**Multi-Stage Build**

Multi-stage builds in Docker allow you to optimize the size and efficiency of your Docker images by using multiple stages to build an application, where each stage can have its own base image and environment. This approach is particularly useful for reducing the final image size by copying only the necessary artifacts from one stage to another, rather than including everything in the final image.

**Key Concepts of Multi-Stage Builds:**

1. **Multiple FROM Statements:**
   * Each stage of the build starts with a FROM statement. You can have as many FROM statements as you need. Each one represents a separate stage in the build process.
2. **Intermediate Stages:**
   * Intermediate stages are used to compile or build the application. These stages can include all the necessary build tools, dependencies, and large files needed for the build process. However, these are not included in the final image.
3. **Final Stage:**
   * The final stage is the one that creates the image that will be deployed. In this stage, you only include the necessary files, libraries, and binaries needed to run the application, excluding everything used during the build process.
4. **Copying Artifacts:**
   * You can copy files from one stage to another using the COPY --from=<stage> directive. This allows you to transfer only the required build artifacts from an earlier stage to the final stage.
5. **Reduced Image Size:**
   * By using multi-stage builds, you can significantly reduce the size of the final Docker image. This results in faster deployments and reduced storage requirements.

**Example of a Multi-Stage Build:**

# Stage 1: Build the application

FROM node:16 AS build //now jo **AS** keyword iskay through hum build ko name de rahay hain like here we set its name as **build** Ab later on agar iss build ko file may call krna hoga toh hum iskay name say call kreinga like here we are using it in **Stage2** COPY command. Or agar name nhi dengay toh phr by default jo **stage1** pay build create hogi wo **0** say consider hogi then **stage2** pay jo build hogi wo **1** and So on.

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

RUN npm run build

# Stage 2: Create the final image

FROM nginx:alpine

COPY --from=build /app/build /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

**Explanation:**

1. **Stage 1 (build):**
   * Starts with a Node.js base image (node:16) and sets up the working directory.
   * Installs the dependencies and builds the application.
   * This stage contains all the tools and dependencies needed to build the app, but these won't be included in the final image.
2. **Stage 2 (Final Stage):**
   * Starts with a lightweight Nginx base image (nginx:alpine).
   * Copies only the built application files (/app/build) from the build stage to the Nginx directory (/usr/share/nginx/html).
   * Exposes port 80 and starts the Nginx server.

**Benefits:**

* **Efficiency:** The final image is smaller and more efficient since it only contains the essential runtime components.
* **Security:** Reduces the attack surface by excluding build tools and unnecessary files.
* **Flexibility:** You can use different base images for different stages, allowing you to optimize each stage for its specific purpose.

Multi-stage builds are a powerful feature for creating lean, production-ready Docker images.

Agar Abhi Multi-stage build smjh nhi ayi toh no problem, You will understand it in the below content.

Q- Toh A/c to above explanation Kya jo multi-stages build use krtay hain wo iss liya krtay hain kay let say we have an application that is running on NodeJS environment but we want it to run on another environment like nginx So is for this type of purpose we use multi-stages build ?

Ans: Haan, multi-stage builds issi type ke purpose ke liye bhi use hotay hain, lekin sirf isi ke liye nahi. Multi-stage builds ka main maqsad yeh hota hai ke aap apne Docker image ko optimize kar sakein by separating the build process from the final runtime environment.

Lekin agar aap ke paas ek Node.js application hai jo sirf development ke liye Node.js environment mein build hota hai, lekin production mein usay sirf static files (HTML, CSS, JavaScript) ke taur par serve karna hai (jaise ke Nginx se), toh multi-stage build ek bohat hi useful technique hai.

**Example:**

1. **Stage 1:** Aap pehle stage mein apna application Node.js environment mein build karte hain. Yeh stage sirf build process ke liye hota hai jismein dependencies install hoti hain, application compile hota hai, aur production-ready files generate hoti hain.
2. **Stage 2:** Doosray stage mein aap sirf woh production-ready files ko copy karte hain aur unhe Nginx jese lightweight server ke saath deploy karte hain. Is stage mein sirf wohi files include hoti hain jo actual run-time ke liye zaroori hain, jese ke static files.

Is tarah, multi-stage builds aap ko allow karte hain ke aap apne application ke build aur runtime environments ko alag alag handle kar sakein. Yeh technique aap ko final Docker image size ko chhota aur zyada efficient banane mein madad deti hai.

**Summary:**

* **Build Stage:** Node.js environment mein application build hota hai.
* **Final Stage:** Sirf production-ready files ko Nginx jese lightweight server ke saath deploy kiya jata hai.

Yeh sirf aik example hai. Multi-stage builds ko different scenarios mein use kiya ja sakta hai depending on your application’s needs.

Q2- A/c to the above explanation its mean another purpose of using multi-stage build is to optimize the Docker Image by only having the neccessary files to run the app, example: Like jo 1st Stage hai usme App ko build krliya but wo final Image nhi build hui and 2nd Stage may usko Final Image build kr rhay hain or usme jo App build ki hai uski srf wohi Build wali files jo App ko run krnay ka liya neccessary hai use ki hai , baki ki jo file hain wo final Image may nhi ayeingi but on the other agar hum normally ek Image ko build krein toh usmay sari hi files jati hai jisme say kuch unneccessary hoti hai . Is my understanding is right or worng ?

Ans: Yes, your understanding is absolutely correct!

