Container Network - Building Docker Image Steps

Building Docker Images from a Container

Building Docker Images from a Container
Building Docker images from a container involves creating a container from an existing image, making modifications, and then committing those changes to form a new image.

In this lab, we'll learn how to create a Docker image by making changes to a container and committing those changes. We'll start with a simple example where we create a file in a container and then commit this change to form a new image.

How It WorksA
When we create a Docker container, it uses a Union File-System (UFS) mount to provide its filesystem. Any changes made to the filesystem within the container are written as new layers, which are owned by the container that created them.

To build a new image, we start with an existing image, make changes to it by modifying the container's filesystem, and then commit these changes to form a new image. This new image can then be used to create further containers, encapsulating the changes made.

alt text

How UFS Works
Base Layer: This is the original, unchanged filesystem, like a basic Linux operating system.

Layers: Each time you make changes (like installing software or creating files), these changes are saved as new layers on top of the base layer.

Union Mount: The union filesystem merges these layers into a single, cohesive filesystem that the container uses.

alt text

The above figure demonstrates how UFS works.

Task
Create a container from the ubuntu:latest image and modify its filesystem by creating a file named HelloWorld.
Commit the changes made in the container to a new image named hw_image.
Remove the modified container to clean up.
Verify the changes by running a new container from the hw_image and checking the existence of the HelloWorld file.
alt text

Solution
Follow these steps to complete the task:

Create a container and modify its filesystem:

Create a container from the ubuntu:latest image and enter the container bash:

docker run -it --name hw_container ubuntu:latest /bin/bash
Create a file named HelloWorld.txt in the container:

touch HelloWorld.txt
Exit the container:

exit
Commit the changes to a new image:

Here we are creating a new image from our container:

docker container commit hw_container hw_image
We can see the new image using the following command:

docker images
Expected output:

term@ubuntu-xhb8pe-6bc6869699-vmmr9:~$ docker container commit hw_container hw_image
sha256:3d6172495dcf3736de54f049f32f21f6816dd2766c4c85647edfe6e2f191238d

term@ubuntu-xhb8pe-6bc6869699-vmmr9:~$ docker images
REPOSITORY   TAG   IMAGE ID   CREATED   SIZE
hw_image   latest   3d6172495dcf   38 seconds ago   76.2MB
ubuntu   latest   bf3dc08bfed0   4 weeks ago   76.2MB
Remove the modified container:

Let's delete the existing container:

docker container rm -vf hw_container
Create new Container from the new Image:

Let's create a new container from the new image that we have created and enter the bash in the new container:

docker run -it --name new_hw_container hw_image /bin/bash
Use the following comand to see the files:

ls
Expected output:

term@ubuntu-xhb8pe-6bc6869699-vmmr9:~$ docker run -it --name new_hw_container hw_image /bin/bash

root@2d9b5c80997a:/# ls
HelloWorld.txt bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
We can see the HelloWorld.txt in this container from our new image!

This output confirms that the file HelloWorld was successfully created in the new image, demonstrating that the modifications made in the original container were correctly committed to the new image.

Cleanup
Remove the container and image:

docker rm new_hw_container
docker rmi hw_image

explain these in a very easiest manner in details with basics also correct if any error occur; what does these codes do with basics of each part and portion with basic example if necessary:

Sure! Let's go step-by-step and **explain everything in a beginner-friendly and detailed way**, including what the commands do, why we use them, and correcting any minor points where needed. Think of this as a full **beginner's guide** to building Docker images **from a running container**.

🔷 What Are Docker Images and Containers?

- **Image** = A *read-only template* that contains everything needed to run an application (like a mini OS, software, libraries).
- **Container** = A *running instance* of an image. Think of it like turning on a computer from a disk image.

🔸 Why Modify a Container and Commit to an Image?

Imagine you start with a clean Ubuntu system (image), install some software or create some files (modifications), and then want to save this modified version as a **new image** so you don't have to repeat the steps every time.

✅ TASK: Create a file inside a container, save changes as an image

---

1️⃣ Create a container from the `ubuntu:latest` image

```bash
docker run -it --name hw_container ubuntu:latest /bin/bash
```

🔍 **Explanation:**

- `docker run`: Run a new container.
- `-it`: Interactive + terminal (lets you type commands inside the container).
- `--name hw_container`: Give your container a name (`hw_container`).
- `ubuntu:latest`: The image you're using (Ubuntu latest version).
- `/bin/bash`: Opens a shell inside the container.

🧠 **Think of it like opening a blank Ubuntu terminal in a temporary computer.**

2️⃣ Create a file inside the container

```bash
touch HelloWorld.txt
```

🔍 Explanation:

- `touch HelloWorld.txt`: Creates an empty file named `HelloWorld.txt`.

3️⃣ Exit the container

```bash
exit
```

🔍 Explanation:

- You're leaving the container terminal, but the container still exists with the changes saved inside it (like hibernating a laptop).

4️⃣ Save the container as a new image

```bash
docker container commit hw_container hw_image
```

🔍 Explanation:

- `docker container commit`: Takes a snapshot of the container and saves it as an image.
- `hw_container`: The container to commit.
- `hw_image`: The name of the new image.

🧠 **This is like saving your customized Ubuntu system as a new ISO image**.

5️⃣ Check that your image was created

```bash
docker images
```

🔍 **Explanation:**

- Lists all images on your system.

Expected Output:

```bash
REPOSITORY   TAG      IMAGE ID     CREATED        SIZE
hw_image     latest   <some-id>    <time>         <size>
ubuntu       latest   <some-id>    <time>         <size>
```

6️⃣ Remove the old container

```bash
docker container rm -vf hw_container
```

🔍 **Explanation:**

- `docker container rm`: Remove a container.
- `-v`: Also remove volumes (not necessary here but harmless).
- `-f`: Force remove (helpful if it's still running or paused).

7️⃣ Run a new container from your new image

```bash
docker run -it --name new_hw_container hw_image /bin/bash
```

🔍 **Explanation:**

- Same as before, but now you're using your new image (`hw_image`) instead of the base Ubuntu.
- It will open a new bash shell inside a container based on the modified image.

8️⃣ Check if the file is still there

```bash
ls
```

🔍 **Expected Output:**

```bash
HelloWorld.txt  bin  boot  dev  etc  ...
```

🎉 **Congrats!** You can see your file (`HelloWorld.txt`) inside the container made from your new image.

🧹 Cleanup: Delete the container and image

```bash
docker rm new_hw_container
docker rmi hw_image
```

🔍 **Explanation:**

- `docker rm`: Remove the container (`new_hw_container`).
- `docker rmi`: Remove the image (`hw_image`).

🎨 Bonus: Visual Summary of Union File System (UFS)

When working with containers:

```less
[Base Layer] (ubuntu:latest)
      ↓
[Layer 1: HelloWorld.txt added]
      ↓
[Container View (Union of all layers)]
```

When you commit:

```csharp
[New Image] = Base Layer + Layer 1 (Your Changes)
```

🔧 Correcting or Clarifying Anything?

Yes — overall, your steps are correct! Just a few small things to remember:

- `HelloWorld` should be `HelloWorld.txt` to be clear.
- `-v` in `docker rm -vf` is often unnecessary unless you used volumes.
- `docker container commit` and `docker commit` are the same (alias).

📦 Real Life Example

Let's say you need a custom Ubuntu image with Python and Git preinstalled:

```bash
docker run -it --name custom_py ubuntu:latest /bin/bash
apt update
apt install -y python3 git
exit
docker commit custom_py ubuntu_py_git
```

Now, **`ubuntu_py_git`** is your own image with Python and Git built-in. You can share or reuse it anytime.

If you'd like, I can turn this into a printable **cheat sheet** or make a visual flowchart for you. Just say the word!

explain these in a very easiest manner in details with basics also correct if any error occur; what does these codes do with basics of each part and portion with basic example if necessary:

Create and Commit an Ubuntu Container with Git Installed

Create and Commit an Ubuntu Container with Git Installed
This session will guide us through creating an Ubuntu container, installing Git, and committing the changes to a new image. Additionally, we'll learn how to set an entrypoint to make using the image more efficient.

Task
We will perform the following steps:

Create an ubuntu container and open a bash session.
Install git inside the container and verify the git.
Create new image by using commit.
Set the entrypoint for the new image to make it easier to use.
Simple Explanation of the Process
In this lab, we will start by creating a container from the Ubuntu image and open a bash session within it. Inside this container, we will install git and verify that the installation was successful by checking the git version. After exiting the container, we will commit these changes to create a new Docker image that includes git. We will run a container from that image.

alt text

Finally, we will set an entrypoint for this new image to make it easier to use git directly without needing to specify the git command each time we start a container from this image.

alt text

Steps
Create an Ubuntu container and open a bash session:

docker run -it --name image-dev ubuntu:latest /bin/bash
This command creates a new container named image-dev from the ubuntu:latest image and opens an interactive bash session.

Install Git inside the container:

```
apt-get update
apt-get install -y git
```

This updates the package list and installs Git in the container.

Verify the Git installation by checking its version:

```
git --version
```

Expected output:

```
root@d15b204bcec5:/# git --version
git version 2.43.0
```

This command confirms that Git was installed correctly.

Exit the container:

```
exit
```

This command exits the interactive bash session and returns to the host terminal.

Review the filesystem changes and commit these changes to create a new image:

```
docker container commit -a "@poridhi" -m "Added git" image-dev ubuntu-git
```

This command commits the changes made in the image-dev container to a new image named ubuntu-git with an author tag and a commit message.

Remove the modified container:

```
docker container rm -vf image-dev
```

This command forcefully removes the image-dev container to clean up.

Verify the new image by checking the Git version in a new container:

```
docker container run --rm ubuntu-git git --version
```

This command runs a temporary container from the ubuntu-git image to verify that Git is installed correctly.

Setting the Entrypoint to Git

Create a new container with the entrypoint set to Git:

```
docker container run --name cmd-git --entrypoint git ubuntu-git
```

This command creates a new container named cmd-git with the entrypoint set to git, showing the standard Git help and exiting.

Commit the new image with the entrypoint:

```
docker container commit -m "Set CMD git" -a "@poridhi" cmd-git ubuntu-git
```

This command commits the changes to the ubuntu-git image, setting the entrypoint to Git.

Remove the modified container:

docker container rm -vf cmd-git
This command forcefully removes the cmd-git container to clean up.

Test the new image:

docker container run --name cmd-git ubuntu-git version
This command runs a new container from the ubuntu-git image, verifying that the entrypoint is set correctly and showing the Git version:

git version 2.43.0
This setup ensures that any container started from the ubuntu-git image will automatically use Git as the entrypoint, making it easier for users to work with Git directly.

Awesome! Let's break everything down step-by-step in the **simplest and most beginner-friendly way possible** — like you're learning Docker for the first time 😊.

🎯 Goal:

You're going to:

1. Start a container using Ubuntu.
2. Install Git inside that container.
3. Save that container as a new image (so you don't have to install Git every time).
4. Make it even easier to use by setting Git as the default command (`entrypoint`).

🧱 Step-by-step Guide with Very Simple Explanations

✅ Step 1: Create a container and open bash

```bash
docker run -it --name image-dev ubuntu:latest /bin/bash
```

🔍 What this does:

- `docker run`: Starts a new container.
- `-it`: Lets you interact with the container's terminal.
- `--name image-dev`: Gives the container a name (`image-dev`).
- `ubuntu:latest`: Uses the official Ubuntu image.
- `/bin/bash`: Opens a bash shell inside the container.

🧠 **Think of it like opening a brand-new virtual Ubuntu computer and going inside its terminal.**

✅ Step 2: Install Git inside the container

```bash
apt-get update
apt-get install -y git
```

🔍 Explanation:

- `apt-get update`: Updates the list of available software packages.
- `apt-get install -y git`: Installs Git automatically without asking for confirmation (`-y` = "yes to all").

📌 You're installing Git just like you would on a regular Ubuntu machine.

✅ Step 3: Check if Git is installed

```bash
git --version
```

🔍 What this does:

- Checks the version of Git installed.
- Output might be:

  ```nginx
  git version 2.43.0
  ```

✅ This confirms Git is working!

✅ Step 4: Exit the container

```bash
exit
```

📌 You leave the container, but all the changes (Git installed) are still inside it.

