Ab basically yaha hum yeh dekhaingay kay application ko as a **root user** run krnay may and **non-root user** run krnay may kya difference hai ? and it is one of important security factor in Containers.

# create a user with permissions to run the app

# -S -> create a system user

# -G -> add the user to a group

# This is done to avoid running the app as root

# If the app is run as root, any vulnerability in the app can be exploited to gain access to the host system

# It's a good practice to run the app as a non-root user

RUN addgroup app && adduser -S -G app app

Now above is the command for running app as a **non-root user**

Ab pehlay yeh smjhlein kay app ko as a **non-root user** and **root-user** run krnay may kya farak hai:

**\*\*1. Root User in Docker Container\*\*:**

- \*\*Powerful Access\*\*: Jab aap apni application ko container ke andar root user ke taur par run karte hain, uska matlab yeh hai ke us container ke andar wo application sab kuch kar sakti hai, jese system files ko modify karna, nayi files create karna, aur system settings change karna.

- \*\*Potential Risk\*\*: Agar application mein koi vulnerability hai (jaise koi security loophole), aur wo application root user ke taur par run ho rahi hai, toh koi attacker is vulnerability ka faida utha kar container ke andar full control hasil kar sakta hai. \*\*Aur agar attacker ne container ke andar root access hasil kar liya, toh uske liye container se host system tak access milna asaan ho sakta hai\*\*.

**\*\*2. Non-Root User in Docker Container\*\*:**

- \*\*Limited Access\*\*: Agar aap application ko non-root user ke taur par run karte hain, toh wo user sirf limited tasks kar sakta hai. Iska matlab yeh hai ke application container ke andar sirf wahi kaam kar sakti hai jo us non-root user ke pass permissions hain.

- \*\*Reduced Risk\*\*: Agar application mein vulnerability ho, aur attacker isko exploit karne ki koshish kare, toh uska access sirf usi non-root user ke level par limited hoga. Wo host system tak ya critical system files tak nahi pahunch sakta.

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

- \*\*Root User Scenario\*\*: Agar app ko root user ke taur par run kiya ja raha hai aur usme koi vulnerability hai, toh attacker container ke andar root access le sakta hai aur container ke boundaries tod kar host system tak pahunch sakta hai.

- \*\*Non-Root User Scenario\*\*: Agar app ko non-root user ke taur par run kiya ja raha hai aur usme koi vulnerability hai, toh attacker sirf us non-root user ke privileges tak hi limited rahega. Uske liye host system tak pahunchna ya system files ko modify karna mushkil ho jayega.

Isliye, Docker container mein application ko non-root user ke taur par run karna ek best practice hai, jo system security ko maintain karne mein madad karta hai.

Now understanding the syntax of running app as **non-root user:**

**\*\*Purpose\*\*:**

The command you're using in your Dockerfile is aimed at creating a new user and group within the Docker container to run your application securely. This is important because running applications as the `root` user inside a container can pose security risks. Running the app as a non-root user limits the potential damage that could be caused if the app is compromised.

\*\*Command Breakdown\*\*:

```Dockerfile

**RUN addgroup app && adduser -S -G app app**

```

1. \*\*`RUN`\*\*:

- This is a Dockerfile instruction that executes a command in the container during the image build process.

2. \*\*`addgroup app`\*\*:

- This command creates a new group called `app` inside the container.

3. \*\*`adduser -S -G app app`\*\*:

- \*\*`adduser`\*\*: This creates a new user.

- \*\*`-S` (System User)\*\*: This flag specifies that the user being created is a system user. System users typically have lower privileges and are used for running background processes or applications rather than logging into the system.

- \*\*`-G app`\*\*: This adds the new user to the `app` group that was created earlier.

- \*\*`app` (final argument)\*\*: This is the name of the new user being created. So, you're creating a system user named `app` and adding it to the `app` group.

**\*\*Why This is Important\*\*:**

- \*\*Security\*\*:

- \*\*Non-Root User\*\*: Running your app as a non-root user (instead of `root`) is a best practice for security. If an attacker finds a vulnerability in your app, they would only have limited access to the system through the non-root user, rather than full control.

- \*\*Preventing Escalation\*\*: Running as a non-root user prevents attackers from easily gaining access to the host system if they exploit a vulnerability in your app.

- \*\*Good Practice\*\*:

- \*\*Isolation\*\*: By creating a dedicated user and group specifically for running the app, you isolate the app’s permissions, reducing the risk of security breaches.

