

Docker

Part 1:-

1. Problem Statement
2. Installation of Docker CLI and Desktop
3. Understanding Images v/s Containers
4. Running Ubuntu Image in Container
5. Multiple Containers
6. Port Mappings
7. Environment Variables
8. Dockerization of Node.js Application
 - a. Dockerfile
 - b. Caching Layers
 - c. Publishing to Hub
9. Docker Compose
 - a. Services
 - b. Port Mapping

c. Env Variables

Part 2:-

Docker Networking

- a. Bridge
- b. Host

Volume Mounting

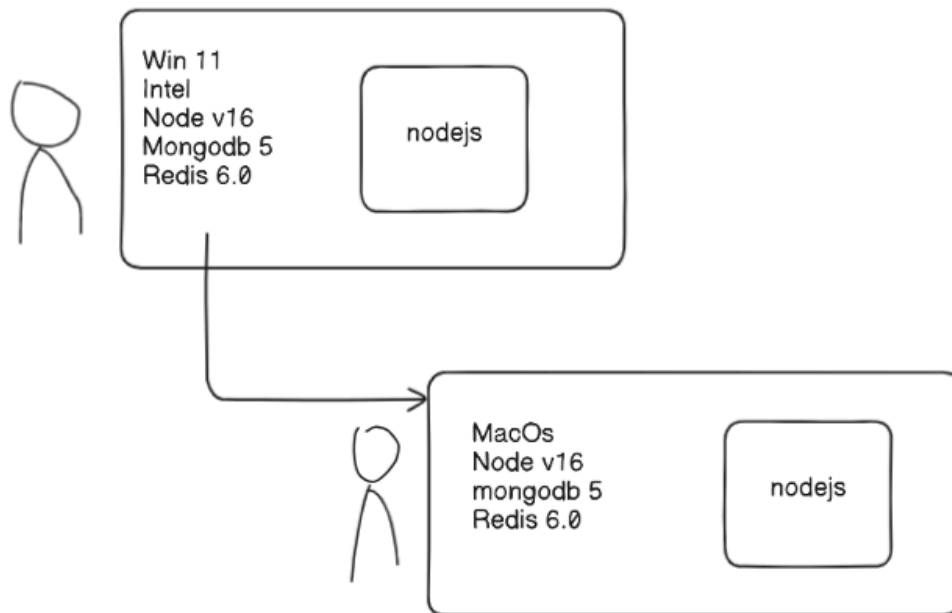
Efficient Caching in Layers

Docker Multi-Stage Builds

Bonus: Amazon Elastic Container Service and ECR

Part-1:- Problem Statement

Multiple Env. -> Replicate



→ If a person using windows system and make a Next.js application and after two years he want to run the exact code but in this time in supposed to use Mac os system and the environment is totally different. Suppose this problem is in a big team like open source Big MNCs where a large number of people coming how many of them can

setup the same environment and the operating system is also different.

→To Solve this problem we use Dockers(Containers)

2. Installation of Docker CLI and Desktop

1. Download Docker Desktop:

- Go to the Docker Hub and download the Docker Desktop installer for Windows.

2. Install Docker Desktop:

- Run the installer you downloaded.
- During installation, ensure that the option for WSL 2 (Windows Subsystem for Linux) is checked. This is recommended for better performance.

3. Enable WSL 2 (if not already enabled):

Open PowerShell as Administrator and run:

bash

```
wsl --install
```

-

If you already have WSL installed, ensure it's updated to WSL 2. You can check the version by running:

bash

```
wsl -l -v
```

-

To set a distribution to WSL 2, use:

bash

Copy code

```
wsl --set-version <Distro> 2
```

-

- Restart your computer if prompted.

4. Start Docker Desktop:

- After installation, launch Docker Desktop. It may take a moment to start as it initializes the WSL backend.

5. Configure Docker:

- Docker Desktop should prompt you to sign in or create a Docker Hub account. You can skip this step if you want to use it locally.

- Configure any settings you need, like CPU and memory allocation in the Docker settings.

6. **Verify Installation:**

Open a terminal (Command Prompt, PowerShell, or Windows Terminal) and run:

bash

Copy code

```
docker --version
```

○

To check if Docker is running, you can run:

bash

Copy code

```
docker run hello-world
```

○

- This command downloads a test image and runs it. If everything is set up correctly, you should see a confirmation message.

○

Docker run -it ubuntu

This command we use in our terminal to download ubuntu image for the first time after ctrl+d to exit

—> then our Containers is created and each container has its unique id

Understanding Images v/s Containers

Images:-

- **Blueprint:** Think of an image as a blueprint or a recipe. It contains everything needed to run an application: the code, libraries, dependencies, and environment settings.
- **Static:** Images don't change. They are built once and can be shared or stored.

Containers:-

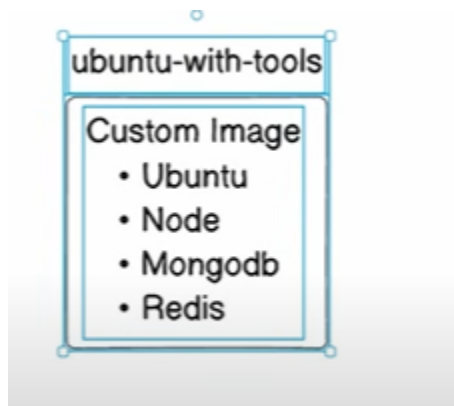
- **Running Instance:** A container is a running instance of an image. It's like using the blueprint to build a house. Once you build it, you can live in it and make changes.
- **Dynamic:** Containers can be started, stopped, modified, and deleted. They can also save changes (if configured to do so).

