that the container will listen on ports 80 and 443 (standard ports for HTTP and HTTPS, respectively).

**\*\*Summary\*\*:**

The `EXPOSE` command in Dockerfile is primarily used for documentation and to define the network ports that the container will listen on. While it doesn't actually publish these ports, it helps in setting up and understanding the container's networking and is an essential part of good Dockerfile practices.



ENV -> The ENV instruction in a Dockerfile is used to set environment variables inside a Docker container. These environment variables can be used by the application running inside the container or by other instructions in the Dockerfile. It helps in configuring the containerized application without hard-coding values directly into the code.

**\*\*Important Features of `ENV`:\*\***

2. \*\*Persistence Across Layers\*\*:

- \*\*How It Works\*\*: The environment variables set using `ENV` persist across all the layers that follow it in the Dockerfile. This means they remain available for subsequent commands and within the final container.

- \*\*Example\*\*:

```Dockerfile

ENV PATH=/usr/local/myapp/bin:$PATH

```

This example adds a directory to the `PATH` environment variable, and it will be available in the entire image.

3. \*\*Parameterizing Builds\*\*:

- \*\*Purpose\*\*: You can use `ENV` to make your Dockerfile more flexible and reusable by setting values that might change based on the environment (e.g., development, testing, production).

- \*\*Example\*\*:

```Dockerfile

ENV DB\_HOST=db.example.com

ENV DB\_USER=root

ENV DB\_PASS=secret

```

These variables could be used to configure database connections in your application.

4. \*\*Overriding Defaults\*\*:

- \*\*Flexibility\*\*: Environment variables set with `ENV` can be overridden at runtime using the `-e` flag with `docker run`. This allows you to change the behavior of your container without modifying the Dockerfile.

- \*\*Example\*\*:

```bash

docker run -e APP\_ENV=development my\_container

```

Here, `APP\_ENV` is overridden to `development` when running the container.

5. \*\*Multi-Line Variables\*\*:

- \*\*Convenience\*\*: You can define multi-line environment variables using the backslash (`\`) for better readability.

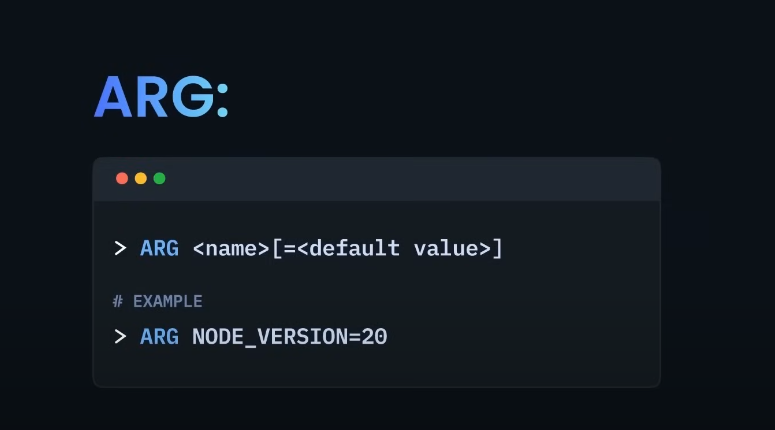
- \*\*Example\*\*:

```Dockerfile

ENV MY\_VAR="This is a \

multi-line variable."

```



**\*\*What is ARG in Dockerfile?\*\***

`ARG` is a Dockerfile instruction that defines a variable that users can pass at build time to customize the image build process. These variables are used during the Docker image build and can be given default values or overridden by providing values during the build command.

**\*\*Important Features of ARG\*\*:**

1. \*\*Build-Time Variables\*\*:

- `ARG` variables are only available during the image build process and are not accessible in the final container.

2. \*\*Default Values\*\*:

- You can set default values for `ARG` variables in the Dockerfile. If no value is provided during the build, the default is used.

3. \*\*Overriding Values\*\*:

- Users can override `ARG` values at build time using the `--build-arg` flag.

- Example: `docker build --build-arg MY\_VAR=value .`

4. \*\*Scope Limitation\*\*:

- `ARG` variables can only be used in the stages of the Dockerfile after they are declared.