- \*\*Limiting Privileges\*\*: The app only has the privileges of the non-root `app` user, which are much more limited than those of the root user.

\*\*Summary\*\*:

The command `RUN addgroup app && adduser -S -G app app` is creating a non-root system user named `app` and a group named `app` within the Docker container. This user will run your application, which is a security measure to prevent potential vulnerabilities in the app from compromising the host system. Running the app as a non-root user is a best practice in Docker and containerized environments.

Q- Now its mean kay ab jo user srf maa create krunga wohi app ko access krskta hai or koi user nhi krsktay like yaha par : 'RUN addgroup app && adduser -S -G app app' maa ek system user app group ma add kr rha hun toh srf wohi app ko fully access krskta hai

* Haan, ab jo specific user (app) aap create kar rahe hain, wahi user us application ko fully access kar sakta hai. Doosre users ke pass wohi access nahi hoga, aur yeh practice aapki application aur system ki security ko badhane ke liye ek important step hai.

Your understanding is close, but there’s a key clarification needed. Let me explain:

**- \*\*System User Definition\*\*:**

- A \*\*system user\*\* is indeed a specific user account that you create within the container. It’s not a general category that applies to all users; rather, it’s a specific user you create with limited privileges, intended for running specific processes or applications.

- In your case, when you run the command `RUN addgroup app && adduser -S -G app app`, you’re creating a specific user named `app` that belongs to a group also named `app`.

**- \*\*Access and Permissions\*\*:**

- This `app` user is the only user that will run the application inside the container if you set it up that way. It’s not that every user accessing the container becomes a system user; rather, the container's application is explicitly run by this `app` user.

- The `app` user will have access to files and directories that it owns or that are accessible to the `app` group. It will not have access to files owned by other users or groups unless those files are world-readable.

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

1. \*\*Specific User (`app`)\*\*:

- The `app` user is a specific, non-root user that you have created within the container. It’s not a general system-wide user that applies to everyone.

2. \*\*Not All Users Become `app`\*\*:

- Other users who might access the container (if allowed) do not automatically become the `app` user. They retain their own identities and permissions.

3. \*\*Access Control\*\*:

- The `app` user’s access is limited to the files and directories that it owns or that are accessible to the `app` group. This limits what the `app` user can do within the container, enhancing security.

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

- \*\*Correct Understanding\*\*: The `app` user you created is a specific, limited-privilege user within the container. It’s not a general user for everyone; rather, it’s a dedicated user meant to run your application securely. Only this `app` user (or processes running as `app`) will have the permissions and access you've defined.

- \*\*Incorrect Assumption\*\*: Not every user accessing the container becomes a system user or part of the `app` group. Other users retain their own identities and permissions.

--------------------------------------------------------------------------------------------------------------

# set the user to run the app

**USER app**

* Ab in the most first explanation humnay yeh toh smjh lia kay User kasay create krna hai or uska maqsad kya hai , but as we know by default docker run app as **root system,** so usko non-root system pa convert krnay kay liya humay user set krwana hoga using **USER** command , now in above command yehi kia hai kay **USER** command ka through Docker ko user set krdia and **app** is the name of user jo humnay create kia hai.
* Ab jab humna user set krdia toh its mean kay docker nay ab root system say non-root system pa switch krlia and ab jitni be command container ma run hongi wo as a **app** user.

Q- ok But muja yaha yeh smjh nhi aya kay chlo thk hai humna uper 'app' user create krliya and usko set be krdia using USER command but ab jasay let suppose humara container run ho rha hai different user will come on it, Ab Docker ko kesay pta chlega kay yeh 'app' user hi hai ?

* So in short iska answer yeh haka, basically jo hum yeh user create krtay hain iska mtlb yeh nhi hota kay jab be koi user Docker container ko access krega toh wo pehlay usko authenticate krega then access dega, Nhi ! basic yeh saray kam from the creation of User to the setting of User using USER command iss lia kia ja rha hai taka Docker jo by default app ko as **root system** run krta hai , wo ussay switch krka as a **non-root system** app ko run kray.
* Or iss tarah jab **non-root system** pa app run hogi toh jo be user ayega wo jab docker container ko access krega toh uskay pss limited access hogi bcuz humari jo app hai wo as **non-root system** run ho rhi hai.