Quick Analogy

- **Image:** A cookbook (you can have many copies).
- **Container:** A dish you prepare from that cookbook (you can eat, change it, or throw it away).

In summary:

- **Image** = Blueprint (static).
- **Container** = House built from the blueprint (dynamic).



```
C:\Users\Dell> docker start hungry_lewin
```

```
hungry_lewin
```

```
C:\Users\Dell> docker stop
```

```
hungry_lewinhungry_lewin
```

C:\Users\Dell>

For Implementing Node js in Docker we can directly take it from hub.docker where in the explore section → Docker official image OR you can manually install it in your own image.

```
C:\Users\Dell> docker run -it node
Unable to find image 'node:latest' locally
latest: Pulling from library/node
8cd46d290033: Pull complete
2e6afa3f266c: Pull complete
2e66a70da0be: Pull complete
1c8ff076d818: Pull complete
71a2ad2ab1a1: Pull complete
8ed09065f016: Pull complete
8cc9946ce160: Pull complete
fa87db89e50a: Pull complete
Digest: sha256:cbe2d5f94110cea9817dd8c5809d05df49b4bd1aac5203f3594d88665ad37988
Status: Downloaded newer image for node:latest
Welcome to Node.js v22.9.0.
Type ".help" for more information.
> console.log('Hello World')
Hello World
undefined
>
```

When we want to run node in ubuntu image it shows the following:-

```
Microsoft Windows [Version 10.0.19045.4894]  
(c) Microsoft Corporation. All rights reserved.  
  
C:\Users\Dell> docker run -it ubuntu  
root@fca345118e2b:/# node  
bash: node: command not found  
root@fca345118e2b:/#
```

Port Mappings

Port mapping in Docker allows you to connect your container's ports to your host machine's ports. This is essential for accessing services running inside containers from outside.

How It Works:

1. ****Containers and Ports****: Each Docker container runs in its own isolated environment. If a service

inside the container listens on a specific port (like a web server on port 80), you need a way to access that service from your host.

2. **Mapping Ports**: When you run a Docker container, you can use the `-p` (or `--publish`) option to map a port from the container to a port on your host. The syntax is:

```
``docker run -p host_port:container_port  
image_name ``
```

- **host_port**: The port on your host machine.

- **container_port**: The port inside the container where the service is running.

3. **Example**: If you run a web server in a container listening on port 80 and want to access it through port 8080 on your host, you would do:

...

```
docker run -p 8080:80 my_web_server
```

...

Now, when you go to `http://localhost:8080` on your web browser, you'll be directed to the web server running inside the container.

Benefits:- ****Access Control****: You can expose only the necessary ports, keeping other services in the container secure.

- ****Multiple Containers****: You can run multiple containers with different services and map them to different host ports, avoiding conflicts

Important Points:

- **Default Behavior**: If you don't specify a host port (e.g., `-p :80``), Docker will randomly assign an available port on the host.
- **Binding to All Interfaces**: By default, the port mapping binds to all interfaces. You can specify an IP to restrict access, like `-p 127.0.0.1:8080:80``, which makes it accessible only from the local machine.

In summary, port mapping in Docker is a straightforward way to connect your containerized applications to the outside world, allowing you to easily access and manage them.



In general when we want to run the localhost server of mailhog of docker in our local browser directly it will not happen instead we can use port mapping

Docker run -it -p 8025:8025 mailhog/mailhog

```
Microsoft Windows [Version 10.0.19045.4894]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Dell> docker run -it -p 8025:8025 mailhog/mailhog
2024/09/23 06:12:25 Using in-memory storage
2024/09/23 06:12:25 [SMTP] Binding to address: 0.0.0.0:1025
[HTTP] Binding to address: 0.0.0.0:8025
2024/09/23 06:12:25 Serving under http://0.0.0.0:8025/
Creating API v1 with WebPath:
Creating API v2 with WebPath:
```



```
last login: Sat Oct 21 12:14:18 on ttys003
piyushgarg@macbook.pro > docker run -it piyushgargdev/mynodeapp
piyushgarg@macbook.pro >
piyushgarg@macbook.pro > docker run -it -p 9000:9000 piyushgargdev/mynodeapp
Server Running at PORT 9000
CACACACAC
piyushgarg@macbook.pro >
piyushgarg@macbook.pro >
piyushgarg@macbook.pro > docker run -it -p 6000:9000 piyushgargdev/mynodeapp
Server Running at PORT 9000
CACACACACACACACACAC
piyushgarg@macbook.pro > docker run -it -p 6000:9000 piyushgargdev/mynodeapp
```

5. Run [mailhog](#) to view emails sent during development

NOTE: Required when `E2E_TEST_MAILHOG_ENABLED` is "1"

```
docker pull mailhog/mailhog
```

```
docker run -d -p 8025:8025 -p 1025:1025 mailhog/mailhog
```

Environment Variables

Environment variables in Docker are used to pass configuration settings to your applications running

inside containers. They help customize behavior without changing the application code. Here's a simple breakdown:

Why Use Environment Variables?

1. **Configuration:** Instead of hardcoding settings (like database URLs, API keys, etc.), you can use environment variables to keep your code flexible and portable.
2. **Security:** Sensitive information (like passwords) can be stored in environment variables instead of in the source code, making it less likely to be exposed.

