🚀 Single Stage Dockerfile

```
FROM node:14
#Set the working directory inside the container
WORKDIR /app
#Copy package.json and package-lock.json to the working directory
COPY package*.json ./
#Install the dependencies
RUN npm install
#Copy the rest of the application code to the working directory
COPY . .
#Expose the port the app runs on
EXPOSE 3000
#Command to run the application
CMD ["npm", "start"]
```

🌟 Multi-Stage Dockerfile

Stage 1: Build

```

FROM node:14 AS builder

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build
```

Stage 2: Run

```

FROM node:14

WORKDIR /app

COPY --from=builder /app .

EXPOSE 3000
CMD ["npm", "start"]
```

📄 Explanation and Reasoning

Stage 1: Build

```
- `FROM node:14 AS builder`: Uses the Node.js image and names this stage `builder`
- `WORKDIR /app`: Sets the working directory.
- `COPY package*.json .`: Copies `package.json` and `package-lock.json`.
- `RUN npm install`: Installs dependencies.
- `COPY . .`: Copies the rest of the code.
- `RUN npm run build`: Builds the application.
```

Stage 2: Run

```
- `FROM node:14`: Uses the Node.js image again for the runtime environment.
- `WORKDIR /app`: Sets the working directory.
- `COPY --from=builder /app .`
  Copies the built application from the builder stage.
- `EXPOSE 3000`: Exposes port 3000.
- `CMD ["npm", "start"]`: Runs the application.
```

💻 Commands on EC2 for Node.js

Here's a breakdown of your commands:

1. `cd SeedUI`: Change directory to SeedUI.
2. `sudo su`: Switch to the root user.
3. `pm2 stop seedFE`: Stop the `seedFE` process managed by PM2.
4. `git checkout dev/main`: Switch to the `dev/main` branch.
5. `git pull origin dev/main`: Pull the latest changes from the remote `dev/main` branch.
6. `git clean -fxd`: Remove untracked files and directories.
7. `npm install`: Install the dependencies.
8. `npm run build`: Build the application.
9. `pm2 restart seedFE`: Restart the `seedFE` process.

📉 Why Multi-Stage Builds Reduce Image Size

Separation of Build and Runtime Environments

- Build Stage: This stage includes all the tools and dependencies needed to build the application. This often includes compilers, build tools, and other development dependencies that are not needed at runtime.
- Runtime Stage: This stage only includes the files and dependencies required to run the application. It does not include any of the build tools or development dependencies.

Copying Only Necessary Files

In the multi-stage build, after the application is built in the first stage (`builder`), only the necessary files (the built application) are copied to the second stage (runtime environment). This means that any temporary files, build caches, and development dependencies are not included in the final image.

Stage 1: Build

```
- Base Image: `node:14` (includes Node.js and npm)
- Dependencies: Installed via `npm install`
- Build Tools: Used during `npm run build`
- Application Code: Copied and built
```

Stage 2: Run

```
- Base Image: `node:14` (includes Node.js and npm)
- Built Application: Copied from the builder stage
- Excluded: Build tools, development dependencies and
any temporary files created during the build process
```

Benefits of Multi-Stage Builds

- Smaller Image Size: By excluding unnecessary files and dependencies, the final image is significantly smaller. This reduces storage costs and improves deployment times.
- Improved Security: A smaller image with fewer components reduces the attack surface, making the application more secure.
- Efficiency: Smaller images are faster to transfer over the network and quicker to start up, leading to more efficient deployments and scaling.

🔍 Difference Between RUN and CMD Commands

RUN Command

Purpose: The `RUN` command is used to execute commands in the container during the image build process. Each `RUN` command creates a new layer in the Docker image.

Example:

```
Code Generated by Sidekick is for learning and experimentation purposes only.
RUN npm install && npm cache clean --force
RUN npm run build
```

Key Points: - `RUN` commands are executed at build time. - They are used to install software packages, set up the environment, and perform any necessary build steps. - The results of `RUN` commands are saved in the Docker image as layers.

CMD Command

Purpose: The `CMD` command specifies the default command to run when a container is started from the image. Unlike `RUN`, it does not execute at build time but at runtime when the container is launched.