**More explanation**

Exactly! You've got it right. Let's summarize and clarify:

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

1. \*\*Switching from Root to Non-Root\*\*:

- By default, Docker containers run as the `root` user, which has full access to everything inside the container. This is why we use the `USER` command in the Dockerfile to switch from running as `root` to running as a specific non-root user (e.g., `app`).

- When you specify `USER app`, Docker switches the execution context from the powerful `root` user to the more restricted `app` user. This means any commands, processes, or applications inside the container will now run with the limited permissions of the `app` user.

2. \*\*Limited Access\*\*:

- The purpose of switching to a non-root user is to limit the permissions and capabilities of the processes running inside the container. This helps enhance security by reducing the risk that an attacker could exploit vulnerabilities to gain full control over the container or the host system.

- Since the container runs as a non-root user (`app`), it won’t have the ability to perform actions that require root privileges, like modifying system-wide files, installing new software, or changing critical configurations.

3. \*\*No Authentication Mechanism\*\*:

- Docker does not implement any authentication mechanism for switching users within the container. When you use the `USER` command, Docker simply runs all processes under the specified user (`app`), but it doesn’t require any login or authentication.

- This means that once the container starts, everything inside runs as `app` automatically, and there’s no user switching or authentication involved. Docker ensures that the environment operates under the permissions and restrictions of the `app` user, but it doesn’t authenticate users in the traditional sense.

# copy package.json and package-lock.json to the working directory

# This is done before copying the rest of the files to take advantage of Docker’s cache

# If the package.json and package-lock.json files haven’t changed, Docker will use the cached dependencies

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

Certainly! Let's break down the command and the explanation step by step.

**`COPY package\*.json ./` Command:**

- \*\*`COPY`\*\*: This is a Dockerfile instruction used to copy files or directories from your local system into the Docker image.

- \*\*`package\*.json`\*\*:

- The `package\*.json` is a pattern that matches both `package.json` and `package-lock.json` files.

- The `\*` (asterisk) is a wildcard character that allows matching multiple files. So, `package\*.json` will match any file that starts with `package` and ends with `.json`, which in this case includes `package.json` and `package-lock.json`.

- \*\*`./`\*\*:

- This represents the current working directory inside the Docker image, which is typically set by a previous `WORKDIR` command.

- The `./` means "copy the files into the current directory" inside the Docker image.

**\*\*Purpose of This Command\*\*:**

1. \*\*Caching Optimization\*\*:

- \*\*Docker Cache\*\*: Docker uses a layer caching mechanism, where each instruction in the Dockerfile creates a new layer. If a layer hasn't changed between builds, Docker can reuse the cached layer instead of rebuilding it, which speeds up the build process.

- \*\*Copying `package.json` First\*\*: By copying `package.json` and `package-lock.json` first, Docker can take advantage of this cache. These files define the dependencies for your Node.js application.

2. \*\*Efficient Dependency Management\*\*:

- \*\*If Dependencies Haven’t Changed\*\*: When Docker sees that `package.json` and `package-lock.json` haven’t changed since the last build, it can skip reinstalling the dependencies (usually done by a `RUN npm install` command) and use the cached layer where the dependencies are already installed.

- \*\*Copying Rest of the Files Later\*\*: After copying `package.json` and `package-lock.json`, you typically copy the rest of the application’s files. Since these other files are copied after the dependencies are installed, changes to them won’t invalidate the cached dependencies, making your builds faster.

**\*\*Example in Context\*\*:**

# Set the working directory

WORKDIR /app

# Copy the package.json and package-lock.json files first

COPY package\*.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application files

**COPY . .**

# Command to run the app

CMD ["node", "main.js"]

**- \*\*Explanation\*\*:**

- \*\*Step 1\*\*: Docker copies `package.json` and `package-lock.json` into the `/app` directory inside the image.

- \*\*Step 2\*\*: Docker runs `npm install` to install the dependencies. If `package.json` and `package-lock.json` haven’t changed, Docker uses the cached layer where dependencies are already installed.

- \*\*Step 3\*\*: Finally, Docker copies the rest of the files. Since this happens after dependencies are installed, changes to application files don’t trigger a reinstall of dependencies.

Summary:

Ab basic package.json and package-lock.json ko pehla copy krnay ka purpose yeh tha kay wo dono Docker cache ma store hojaye and on the next lines jab **npm install** ki command RUN ho toh as we know kay it is used to create **package.json and install dependencies** toh at that time toh wo **COPY package\*.json ./** layer hai usme check kray agar same package.json and package-lock.json ho toh wo new files na create kray blka unhi ko use krlay .