How to Set Environment Variables

Using the `-e` Flag: When you run a container, you can set an environment variable using the `-e` flag:

bash

Copy code

```
docker run -e VARIABLE_NAME=value  
image_name
```

Example:

bash

Copy code

```
docker run -e  
DB_PASSWORD=mysecretpassword my_app
```

1.

2. **Using an .env File:** You can also store environment variables in a file and pass them to Docker using the `--env-file` option:

Create a file named `.env`:

makefile

Copy code

```
DB_USER=myuser
```

```
DB_PASSWORD=mysecretpassword
```

○

○ Run the container with the `--env-file` option:

bash

Copy code

```
docker run --env-file .env my_app
```

Environment variables in Docker are used to pass configuration settings to your applications running inside containers. They help customize

behavior without changing the application code. Here's a simple breakdown:

Best Practices

- **Keep Sensitive Data Secure:** Avoid hardcoding sensitive information in your images or code.
- **Use Default Values:** In your application, consider using default values if environment variables aren't set, which helps avoid errors.
- **Documentation:** Document the required environment variables for your application to help others understand what they need to set up.

In summary, environment variables are a powerful way to configure your Docker containers flexibly and securely, making your applications easier to manage and deploy

Dockerization of Node.js Application

Open vs code by terminal

```
C:\Users\Dell> cd C:\Users\Dell\Desktop\Coding  
C:\Users\Dell\Desktop\Coding> mkdir docker-node  
C:\Users\Dell\Desktop\Coding> code .  
C:\Users\Dell\Desktop\Coding>
```

—> It will automatically go to vs code then go to terminal and write npm init and after click enter button only.

```
PS C:\Users\Dell\Desktop\Coding> npm init  
This utility will walk you through creating a package.json file.  
It only covers the most common items, and tries to guess sensible defaults.  
  
See `npm help init` for definitive documentation on these fields  
and exactly what they do.  
  
Use `npm install <pkg>` afterwards to install a package and  
save it as a dependency in the package.json file.
```

—>After this install express

```
PS C:\Users\Dell\Desktop\Coding> npm i express  
  
added 65 packages, and audited 66 packages in 5s  
  
13 packages are looking for funding  
  run `npm fund` for details  
  
found 0 vulnerabilities  
PS C:\Users\Dell\Desktop\Coding> █
```

```
node.js  
package  
main.js  
package-lock
```

```
Dockerfile
```

```
Welcome JS main.js X Dockerfile docker-compose.yml
JS main.js > ...
1  const express = require('express')
2  const app = express();
3
4  const PORT = process.env.PORT || 8000
5
6  app.get("/", (req,res) =>{
7    return res.json({message: "Hey ,I am Node js in Container"})
8  })
9
10 app.listen(PORT , ()=> console.log(`Server started on Port: ${PORT}`));
```

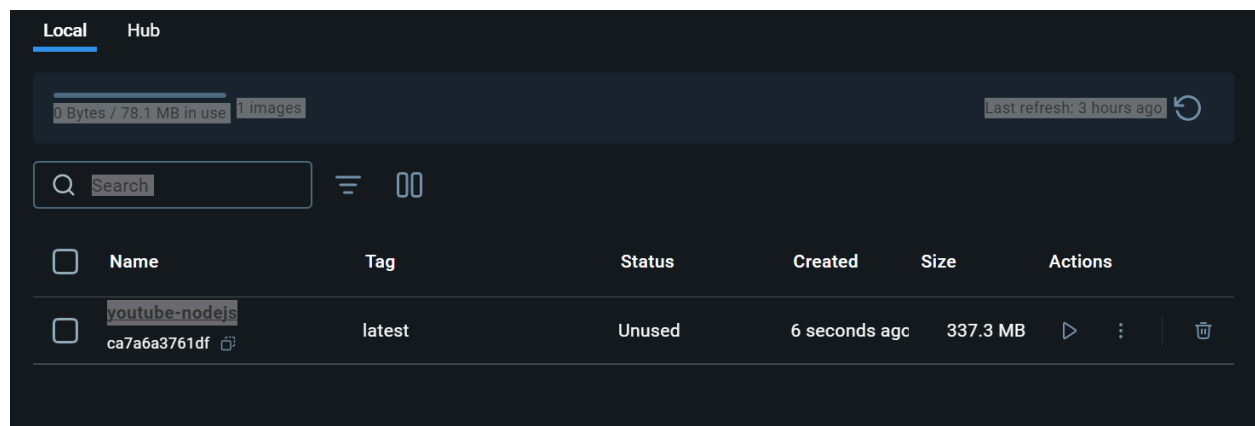
—>Go to Browser and search Ubuntu node js installation

```
Welcome JS main.js Dockerfile X
Dockerfile
1  FROM ubuntu
2
3  RUN apt-get update
4  RUN apt-get install -y curl
5  RUN curl -sL https://deb.nodesource.com/setup\_18.x | bash -
6  RUN apt-get upgrade -y
7  RUN apt-get install -y nodejs
8
9  COPY package.json package.json
10 COPY package-lock.json package-lock.json
11 COPY main.js main.js
12
13 RUN npm install
14
15 ENTRYPOINT [ "node","main.js"]
```