✅ Step 5: Save (commit) the modified container as a new image

```bash
docker container commit -a "@poridhi" -m "Added git" image-dev ubuntu-git
```

🔍 What this does:

- `docker container commit`: Saves your container as a new Docker image.
- `-a "@poridhi"`: Sets the author name.

- `-m "Added git"`: Adds a commit message.
- `image-dev`: The container you modified.
- `ubuntu-git`: The name of the new image.

🧠 **You just saved a custom Ubuntu with Git into your own image!**

✅ Step 6: Remove the old container

```bash
docker container rm -vf image-dev
```

🔍 Explanation:

- `rm`: Remove container.
- `-v`: Remove volumes (if any).
- `-f`: Force delete, even if it's running or paused.

🧹 This cleans up your old container (safe to delete now since we saved the changes in an image).

✅ Step 7: Test your new image by checking Git

```bash
docker container run --rm ubuntu-git git --version
```

🔍 Explanation:

- `run`: Run a new container.
- `--rm`: Automatically remove this temporary container after it finishes.
- `ubuntu-git`: Your custom image.
- `git --version`: Run `git --version` directly.

✅ This confirms your image has Git built-in and works fine.

🧠 Bonus: Set Git as the Default Command (Entrypoint)

This makes it **super easy** to use Git without typing `git` every time.

✅ Step 8: Run a container with Git as the entrypoint

```bash
docker container run --name cmd-git --entrypoint git ubuntu-git
```

🔍 What this does:

- `--entrypoint git`: Sets the default command inside the container to `git`.
- You'll see Git help output (like a guide of all git commands).

✅ Step 9: Save this entrypoint setting into a new image

```bash
docker container commit -m "Set CMD git" -a "@poridhi" cmd-git ubuntu-git
```

🔍 Explanation:

- Saves this container with `git` as the default command.
- Now every time you run a container from `ubuntu-git`, it behaves like running the `git` command automatically.

✅ Step 10: Delete the container

```bash
docker container rm -vf cmd-git
```

📌 Clean up the container — your updated image already saved the entrypoint setting.

✅ Step 11: Final Test – Run your new image

```bash
docker container run --name cmd-git ubuntu-git version
```

🔍 What this does:

- `ubuntu-git`: Uses your custom image.
- `version`: Runs as if you typed: `git version` (because entrypoint is set to `git`).

✅ Output:

```nginx
git version 2.43.0
```

🎉 You've successfully created a custom image with Git and made it even easier to use!

📌 Summary in Human Words

Step	Action	Why
1	Create container from Ubuntu	Get a clean Ubuntu to work on

Step	Action	Why
2	Install Git	Modify the system
3	Check Git	Make sure it’s there
4	Exit	Done with the setup
5	Commit container to image	Save your customized version
6	Remove container	Clean up unused stuff
7	Test new image	Check if Git works
8–11	Set Entrypoint to Git	So you don’t need to type `git` every time

---

Would you like a **one-page printable cheat sheet**, **visual flowchart**, or a **short video-style explanation** for this too? Just let me know!

explain these word by word and don't miss any important part in a very easiest manner in details with basics also correct if any error occur; what does these codes do with basics of each part and portion with basic example if necessary:

Reviewing Filesystem Changes

Reviewing filesystem changes
In this lab, we will learn how to review filesystem changes made inside a Docker container. Docker provides a command to list all changes to the filesystem, including added, changed, or deleted files and directories.

When you read a file from a union filesystem, that file will be read from the topmost layer where it exists. If a file was not created or changed on the top layer, the read will fall through the layers until it reaches a layer where that file does exist. Here is a simple example:

alt text

The layer functionality is hidden by the UFS. No special actions are required by the software running in a container to utilize these features. The UFS manages the complexity of handling files across multiple layers.

Most union filesystems use something called copy-on-write, which is easier to under stand if you think of it as copy-on-change. When a file in a read-only layer (not the top layer) is modified, the whole file is first copied from the read-only layer into the writable layer before the change is made.

alt text

In this illustration, files are added, changed, deleted, and added again over a range of three layers.

Task
Create a container and add a new file and review changes
Create another container and delete an existing file and review changes
Create yet another container and change an existing file and review changes
Clean up our workspace

Steps
Follow these steps to complete the task:

Create a container and add a new file:

alt text

Command:

docker container run --name tweak-a busybox:latest touch /HelloWorld
This command runs a new container named tweak-a from the busybox:latest image and creates an empty file named /HelloWorld inside the container.

Review the filesystem changes:

docker container diff tweak-a
This command lists all the changes made to the filesystem of the tweak-a container. You should see:

Expected Output:

A /HelloWorld
This indicates that the file /HelloWorld was added.

Create another container and delete an existing file:

alt text

Command:

docker container run --name tweak-d busybox:latest rm /bin/vi
This command runs a new container named tweak-d from the busybox:latest image and deletes the file /bin/vi.

Review the filesystem changes:

docker container diff tweak-d
This command lists all the changes made to the filesystem of the tweak-d container. You should see:

Expected Output:

C /bin
D /bin/vi
This indicates that the directory /bin was changed and the file /bin/vi was deleted.

Create yet another container and change an existing file:

alt text

Command:

docker container run --name tweak-c busybox:latest touch /bin/vi

This command runs a new container named tweak-c from the busybox:latest image and updates the modification time of the file /bin/vi.

Review the filesystem changes:

docker container diff tweak-c
This command lists all the changes made to the filesystem of the tweak-c container. You should see:

Expected Output:

C /bin
C /bin/busybox
This indicates that the directories /bin and /bin/busybox were changed.

Clean up our workspace by removing the containers:

docker container rm -vf tweak-a
docker container rm -vf tweak-d
docker container rm -vf tweak-c
These commands forcefully remove the tweak-a, tweak-d, and tweak-c containers, cleaning up our workspace.

Explanation of docker container diff Output
Lines that start with an A indicate files that were added.
Lines that start with a C indicate files that were changed.
Lines that start with a D indicate files that were deleted.
By following these steps, you'll be able to track and understand filesystem changes within Docker containers. This is particularly useful for debugging and for ensuring that your containerized applications behave as expected.

Commands Summary
Create a container and add a new file:

docker container run --name tweak-a busybox:latest touch /HelloWorld
Review the filesystem changes:

docker container diff tweak-a
Create another container and delete an existing file:

docker container run --name tweak-d busybox:latest rm /bin/vi
Review the filesystem changes:

docker container diff tweak-d
Create yet another container and change an existing file:

docker container run --name tweak-c busybox:latest touch /bin/vi
Review the filesystem changes:

docker container diff tweak-c
Clean up your workspace:

```
docker container rm -vf tweak-a
docker container rm -vf tweak-d
docker container rm -vf tweak-c
```

Absolutely! Let's go **word by word**, **step by step**, in a **very easy and detailed way** — explaining every part, correcting small mistakes, and giving **basic examples** where needed.

🎯 Topic: Reviewing Filesystem Changes in Docker

In Docker, when you **run a container** and do things like **create, change, or delete files**, Docker tracks these changes. You can use the command **`docker container diff`** to **see exactly what changed** inside the container.

📦 Basics First: What is Union Filesystem (UFS)?

Docker containers use a special layered system called **Union File System** (UFS). Here's how it works:

- **Base Layer**: Original image like **`busybox`** or **`ubuntu`** — **read-only**.
- **Writable Layer**: When you run a container, a new layer is added on top — this is where your changes go.
- You can read and write in this top layer.
- Files from the base image can't be changed directly — Docker **copies them to the top layer** first, then applies changes. This is called **copy-on-write** (or **copy-on-change**).

🧪 LAB: Let's Do the Steps One by One

🧾 Step 1: Create a container and add a new file

```bash
docker container run --name tweak-a busybox:latest touch /HelloWorld
```

🔍 Breakdown:

- **`docker container run`**: Create and start a new container.
- **`--name tweak-a`**: Give the container a name: **`tweak-a`**.
- **`busybox:latest`**: Use the **`busybox`** image (a very small Linux system).
- **`touch /HelloWorld`**: Inside the container, create an empty file **`/HelloWorld`**.

📌 This means: **You just started a container and created a file inside it.**

🔍 Step 1.1: Check what changed

```bash
docker container diff tweak-a
```

This shows:

```css
A /HelloWorld
```

🔤 **What this means:**

- `A` = **Added**
- `/HelloWorld` = This file was **added**.

✅ **Docker saw that you created a new file** in the container.

🧾 Step 2: Create a container and delete a file

```bash
docker container run --name tweak-d busybox:latest rm /bin/vi
```

🔍 **Breakdown:**

- `--name tweak-d`: Name the container `tweak-d`.
- `rm /bin/vi`: Deletes the `vi` program (a text editor).

📌 So here, you're starting a new container and **deleting an existing file**.

🔍 Step 2.1: Check what changed

```bash
docker container diff tweak-d
```

Expected Output:

```bash
C /bin
D /bin/vi
```

🔤 **What this means:**

- `C` = **Changed** → The `/bin` directory was changed (because a file was deleted inside it).
- `D` = **Deleted** → `/bin/vi` was removed.

✅ **Docker detected you deleted a file** and changed the directory it was in.

🧾 Step 3: Create a container and change a file

```bash
docker container run --name tweak-c busybox:latest touch /bin/vi
```

🔍 Breakdown:

- `--name tweak-c`: Name the container `tweak-c`.
- `touch /bin/vi`: This **updates the modification time** of the `/bin/vi` file, even though it doesn't change the content.

📌 So, the file wasn't deleted or added, but it was **modified**.

🔍 Step 3.1: Check what changed

```bash
docker container diff tweak-c
```

Expected Output:

```bash
C /bin
C /bin/busybox
```

🔤 What this means:

- `/bin` and `/bin/busybox` were **changed**. Likely because `vi` is a link or alias pointing to `busybox`, so modifying `vi` affected `busybox`.

✅ **Docker detected a change, even if it's small**, like touching a file.

🧹 Final Step: Clean Up

```bash
docker container rm -vf tweak-a
docker container rm -vf tweak-d
docker container rm -vf tweak-c
```

🔍 Breakdown:

- `docker container rm`: Remove (delete) a container.
- `-v`: Remove any volumes attached (not necessary here, but okay).
- `-f`: Force removal (even if the container is running or paused).

📌 This just cleans everything up from your system.

🧠 Bonus: Understanding `docker container diff` Output

Here's how to read the output:

Code	Meaning	Example Output	Meaning
`A`	Added	`A /HelloWorld`	A file or directory was added
`C`	Changed	`C /bin`	A file/directory was modified
`D`	Deleted	`D /bin/vi`	A file was deleted from the system

📘 Full Commands Summary with Explanation

Purpose	Command	What It Does
Add a file	`docker container run --name tweak-a busybox:latest touch /HelloWorld`	Creates a file inside a container
View changes (added)	`docker container diff tweak-a`	Shows added files
Delete a file	`docker container run --name tweak-d busybox:latest rm /bin/vi`	Deletes a file
View changes (deleted)	`docker container diff tweak-d`	Shows deleted files
Modify a file	`docker container run --name tweak-c busybox:latest touch /bin/vi`	Touches a file (changes it)
View changes (changed)	`docker container diff tweak-c`	Shows changed files
Clean up all containers	`docker container rm -vf tweak-a tweak-d tweak-c`	Removes all containers

🧪 Example Analogy (for fun)

Let's say:

- You take a photo (this is your base image).
- You draw a mustache on it (this is your added file).
- You erase part of the photo (this is a deleted file).
- You color the photo a bit (this is a changed file).

Docker lets you **track all these changes** step-by-step and **see exactly what happened** using `docker container diff`.

Would you like me to turn this into a **printable cheat sheet**, **infographic**, or **flashcards**? I'm happy to help you study or practice this any way you like!

explain these word by word and don't miss any important part in a very easiest manner in details with basics also correct if any error occur; what does these codes do with basics of each part and portion with basic example if necessary:

Modifying Docker Image Attributes

Modifying Docker Image Attributes
This lab aims to deepen your understanding of Docker image attributes and how they can be modified and inherited across different layers of an image.

By the end of this exercise, you will be familiar with setting environment variables, working directories, exposed ports, volume definitions, container entrypoints, and commands.

Description
When you use docker container commit, you create a new layer for an image. This layer includes not only a snapshot of the filesystem but also metadata about the execution context. The following parameters, if set for a container, will be carried forward to the new image:

Environment variables
Working directory
Exposed ports
Volume definitions
Container entrypoint
Command and arguments
If these parameters are not explicitly set, they will be inherited from the original image.