# sometimes the ownership of the files in the working directory is changed to root

# and thus the app can't access the files and throws an error -> EACCES: permission denied

# to avoid this, change the ownership of the files to the root user

**USER root**

# change the ownership of the /app directory to the app user

# chown -R <user>:<group> <directory>

# chown command changes the user and/or group ownership of for given file.

**RUN chown -R app:app .**

# change the user back to the app user

**USER app**

Now in the above code-snippet you are looking that pehla uper wali commands may humna USER set kia hai **app** for switching from root to non-root system , but yaha ab hum again **root** system pa user ko set kr rhay hain toh uski kya waja hai ? toh uski waja yeh haka as you have in above COPY command where we are copying **package\*.json** files ab usme bht sari files hongi jo srf root user access krskta hai , toh issi lia after copying those file humna USER ko again **root** pa set kia or then phr root system may hi rehty huye using this command **RUN chown -R app:app .**  humnay yeh specify krdia kay jitni be files **/app** directory may hain mtlb kay **root** **system** pa hain unki jo access hai wo **app** user and **app** group ko be mil jaye. Takay at the time of run time jab **app** user kisi be **root system** file ko access krnay chahy toh easily hojaye without any ownership error. Or phr after changing ownership of all file to **app** user , then back root system say non-system pay switch krdia by using **USER app** .

Understanding more about this command **RUN chown -R app:app . :**

Certainly! Let's break down the command `RUN chown -R app:app .` and explain what it does:

**- \*\*`RUN`\*\*:**

- This is a Dockerfile instruction used to execute a command in the container's environment during the build process. The result of this command is then saved in the Docker image.

**- \*\*`chown`\*\*:** `chown` stands for "change owner." It's a Unix/Linux command used to change the ownership of files or directories. Ownership includes both the user and the group that owns the file or directory

**- \*\*`-R`\*\*:**

- This flag stands for "recursive." It means that the command should apply not just to the specified directory, but also to all of its contents—subdirectories and files—recursively.

**- \*\*`app:app`\*\*:**

- The first `app` refers to the \*\*user\*\* to whom ownership of the files and directories should be given.

- The second `app` refers to the \*\*group\*\* to whom ownership should also be assigned.

- So, `app:app` means that both the user and group ownership will be changed to `app`.

**- \*\*`.` (dot)\*\*:**

- The dot (`.`) represents the current directory. This means the command will apply to the current directory and all its contents.

**\*\*Putting It All Together\*\*:**

- \*\*`RUN chown -R app:app .`\*\*:

- This command changes the ownership of the current directory and all its subdirectories and files to the `app` user and the `app` group.

- \*\*`-R`\*\* ensures that the ownership change is applied recursively to everything within the current directory.

\*\*Why This Is Important\*\*:

- \*\*Ensuring Correct Permissions\*\*:

- When you copy files into the Docker image, they might be owned by `root` by default. If your application is supposed to run as a non-root user (like `app`), those files need to be accessible to that user.

Q- kya hum isme yeh be toh krsktay thy kay pehla USER as app ko set naa krtay then after copying the package\*.json file hum ownership change krkay to app user then finally USER app set krdetay ?

* Yes exactly we can do this as well .

# install dependencies

**RUN npm install**

Now it will simply install node-modules and dependencies , also remember jo dependencies package.json ma already hongi wo nhi install hongi bcuz Docker have already cached them.

Q- Ab main question yeh haka jasa uper humnay yeh sikha kay kuch files package.json may asi be hongi that requires root system access, toh agar usi understanding ka A/c sochay toh phr jab npm install ka through node-modules and dependencies install hongi toh at that time possible hai kay some file require root system?

Your thinking is correct to some extent, but there are important nuances to consider. Let’s clarify this:

When you run `npm install` to install Node.js dependencies, the following points are relevant:

**1. \*\*Default Behavior of `npm install`:\*\***

- By default, `npm install` does not require root privileges to install packages in the working directory, such as `/app/node\_modules`. It installs dependencies in the local project directory (e.g., `/app/node\_modules`), which the current user (in your case, the `app` user) should have permission to write to.

- If the `app` user has the correct ownership of the working directory and its subdirectories (like `/app`), `npm install` should proceed without any issues, even if run as a non-root user.