→ Building image of our own node js application

```
found 0 vulnerabilities
PS C:\Users\Dell\Desktop\Coding> docker build -t youtube-nodejs .
[+] Building 191.5s (16/16) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 368B
=> [internal] load metadata for docker.io/library/ubuntu:latest
=> [auth] library/ubuntu:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 28.13kB
=> [ 1/10] FROM docker.io/library/ubuntu:latest@sha256:dfc10878be8d8fc9c61cbff33166cb1d1fe44391539243703c727668
=> => resolve docker.io/library/ubuntu:latest@sha256:dfc10878be8d8fc9c61cbff33166cb1d1fe44391539243703c72766894
=> => sha256:dfc10878be8d8fc9c61cbff33166cb1d1fe44391539243703c72766894fa834a 1.34kB / 1.34kB
=> => sha256:77d57fd89366f7d16615794a5b53e124d742404e20f035c22032233f1826bd6a 424B / 424B
=> => sha256:b1e9cef3f2977f8bdd19eb9ae04f83b315f80fe4f5c5651fedf41482c12432f7 2.30kB / 2.30kB
=> => sha256:dafa2b0c44d2cfb0be6721f079092ddf15dc8bc537fb07fe7c3264c15cb2e8e6 29.75MB / 29.75MB
=> => extracting sha256:dafa2b0c44d2cfb0be6721f079092ddf15dc8bc537fb07fe7c3264c15cb2e8e6
=> [ 2/10] RUN apt-get update
=> [ 3/10] RUN apt-get install -y curl
=> [ 4/10] RUN curl -sL https://deb.nodesource.com/setup_18.x | bash -
=> [ 5/10] RUN apt-get upgrade -y
=> [ 6/10] RUN apt-get install -y nodejs
=> [ 7/10] COPY package.json package.json
=> [ 8/10] COPY package-lock.json package-lock.json
=> [ 9/10] COPY main.js main.js
```

→ Now this is my custom image



	Name	Tag	Status	Created	Size	Actions
<input type="checkbox"/>	youtube-nodejs ca7a6a3761df	latest	Unused	6 seconds ago	337.3 MB	

→ Port Mapping


```
C:\Users\Dell\Desktop\Coding> docker run -it youtube-nodejs
server started on Port: 8000

C:\Users\Dell\Desktop\Coding> docker run -it -p 8000:8000 youtube-nodejs
server started on Port: 8000
```

```
(c) Microsoft Corporation. All rights reserved.

C:\Users\Dell> docker exec -it dd4fba4ea666 bash
root@dd4fba4ea666:/# ls
bin  dev  home  lib64  media  node_modules  package-lock.json  proc  run  srv  tmp  var
boot  etc  lib  main.js  mnt  opt  package.json  root  sbin  sys  usr
root@dd4fba4ea666:/# cat main.js
const express = require('express')
const app = express();

const PORT = process.env.PORT || 8000

app.get("/", (req,res) =>{
  return res.json({message: "Hey ,I am Node js in Container"})
})

app.listen(PORT , ()=> console.log(`Server started on Port: ${PORT}`));root@dd4fba4ea666:/#
```

HTTPlocalhost:8000

Save

Share

GETlocalhost:8000

Send

Params

Authorization

Headers (8)

Body

Scripts

Settings

Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (7)

Test Results

200 OK • 1182 ms • 279 B •

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "message": "Hey ,I am Node js in Container"
3 }
```

→ Use of Environment variables

```
C:\Users\Dell\Desktop\Coding> docker run -it -e PORT=4000 -p 4000:4000 youtube-nodejs
Server started on Port: 4000
```

```
main.js  Dockerfile X
Dockerfile > ...
1  FROM node
2
3
4  COPY package.json package.json
5  COPY package-lock.json package-lock.json
6  COPY main.js main.js
7
8  RUN npm install
9
10 ENTRYPOINT [ "node", "main.js" ]
```

→ Go to Docker hub website and made a repository paste the name

```
What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview
C:\Users\Dell\Desktop\Coding> docker build -t vishwajeetpandey/youtube-nodejs .
[+] Building 3.9s (16/16) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 368B
=> [internal] load metadata for docker.io/library/ubuntu:latest
=> [auth] library/ubuntu:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [ 1/10] FROM docker.io/library/ubuntu:latest@sha256:dfc10878be8d8fc9c61cbff33166cb1d1fe44391539243703c72766
```

Push the code

```
Microsoft Windows [Version 10.0.19045.4894]
(c) Microsoft Corporation. All rights reserved.

C:\Users\De11> docker push vishwajeetpandey/youtube-nodejs:latest
The push refers to repository [docker.io/vishwajeetpandey/youtube-nodejs]
55efdfdca696: Pushed
621d750228e4: Pushed
74b205c6de7b: Pushed
4729769cdf53: Pushed
acdda8f81f25: Pushed
2a3b8a98d809: Pushed
d544df2d849f: Pushed
1ab5d4c425cb: Pushed
6cb5812fff97: Pushed
b15b682e901d: Mounted from library/ubuntu
latest: digest: sha256:d0f6944995a3d8ee1c2a55e03301d1bb96e73a75751274f906ff8f05dbd30d79 size: 2418