This lab will provide you with hands-on experience in modifying these attributes and observing how they are inherited across image layers.

Task
alt text

Create a Container with Environment Variables

Run a Docker container from the busybox:latest image, setting two environment variables.
Commit the running container to a new image.
Modify the Entrypoint and Command

Run a new container from the previously committed image, setting a new entrypoint and command.
Commit this container to update the image.
Verify Inheritance of Attributes

Run a container from the final image without specifying any command or entrypoint to verify that the environment variables and the entrypoint/command are inherited correctly.
Solution Steps
Create a Container with Environment Variables

Run the container:

docker run --name container1 -e ENV_EXAMPLE1=value1 -e ENV_EXAMPLE2=value2 busybox:latest

This command creates a new container named container1 from the busybox:latest image and sets two environment variables, ENV_EXAMPLE1 and ENV_EXAMPLE2.

Expected output:

term@ubuntu-x1brs3-5444b5f5fc-j5xpq:~$ docker run --name container1 -e ENV_EXAMPLE1=value1 -e ENV_EXAMPLE2=value2 busybox:latest

Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
ec562eabd705: Pull complete
Digest: sha256:9ae97d36d26566ff84e8893c64a6dc4fe8ca6d1144bf5b87b2b85a32def253c7
Status: Downloaded newer image for busybox:latest
Commit the container to a new image:

docker commit container1 new-image
This command commits the container1 container to a new image named new-image.

Varify the image creation using the following command:

docker images
Expected output:

term@ubuntu-x1brs3-5444b5f5fc-j5xpq:~$ docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
new-image	latest	7865e738cc85	11 seconds ago	4.26MB
busybox	latest	65ad0d468eb1	12 months ago	4.26MB

Modify the Entrypoint and Command

Run a new container with a specific entrypoint and command:

docker run --name container2 --entrypoint "/bin/sh" new-image -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"
This command runs a new container named container2 from the new-image image, setting the entrypoint to /bin/sh and the command to -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2". This setup will print the values of the environment variables.

Expected output:

term@ubuntu-x1brs3-5444b5f5fc-j5xpq:~$ docker run --name container2 --entrypoint "/bin/sh" new-image -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"

value1 value2
Commit this container to update the image:

docker commit container2 new-image
This will commit the container container2 to the new-image image, updating the image with the new entrypoint and command. The updated image will have the entrypoint and the command with it.

Verify the new image's entrypoint and command settings:

docker inspect --format '{{ .Config.Entrypoint }}' new-image
docker inspect --format '{{ .Config.Cmd }}' new-image
Expected output:

term@ubuntu-x1brs3-5444b5f5fc-j5xpq:~$ docker inspect --format '{{ .Config.Entrypoint }}' new-image

[/bin/sh]

term@ubuntu-x1brs3-5444b5f5fc-j5xpq:~$ docker inspect --format '{{ .Config.Cmd }}' new-image

[-c echo $ENV_EXAMPLE1 $ENV_EXAMPLE2]
Verify Inheritance of Attributes

Run a container from the final image to verify the inherited behavior:

docker run --rm new-image
This command runs a container from the final new-image image, verifying that the environment variables and the entrypoint/command are inherited correctly.

Expected output:

term@ubuntu-x1brs3-5444b5f5fc-j5xpq:~$ docker run --rm new-image
value1 value2
By completing this lab, we hope, you have a practical understanding of how to modify and verify Docker image attributes, and how these changes are inherited across image layers.

Absolutely! Let's go through everything **word by word, step by step**, and explain it in the **simplest way possible**, including what each part of the code does, what it's for, and correcting or clarifying anything along the way.

🧱 What You're Learning

You're learning how to:

- Set **environment variables** in a container.
- **Commit** that container to a new image.
- Set or change an image's **entrypoint** and **command**.
- **Verify** that these settings are inherited when using the new image.

These things define **how the container behaves** when you run it.

📌 Some Basic Terms First

Term	Simple Meaning
Image	A blueprint (template) for containers
Container	A running version of an image
Commit	Save a container's state to a new image
Entrypoint	The default program that runs in a container
Command (CMD)	Arguments passed to the entrypoint
Environment Variables	Settings stored inside the container, like `` `$ENV_EXAMPLE1=value1` ``

🔧 Step-by-Step with Easy Explanation

✅ Step 1: Create a container with environment variables

```bash
docker run --name container1 -e ENV_EXAMPLE1=value1 -e ENV_EXAMPLE2=value2 busybox:latest
```

🔍 Word by Word Breakdown:

- `` `docker run` `` → Run a new container.
- `` `--name container1` `` → Name it `` `container1` `` for easy reference.
- `` `-e ENV_EXAMPLE1=value1` `` → Set environment variable `` `ENV_EXAMPLE1` `` to `` `value1` ``.
- `` `-e ENV_EXAMPLE2=value2` `` → Set `` `ENV_EXAMPLE2` `` to `` `value2` ``.
- `` `busybox:latest` `` → Use the lightweight Linux image `` `busybox` ``.

📌 This just **starts a container** and sets 2 environment variables inside it.

📝 This container will **exit immediately** since you didn't give it anything to do — that's normal.

✅ Step 2: Commit this container as a new image

```bash
docker commit container1 new-image
```

🔍 What this does:

- `` `docker commit` `` → Take a snapshot of the container and save it as an image.
- `` `container1` `` → The container to save.
- `` `new-image` `` → The name of your new image.

✅ Now you have a new image called `` `new-image` `` that includes the environment variables.

✅ Step 3: Check if the image was created

```bash
docker images
```

🔍 This shows all images on your system. You should see:

```arduino
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
new-image    latest    abc123...       few secs ago    ~4MB
```

✅ Step 4: Run a new container and set Entrypoint and Command

```bash
docker run --name container2 --entrypoint "/bin/sh" new-image -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2
```

🔍 Breakdown:

- `` `--name container2` `` → Name this new container `` `container2` ``.
- `` `--entrypoint "/bin/sh"` `` → Set the program to run as `` `sh` `` (shell).
- `` `new-image` `` → Use your custom image with env variables.
- `` `-c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"` `` → Run the shell command to print the variables.

🧠 The `` `\$` `` is used to **escape the `` `$` `` sign**, so your shell doesn't interpret it too early.

✅ You should see:

```nginx
value1 value2
```

✅ Step 5: Commit this new container to update the image

```bash
docker commit container2 new-image
```

🔍 **This saves the entrypoint and command** from container2 into your image called `` `new-image` ``.

So now `` `new-image` `` will:

- Have environment variables.
- Automatically run `` `/bin/sh -c "echo $ENV_EXAMPLE1 $ENV_EXAMPLE2"` `` when started.

✅ Step 6: Verify Entrypoint and Command settings

🔍 To see the Entrypoint:

```bash
docker inspect --format '{{ .Config.Entrypoint }}' new-image
```

Expected output:

```bash
[/bin/sh]
```

🔍 To see the Command (CMD):

```bash
docker inspect --format '{{ .Config.Cmd }}' new-image
```

Expected output:

```bash
[-c echo $ENV_EXAMPLE1 $ENV_EXAMPLE2]
```

🧠 This tells you what your image will automatically run when you launch a container from it.

✅ Step 7: Run the image and test inheritance

```bash
docker run --rm new-image
```

🔍 What this does:

- **`--rm`** → Automatically deletes the container after it runs.
- **`new-image`** → Run your custom image.

✅ Since the image has both **entrypoint** and **command saved**, it runs:

```bash
/bin/sh -c "echo $ENV_EXAMPLE1 $ENV_EXAMPLE2"
```

Output:

```nginx
value1 value2
```

🧠 Concept Recap: What Got Inherited?

When you committed the containers:

- **Environment Variables** got saved ✅
- **Entrypoint** (`/bin/sh`) got saved ✅
- **Command** (`-c echo...`) got saved ✅

These now run automatically when using your `new-image`.

🧹 Optional: Cleanup Commands

If you want to clean up afterwards:

```
bash

docker container rm container1 container2
docker rmi new-image
```

🔍 Summary Table

Task	Command
Create container with env vars	`docker run --name container1 -e ENV_EXAMPLE1=value1 -e ENV_EXAMPLE2=value2 busybox:latest`
Commit container to image	`docker commit container1 new-image`
Run with entrypoint + command	`docker run --name container2 --entrypoint "/bin/sh" new-image -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"`
Commit again with entrypoint+cmd	`docker commit container2 new-image`
Inspect entrypoint	`docker inspect --format '{{ .Config.Entrypoint }}' new-image`
Inspect command	`docker inspect --format '{{ .Config.Cmd }}' new-image`
Run image and test inherited behavior	`docker run --rm new-image`

Would you like this in the form of:

- A **flowchart**?
- A **cheat sheet**?
- An **interactive exercise idea**?

Just say the word!

explain these word by word and don't miss any important part in a very easiest manner in details with basics also correct if any error occur; what does these codes do with basics of each part and portion with basic example if necessary:

Exploring Docker Image Layers and Size Management

Exploring Docker Image Layers and Size Management
Docker images are built from layers, where each layer represents a set of filesystem changes. The size of an image on disk is the sum of the sizes of its component layers. Docker allows you to commit changes to a running container, creating new image layers.

This lab will guide you through these concepts with hands-on practices, focusing on creating and modifying a Docker image with ubuntu as the base image. You will install and remove software within containers, observe the changes in image sizes, and understand the impact of Docker's Union File System (UFS) on image size.

alt text

Task: Building and Modifying Docker Images
Here, you will create a Docker image from the official Ubuntu image, install Git within a container, and commit the changes to create a new image. You will then modify this image by removing Git and observe how Docker manages image layers and size.

Steps:
Pull the Ubuntu Image:

Pull the ubuntu image from Docker Hub.

docker pull ubuntu
This command fetches the latest ubuntu image from Docker Hub and stores it in your local Docker repository.

Create a Container and Install Git:

Create a container from the ubuntu image and install Git.

docker run -d --name ubuntu-git-container ubuntu sleep infinity
docker exec -it ubuntu-git-container apt-get update
docker exec -it ubuntu-git-container apt-get install -y git
These commands run a container named ubuntu-git-container from the ubuntu image and install Git inside the container. The sleep infinity command keeps the container running. The apt-get update and apt-get install -y git commands update the package list and install Git, respectively.

Commit the Changes to Create a New Image:

Commit the container to create a new image with Git installed.

docker commit ubuntu-git-container ubuntu-git:1.0
docker tag ubuntu-git:1.0 ubuntu-git:latest
These commands commit the current state of the ubuntu-git-container container to a new image named ubuntu-git with a tag 1.0, and then tag this image as latest.

Check Image Sizes:

Check the sizes of all the images created.

docker images
Expected output:

term@ubuntu-71gmcp-db867df5c-4ddm9:~$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu-git 1.0 2c0f3da09652 6 seconds ago 198MB
ubuntu-git latest 2c0f3da09652 6 seconds ago 198MB
ubuntu latest 17c0145030df 7 days ago 76.2MB
Remove Git:

Create a new container from the ubuntu-git image and remove Git.

docker run --name ubuntu-git-remove --entrypoint /bin/bash ubuntu-git:latest -c "apt-get remove -y git"
This command runs a container named ubuntu-git-remove from the ubuntu-git:latest image with an entrypoint set to /bin/bash, and removes Git from the container.

Commit the Changes to Create a New Image with Git Removed:

Commit the container to create a new image with Git removed.

docker commit ubuntu-git-remove ubuntu-git:2.0
docker tag ubuntu-git:2.0 ubuntu-git:latest
These commands commit the current state of the ubuntu-git-remove container to a new image named ubuntu-git:removed and reassign the latest tag to this new image.

Check Image Sizes:

Check the sizes of all the images created.

docker images
Expected output:

term@ubuntu-71gmcp-db867df5c-4ddm9:~$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu-git 2.0 6acdc1e0df8e 7 seconds ago 198MB
ubuntu-git latest 6acdc1e0df8e 7 seconds ago 198MB
ubuntu-git 1.0 2c0f3da09652 51 seconds ago 198MB
ubuntu latest 17c0145030df 7 days ago 76.2MB
Notice that even though you removed Git, the image actually same in size. Although you could examine the specific changes with docker diff, you should be quick to realize that the reason for the increase has to do with the union file system.

Remember, UFS will mark a file as deleted by actually adding a file to the top layer. The original file and any copies that existed in other layers will still be present in the image. When a file is deleted, a delete record is written to the top layer, which overshadows any versions of that file on lower layers.