\*\*Example\*\*:

```Dockerfile

ARG VERSION=latest

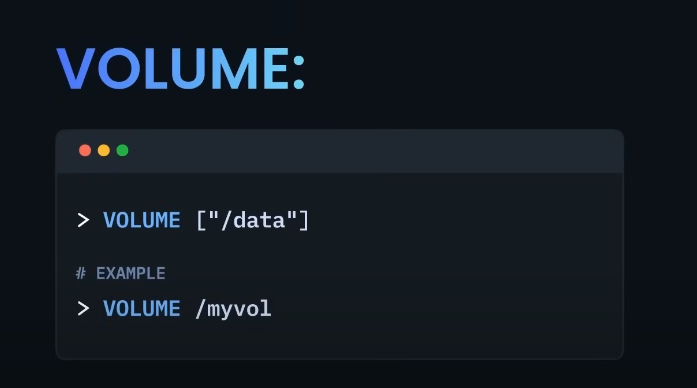
FROM ubuntu:$VERSION

```

Here, `VERSION` is an `ARG` variable that defaults to `latest` but can be overridden at build time.

# Basically yeh jo ARG hai isko build argument be kehtay kiu kay yeh build time pa access hota hai docker file may. Acha yeh basiaclly simple variables ki tarah hai jasay but jo difference hai ya isko use krnay ka kya maqsaad hai.

# Toh basically iska faida yeh haka we can directly assign value also like below we can directly assign 'Node version' , but let suppose kay we want to change the version of node or the base Image at 'build time' so you would do that .



VOLUME -> The VOLUME command in a Dockerfile is used to create a mount point with a specified path and mark it as a location where data can be stored outside the container's filesystem. This allows data to persist even after the container is removed.

basically yeh container ma hum point define krdetay hain kay iss point pa data store hoga. like humnay ek HTML form ka container bnaya hai toh usme jab yeh point define krdia toh later jo be forms ka data hoga wo iss point pa store hojayega takay agr incase container crash hojaye ya remove hojaye but data toh Volume may store hoga na.

Key Features of the VOLUME Command:

Persistent Data Storage:

Ensures that data stored in the specified directory is not lost when the container stops or is deleted.

Host-Container Data Sharing:

Allows sharing of data between the host system and the container, making it accessible both inside and outside the container.

Shared Volumes Across Containers:

Multiple containers can share the same volume, enabling them to read and write to the same data.

Decoupling Data from Containers:

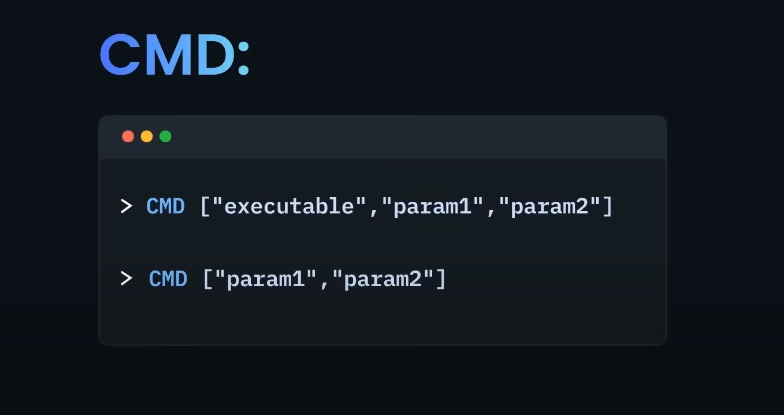
Keeps the data separate from the container's lifecycle, making it easier to manage and back up.

Example:

Dockerfile

**VOLUME /data**

This command creates a volume at /data in the container, which will be persisted outside the container's filesystem.



**\*\*CMD Command in Dockerfile\*\***

The `CMD` command in a Dockerfile specifies the default command that should run when a container is started from the Docker image. It sets the command that will be executed inside the container, but it can be overridden if a different command is specified at runtime.

**\*\*Important Features of `CMD`\*\*:**

1. \*\*Default Command\*\*:

- `CMD` provides a default command that runs when the container starts. If you don't specify a command when running the container, Docker uses the one defined by `CMD`.

2. \*\*Overridable\*\*:

- The command specified by `CMD` can be overridden by providing a command in the `docker run` command. For example, `docker run <image> <different-command>` will replace the `CMD` command.