Example:

```
Code Generated by Sidekick is for learning and experimentation purposes only.
CMD ["npm", "start"]
```

Key Points: - `CMD` commands are executed at runtime. - They define the default behavior of the container when it starts. - Only one `CMD` instruction is allowed in a Dockerfile. If multiple `CMD` instructions are specified, only the last one will be used. - `CMD` can be overridden by specifying a different command when running the container (`docker run <image> <command>`).

🌐 Concepts of EXPOSE and PUBLISH

EXPOSE Command

Purpose: The `EXPOSE` command in a Dockerfile is used to inform Docker that the container will listen on the specified network ports at runtime. It does not actually publish the port to the host machine; it just serves as documentation and metadata.

Usage in Dockerfile:

```
EXPOSE 3000
```

Key Points: - Informational: It tells Docker and anyone reading the Dockerfile which ports the application inside the container will use. - No Port Binding: It does not bind the port to the host machine. It only makes the port available to other containers in the same Docker network. - Multiple Ports: You can specify multiple ports by using multiple `EXPOSE` commands.

When to Use: - Use `EXPOSE` when you want to document which ports your application uses. - It is useful for internal communication between containers in the same Docker network.

PUBLISH Command (or `-p` flag)

Purpose: The `PUBLISH` command (or the `-p` flag in `docker run`) is used to map a port on the host machine to a port in the container. This makes the application accessible from outside the Docker host.

Usage in `docker run` :

```
docker run -p 8080:3000 my-node-app
```

Key Points: - Port Binding: It binds a port on the host machine (e.g., 8080) to a port in the container (e.g., 3000). - External Access: It allows access to the containerized application from outside the Docker host. - Syntax: The syntax is `-p <host_port>:<container_port>` . You can also specify the IP address to bind to, e.g., `-p 127.0.0.1:8080:3000` .

When to Use: - Use `PUBLISH` when you need to expose your containerized application to external clients or users. - It is essential for making your application accessible from outside the Docker host, such as from a web browser or an API client.

🛠️ Docker Commands

Dockerfile Creation for Node.js App

Let's break down the commands `docker build`, `docker pull`, `docker push`, and how they relate to each other in a simple and easy-to-understand way.

`docker build`

Purpose: The `docker build` command is used to create a Docker image from a Dockerfile and a "context" (a set of files at a specified location).

Basic Usage:

```
docker build -t my-image:tag .
```

Explanation: - `docker build`: The command to build a Docker image. - `-t my-image:tag`: Tags the image with a name (`my-image`) and a tag (`tag`). If you don't specify a tag, it defaults to `latest`. - `.`: The build context, usually the current directory, containing the Dockerfile and other necessary files.

`docker pull`

Purpose: The `docker pull` command is used to download a Docker image from a Docker registry (like Docker Hub) to your local machine.

Basic Usage:

```
docker pull my-image:tag
```

Explanation: - `docker pull`: The command to pull a Docker image. - `my-image:tag`: Specifies the name and tag of the image to pull. If you don't specify a tag, it defaults to `latest`.

`docker push`

Purpose: The `docker push` command is used to upload a Docker image from your local machine to a Docker registry.

Basic Usage:

```
docker push my-image:tag
```

Explanation: - `docker push`: The command to push a Docker image. - `my-image:tag`: Specifies the name and tag of the image to push.

How They Work Together

1. Building an Image (`docker build`):
    - You create a Dockerfile and use `docker build` to create an image from it.
    - Example:

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

This builds an image named `my-node-app` with the tag `1.0`.

2. Pulling an Image (`docker pull`):
    - You can download an existing image from a Docker registry to your local machine.
    - Example:

```
docker pull node:14
```

This pulls the official Node.js image with the tag `14` from Docker Hub.

3. Pushing an Image (`docker push`):
    - After building an image, you can upload it to a Docker registry.
    - Example:

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

This pushes the `my-node-app` image with the tag `1.0` to the configured Docker registry.

Related Combinations

1. Build and Push:
    - Build an image and then push it to a registry.
    - Example:

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

2. Pull and Run:
    - Pull an image from a registry and then run a container from it.
    - Example:

```
docker pull node:14
docker run -it node:14
```

3. Build with Pull:

- Build an image and ensure the base image is up-to-date by pulling it first.
- Example:

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

The `--pull` flag ensures that Docker pulls the latest version of the base image before building.

Summary

```
- `docker build`: Creates a Docker image from a Dockerfile.
- `docker pull`: Downloads a Docker image from a registry.
- `docker push`: Uploads a Docker image to a registry.
```