It’s important to minimize image size for the sake of the people and systems that will be consuming your images. If you can avoid causing long download times and significant disk usage with smart image creation, then your consumers will benefit.

Absolutely! Let's now go **word by word**, **command by command**, and explain **every part** of this Docker image and size management lab in the **simplest way possible**, with **basic examples** and **important clarifications** wherever needed. 🧠✨

🧱 Understanding the Core Idea

Docker images are made of **layers**. Each time you:

- Add a file ✅
- Install something ✅
- Delete something ❌ (but not really deleting it physically)...

...a new **layer** is added on top.

> Think of a Docker image like a **stack of transparent sheets**. Each sheet (layer) has some changes. What you "see" is the sum of all the sheets layered on top of each other.

🎯 What You'll Learn in This Lab

- How Docker layers grow when you install/remove things.
- How committing containers creates new images.
- Why even after you **remove** something like Git, **the image size stays the same.**
- How UFS (Union File System) works behind the scenes.

🧪 TASKS (Step-by-Step Breakdown)

✅ Step 1: Pull the Ubuntu Image

```bash
docker pull ubuntu
```

🔍 Word-by-word:

- `docker pull` = Download an image.
- `ubuntu` = The name of the image (the latest Ubuntu Linux system).

📌 This **downloads the latest Ubuntu image** from Docker Hub (official online image library).

✅ Step 2: Create a Container & Install Git

```bash
docker run -d --name ubuntu-git-container ubuntu sleep infinity
```

🔍 **Word-by-word:**

- `docker run` = Start a new container.
- `-d` = Run in **detached mode** (in the background).
- `--name ubuntu-git-container` = Name this container for easy use.
- `ubuntu` = Use the Ubuntu image.
- `sleep infinity` = Keep the container running forever by doing nothing.

🧠 **Why use `sleep infinity`?** So the container **doesn't exit immediately**, and you can go inside it to install stuff.

```bash
docker exec -it ubuntu-git-container apt-get update
```

🔍 **What this does:**

- `docker exec -it` = Run a command **inside a running container**.
- `ubuntu-git-container` = The container name.
- `apt-get update` = Update the list of packages (required before installing anything).

```bash
docker exec -it ubuntu-git-container apt-get install -y git
```

🔍 **What this does:**

- `apt-get install -y git` = Installs Git.
- `-y` = Automatically say "yes" to prompts.

✅ You now have a **container with Git installed**.

✅ Step 3: Commit the Container to Create a New Image

```bash
docker commit ubuntu-git-container ubuntu-git:1.0
```

🔍 **Explanation:**

- `docker commit` = Save the current state of a container as a new image.
- `ubuntu-git-container` = The container with Git installed.

- `ubuntu-git:1.0` = New image name and version tag.

```bash
docker tag ubuntu-git:1.0 ubuntu-git:latest
```

🔍 What this does:

- `docker tag` = Assign another name/tag to the same image.
- Now you have two tags (`1.0` and `latest`) for the **same image**.

🧠 Think of it like giving the same photo two names.

✅ Step 4: Check Image Sizes

```bash
docker images
```

🔍 What this does:

- Lists all images on your system with their size.

🧾 Example output:

```nginx
REPOSITORY   TAG      IMAGE ID     CREATED        SIZE
ubuntu-git   1.0      abc123...    seconds ago    198MB
ubuntu       latest   xyz456...    7 days ago     76.2MB
```

✅ Your custom image (`ubuntu-git`) is now **198MB**, because Git was added.

✅ Step 5: Remove Git in a New Container

```bash
docker run --name ubuntu-git-remove --entrypoint /bin/bash ubuntu-git:latest -c "apt-get remove -y
```

🔍 Word-by-word:

- `docker run` = Start a new container.
- `--name ubuntu-git-remove` = Name this container.
- `--entrypoint /bin/bash` = Use Bash as the main program inside the container.
- `ubuntu-git:latest` = Use the image we created with Git.
- `-c "apt-get remove -y git"` = Run a Bash command to remove Git.

📌 This **removes Git** — but only in the top layer of the container.

✅ Step 6: Commit This Container (Without Git)

```bash
docker commit ubuntu-git-remove ubuntu-git:2.0
docker tag ubuntu-git:2.0 ubuntu-git:latest
```

🔍 What this does:

- Create a new image (`` `ubuntu-git:2.0` ``) without Git.
- Tag it as the new `` `latest` ``.

📌 But wait… even though Git is removed — **the image size will still be big**.

✅ Step 7: Check Image Sizes Again

```bash
docker images
```

🧾 Expected output:

```arduino
REPOSITORY   TAG      IMAGE ID     CREATED        SIZE
ubuntu-git   2.0      def789...    seconds ago    198MB
ubuntu-git   latest   def789...    seconds ago    198MB
ubuntu-git   1.0      abc123...    1 min ago      198MB
ubuntu       latest   xyz456...    7 days ago     76.2MB
```

🤔 Even after removing Git — the image size is **still 198MB**. Why?

🧠 Why the Size Didn't Shrink — UFS Explained

Docker uses a **Union File System**, which:

- Doesn't actually delete files.
- It just adds a **"delete marker"** in the top layer.
- Lower layers (with the original files) are still there.

📌 **When you remove Git**, Docker doesn't erase its files from older layers. It just **marks them as hidden** in the new layer.

> 🧃 Think of it like pouring juice in layers:
>
> - Git was added in one layer ✅
> - Then removed in the next ❌
> - But the liquid from the old layer still exists at the bottom — just not visible.

🧠 Summary: What You Learned

Step	Action	Key Point
1	Pull Ubuntu	Get a fresh base image
2	Install Git in a container	Create a customized container
3	Commit it	Save your Git image
4	Remove Git in a new container	Simulate modification
5	Commit again	Save "Git-removed" image
6	Check image sizes	Size didn't change due to UFS
✅	Main Concept	Removing files doesn't reduce image size!

💡 Pro Tip for Real Optimization

If you **really** want a smaller image:

- Use a **multi-stage Dockerfile**
- Use **Alpine Linux** (very small image)
- Don't install unnecessary packages
- Use `docker system prune` to clean up unused layers (careful!)

Would you like:

- A visual diagram of UFS layers?
- A Dockerfile-based version of this experiment?
- A cleanup script to remove old images and containers?

Let me know, happy to help you master Docker layer by layer! 🐳

explain these with example and real life implementation:
Use a multi-stage Dockerfile

Absolutely! Let's go step-by-step and **explain multi-stage Dockerfiles** in a super simple way, with examples and a real-life use case.

🧠 What Is a Multi-Stage Dockerfile?

A **multi-stage Dockerfile** is a way to **build smaller, cleaner, and more secure Docker images**.

🔥 Problem with normal Dockerfiles:

When you compile or build software inside a Dockerfile (like using `npm`, `javac`, or `gcc`), the final image includes:

- Compilers
- Source code
- Build tools
- Temporary files

⚠️ This makes the final image **BIG** and sometimes **unsafe**.

✅ Multi-Stage Dockerfile: The Solution

In a **multi-stage Dockerfile**, you:

1. Use one image for **building** (e.g., has tools and source code).
2. Use another image for **running only** (copy only the final result).

This way:

- You keep your final image **lightweight**
- You don't include compilers or unused files

📦 Example: Build and Run a Go App (Tiny Final Image)

Let's say you have a Go application with this file:

📁 `main.go`

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello from Go!")
}
```

🐳 Multi-Stage Dockerfile

```dockerfile

# -------- Stage 1: Build Stage --------
FROM golang:1.20 AS builder

WORKDIR /app

COPY main.go .

RUN go build -o myapp

# -------- Stage 2: Run Stage --------
FROM alpine:latest
```

```
WORKDIR /app

# Copy only the compiled binary from builder stage
COPY --from=builder /app/myapp .

# Run the app
ENTRYPOINT ["./myapp"]
```

🔍 Explanation Line by Line

🧱 Stage 1 – Build the binary

```dockerfile
FROM golang:1.20 AS builder
```

→ Start from the official Go image, call this stage "builder".

```dockerfile
WORKDIR /app
```

→ Set working directory inside the container.

```dockerfile
COPY main.go .
```

→ Copy your Go file into the container.

```dockerfile
RUN go build -o myapp
```

→ Build the Go app. Output is `myapp` binary.

🥽 Stage 2 – Run only the result

```dockerfile
FROM alpine:latest
```

→ Start a fresh, clean Alpine Linux image (only 5MB).

```dockerfile
WORKDIR /app
```

→ Set working directory again.

```dockerfile
COPY --from=builder /app/myapp .
```

→ Only copy the final binary from the builder stage. Nothing else.

```dockerfile
ENTRYPOINT ["./myapp"]
```

→ Run the binary when container starts.

✅ Build and Run

```bash
docker build -t go-multistage-app .
docker run --rm go-multistage-app
```

💡 Output:

```csharp
Hello from Go!
```

📊 Real-Life Benefits

Feature	Normal Dockerfile	Multi-Stage Dockerfile
Size	Large (includes tools, temp files)	Small (only final result)
Security	Risky (includes tools)	Safer (minimal surface)
Build Process	Mixed in one image	Clean separation
CI/CD Friendly	Less	✅ Yes

🛠️ Real-Life Use Cases

1. Node.js Frontend App (React or Angular)

- Stage 1: Use `node:18` to build (`npm install && npm run build`)
- Stage 2: Use `nginx:alpine` to serve `build/` folder

2. Java / Spring Boot App

- Stage 1: Use `openjdk` with Maven or Gradle to build `.jar`
- Stage 2: Use `openjdk:runtime` to run the JAR only

3. Python with C/C++ Dependencies

- Stage 1: Use full build tools like `gcc`, `pip`, `setuptools`
- Stage 2: Copy only the installed packages and source

🚀 Bonus: Multi-Stage + .dockerignore

Add a `.dockerignore` file to ignore:

```bash
*.log
node_modules
build
.git
```

This keeps build context clean and fast.

🧠 Summary

Concept	Description
Multi-stage build	Build in one stage, run in another
Why	Smaller, cleaner, safer images
Command	`COPY --from=<stage>` to bring only what you need
Best Use	Apps that need build steps like Go, Node.js, Java, C++

Would you like me to generate:

- A real multi-stage Dockerfile for a React, Python, or Spring Boot app?
- Or a template you can reuse?

Let me know your tech stack!

explain these word by word and don't miss any important part in a very easiest manner in details with basics also correct if any error occur; what does these codes do with basics of each part and portion with basic example if necessary:

Containerize a Single-Container App

Containerize a Single-Container App
In this guide, we will walk through the process of containerizing a simple Node.js application. Containerization involves packaging an application and all its dependencies, libraries, and configurations into a single package known as a container. This ensures the app runs consistently across different environments, whether it's your local machine, a server, or the cloud.

Docker is the tool we'll use for containerization. It allows developers to automate the deployment of applications inside lightweight, portable containers. We'll also explore how Docker layers work and their significance in the containerization process.

Creating the Application Code
First, let's create a basic Node.js application. We will begin by creating a directory for the app and navigating into it:

```
mkdir my-node-app
cd my-node-app
```

Next, create a file named app.js with the following content:

```
// app.js
const http = require('http');

const hostname = '0.0.0.0';
const port = 8080;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

This is a simple HTTP server that responds with "Hello, World!".

Creating the Dockerfile
A Dockerfile is a script that contains instructions for Docker to build your container image. Create a Dockerfile in the same directory with the following content:

```
# Use an official Node.js runtime as the base image
FROM node:14-alpine

# Set the working directory
WORKDIR /usr/src/app

# Copy the application code
COPY app.js .

# Install the necessary Node.js package
RUN npm install http@0.0.1-security

# Expose the port the app runs on
EXPOSE 8080

# Define the command to run the app
CMD ["node", "app.js"]
```

FROM: Specifies the base image (Node.js with Alpine Linux, a lightweight version of Linux).
WORKDIR: Sets the working directory inside the container.

COPY: Copies app.js from the local machine to the container.
RUN: Installs the necessary Node.js package.
EXPOSE: Informs Docker the container listens on port 8080.
CMD: Defines the command to run the app.
Containerizing the App / Building the Image
Now that the Dockerfile is ready, we can build the Docker image. In your terminal, run:

docker build -t my-node-app:1.0 .
This command tells Docker to build an image named my-node-app with the version 1.0 using the current directory.