3. \*\*Syntax\*\*:

- `CMD` can be written in three forms:

- \*\*Exec form\*\* (recommended): `CMD ["executable", "param1", "param2"]`

- \*\*Shell form\*\*: `CMD command param1 param2`

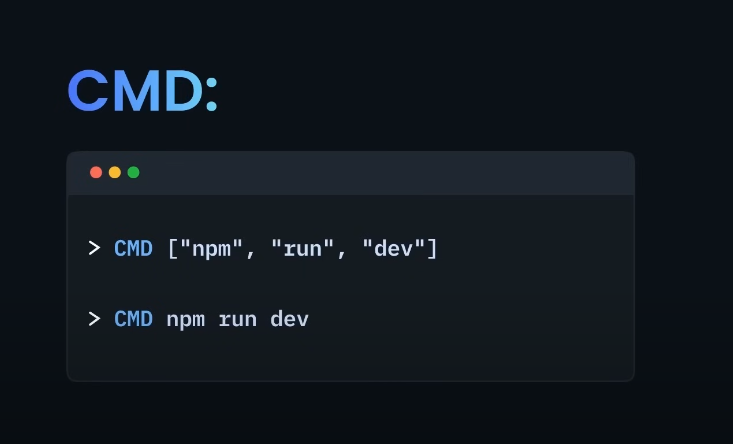
- \*\*Inline\*\*: `CMD command` (uses the shell form under the hood)

4. \*\*Single Use\*\*:

- Only the last `CMD` in a Dockerfile is effective if multiple `CMD` instructions are specified; the previous ones are ignored.

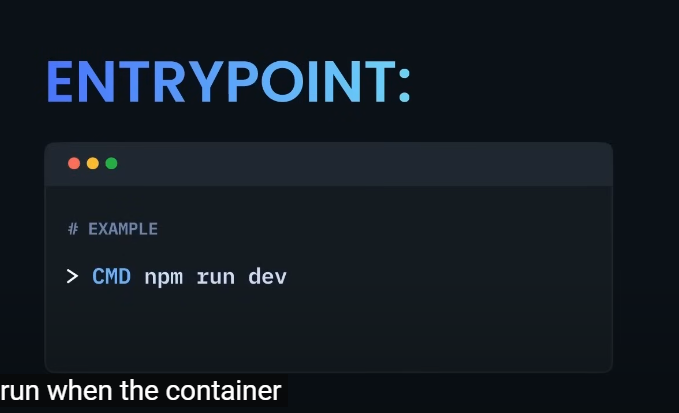
\*\*Summary\*\*:

The `CMD` command sets the default command that runs when a Docker container starts. It’s overridable, allowing flexibility at runtime, and should be used to define the primary action or process that the container is expected to execute.



Like in above example you can see that jo 1st parameter hai usme jo be runtime hai jispa humari application run hogi uski executeable command pass krni hai in our case it is Node . or Ab jasay humna yeh command as a default rakhdi hai toh it will automatically execute when container will start.

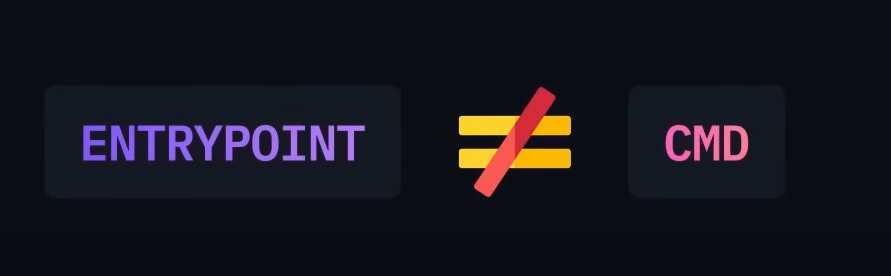
Acha yeh jo uper CMD command 2 dafa hai , actually these both are two syntax of writing CMD command 1st one is called **executable form** (recommended to use) and 2nd one is called **shell form.** Both do similar task.



ENTRYPOINT \_-> now entrypoint is also the command that will execute by default when you will start the container.

The ENTRYPOINT command in a Dockerfile specifies the main command that will always run when a container starts. It defines the executable that should be run as soon as the container is launched.