**2. \*\*Root Privileges Requirement:\*\***

- \*\*When Root Might Be Needed:\*\* Root privileges are generally required when you try to install packages globally (e.g., using `npm install -g`) because this affects system-wide directories like `/usr/local/lib`. However, in the context of a Docker container where dependencies are installed locally within the project directory, root access is typically not necessary.

- \*\*Permissions Issues:\*\* If there are files or directories in the working directory that the `app` user does not have write access to, `npm install` might fail. This is why it's important to ensure that the `app` user has ownership of the relevant directories before running `npm install`.

**3. \*\*Best Practice:\*\***

- The common best practice in Docker containers is to set the appropriate file ownership (using `chown`) for the application user (e.g., `app`) before running commands like `npm install`. This avoids the need for root access and adheres to the principle of least privilege.

\*\*Conclusion:\*\*

In short , iska answer yeh haka actually we have already change the ownership of working directory to **app** user so iss liya jab basically hum just **npm Install** run krtay hain toh wo by default jo current working directory hai usma install krta hai sari files wagera ko and in our case it is **app** , therefore now we will not face any error bcuz jo **app** user hai it have the permission to access all the files in the **app** directory. But agr hum **npm -g install** iss command ko run krtay toh phr npm sari files ko globally jo local machine ki files hai uspa install krta that can become a issue.

# copy the rest of the files to the working directory

**COPY . .**

This command simply says that jo current directory may Docker file create hui hai usi directory my jitni be files hain sb ko copy krlo and then usko jo **app** working directory hai usme paste krdo. So as we have created Dockerfile in **src** directory in project so usme jitni be files hai wo copy krlega .

# expose port 5173 to tell Docker that the container listens on the specified network ports at runtime

**EXPOSE 5173**

This command is telling that kay jo be Container create hoga using this Docker Image jo **5173** network port pa hi run hoga bs.

Q- Ab question yeh haka yeh jo network port expose krna hai yeh apni marzi say krna hai ya iska what is the proper way of doing this ?

Exposing network ports in Docker is an important aspect of containerized applications, especially when your application needs to be accessible from outside the container. However, it should be done thoughtfully and securely. Here’s how to approach this:

**\*\*1. Purpose of Exposing Ports:\*\***

- \*\*Exposing Ports\*\*: When you expose a port in a Docker container, you are making that port accessible from outside the container. This is typically necessary when your containerized application (e.g., a web server, database, or API) needs to interact with the outside world, such as serving web pages or accepting connections from other services.

**\*\*2. Proper Way to Expose Ports:\*\***

- \*\*Identify Required Ports\*\*: Only expose the ports that are necessary for your application to function. For example, if you’re running a web server, you might expose port `80` for HTTP or port `443` for HTTPS. Avoid exposing unnecessary ports as it can increase security risks.

- \*\*Use `EXPOSE` in Dockerfile\*\*:

- The `EXPOSE` instruction in a Dockerfile is used to inform Docker that the container listens on the specified network ports at runtime. However, using `EXPOSE` alone does not publish the port to the host machine. It’s mainly documentation for anyone using the image.

- Example:

**EXPOSE 80**

- \*\*Map Ports During Runtime\*\*: To actually make a port accessible from outside the container, you need to map the container’s port to a port on the host machine using the `-p` flag in the `docker run` command.

- Example:

**docker run -p 8080:80 my-container**

- In this example, port `80` inside the container is mapped to port `8080` on the host machine, making the application accessible at `http://localhost:8080`.

**\*\*3. Security Considerations:\*\***

- \*\*Limit Exposure\*\*: Only expose and map the ports that are absolutely necessary. Exposing too many ports increases the attack surface and can lead to security vulnerabilities.

- \*\*Use Firewalls and Network Policies\*\*: Implement firewalls or Docker’s built-in network policies to control which external systems can access the exposed ports.