To verify that the image was created, use the following command:

docker images

This will display a list of available images.

Pushing the Image (Optional)
If you'd like to share your image, you can push it to Docker Hub. Follow these steps:

Log in to Docker Hub:

docker login
Tag the image with your Docker ID:

docker tag my-node-app:1.0 <docker-id>/my-node-app:1.0
Push the image:

docker push <docker-id>/my-node-app:1.0
This will make the image available for others to pull and use.

Running the App
To run the containerized app, execute the following command:

docker run -d --name my-node-app-container -p 80:8080 my-node-app:1.0
-d: Runs the container in detached mode (in the background).
--name: Assigns a name to the running container.
-p 80:8080: Maps port 8080 inside the container to port 80 on your machine.
Check that the container is running by typing:

docker ps

This will list all active containers.

Testing the App
To test the app, you can use the curl command or open a browser and go to http://localhost. Using curl, run:

curl http://localhost:80

Docker Layers and Efficiency
Each instruction in the Dockerfile creates a layer. These layers are cached and reused during subsequent builds if the content doesn't change, which improves build speed and reduces storage space. For example:

FROM creates a base layer.
COPY adds a new layer with the application files.
RUN installs dependencies in a separate layer.
You can inspect the layers of your image by running:

```
docker history my-node-app:1.0
```

This command displays each command from the Dockerfile and the corresponding layer it created.

Optimizing Docker Images: Multi-Stage Builds
To reduce image size, you can use multi-stage builds, especially for production environments. In multi-stage builds, unnecessary files (such as development dependencies) are not included in the final image.

Here's an example of a multi-stage Dockerfile:

```
# Stage 1: Build
FROM node:14-alpine AS builder
WORKDIR /usr/src/app
COPY app.js .
RUN npm install http@0.0.1-security

# Stage 2: Production
FROM node:14-alpine
WORKDIR /usr/src/app
COPY --from=builder /usr/src/app .
EXPOSE 8080
CMD ["node", "app.js"]
```

In this setup, the first stage builds the app, and the second stage creates a minimal image without including the build tools, reducing the final image size.

Container Management and Best Practices
Restart Policies
You can ensure that containers automatically restart if they crash or your system reboots. Add the --restart flag when running your container:

```
docker run -d --name my-node-app-container --restart always -p 80:8080 my-node-app:1.0
```

Environment Variables
Environment variables allow you to pass dynamic configurations into your container at runtime. For example:

```
docker run -d -e NODE_ENV=production -p 80:8080 my-node-app:1.0
```

Security Considerations

Use Official Base Images: Always start with trusted base images to reduce security risks.
Scan for Vulnerabilities: Regularly scan your images for known vulnerabilities using tools like Trivy or Docker Scan.
Limit Privileges: Run containers with the least privileges needed by using Docker's security options.
Conclusion
Containerizing applications with Docker ensures consistency, portability, and ease of deployment across various environments. By following best practices like using official base images, optimizing Docker layers, and employing security measures, you can create efficient and secure containerized applications.

This guide provides a clear step-by-step process for containerizing a Node.js app while explaining Docker's functionality, Dockerfile instructions, and optimization strategies. Happy containerizing!

Awesome! Let's now go through the **entire process word by word**, step by step, and explain **each line of code and concept** in the **easiest way possible**, with **real-life comparisons** and **corrections where needed**. We'll focus on **containerizing a single-container Node.js app using Docker**.

🧱 What is Containerization?

Containerization means packaging your app and everything it needs (code, dependencies, libraries, OS-level tools) into a single unit called a **container**.

🧠 **Real-life example**: Think of it like a zip file that runs the same way **everywhere** — your laptop, cloud server, or someone else's machine.

🛠️ Step-by-Step Process to Containerize a Node.js App

🔹 1. Create the Project Folder

```bash
mkdir my-node-app
cd my-node-app
```

📝 **Explanation:**

- `mkdir` creates a new folder named `my-node-app`.
- `cd` moves you inside that folder.

📦 **Think of this folder as the workspace** for your Node.js project.

🔹 2. Create the App File

Create a file called `app.js` with this content:

```js
// app.js
const http = require('http');

const hostname = '0.0.0.0';
const port = 8080;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

🧠 **What it does**:

- It starts a simple web server.
- It listens on port **8080**.
- It responds to every request with: `Hello, World!`.

🔹 3. Create the Dockerfile

```Dockerfile
# Use an official Node.js runtime as the base image
FROM node:14-alpine

# Set the working directory
WORKDIR /usr/src/app

# Copy the application code
COPY app.js .

# Install the necessary Node.js package
RUN npm install http@0.0.1-security

# Expose the port the app runs on
EXPOSE 8080

# Define the command to run the app
CMD ["node", "app.js"]
```

🧠 Word-by-Word Breakdown of Dockerfile:

Line	Explanation
`FROM node:14-alpine`	Use a small official Node.js image. `alpine` means it's super lightweight.
`WORKDIR /usr/src/app`	Inside the container, all commands will run in this folder.
`COPY app.js .`	Copy your `app.js` from your computer into the container.
`RUN npm install http@0.0.1-security`	Install the basic `http` module. *(Note: This is optional! `http` is already built-in to Node.js.)*
`EXPOSE 8080`	Let Docker know that the app will run on port 8080.

Line	Explanation
`CMD ["node", "app.js"]`	When the container starts, it should run `node app.js`.

📝 **Note:** You don't need to install `http@0.0.1-security`. It's built-in to Node.js. So this `RUN` line can be removed.

🔹 4. Build the Docker Image

```bash
docker build -t my-node-app:1.0 .
```

🧠 **Word-by-Word**:

- `docker build` = build an image
- `-t my-node-app:1.0` = name and tag the image (like version 1.0)
- `.` = use current directory (where your Dockerfile is)

🔹 5. Check the Image

```bash
docker images
```

👀 Shows something like:

```perl
REPOSITORY     TAG       IMAGE ID      SIZE
my-node-app    1.0       abc123...     100MB
```

🔹 6. Run the Container

```bash
docker run -d --name my-node-app-container -p 80:8080 my-node-app:1.0
```

🧠 **Word-by-Word**:

- `docker run` = start a container
- `-d` = run in the background (detached mode)
- `--name my-node-app-container` = name your container
- `-p 80:8080` = map container port 8080 to your computer's port 80
- `my-node-app:1.0` = run the image we just built

📌 Now your app is accessible at `http://localhost`.

🔹 7. Check Running Containers

```bash
docker ps
```

Shows all running containers and their info.

🔹 8. Test the App

```bash
curl http://localhost
```

Or open a browser and go to http://localhost

✅ You should see:

```
Hello, World!
```

📦 Understanding Docker Layers

Each line in your Dockerfile creates a new **layer**. Example:

Layer Created By	Description
`FROM`	Base OS + Node.js
`COPY`	Your app.js
`RUN`	Dependencies
`CMD`	Startup instructions

Layers help with **caching and optimization** — unchanged layers don't rebuild, saving time.

You can view layers using:

```bash
docker history my-node-app:1.0
```

📉 Optimize Image Size with Multi-Stage Build (Optional)

Use this **only if you're doing complex builds**, but here's a basic example anyway:

```Dockerfile
# Stage 1: Build (if you had to compile or bundle things)
FROM node:14-alpine AS builder
WORKDIR /app
COPY app.js .
RUN npm install http@0.0.1-security
```

```
# Stage 2: Run
FROM node:14-alpine
WORKDIR /app
COPY --from=builder /app .
EXPOSE 8080
CMD ["node", "app.js"]
```

✅ This removes unnecessary tools from the final image = smaller, faster image.

🔐 Best Practices

Best Practice	Example
Restart policy	`--restart always` = auto-restarts if crash
Environment variables	`-e NODE_ENV=production`
Security scans	Use `docker scan` or `trivy`
Official base image	`node:14-alpine` is safe and trusted

🎉 Final Recap

Step	Command / File	Purpose
1	`app.js`	Simple Node.js server
2	`Dockerfile`	Build instructions
3	`docker build -t my-node-app:1.0 .`	Create image
4	`docker run -d -p 80:8080 ...`	Run container
5	`docker ps` / `curl`	Test app
6	`docker history`	View layers
7	`--restart` / `-e`	Production readiness

Would you like:

- A template for a **Python**, **Java**, or **React** version of this?
- A **GitHub Actions workflow** to auto-build and deploy it?
- Or a **diagram showing how this works behind the scenes**?

Just say the word!

also include and explain this:
Pushing the Image (Optional)
If you'd like to share your image, you can push it to Docker Hub. Follow these steps:

Log in to Docker Hub:

docker login
Tag the image with your Docker ID:

docker tag my-node-app:1.0 <docker-id>/my-node-app:1.0
Push the image:

docker push <docker-id>/my-node-app:1.0
This will make the image available for others to pull and use.

Great! Let's now explain the **image pushing process to Docker Hub** word by word, as clearly and simply as possible. We'll also include examples and real-life comparisons so it's very easy to follow.

🚀 Pushing Docker Images to Docker Hub

After building your Docker image locally, you might want to **share it** with:

- Your team
- The public
- A cloud provider (like Kubernetes, AWS, etc.)

For that, you **push the image to Docker Hub**, which is like the **GitHub for Docker images**.

🌐 What is Docker Hub?

Docker Hub is an online registry where you can **store and share images**.
🧠 Think of it as a cloud shelf where your container lives.

🔐 Step 1: Log in to Docker Hub

```bash
docker login
```

🔍 Explanation:

- This opens a login prompt in your terminal.
- Enter your **Docker Hub username** and **password**.
- It saves your credentials so you can push and pull images.

🧠 **Real-world analogy**: Like logging into your email before sending a message.

🏷️ Step 2: Tag the Image

```bash
docker tag my-node-app:1.0 <your-docker-id>/my-node-app:1.0
```

🔍 **Word-by-word:**

- `docker tag` = Create a new name (tag) for the existing image.
- `my-node-app:1.0` = This is your **local image**.
- `<your-docker-id>/my-node-app:1.0` = The **target name** that matches your Docker Hub ID.

📌 **Important:** The image name **must start with your Docker Hub ID**, or Docker Hub won't accept it.

🧠 **Example**: If your Docker ID is `john123`, the command would be:

```bash
docker tag my-node-app:1.0 john123/my-node-app:1.0
```

📤 Step 3: Push the Image

```bash
docker push <your-docker-id>/my-node-app:1.0
```

🔍 **What this does:**

- Sends the image to your Docker Hub account.
- Others can now **pull and use it** from anywhere.

🧠 **Example**:

```bash
docker push john123/my-node-app:1.0
```

✅ Final Output:

Your image is now available at:

```bash
https://hub.docker.com/r/<your-docker-id>/my-node-app
```

💡 Others can run your app using:

```bash
docker pull <your-docker-id>/my-node-app:1.0
docker run -p 80:8080 <your-docker-id>/my-node-app:1.0
```

💬 Summary of Commands

Purpose	Command
Login to Docker Hub	`docker login`
Tag local image for Docker Hub	`docker tag my-node-app:1.0 yourid/my-node-app:1.0`
Push the image online	`docker push yourid/my-node-app:1.0`

🧠 Why Tag Before Push?

Docker doesn't know where to push your image unless it's tagged correctly.

- `my-node-app:1.0` → Only local
- `john123/my-node-app:1.0` → Can be pushed to your Docker Hub repo

🎓 Real-Life Use Case

Let's say you made an app and want your friend to run it on their laptop. You tell them:

> "Just run `docker pull john123/my-node-app:1.0` and you're done!"

You don't need to:

- Send code
- Share dependencies
- Explain setup

They get the **same app**, running the **same way**, instantly.

Would you like:

- Help setting up your own Docker Hub account?
- A GitHub Actions CI pipeline that **builds and pushes** your image automatically?

Just ask — happy to help!

also include and explain this:
Container Management and Best Practices
Restart Policies
You can ensure that containers automatically restart if they crash or your system reboots. Add the --restart flag when running your container:

docker run -d --name my-node-app-container --restart always -p 80:8080 my-node-app:1.0
Environment Variables
Environment variables allow you to pass dynamic configurations into your container at runtime. For example:

docker run -d -e NODE_ENV=production -p 80:8080 my-node-app:1.0

Security Considerations
Use Official Base Images: Always start with trusted base images to reduce security risks.
Scan for Vulnerabilities: Regularly scan your images for known vulnerabilities using tools like Trivy or Docker Scan.
Limit Privileges: Run containers with the least privileges needed by using Docker's security options.

Awesome! Now let's complete the guide by explaining **Container Management and Best Practices** — step by step, word by word — in the **easiest and clearest way possible**, with real-life examples.

🛠️ Container Management and Best Practices

Managing your containers properly ensures they:

- Run reliably ✅
- Stay secure 🔐
- Are easy to configure and deploy ⚙️

Let's go through the main sections below.

🔁 1. Restart Policies

📌 Problem:

What happens if your container crashes? Or your server restarts?

By default: **Docker containers do not restart** automatically.

✅ Solution:

Use the `--restart` flag when running your container.

🧪 Example:

```bash
docker run -d --name my-node-app-container --restart always -p 80:8080 my-node-app:1.0
```

🔍 Word-by-word Explanation:

- `docker run`: Start a new container.
- `-d`: Detached mode (runs in the background).
- `--name my-node-app-container`: Give your container a name.
- `--restart always`: Restart the container if it crashes or if Docker restarts.
- `-p 80:8080`: Map port 8080 in container to 80 on your computer.
- `my-node-app:1.0`: Your custom image.

📘 Restart Policy Options:

Policy	Description
`no` (default)	Don't restart automatically
`on-failure`	Restart only if the container exits with an error
`always`	Restart always, even after system reboot
`unless-stopped`	Restart unless you manually stop it

🌎 2. Environment Variables

Environment variables let you **configure your app without changing the code**.

🧪 Example:

```bash
docker run -d -e NODE_ENV=production -p 80:8080 my-node-app:1.0
```

🔍 Word-by-word:

- `-e NODE_ENV=production`: Set environment variable `NODE_ENV` to `production`.

🧠 **Inside the container**, your Node.js app can use:

```js
process.env.NODE_ENV
```

🧠 Real-Life Use Case:

You can use different settings for:

- Development (`NODE_ENV=development`)
- Production (`NODE_ENV=production`)
- Testing (`NODE_ENV=test`)

Without changing your code — just change the variable when you run the container.

🔐 3. Security Best Practices

Keeping your Docker containers secure is **very important**. Here's how:

🔹 A. Use Official Base Images

```dockerfile
FROM node:14-alpine
```

✅ These are:

- Maintained by trusted organizations (like Node.js or Docker).
- Regularly updated for security.

🚫 Don't use random images like `someguy/node-custom123`, unless you trust the source.

🔹 B. Scan for Vulnerabilities

Use tools to check your images for known security issues.

🧪 Tools:

- **Trivy** (lightweight scanner)
- **Docker Scan** (built-in tool)

```bash
docker scan my-node-app:1.0
```

✅ This checks your image for:

- OS vulnerabilities
- Known package risks

🔹 C. Limit Privileges

By default, containers run with high privileges. You should reduce them.

🧪 Example:

```bash
docker run --user 1001:1001 my-node-app:1.0
```

🧠 This runs the container with a **non-root user** (user ID 1001), improving security.

📌 Also consider:

- `--read-only`: Prevent writing to the container filesystem.
- `--cap-drop all`: Remove all extra Linux capabilities.

✅ Final Recap: Best Practices

Area	Best Practice	Example
🔁 Restart	Auto-restart container	`--restart always`
⚙️ Configuration	Use env variables	`-e NODE_ENV=production`
🧱 Base Image	Use official images	`FROM node:alpine`
🧪 Scanning	Scan for issues	`docker scan image-name`
🔐 Privileges	Run as non-root	`--user 1001:1001`

🧠 Bonus: Combine All Best Practices

```bash
docker run -d \
  --name my-node-app-container \
  --restart always \
  -e NODE_ENV=production \
  --user 1001:1001 \
  -p 80:8080 \
  my-node-app:1.0
```

This runs your container:

- In production mode
- With a restart policy
- With limited user access
- On the correct port

Would you like:

- A **Docker security checklist**?
- **Templates** for different environments (dev, prod)?
- A GitHub Actions workflow that builds, scans, and pushes?

Just let me know! 🚀

what is Run as non-root --user 1001:1001

Great question! Let me explain `--user 1001:1001` in a **very simple and clear way**, with **real-life analogies** and **why it's important**.

🧠 What Does `--user 1001:1001` Mean?

When you run a Docker container, it **normally runs as root** (superuser inside the container).

🔥 Problem:

Running as root is **risky**, because if someone hacks your app, they can:

- Access system files
- Install malicious software
- Escape the container (in rare but possible cases)

So we **reduce the risk** by running the app as a **non-root user**.

✅ `--user` Flag: Run as a Specific User

```bash
docker run --user 1001:1001 my-node-app:1.0
```

🔍 What this means:

- `--user` = Run the container as a specific **user and group**
- `1001:1001` =
  - `1001` = UID (User ID)
  - `1001` = GID (Group ID)

So Docker starts the container **as user ID 1001**, instead of root (which is 0).

📘 Real-Life Analogy

Imagine:

- Root = Super Admin (can do anything — delete, install, format)
- Normal user (ID 1001) = Staff user (limited access)

You wouldn’t give every intern super admin access, right?
Same idea with containers!

🧪 How to Check Users in a Linux System (inside a container)

If you open a shell in the container:

```bash
docker run -it my-node-app:1.0 /bin/sh
```

You can run:

```bash
cat /etc/passwd
```

This shows all users. You might see:

```ruby
root:x:0:0:root:/root:/bin/sh
node:x:1001:1001::/home/node:/bin/sh
```

This means:

- `root` has UID `0`
- `node` has UID `1001`

So if your image already has a non-root user (`node`), you can run as:

```bash
docker run --user node my-node-app:1.0
```

Or by UID:

```bash
docker run --user 1001:1001 my-node-app:1.0
```

🔐 Why Is This a Good Security Practice?

Risk when using root	Safer with non-root
Full control over the container (can be abused)	Limited control inside container
Can break system if escaped	Reduced access = limited damage
Can modify sensitive files	Files are protected

🧠 Tip: Custom Users in Your Dockerfile

You can also create your own user inside the Dockerfile like this:

```dockerfile
FROM node:14-alpine

# Create a new user
RUN addgroup -g 1001 appgroup && adduser -D -u 1001 -G appgroup appuser

# Set working directory and switch to non-root user
WORKDIR /usr/src/app
COPY app.js .
USER appuser

CMD ["node", "app.js"]
```

Then run it without needing `--user`, because the image defaults to `appuser`.

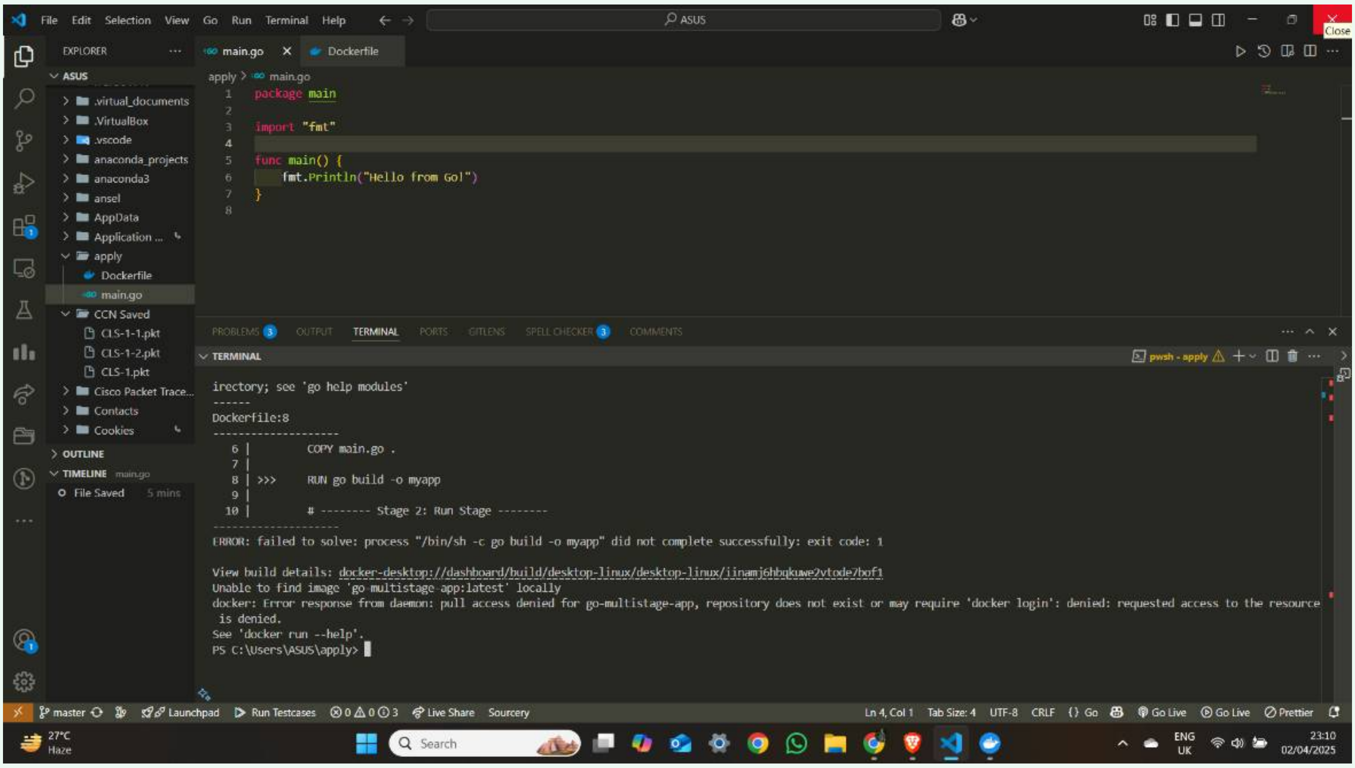
✅ Summary

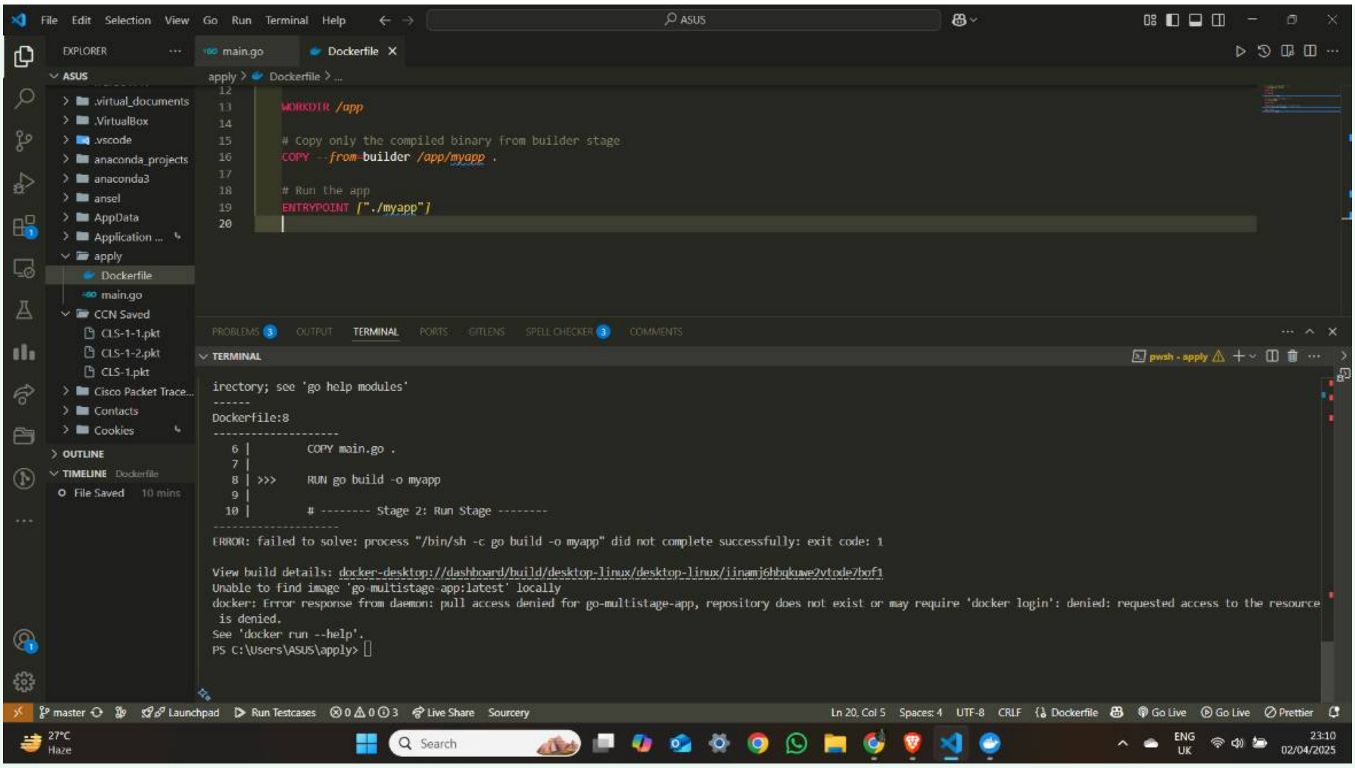
Term	Meaning
`--user`	Run the container as a specific user
`1001:1001`	User ID 1001 and Group ID 1001 (non-root)
Why use it?	Improves security by limiting privileges
Real-world analogy	Don’t give everyone admin access!

Would you like:

- A cheat sheet for Docker security best practices?
- Help setting up a custom user in your own Dockerfile?

Just let me know!