C:\Users\De11> IN USE
```

node.js
package
main.js
package-lock









Dockerfile
UBUNTU

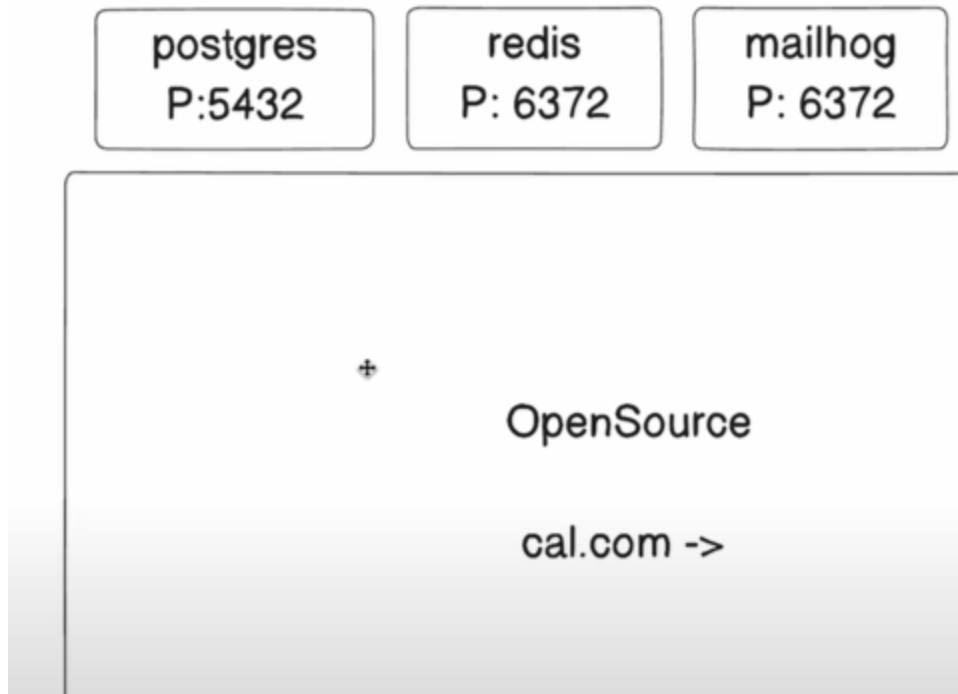
node v.18.0

ubuntu

```
Welcome JS main.js Dockerfile docker-compose.yml X
docker-compose.yml
1 version: '3.8'
2
3 services:
4   postgres:
5     image: postgres # hub.docker.com
6     ports:
7       - '5432:5432'
8     environment:
9       POSTGRES_USER: postgres
10      POSTGRES_DB: review
11      POSTGRES_PASSWORD: password
12
13   redis:
14     image: redis
15     ports:
16       - '6379:6379'
17
```

```
PS C:\Users\Dell\Desktop\Coding> docker compose up
time="2024-09-23T14:49:08+05:30" level=warning msg="C:\\Users\\Dell\\Desktop\\Coding\\docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion"
[+] Running 2/2
 ✓ redis Pulled
 ✓ postgres Pulled
[+] Running 3/3
```

<input type="checkbox"/>	Name ↑	Image	Status	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	 coding		Running (2/2)		0.76%	6 minutes ago	<input type="checkbox"/> ⋮ 
<input type="checkbox"/>	 postgres: fdbc9a4c	postgres	Running	5432:5432 	0.01%	6 minutes ago	<input type="checkbox"/> ⋮ 
<input type="checkbox"/>	 redis-1 6a71e9d0	redis	Running	6379:6379 	0.75%	6 minutes ago	<input type="checkbox"/> ⋮ 



```
C:\Users\Dell> docker run -it --name my_container busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
2fce1e0cdfc5: Pull complete
Digest: sha256:c230832bd3b0be59a6c47ed64294f9ce71e91b327957920b6929a0caa8353140
Status: Downloaded newer image for busybox:latest
/ # ping google.com
PING google.com (142.250.76.174): 56 data bytes
64 bytes from 142.250.76.174: seq=0 ttl=63 time=27.621 ms
64 bytes from 142.250.76.174: seq=1 ttl=63 time=24.922 ms
64 bytes from 142.250.76.174: seq=2 ttl=63 time=21.963 ms
64 bytes from 142.250.76.174: seq=3 ttl=63 time=22.509 ms
64 bytes from 142.250.76.174: seq=4 ttl=63 time=22.875 ms
64 bytes from 142.250.76.174: seq=5 ttl=63 time=22.546 ms
64 bytes from 142.250.76.174: seq=6 ttl=63 time=24.073 ms
64 bytes from 142.250.76.174: seq=7 ttl=63 time=25.123 ms
64 bytes from 142.250.76.174: seq=8 ttl=63 time=24.050 ms
64 bytes from 142.250.76.174: seq=9 ttl=63 time=39.305 ms
64 bytes from 142.250.76.174: seq=10 ttl=63 time=23.740 ms
64 bytes from 142.250.76.174: seq=11 ttl=63 time=24.263 ms
64 bytes from 142.250.76.174: seq=12 ttl=63 time=23.571 ms
64 bytes from 142.250.76.174: seq=13 ttl=63 time=24.024 ms
^C
```

Part-2

Docker Networking

- c. Bridge
- d. Host

Docker networking is a crucial aspect of container orchestration, enabling communication between containers, the host system, and external networks. Here's a breakdown of the key concepts:

1. **Networking Modes** Docker offers several networking modes, each suited for different use cases:

- **Bridge Network**: This is the default network mode. Containers are connected to a private internal network (the bridge), and can communicate with each other using their

IP addresses. Ports can be mapped to the host for external access.

- **Host Network**: In this mode, a container shares the host's networking namespace. This means it can access the host's network stack directly, without any isolation. It's useful for performance-sensitive applications but reduces isolation.

- **None Network**: This disables all networking for the container. It's used for scenarios where no network connectivity is needed.

- **Overlay Network**: Used primarily in Docker Swarm, overlay networks allow containers running on different hosts to communicate securely. Docker creates an internal virtual network that spans multiple hosts.

- **Macvlan Network**: This allows you to assign a MAC address to a container, making it appear as a physical device on the network. This is useful for legacy

applications or when you need to integrate with existing network configurations.

2. **Network Drivers**

Each network mode uses different drivers. Common drivers include:

- **bridge**: The default driver for bridge networks.
- **overlay**: For multi-host communication in Swarm mode.
- **macvlan**: For direct integration with the physical network.
- **host**: Direct use of the host's network stack.

3. **Container Communication**

Containers can communicate in various ways:

- **Using IP Addresses**: Each container on a bridge network has an IP address. Containers can communicate directly using these addresses.

- **Container Names**: Docker's built-in DNS service allows containers to resolve each other by name. This simplifies communication, as you can use the container name instead of the IP.

4. **Port Mapping**

To allow external access to a container's services, you can map container ports to host ports using the `-p` flag when starting a container:

```
```bashdocker run -p <host_port>:<container_port>  
<image_name>```
```

### ### 5. **\*\*Network Configuration\*\***

You can create custom networks using:

```
```bash
```

```
docker network create <network_name>```
```

Custom networks allow you to fine-tune settings like subnet, gateway, and network driver.

6. ****Inspecting Networks****You can inspect existing networks to see their configurations and connected containers:

```
```bashdocker network inspect <network_name>```
```

### ### 7. **\*\*Security and Isolation\*\***

Docker networks provide security through isolation. Containers on a bridge network cannot communicate with those on another bridge network unless explicitly connected.

### ### 8. **\*\*Networking Tools\*\***Docker also integrates with networking tools like:

- **\*\*Docker Compose\*\***: Simplifies multi-container setups and allows you to define networks in a ``docker-compose.yml`` file.

- **\*\*Docker Swarm\*\***: Enhances orchestration with built-in overlay networking for scaling applications across multiple hosts.

### ### Conclusion

Docker networking is flexible and powerful, enabling seamless communication between containers, while providing options for isolation and performance.

Understanding these concepts is vital for effectively deploying and managing containerized applications. If you have specific questions or need more detail on any aspect, feel free to ask!

—> Go to the docker networking website

```
C:\Users\De11> docker network ls
NETWORK ID NAME DRIVER SCOPE
c8a87cc9d6ca bridge bridge local
b569d0c5376f host host local
4a733316bd04 none null local

C:\Users\De11>
```

—> In bridge the default nature of docker to connect it with external network through ip address

—> In host there is no need of port mapping because In host docker and our local machine work on the same port.

```
\Users\Dell> docker network ls
NETWORK ID NAME DRIVER SCOPE
a87cc9d6ca bridge bridge local
69d0c5376f host host local
733316bd04 none null local

\Users\Dell> docker run -it --network=host busybox
#
```

```
C:\Users\Dell> docker run -it --network=none busybox
/ # ping google.com
ping: bad address 'google.com'
/ #
```

CUSTOM NETWORK:-

Microsoft Windows [Version 10.0.19045.4894]  
(c) Microsoft Corporation. All rights reserved.

```
C:\Users\Dell> docker network create -d bridge youtube
528b52e0aced803d36f205e367db930a4786f8c3296e5ea26832d6da0d297780
```

```
C:\Users\Dell> docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
c8a87cc9d6ca	bridge	bridge	local
b569d0c5376f	host	host	local
4a733316bd04	none	null	local
528b52e0aced	youtube	bridge	local

```
C:\Users\Dell>
```

```
C:\Users\Dell> docker run -it --network=youtube --name Tony_stark ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
dafa2b0c44d2: Already exists
Digest: sha256:dfc10878be8d8fc9c61cbff33166cb1d1fe44391539243703c72766894fa834a
Status: Downloaded newer image for ubuntu:latest
root@717f27f16e1e:/#
```

```
C:\Users\Dell>docker run -it --network=youtube --name dr_strange busybox
/ # Tony_stark
sh: Tony_stark: not found
/ # Tony_stark_stark
sh: Tony_stark_stark: not found
/ # ping Tony_stark
PING Tony_stark (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=2.087 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.204 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.323 ms
64 bytes from 172.18.0.2: seq=3 ttl=64 time=0.175 ms
64 bytes from 172.18.0.2: seq=4 ttl=64 time=0.372 ms
64 bytes from 172.18.0.2: seq=5 ttl=64 time=0.176 ms
64 bytes from 172.18.0.2: seq=6 ttl=64 time=0.082 ms
64 bytes from 172.18.0.2: seq=7 ttl=64 time=0.254 ms
64 bytes from 172.18.0.2: seq=8 ttl=64 time=0.185 ms
^C
--- Tony_stark ping statistics ---
9 packets transmitted, 9 packets received, 0% packet loss
round-trip min/avg/max = 0.082/0.428/2.087 ms
/ #
```

—> They both working on the same network and there is no need of ip address and only ping the host name

```

C:\Users\Dell> docker inspect youtube
[
 {
 "Name": "youtube",
 "Id": "528b52e0aced803d36f205e367db930a4786f8c3296e5ea26832d6da0d297780",
 "Created": "2024-09-24T07:04:00.397332422Z",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
 "Driver": "default",
 "Options": {},
 "Config": [
 {
 "Subnet": "172.18.0.0/16",
 "Gateway": "172.18.0.1"
 }
]
 },
 "Internal": false,
 "Attachable": false,
 "Ingress": false,
 "ConfigFrom": {
 "Network": ""
 },
 "ConfigOnly": false,
 "Containers": {
 "717f27f16e1ec9e6ba20416016190967c33213a3a2391257f665a9b604102bce": {
 "Name": "Tony stark",

```

# Volume Mounting

Volume mounting in Docker allows you to persist data generated by and used by containers, ensuring that data remains intact even after containers are stopped or removed. There are two main types of mounts:

### 1. **Volumes**- **Definition**: Managed by Docker, volumes are stored in a part of the host filesystem that's isolated from the container's filesystem.



- **Creation**: You can create a volume using the command:

```
```bash docker volume create <volume_name> ```
```

- **Usage**: Volumes are mounted into containers using the `-v` or `--mount` flag:

```
```bash docker run -v <volume_name>:/path/in/container <image_name> ```
```

### 2. **Bind Mounts**- **Definition**: Bind mounts link a directory on the host to a directory in the container. Changes in either the host or the container are reflected in real-time.

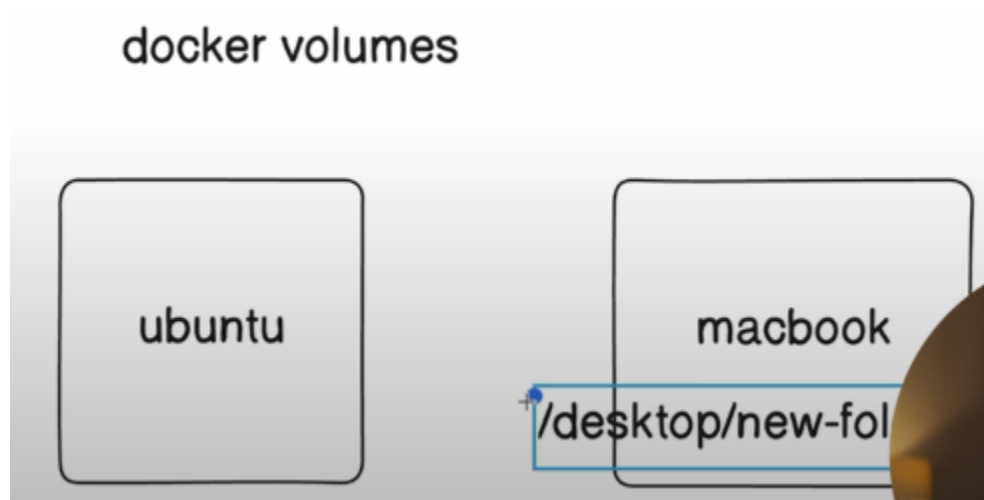
- **Usage**: You specify the host directory when running a container:

```
```bash docker run -v /host/path:/container/path <image_name> ```
```

Advantages of Volume Mounting

- **Data Persistence**: Volumes persist beyond the lifecycle of individual containers.

- **Ease of Backup and Migration**: Volumes can be easily backed up, shared, or migrated across different environments.
- **Isolation**: Using volumes keeps container filesystems clean and reduces complexity.



Efficient caching in Docker layers is a key aspect of how Docker images are built and stored. Understanding this concept can significantly speed up the development and deployment process. Here's a breakdown of how it works:

1. **Image Layers** Docker images are composed of a series of read-only layers stacked on top of each other. Each layer corresponds to a specific instruction in the Dockerfile. For example, when you run a command like `RUN`, `COPY`, or `ADD`, Docker creates a new layer.

2. **Layer Caching Mechanism** When building an image, Docker checks if it has previously built a layer that matches the current instruction. If it finds a match, it uses the cached layer instead of rebuilding it. This is where efficient caching comes into play:

- **Identical Instructions**: If the command and all its context (e.g., files copied or the base image) are the same as a previous build, Docker uses the cached layer.

- **Layer Invalidations**: If any part of the instruction changes (e.g., a file used in a `COPY` command is modified), Docker will invalidate the cache for that layer and all subsequent layers. It will rebuild them, but earlier layers will still be cached.

3. **Best Practices for Efficient Caching**

To maximize caching efficiency, consider the following best practices:

- **Order of Instructions**: Place the most stable instructions (like installing dependencies) at the top of the Dockerfile. This way, changes in application code won't affect those layers.

```
```dockerfile # Good ordering for caching FROM node:14
```