**4. Summary:**

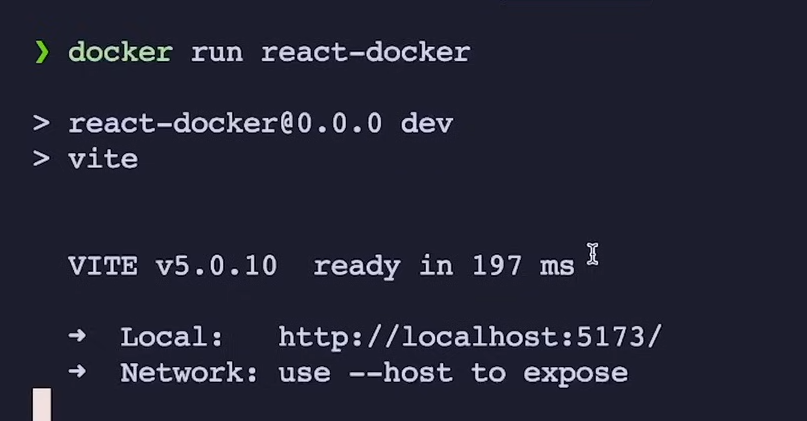
- \*\*Best Practice\*\*: Expose only the necessary ports that your application needs to communicate with the outside world. Document them using the `EXPOSE` instruction in the Dockerfile for clarity, but control actual exposure through port mapping (`-p` flag) at runtime.

- \*\*Security\*\*: Always be mindful of security risks when exposing ports, and ensure that unnecessary ports are not left open.

# command to run the app

**CMD npm run dev**

This command will simply execute when the container will run.



Now you can see that after running the app on terminal , it also provide localhost link. But if we click on it so we will not navigate to it. So why this happening ?

* Iski waja yeh haka yeh toh thk hai kay humnay container may port ko expose krdia and Docker also know kay humari application **5173** port pa run hogi but jo humari host machine/server hai it don’t know kay kis port pay our container will run kiu kay container ek isolated environment ma work krta hai isi lia it don’t expose its port , so we have to explicitly tell the host machine kay container kis port pa work krega.

You're absolutely on the right track with your explanation! Let me clarify and expand on it:

**\*\*Why Can't You Access the Application via Localhost?\*\***

**- \*\*Container Isolation\*\*:**

- Docker containers run in an isolated environment, which means that the network inside the container is separate from the host machine's network. The application inside the container might be running on port `5173`, but this port is not automatically accessible from the host machine because of this isolation.

**- \*\*Port Mapping\*\*:**

- Even though the application is running on `localhost:5173` inside the container, the host machine doesn’t know about it unless you explicitly map the container’s port to a port on the host machine.

- Without this mapping, trying to access `http://localhost:5173/` from your browser on the host machine won’t work because the host machine doesn’t know to forward traffic on port `5173` to the container.

**\*\*Solution: Port Mapping with `-p` Flag\*\***

To make the application accessible from your host machine, you need to map the container’s port to a port on the host. This is done using the `-p` flag when running the Docker container.

\*\*How to Map Ports\*\*:

```

**docker run -p 5173:5173 react-docker**

```

- \*\*`-p 5173:5173`\*\*: This maps port `5173` on your host machine to port `5173` in the container.

- The first `5173` is the port on your \*\*host machine\*\*.

- The second `5173` is the port inside the \*\*Docker container\*\* where your application is running.

**\*\*What Happens After Port Mapping?\*\***

- \*\*Host to Container Traffic\*\*: With this mapping, when you visit `http://localhost:5173/` on your host machine, Docker knows to forward this request to port `5173` inside the container, allowing you to access your application.

**Summary:**

Ab basic yeh hua yeh kay jab humnay port mapping ki using **-p** flag toh in simple terms humnay host machine ko yeh btadia kay jasay hi [**http://localhost:5173**](http://localhost:5173)iss URL say koi request ayegi toh port **5173** pa jo container hai usko request forward hojayegi, then Docker will receive the request and request to Container on 5173 port then Container will return the application running inside it as response.

**What is port Mapping in Docker in detail ?**

\*\*Port Mapping\*\* is a way to make an application running inside a Docker container accessible from outside the container (e.g., from your local machine or the internet).

**\*\*How It Works\*\*:**

- \*\*Inside the Container\*\*: Your application runs on a specific port, like `5173`.

- \*\*Outside the Container\*\*: By default, this port is not accessible from your local machine or the outside world because the container is isolated.

**\*\*Port Mapping with `-p`\*\*:**

- When you run a container, you use the `-p` flag to map a port on your local machine (host) to a port inside the container.

- Example: `-p 8080:80`

- This maps port `80` inside the container to port `8080` on your local machine.

- Now, if your application is running on port `80` inside the container, you can access it via `http://localhost:8080` on your local machine.