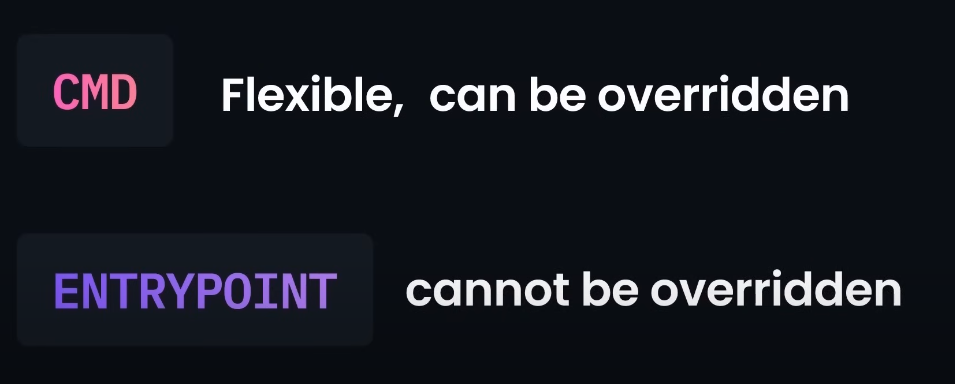
Now Here a Question kay Kya CMD and ENTRYPOINT same nhi hai ? bcuz both are doing same work :



No ENTRYPOINT is not same as CMD , there is a key difference between both:

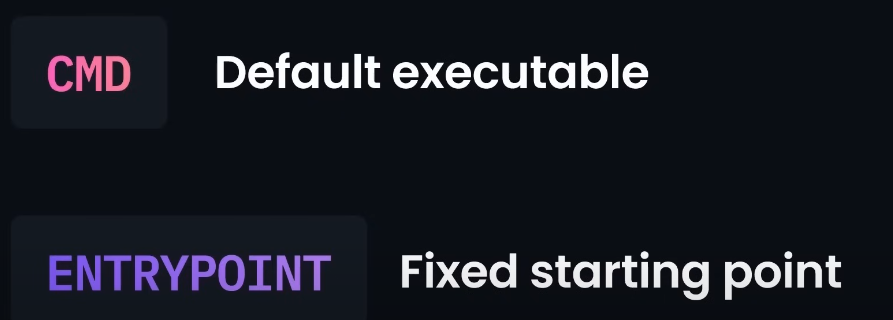


In simple terms both EntryPoint and CMD is a default instruction that run when container starts but there is a slight difference between them.



CMD -> jo isme default command ayegi wo runtime pay overridden hoskti hai.

ENTRYPOINT -> But jo entrypoint ka through command hogi wo easily override nhi hogi.



But agar CMD and ENTRYPOINT dono use hongay toh phir kya hoga?

Agar `CMD` aur `ENTRYPOINT` dono ek Dockerfile mein use hote hain, toh unka interaction kuch is tarah hota hai:

**\*\*How `CMD` and `ENTRYPOINT` Work Together\*\*:**

1. \*\*`ENTRYPOINT` Defines the Main Command\*\*:

- `ENTRYPOINT` command specify karta hai ke container start hote hi konsa command ya executable run hoga. Yeh command immutable hota hai, matlab yeh hamesha run hoga jab container start hoga.

2. \*\*`CMD` Provides Default Arguments\*\*:

- `CMD` typically `ENTRYPOINT` ke command ke liye default arguments provide karta hai. Agar `ENTRYPOINT` ke baad `CMD` specify hoti hai, toh `CMD` ka value as an argument pass hota hai `ENTRYPOINT` ke command ke saath.

### \*\*Example\*\*:

```Dockerfile

ENTRYPOINT ["python"]

CMD ["app.py"]

```

In this example:

- \*\*`ENTRYPOINT`\*\*: Defines `python` as the command that will always run when the container starts.

- \*\*`CMD`\*\*: Provides the default argument `app.py`.

\*\*When the Container Runs\*\*:

- \*\*Result\*\*: The container will execute `python app.py`. Yani `python` command ke saath `app.py` file run hogi, jo `CMD` se aayi hai.

\*\*Key Points\*\*:

- \*\*Override CMD\*\*: Agar container run karte waqt `CMD` se diya gaya argument override kiya jaye, toh aap naya argument de sakte hain. For example: `docker run mycontainer hello.py` will run `python hello.py` instead of `python app.py`.

- \*\*Immutable `ENTRYPOINT`\*\*: `ENTRYPOINT` change nahi hota jab tak specifically `--entrypoint` flag use na kiya jaye `docker run` command mein.

\*\*Summary\*\*:

Agar `ENTRYPOINT` aur `CMD` dono use hote hain, toh `ENTRYPOINT` command ko define karta hai aur `CMD` uske liye default arguments provide karta hai. `CMD` ke arguments ko runtime par override kiya ja sakta hai, lekin `ENTRYPOINT` hamesha execute hoga.