Multi-stage builds are indeed used to optimize Docker images by including only the necessary files and excluding everything else that isn't needed to run the application. Here's how your explanation breaks down:

1. **First Stage:**
   * In the first stage, you build your application, which might involve compiling code, installing dependencies, and generating build artifacts (like static files).
   * This stage contains all the files, tools, and libraries needed for building the app, but these are not necessary for running the app.
2. **Second Stage (Final Stage):**
   * In the second stage, you create the final Docker image that will actually be used in production.
   * Here, you only copy the necessary files (e.g., the build artifacts) from the first stage and include them in the final image.
   * All the unnecessary files, tools, and dependencies used in the first stage are not included in the final image, making it much smaller and more efficient.
3. **Without Multi-Stage Builds:**
   * If you were to build a Docker image without using multi-stage builds, all the files from the build process, including unnecessary ones, would be included in the final image. This would result in a larger image size and potentially more security vulnerabilities, as more unnecessary files and dependencies are exposed.

Hope So now you have understood the purpose of using multi-stages build . That we mainly use this technique to optimize our Docker Image . Acha isme ek cheez or hai wo yeh kay like we can create multi-stages let say we have create 3 stages so basically un Sab stages may app hi build ho rhi hogi BUT jo last stage hogi wo **final Image building** ki stage hogi jisme srf wohi files hongi that are most necessary for running an app. Ab yeh jo multi-stages hotay hain These stages may include steps like installing dependencies, compiling code, running tests, or generating build artifacts (like static files, binaries, etc.). However, these intermediate stages are mainly for preparing the necessary files, and they don't contribute directly to the final image that will be deployed.

Example:

# Stage 1: Build the application

FROM node:16 AS build

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

RUN npm run build

# Stage 2: Optimize the build (optional)

FROM node:16 AS optimize

WORKDIR /app

COPY --from=build /app/build /app/build

RUN npm run optimize

# Stage 3: Final image

FROM nginx:alpine

COPY --from=optimize /app/build /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

Q3- explain artifacts and binaries in the context of Docker ?

**1. Artifacts:**

* **Definition:** Artifacts are the files or outputs produced as a result of the build process. These could include compiled code, static assets, configuration files, or any other files that are necessary for the application to function in its final, deployed state.

**2. Binaries:**

* **Definition:** Binaries are compiled executable files that the application needs to run. These files are the result of source code being compiled into machine code that can be executed by the operating system.

**Usage in Docker:**

* **Artifacts in Multi-Stage Builds:**
  + In a multi-stage build, you might generate artifacts like static files (e.g., HTML, CSS, JavaScript) in the first stage. These files are then copied to the final stage, where they are served by a web server like Nginx.
  + The artifacts are the "finished" pieces of your application that don't need any further processing and are ready to be deployed.
* **Binaries in Multi-Stage Builds:**
  + When building an application that compiles down to binaries (e.g., C++, Go, Rust applications), you compile the source code in the first stage and then copy only the resulting binary to the final stage.
  + The final Docker image will contain just the binary and any necessary runtime dependencies, without including the source code or the tools used to create the binary.

Q4- Toh agar hum application may kuch change krtay hain toh wo DockerFile jab dubara say create hogi toh let say if we have created 3 stages like 1st for build 2nd for 2nd build and 3rd for final Image toh yeh sab duabara say build hongay ya srf koi specific ?

In Docker, when you rebuild an image after making changes to your application, Docker will try to use its **layer caching** mechanism to avoid rebuilding stages that haven't changed. Here’s how it works:

**Layer Caching in Docker:**

* **Docker Layers:** Each command in a Dockerfile (like RUN, COPY, FROM, etc.) creates a new layer in the Docker image. Docker caches these layers to speed up the build process.
* **Cache Reuse:** If Docker detects that a particular layer has not changed (meaning the command and the files involved haven't changed), it will reuse the cached version of that layer instead of rebuilding it.

**How This Applies to Multi-Stage Builds:**

* **Stage 1 (Build):** If you make changes in your source code or dependencies that are part of the build process (e.g., you modify a file that's copied or change the RUN command), Docker will rebuild this stage. If nothing changes in this stage, Docker will reuse the cached layers.
* **Stage 2 (Second Build):** Similar to the first stage, if there are any changes in the commands or files used in this stage, Docker will rebuild it. If there are no changes, Docker will skip rebuilding this stage and use the cached layers.
* **Stage 3 (Final Image):** The final stage will be rebuilt only if the files or commands involved in this stage have changed. If the changes are only in the first or second stage, Docker will rebuild those stages and then proceed to create the final image, but it will reuse layers in the final stage if they remain unchanged.

**Example Scenario:**

* **Change in Source Code:**
  + If you change a file in your source code, Docker will rebuild the stage where that file is copied or used. For instance, if the source code is copied in the first stage, Docker will rebuild the first stage.
  + The subsequent stages will also be rebuilt if they depend on the output of the first stage.
* **Change in Final Stage Configuration:**
  + If you only change something in the final stage (e.g., a configuration file or a command in the final Dockerfile stage), Docker will reuse the earlier stages' cached layers and only rebuild the final stage.

**Summary:**

* Docker rebuilds only the stages where changes have been detected.
* If a stage has no changes (in commands or files), Docker will reuse the cached version of that stage.
* This caching mechanism helps in speeding up the build process and avoids unnecessary rebuilding of stages that haven’t changed.

So, in your example with three stages, only the specific stages where changes occurred will be rebuilt. The other stages will use the cached layers from the previous build, making the process more efficient.