**\*\*Why It’s Important\*\*:**

- \*\*Access Your App\*\*: Port mapping allows you to interact with your application from outside the container, making it accessible for development, testing, or production use.

**\*\*Simple Analogy\*\*:**

Think of port mapping as connecting a specific "door" on your local machine to a "door" inside the container. When you knock on the door on your local machine, it automatically opens the corresponding door inside the container, allowing you to interact with the application running th

In summary, \*\*port mapping\*\* connects ports from your Docker container to your host machine, making containerized applications accessible from outside the container.

ere.

Q- it means that A/c to the above example which is : -p 8080:80 when we do port mapping means to tell the host machine that when anyone try to access http://localhost:8080 this URL you should send the request to container running at port 80 ?

**\*\*Explanation Based on Your Example (`-p 8080:80`)\*\*:**

- \*\*Host Machine\*\*: When someone tries to access `http://localhost:8080` on your host machine (your computer), the host machine receives the request on port `8080`.

- \*\*Port Mapping\*\*: The `-p 8080:80` option tells Docker to map port `8080` on the host machine to port `80` inside the Docker container.

- \*\*Request Forwarding\*\*: When the host machine receives a request on `http://localhost:8080`, it forwards that request to the Docker container's port `80`.

- \*\*Container\*\*: Inside the container, your application is listening on port `80`, so it receives the request and processes it.

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

- Yes, you're correct! Port mapping like `-p 8080:80` instructs the host machine to forward requests from `http://localhost:8080` to the container's port `80`, where the application is running.

Now as you have experience that like we have created one container , now let suppose I am continuously making changes in app, ab agar un changes ko container may reflect krna hai toh its mean kay dubara say ek Docker image build krni hogi, Now agar ek do dafa change krna ho app toh smjh ata hai bnda dubara say image create krlay , But agar frequent changes ho rhay hain app may and you to automate this process kay jo changes maa local system may app pay krunga wo automatically reflect hotay jayen container may be .

So to achieve this purpose we can use **Docker Compose** as well which is latest tool, but right now we will learning how to automate this process manually.

\*\*Command\*\*:

**Docker run -p 5173:5173 -v “${pwd):/app” -v /app/node\_modules react-docker**

\*\*1. `docker run`\*\*:

- \*\*What It Does\*\*: Starts a new container from the `react-docker` image.

\*\*2. `-p 5173:5173`\*\*:

- \*\*Port Mapping\*\*: This maps port `5173` on your host machine (your computer) to port `5173` inside the Docker container.

- \*\*Purpose\*\*: This allows you to access the application running inside the container on port `5173` by going to `http://localhost:5173` on your host machine.

\*\*3. `-v "$(pwd):/app"`\*\*:

- \*\*Volume Mounting (First One)\*\*:

- \*\*`-v "$(pwd):/app"`\*\*: This mounts the current directory on your host machine (where you run the command) to the `/app` directory inside the container.

- \*\*Purpose\*\*: This allows you to share your local files with the container. Any changes you make to the files on your host machine will be reflected inside the container, and vice versa.

\*\*4. `-v /app/node\_modules`\*\*:

- \*\*Volume Mounting (Second One)\*\*:

- \*\*`-v /app/node\_modules`\*\*: This is an anonymous volume that mounts the `node\_modules` directory inside the container.

- \*\*Purpose\*\*: It prevents the `node\_modules` directory from being overwritten by the empty directory in your host machine (since the previous `-v` command syncs the entire `/app` directory). This ensures that the container uses the `node\_modules` installed inside the container, not from your host machine.

\*\*5. `react-docker`\*\*:

- \*\*Image Name\*\*: This is the name of the Docker image you’re running. The container will start from this image.

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

Ab basic first we have mounted volume (it’s a process jokay humari jo host machine pay working directory hoti hai usko link krwata hai container kay andar jo WorkDir hoti hai) , Ab yaha be yehi hua kay when we put this command **-v “$(pwd): /app”** so basically it mounts our present working directory (pwd) into the **/app** WORKDIR in container. And now its allow kay jo be changes present working directory ma hongay on local machine wohi container mabi reflect kreingay. Or inkay darmayain jo syncing that is done by **Volume**. Or basically jo be changes hongay wo Volume may persist rahegay takay agar in future container crash hojaye toh jo Volume ma data hai usko krka **react-docker** image ka through new container ban jayega. This is also known as **Bind Mount.** Bcuz we are binding local machine file with container working directory through Volume.