```
PS C:\Users\ASUS> docker build -t go-multistage-app .
>> docker run --rm go-multistage-app
>>
ERROR: error during connect: Head "http://%2F%2F.%2Fpipe%2FdockerDesktopLinuxEngine/_ping": open //./pipe/dockerDesktopLinuxEngine: The system cannot find the file specified.
docker: error during connect: Head "http://%2F%2F.%2Fpipe%2FdockerDesktopLinuxEngine/_ping": open //./pipe/dockerDesktopLinuxEngine: The system cannot find the file specified.
See 'docker run --help'.
PS C:\Users\ASUS> cd apply
PS C:\Users\ASUS\apply> docker build -t go-multistage-app .
>> docker run --rm go-multistage-app
>>
ERROR: error during connect: Head "http://%2F%2F.%2Fpipe%2FdockerDesktopLinuxEngine/_ping": open //./pipe/dockerDesktopLinuxEngine: The system cannot find the file specified.
docker: error during connect: Head "http://%2F%2F.%2Fpipe%2FdockerDesktopLinuxEngine/_ping": open //./pipe/dockerDesktopLinuxEngine: The system cannot find the file specified.
See 'docker run --help'.
PS C:\Users\ASUS\apply> docker build -t go-multistage-app .
>> docker run --rm go-multistage-app
>>
[+] Building 80.1s (11/12)                                   docker:desktop-linux
 => [internal] load build definition from Dockerfile                          0.0s
 => => transferring dockerfile: 448B                                          0.0s
 => [internal] load metadata for docker.io/library/alpine:latest              3.6s
 => [internal] load metadata for docker.io/library/golang:1.20                3.9s
 => [internal] load .dockerignore                                             0.0s
 => => transferring context: 2B                                               0.0s
 => [builder 1/4] FROM docker.io/library/golang:1.20@sha256:8f9af7094d0cb27cc783c697  74.9s
 => => resolve docker.io/library/golang:1.20@sha256:8f9af7094d0cb27cc783c697ac5ba25ef  0.0s
 => => sha256:8f9af7094d0cb27cc783c697ac5ba25efdc4da35f8526db21f7aebb 2.36kB / 2.36kB  0.0s
 => => sha256:68e92d11b04ec0fe48e60d59964704aca234084f87af5d1a068c 64.14MB / 64.14MB
```

```
16.2s
 => => sha256:c1a446d8f0c5658cc9af7b207a521995cf26bcfe1c9e6a4e148ba4e 1.79kB / 1.79kB  0.0s
 => => sha256:d5beeac3653f7c94aeda867588172915ce848d8d49d4ca24d242455 2.75kB / 2.75kB
0.0s
 => => sha256:6a299ae9cfd996c1149a699d36cdaa76fa332c8e9d66d6678fa9 49.58MB / 49.58MB
49.2s
 => => sha256:e08e8703b2fb5e50153f792f3192087d26970d262806b397049d 24.05MB / 24.05MB
35.7s
 => => sha256:4105062d1ee61f98bb0f7f6c0ac998caf55ff9a0e0fda9aa1fdd 92.37MB / 92.37MB  35.0s
 => => sha256:870a28135dd085067b84c7b2af0529d02eab744a63d2834ad2 100.45MB / 100.45MB
70.7s
 => => sha256:ffe80ff754489bcce87b4c3580467de7a34764fc1ad5831f6eed096c78 173B / 173B
36.3s
 => => sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb5577484a6d75e68dc38e8 32B / 32B
36.7s
 => => extracting sha256:6a299ae9cfd996c1149a699d36cdaa76fa332c8e9d66d6678fa9a231d9ea
2.2s
 => => extracting sha256:e08e8703b2fb5e50153f792f3192087d26970d262806b397049d61b9a14b
0.5s
 => => extracting sha256:68e92d11b04ec0fe48e60d59964704aca234084f87af5d1a068c49456b37
2.5s
 => => extracting sha256:4105062d1ee61f98bb0f7f6c0ac998caf55ff9a0e0fda9aa1fddbecc1f59  2.1s
 => => extracting sha256:870a28135dd085067b84c7b2af0529d02eab744a63d2834ad2780934573b
3.8s
 => => extracting sha256:ffe80ff754489bcce87b4c3580467de7a34764fc1ad5831f6eed096c7832  0.0s
 => => extracting sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb5577484a6d75e68dc38e8
0.0s
 => [internal] load build context                                            0.0s
 => => transferring context: 119B                                            0.0s
 => [stage-1 1/3] FROM docker.io/library/alpine:latest@sha256:a8560b36e8b8210634f77d  39.9s
 => => resolve docker.io/library/alpine:latest@sha256:a8560b36e8b8210634f77d9f7f9efd7  0.0s
 => => sha256:a8560b36e8b8210634f77d9f7f9efd7ffa463e380b75e2e74aff451 9.22kB / 9.22kB  0.0s
 => => sha256:1c4eef651f65e2f7daee7ee785882ac164b02b78fb74503052a26dc 1.02kB / 1.02kB  0.0s
 => => sha256:aded1e1a5b3705116fa0a92ba074a5e0b0031647d9c315983ccba2ee542 581B / 581B
0.0s
 => => sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e 3.64MB / 3.64MB
39.7s
 => => extracting sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5
0.1s
 => [stage-1 2/3] WORKDIR /app                                               0.0s
 => [builder 2/4] WORKDIR /app                                               0.8s
 => [builder 3/4] COPY main.go .                                             0.0s
 => ERROR [builder 4/4] RUN go build -o myapp                                0.3s
------
 > [builder 4/4] RUN go build -o myapp:
0.241 go: go.mod file not found in current directory or any parent dodules'
------
Dockerfile:8
--------------------
   6 |         COPY main.go .
   7 |
```

```
  8 | >>>     RUN go build -o myapp
  9 |
 10 |        # -------- Stage 2: Run Stage --------
--------------------
ERROR: failed to solve: process "/bin/sh -c go build -o myapp" did not complete successfully: exit
code: 1

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-
linux/xrk6s1s9hulxrzjhpx9udjou6
Unable to find image 'go-multistage-app:latest' locally
docker: Error response from daemon: pull access denied for go-multistage-app, repository does not
exist or may require 'docker login': denied: requested access to the resource is denied.
See 'docker run --help'.
PS C:\Users\ASUS\apply> docker build -t go-multistage-app .
>> docker run --rm go-multistage-app
>>
[+] Building 1.4s (11/12)                    docker:desktop-linux
 => [internal] load build definition from Dockerfile          0.0s
 => => transferring dockerfile: 448B                          0.0s
 => [internal] load metadata for docker.io/library/golang:1.2  1.1s
 => [internal] load metadata for docker.io/library/alpine:lat  1.0s
 => [internal] load .dockerignore                             0.0s
 => => transferring context: 2B                               0.0s
 => [builder 1/4] FROM docker.io/library/golang:1.20@sha256:8  0.0s
 => [internal] load build context                             0.0s
 => => transferring context: 119B                             0.0s
 => [stage-1 1/3] FROM docker.io/library/alpine:latest@sha256  0.0s
 => CACHED [stage-1 2/3] WORKDIR /app                         0.0s
 => CACHED [builder 2/4] WORKDIR /app                         0.0s
 => CACHED [builder 3/4] COPY main.go .                       0.0s
irectory; see 'go help modules'
------
Dockerfile:8
--------------------
  6 |       COPY main.go .
  7 |
  8 | >>>     RUN go build -o myapp
  9 |
 10 |        # -------- Stage 2: Run Stage --------
--------------------
ERROR: failed to solve: process "/bin/sh -c go build -o myapp" did not complete successfully: exit
code: 1

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-
linux/iinamj6hbqkuwe2vtode7bof1
Unable to find image 'go-multistage-app:latest' locally
docker: Error response from daemon: pull access denied for go-multistage-app, repository does not
exist or may require 'docker login': denied: requested access to the resource is denied.
See 'docker run --help'.
PS C:\Users\ASUS\apply>
```

```
main.go:
package main

import "fmt"

func main() {
   fmt.Println("Hello from Go!")
}

Dockerfile:
# -------- Stage 1: Build Stage --------
   FROM golang:1.20 AS builder

   WORKDIR /app

   COPY main.go .

   RUN go build -o myapp

   # -------- Stage 2: Run Stage --------
   FROM alpine:latest

   WORKDIR /app

   # Copy only the compiled binary from builder stage
   COPY --from=builder /app/myapp .

   # Run the app
   ENTRYPOINT ["./myapp"]
```

Thanks for sharing the full details and screenshots! I can clearly see the issue with your Go Docker multi-stage build, and I'll walk you through the **problem, cause, and exact fix** — step by step.

❌ Problem (from error logs):

```vbnet
ERROR: failed to solve: process "/bin/sh -c go build -o myapp" did not complete successfully: exit 
go: go.mod file not found in current directory or any parent directory; see 'go help modules'
```

🔍 Cause:

You're using the **Go module system**, but you don't have a `go.mod` file. Since Go 1.11+, Go expects a `go.mod` file to manage dependencies (even for simple apps).

In a multi-stage Docker build, when you run:

```Dockerfile
RUN go build -o myapp
```

Go tries to build using modules — but there's **no module file** (`go.mod`), so it fails.

✅ Solution Options:

🔧 Option 1: Use `GO111MODULE=off` for small simple apps (quick fix)

Update your Dockerfile's build line to this:

```Dockerfile
RUN GO111MODULE=off go build -o myapp
```

This tells Go: "Don't use modules. Just compile the file directly."

✅ Recommended Updated Dockerfile:

```Dockerfile
# -------- Stage 1: Build Stage --------
FROM golang:1.20 AS builder

WORKDIR /app

COPY main.go .

# Disable module mode and build app
RUN GO111MODULE=off go build -o myapp

# -------- Stage 2: Run Stage --------
FROM alpine:latest

WORKDIR /app

COPY --from=builder /app/myapp .

ENTRYPOINT ["./myapp"]
```

🔧 Option 2 (Better for larger Go apps): Add a go.mod file

If your project grows, create a proper Go module:

```bash
go mod init myapp
```

This will generate a `go.mod` file like:

```go
module myapp
```

```
go 1.20
```

Then your Docker build will work with the original command:

```Dockerfile
RUN go build -o myapp
```

🧪 How to Test Again:

After fixing the Dockerfile:

Step 1: Rebuild the image

```bash
docker build -t go-multistage-app .
```

Step 2: Run the container

```bash
docker run --rm go-multistage-app
```

Expected output:

```csharp
Hello from Go!
```

🛠️ Extra Note on Docker Daemon Error:

From earlier:

```arduino
ERROR: error during connect: Head "...dockerDesktopLinuxEngine": The system cannot find the file spe
```

🔍 This happens if **Docker Desktop is not running** or the **Linux engine backend is not available**.

✅ **Fix**:

- Open Docker Desktop manually (from Start Menu or taskbar).
- Wait until it shows `` `Docker Desktop is running` ``.
- Then retry the build/run.

✅ Summary

Problem	Fix
Missing `go.mod`	Add `GO111MODULE=off` to disable module mode
Docker engine not connected	Make sure Docker Desktop is running
Build failed	Use correct build command and Dockerfile structure

---

Would you like:

- A version with `go.mod` setup?
- A GitHub repo template for this Go multi-stage app?

Just let me know and I'll generate it for you!