```
WORKDIR /app # Install dependencies first COPY package*.json ./ RUN npm
install # Copy application codeCOPY``
```

- **Minimize Changes**: Try to keep the files you copy into the image stable. Frequent changes to files in earlier layers can lead to cache invalidation of many subsequent layers.

- **Multi-Stage Builds**: Use multi-stage builds to reduce the size of the final image and ensure that only necessary artifacts are copied over, which can also help optimize caching.

### 4. **Example of Layer Caching in Action** Consider the following Dockerfile:

FROM ubuntu:20.04

# Install dependencies

RUN apt-get update && apt-get install -y python3

# Copy application files

COPY app.py /app/app.py

# Set the working directory

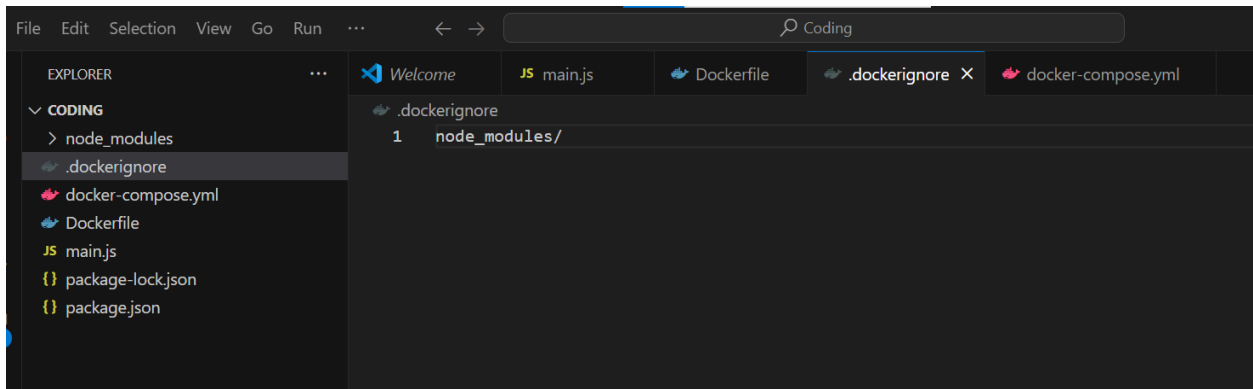
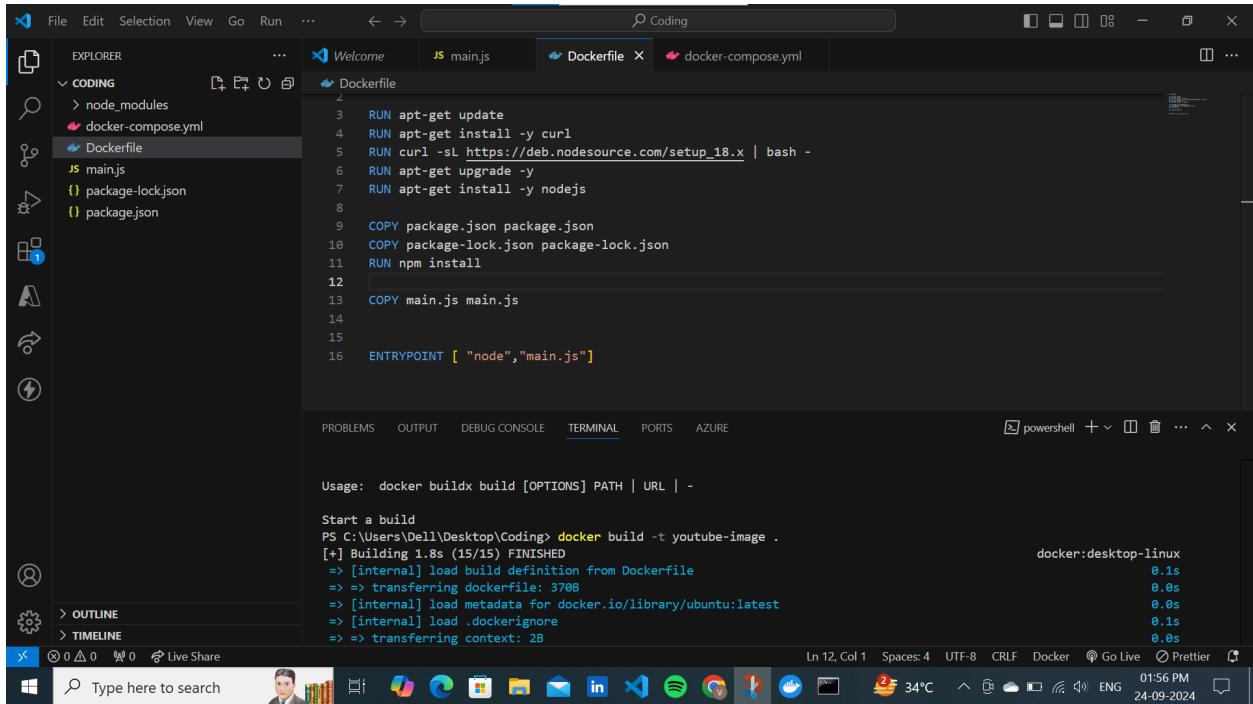
```
WORKDIR /app
```

```
Run the application
```

```
CMD ["python3", "app.py"]
```

### 5. **Conclusion** Efficient caching in Docker layers is crucial for optimizing build times and resource usage. By structuring your Dockerfile thoughtfully and leveraging layer caching, you can streamline your development workflow and create more efficient images. If you have specific scenarios or further questions in mind, feel free to ask!

Caching in docker can be achieved by taking npm install uper to main.js because whenever we change main.js we don't need to again install node js and that increases its speed.





...WelcomeJS main.jsDockerfile X.dockerignoredocker-compose.yml

Dockerfile23RUN apt-get update4RUN apt-get install -y curl5RUN curl -sL https://deb.nodesource.com/setup\_18.x | bash -6RUN apt-get upgrade -y7RUN apt-get install -y nodejs89COPY package.json package.json10COPY package-lock.json package-lock.json11RUN npm install1213COPY . .141516ENTRYPOINT [ "node","main.js"]

PROBLEMSOUTPUTDEBUG CONSOLETERMINALPORTSAZURE

```
=> exporting to image
=> => exporting layers
=> => writing image sha256:019c46be9cd0b0c3ea6cef555717bb31be5b6868634dfa98c874d916dd02276f
=> => naming to docker.io/library/youtube-image

What's next:
 View a summary of image vulnerabilities and recommendations → docker scout quickview
PS C:\Users\Dell\Desktop\Coding> docker build -t youtube-image .
[+] Building 2.0s (15/15) FINISHED
```