Now what is the purpose creating this volume **-v /app/node\_modules** ?

its mean -v /app/node\_modules by doing this we have created another volume jismay saray node modules hain or yeh iss lia kia hai taka jo current working directory mount hogi from local machine to container toh usmay be ek node\_modules ki file hogi but wo empty hogi , toh iss lia kahin wo empty file overwrite naa krday node\_modules ki file ko jo container may hai. toh isi lia ek alag say volume create krliya container jisme /app/node\_moudles may jo node modules hain wo store krliya and yeh ensure krtay hain kay ab jo node\_modules directory hai inside container it will be protected now.

**\*\*Volume Mounting with `-v /app/node\_modules`\*\*:**

1. \*\*Mounting the Current Working Directory\*\*:

- When you use `-v "$(pwd):/app"`, you are mounting your local working directory (where your source code is) to the `/app` directory inside the container.

- This means that everything in your local directory, including a potentially empty `node\_modules` directory, will be mirrored inside the container.

2. \*\*The Problem\*\*:

- If your local directory contains an empty `node\_modules` folder, it could overwrite the `node\_modules` directory inside the container when the volume is mounted.

- This would be an issue because the container’s `node\_modules` directory contains all the installed dependencies required for your application to run.

3. \*\*Solution with `-v /app/node\_modules`\*\*:

- By using `-v /app/node\_modules`, you are creating a separate volume specifically for the `node\_modules` directory inside the container.

- This volume is independent of the local directory that is being mounted. It ensures that the `node\_modules` directory inside the container is preserved and not overwritten by the empty `node\_modules` directory from your local machine.

\*\*Summary\*\*:

- This ensures that the dependencies inside the container remain intact and are not accidentally removed when the local directory is mounted to `/app` inside the container.

**What Volume Mounting ?**

\*\*Volume Mounting\*\* refers to the process of making a directory or file on your host machine (your local computer) accessible within a Docker container. This is done by linking a directory or file from your host system to a directory or file inside the container.

**\*\*Key Concepts of Volume Mounting\*\*:**

1. \*\*Host Directory to Container Directory\*\*:

- When you mount a volume, you specify a directory on your host machine that you want to share with a directory inside the container.

- This allows files and data to be shared between the host and the container. Changes made in the mounted directory on the host machine will be reflected inside the container, and vice versa.

2. \*\*Two Types of Mounts\*\*:

- \*\*Bind Mounts\*\*: These are specific paths on your host machine that are mounted into the container. For example, `-v /path/on/host:/path/in/container`.

- \*\*Named Volumes\*\*: These are Docker-managed volumes that are stored in a specific location on your host machine, but you don’t usually specify the exact path. Docker manages these volumes, and they are typically used for data persistence.

3. \*\*Why Volume Mounting is Useful\*\*:

- \*\*Data Persistence\*\*: Volumes allow data to persist beyond the lifecycle of a container. When a container is deleted, the data in a volume can still be accessed.

- \*\*Development\*\*: In development, volume mounting is used to sync code between your host machine and the container, so you can see changes live without rebuilding the image.

- \*\*Configuration Sharing\*\*: Volumes can be used to share configuration files or other data between the host and container.

### \*\*Example of Volume Mounting\*\*:

```bash

docker run -v /path/on/host:/path/in/container my-image

```

-v /path/on/host:/path/in/container:

- \*\*`/path/on/host`\*\*: This is a directory on your host machine.

- \*\*`/path/in/container`\*\*: This is the directory inside the container where the host directory will be mounted.

- The container will see the contents of `/path/on/host` inside `/path/in/container`, and any changes will be synchronized.

\*\*Use Case Example\*\*:

- \*\*Web Development\*\*:

- You’re working on a website project stored at `/myproject` on your host machine.

- You run a Docker container for your web server, and you want the server to use your local files directly.

- You use volume mounting: `docker run -v /myproject:/var/www/html my-web-server`.

- Now, any changes you make to the files in `/myproject` on your host are immediately reflected inside the container at `/var/www/html`, and the web server will serve the latest content without needing to rebuild the Docker image.

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

- \*\*Volume Mounting\*\* is the process of linking directories or files between your host machine and a Docker container.

- It allows for data sharing and synchronization between the host and container, making it ideal for persistent storage, live development, and configuration sharing.

- The mounted directory on the host is directly accessible inside the container, allowing for real-time interaction between the two environments.