```
root@83ebfd456292598d:~/code# mkdir my-node-app
root@83ebfd456292598d:~/code# cd my-node-app
root@83ebfd456292598d:~/code/my-node-app# docker build -t my-node-app:1.0 .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

unable to prepare context: unable to evaluate symlinks in Dockerfile path: lstat /root/code/my-node-app/Dockerfile: no such file or directory
root@83ebfd456292598d:~/code/my-node-app# docker build -t my-node-app:1.0 .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

unable to prepare context: unable to evaluate symlinks in Dockerfile path: lstat /root/code/my-node-app/Dockerfile: no such file or directory
root@83ebfd456292598d:~/code/my-node-app# docker build -t my-node-app:1.0 .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

unable to prepare context: unable to evaluate symlinks in Dockerfile path: lstat /root/code/my-node-app/Dockerfile: no such file or directory
root@83ebfd456292598d:~/code/my-node-app# docker build -t my-node-app:1.0 .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  3.072kB
Step 1/6 : FROM node:14-alpine
14-alpine: Pulling from library/node
f56be85fc22e: Pull complete
8f665685b215: Pull complete
```

```
e5fca6c395a6: Pull complete
561cb69653d5: Pull complete
Digest: sha256:434215b487a329c9e867202ff89e704d3a75e554822e07f3e0c0f9e606121b33
Status: Downloaded newer image for node:14-alpine
 ---> 0dac3dc27b1a
Step 2/6 : WORKDIR /usr/src/app
 ---> Running in 9d995ead2805
Removing intermediate container 9d995ead2805
 ---> 31dc12badb5f
Step 3/6 : COPY app.js .
 ---> c9338dde16d4
Step 4/6 : RUN npm install http@0.0.1-security
 ---> Running in 20181431d4a8
npm WARN saveError ENOENT: no such file or directory, open '/usr/src/app/package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open '/usr/src/app/package.json'
npm WARN app No description
npm WARN app No repository field.
npm WARN app No README data
npm WARN app No license field.

+ http@0.0.1-security
added 1 package and audited 1 package in 0.869s
found 0 vulnerabilities

Removing intermediate container 20181431d4a8
 ---> 1861609ffd13
Step 5/6 : EXPOSE 8080
 ---> Running in a128b4aa1563
Removing intermediate container a128b4aa1563
 ---> d060d7a09d0c
Step 6/6 : CMD ["node", "app.js"]
 ---> Running in 46fc8fbce61d
Removing intermediate container 46fc8fbce61d
 ---> 15092796b047
Successfully built 15092796b047
Successfully tagged my-node-app:1.0
root@83ebfd456292598d:~/code/my-node-app# docker images
REPOSITORY    TAG         IMAGE ID       CREATED          SIZE
my-node-app   1.0         15092796b047   10 seconds ago   119MB
node          14-alpine   0dac3dc27b1a   2 years ago      119MB
root@83ebfd456292598d:~/code/my-node-app# docker run -d --name my-node-app-container -p
80:8080 my-node-app:1.0
473e519c8d4183fbf1c54a467c43e476e983c83c9838f7927f9d71ac9cd4a381
root@83ebfd456292598d:~/code/my-node-app# docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS
NAMES
473e519c8d41   my-node-app:1.0   "docker-entrypoint.s…"   7 seconds ago    Up 6 seconds
0.0.0.0:80->8080/tcp, :::80->8080/tcp   my-node-app-container
root@83ebfd456292598d:~/code/my-node-app# curl http://localhost
Hello, World!
```

```
root@83ebfd456292598d:~/code/my-node-app# docker history my-node-app:1.0
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
15092796b047   55 seconds ago   /bin/sh -c #(nop)  CMD ["node" "app.js"]        0B
d060d7a09d0c   55 seconds ago   /bin/sh -c #(nop)  EXPOSE 8080                  0B
1861609ffd13   55 seconds ago   /bin/sh -c npm install http@0.0.1-security      8.26kB
c9338dde16d4   57 seconds ago   /bin/sh -c #(nop) COPY file:08eb748da9207b81…   350B
31dc12badb5f   57 seconds ago   /bin/sh -c #(nop) WORKDIR /usr/src/app          0B
0dac3dc27b1a   2 years ago      /bin/sh -c #(nop)  CMD ["node"]                 0B
<missing>      2 years ago      /bin/sh -c #(nop)  ENTRYPOINT ["docker-entry…   0B
<missing>      2 years ago      /bin/sh -c #(nop) COPY file:4d192565a7220e13…   388B
<missing>      2 years ago      /bin/sh -c apk add --no-cache --virtual .bui…   7.85MB
<missing>      2 years ago      /bin/sh -c #(nop)  ENV YARN_VERSION=1.22.19     0B
<missing>      2 years ago      /bin/sh -c addgroup -g 1000 node     && addu…   104MB
<missing>      2 years ago      /bin/sh -c #(nop)  ENV NODE_VERSION=14.21.3     0B
<missing>      2 years ago      /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      2 years ago      /bin/sh -c #(nop) ADD file:9a4f77dfaba7fd2aa…   7.05MB
root@83ebfd456292598d:~/code/my-node-app# docker build -t my-node-app:2.0 .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  3.072kB
Step 1/9 : FROM node:14-alpine AS builder
 ---> 0dac3dc27b1a
Step 2/9 : WORKDIR /app
 ---> Running in 9f0be860ef6c
Removing intermediate container 9f0be860ef6c
 ---> da7948db9349
Step 3/9 : COPY app.js .
 ---> 085e0a25772f
Step 4/9 : RUN npm install http@0.0.1-security
 ---> Running in e33937811037
npm WARN saveError ENOENT: no such file or directory, open '/app/package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open '/app/package.json'
npm WARN app No description
npm WARN app No repository field.
npm WARN app No README data
npm WARN app No license field.

+ http@0.0.1-security
added 1 package and audited 1 package in 2.155s
found 0 vulnerabilities

Removing intermediate container e33937811037
 ---> 960c787301d9
Step 5/9 : FROM node:14-alpine
 ---> 0dac3dc27b1a
Step 6/9 : WORKDIR /app
 ---> Using cache
 ---> da7948db9349
```

```
Step 7/9 : COPY --from=builder /app .
 ---> 01a0daeb9597
Step 8/9 : EXPOSE 8080
 ---> Running in ec4b5611da7b
Removing intermediate container ec4b5611da7b
 ---> 377898bbd1d1
Step 9/9 : CMD ["node", "app.js"]
 ---> Running in aee833642b47
Removing intermediate container aee833642b47
 ---> 6f1ae6f0c735
Successfully built 6f1ae6f0c735
Successfully tagged my-node-app:2.0
root@83ebfd456292598d:~/code/my-node-app# docker images
REPOSITORY    TAG       IMAGE ID       CREATED              SIZE
my-node-app   2.0       6f1ae6f0c735   6 seconds ago        119MB
<none>        <none>    960c787301d9   6 seconds ago        119MB
my-node-app   1.0       15092796b047   About a minute ago   119MB
node          14-alpine   0dac3dc27b1a   2 years ago        119MB
root@83ebfd456292598d:~/code/my-node-app# docker scan my-node-app:1.0
docker: 'scan' is not a docker command.
See 'docker --help'
root@83ebfd456292598d:~/code/my-node-app# docker run -it my-node-app:1.0 /bin/sh
/usr/src/app # cat /etc/passwd
root:x:0:0:root:/root:/bin/ash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/mail:/sbin/nologin
news:x:9:13:news:/usr/lib/news:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucppublic:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
man:x:13:15:man:/usr/man:/sbin/nologin
postmaster:x:14:12:postmaster:/var/mail:/sbin/nologin
cron:x:16:16:cron:/var/spool/cron:/sbin/nologin
ftp:x:21:21::/var/lib/ftp:/sbin/nologin
sshd:x:22:22:sshd:/dev/null:/sbin/nologin
at:x:25:25:at:/var/spool/cron/atjobs:/sbin/nologin
squid:x:31:31:Squid:/var/cache/squid:/sbin/nologin
xfs:x:33:33:X Font Server:/etc/X11/fs:/sbin/nologin
games:x:35:35:games:/usr/games:/sbin/nologin
cyrus:x:85:12::/usr/cyrus:/sbin/nologin
vpopmail:x:89:89::/var/vpopmail:/sbin/nologin
ntp:x:123:123:NTP:/var/empty:/sbin/nologin
smmsp:x:209:209:smmsp:/var/spool/mqueue:/sbin/nologin
guest:x:405:100:guest:/dev/null:/sbin/nologin
nobody:x:65534:65534:nobody:/:/sbin/nologin
```

```
node:x:1000:1000:Linux User,,,:/home/node:/bin/sh
/usr/src/app #
```

Great job! You're doing amazing progress. Let me break down everything that just happened in your terminal — **what went right** ✅, **what can be improved** 🚀, and **what to do next** 🔧 — all in super simple terms.

✅ What You Did Successfully

✅ 1. Built a Docker Image from a Node.js App

You used a Dockerfile to build an image:

```bash
docker build -t my-node-app:1.0 .
```

and later:

```bash
docker build -t my-node-app:2.0 .
```

✅ **Both were successful** and created images `` `my-node-app:1.0` `` and `` `my-node-app:2.0` ``. Great!

✅ 2. Started the App in a Container

You ran:

```bash
docker run -d --name my-node-app-container -p 80:8080 my-node-app:1.0
```

✅ Your app was running inside a container and responded with:

```
Hello, World!
```

when accessed using:

```bash
curl http://localhost
```

✅ 3. Explored Docker Layers

Using:

```bash
docker history my-node-app:1.0
```

You viewed the history of how the image was built — **step by step**. Every `RUN`, `COPY`, and `CMD` created a layer.

✅ 4. Checked Container User Info

You went inside the container:

```bash
docker run -it my-node-app:1.0 /bin/sh
```

And listed all users using:

```bash
cat /etc/passwd
```

✅ You confirmed that the image has a **non-root user named `node`** with:

```bash
node:x:1000:1000:Linux User,,,:/home/node:/bin/sh
```

So now you can safely run the container using:

```bash
docker run -u node my-node-app:1.0
```

🚫 What Didn't Work (And How to Fix It)

❌ 1. Docker Scan Not Found

You tried:

```bash
docker scan my-node-app:1.0
```

And got:

```vbnet
docker: 'scan' is not a docker command.
```

🔧 Fix:

You need to install Docker Scan manually.

For Docker Desktop:

- Open Docker Desktop
- Go to **Settings > Extensions**
- Enable or install **Docker Scan**

Or use Trivy (lightweight alternative):

Install Trivy (recommended):

```bash
brew install aquasecurity/trivy/trivy    # for macOS
sudo apt install trivy                    # for Ubuntu/Debian
```

Then scan:

```bash
trivy image my-node-app:1.0
```

🧠 Extra Tips and Best Practices

🔐 Run as Non-Root User

You confirmed `node` user exists. So update your Dockerfile to use it:

Dockerfile (updated):

```Dockerfile
FROM node:14-alpine

WORKDIR /usr/src/app

COPY app.js .

RUN npm install http@0.0.1-security

USER node  # run as non-root

EXPOSE 8080

CMD ["node", "app.js"]
```

Then rebuild:

```bash
docker build -t my-node-app:secure .
```

🔁 Use Restart Policy

```bash
docker run -d --name my-node-app-container --restart always -p 80:8080 my-node-app:1.0
```

This makes sure your app restarts if it crashes or if the system restarts.

🌍 Push to Docker Hub (Optional Sharing)

If you want to share your image:

```bash
docker login
docker tag my-node-app:1.0 yourDockerID/my-node-app:1.0
docker push yourDockerID/my-node-app:1.0
```

Then anyone can use:

```bash
docker pull yourDockerID/my-node-app:1.0
```

✅ Summary (What You Accomplished)

Task	Status
Create Docker image	✅ Done
Run container and test	✅ Done
Use multi-stage build	✅ Done
Checked build layers	✅ Done
Checked system users	✅ Done
Verified non-root user	✅ Done
Tried Docker Scan	⚠️ Needs extra install
Built `:2.0` version	✅ Done
Ready to push to Docker Hub	✅✅✅

Would you like me to:

- Help you write a production-ready Dockerfile?
- Set up a `.dockerignore` file?
- Or give you a GitHub Actions workflow for CI/CD?

Let me know — I’m happy to help! 💻🐳