🧱 What is a Multi-Stage Build in Docker?

> **Multi-stage build** allows you to use **multiple `FROM` statements** in a single Dockerfile to:
>
> - Build the app in one stage 🏗️
> - Copy only what's needed to a smaller final image 📦

❓ Why Do We Need It?

✅ Main Goals:

🚀 **Benefit**	💬 **Why it Matters**
⚡ Smaller Images	Only copy what's needed into final image
🔐 More Secure	No dev tools or secrets in production image
🛠️ Cleaner CI/CD	Separate build & runtime environment
🗂️ Better Layer Caching	Speeds up builds
🌍 Environment Separation	One image builds everything!

🧠 Real-World Analogy

Imagine:

- 🏗️ Stage 1 = Construction site (messy, heavy tools)
- 🏠 Stage 2 = Finished house (clean, cozy)

You **build** in the messy environment, but **only move the furniture** into the clean house. 🧹

🧪 Multi-Stage Build Syntax

```
# 🔨 Stage 1: Build Stage
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# 📦 Stage 2: Final Production Image
FROM node:20-alpine
WORKDIR /app

# Copy only final build artifacts (no source or node_modules)
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package.json ./
RUN npm ci --omit=dev

# Set env vars, port and run
ENV PORT=3000
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

🔍 Key Concepts Explained

Keyword	Meaning
`AS builder`	Give a name to this stage
`--from=builder`	Copy files from previous stage
`npm ci --omit=dev`	Install only production deps
`COPY . .`	Used only in build stage to avoid code bloat in final image

---

📦 Before vs After: Image Size

Approach	Image Size	Contents
😵 Traditional Single Build	~900MB	Full source code + dev dependencies
😍 Multi-Stage Build	~200MB	Just built app + runtime dependencies

---

💥 Real Project Example: React App

```
# Step 1: Build React App
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Step 2: Serve using NGINX
FROM nginx:alpine
COPY --from=builder /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

☑ This builds the app with Node.js, and serves the static files via NGINX (no Node.js in final image!)

🎯 Common Multi-Stage Use Cases

Use Case	Description
💻 Frontend builds	Use `node` + `nginx` combo
🔧 Backend builds	Build with TS/Go/Rust, then copy binaries only
🧪 Testing stage	Add test/linting in one stage, skip in final
📦 CI/CD pipelines	Clean, reproducible builds across stages

🧰 Pro Tips & Best Practices

💡 Tip	✅ Recommendation
Use `--omit=dev`	Strip dev-only packages in final stage
Use `.dockerignore`	Exclude `node_modules`, `.git`, `tests/`, etc
Use labels	Add metadata like version, author, etc
Don't copy everything	Use exact `COPY` paths for size control
Use named stages	Easier to copy from (`--from=builder`)
Keep final image minimal	Just enough to run your app (no tools!)

🔄 Combine with Docker Compose

You can define multi-stage builds in your Dockerfile and just run:

```
docker-compose build
docker-compose up
```

Your services will use the optimized final image automatically 🧠✅

🛠️ Example Multi-Stage for TypeScript API

```
# Stage 1: Compile TS
FROM node:20-alpine AS builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

# Stage 2: Run with only JS output
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package*.json ./
```

```
RUN npm ci --omit=dev
CMD ["node", "dist/server.js"]
```

☑ Summary: When to Use Multi-Stage Builds?

☑ Always use if:

- You're using **build tools** like `tsc`, `webpack`, `vite`
- You want **minimal production images**
- You want to **separate testing/staging/building**
- You want **faster CI builds** & smaller attack surface

🧾 Final TL;DR Cheatsheet

Stage	Purpose	Base Image	Output
Stage 1 (builder)	Build, compile, test	`node`, `golang`, `rust`, etc.	`/dist`, `/build`, etc.
Stage 2 (prod)	Serve/run app only	`node:alpine`, `nginx`, etc.	Final slim image

---

📦 Full Dockerfile (Context Recap)

```
FROM node:20-alpine3.19 as base

# Stage 1: Build Stuff
FROM base as builder

WORKDIR /home/build

COPY package*.json .
COPY tsconfig.json .

RUN npm install

COPY src/ src/

RUN npm run build

# Stage 2: Runner
FROM base as runner

WORKDIR /home/app

COPY --from=builder /home/build/dist dist/
COPY --from=builder /home/build/package*.json .

RUN npm install --omit=dev
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nodejs

USER nodejs

EXPOSE 8000
ENV PORT=8000

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

🎯 A2Z Breakdown of Each Section

🧱 `FROM node:20-alpine3.19 as base`

🧠 What it does:

- Starts from a **minimal Node.js 20 Alpine image**
- Alpine is lightweight (~5MB), good for small, fast images
- `as base` names this stage for reuse

> 🧩 Think of `base` like a shared template that both stages use.

🔨 Stage 1: Builder

```
FROM base as builder
WORKDIR /home/build
```

🔧 What happens here:

- We **switch to a new build stage**, using `base` image
- `WORKDIR /home/build` sets a directory for our build process

```
COPY package*.json .
COPY tsconfig.json .
RUN npm install
```

📦 Install dependencies:

- `package*.json` copied to install dependencies
- `tsconfig.json` is required for TypeScript compilation
- `npm install` installs **all dependencies** (dev + prod)

```
COPY src/ src/
RUN npm run build
```

🛠️ **Build your app:**

- Copies your app's TypeScript code
- `npm run build` compiles TS into JS → typically inside `/dist`

✅ End Result of Stage 1:

> A folder `/home/build/dist` with compiled production-ready JS output.

🚀 Stage 2: Runner

```
FROM base as runner
WORKDIR /home/app
```

🗂️ **What it does:**

- We now create a fresh container just for **running** the app.
- `WORKDIR /home/app` is where your app will run from.

```
COPY --from=builder /home/build/dist dist/
COPY --from=builder /home/build/package*.json .
```

📂 **Copy built artifacts only:**

- Only copy the `dist/` folder and package files (no source, no tsconfig)
- Ensures the final image is **slim & clean**

```
RUN npm install --omit=dev
```

🔐 **Production-only install:**

- Installs **only prod dependencies** (no dev tools like `eslint`, `tsc`, etc.)
- Keeps final image light and secure ✅

👮 Add Secure Non-Root User

```
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nodejs
USER nodejs
```

Why?

- Running as `root` is dangerous in containers ✖
- We create a user `nodejs` with **limited permissions** for safety
- UID/GID `1001` is just an arbitrary non-root system user

Port & Env Setup

```
EXPOSE 8000
ENV PORT=8000
```

- `EXPOSE 8000`: Documents that the app uses port 8000
- `ENV PORT=8000`: Sets the default port for app to use internally

> You still need to use `-p` to map it to host: `docker run -p 8000:8000 <image>`

Start the App

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

Default entrypoint when container runs

- This triggers your `"start"` script from `package.json`:

  ```
  "start": "node dist/index.js"
  ```

Summary Table

◇ **Section**	**Purpose**
`FROM base`	Reuse image to reduce duplication
`builder`	Compiles TypeScript into JS
`runner`	Runs a minimal production image
`npm install` in builder	Installs full deps for building

🔹 Section	🔍 Purpose
`npm install --omit=dev` in runner	Installs only what's needed to run
`COPY --from=builder`	Efficient file copy without rebuild
`USER nodejs`	Enhances container security

📊 Resulting Benefits

🚀 Benefit	✅ Achieved
Small Image	✅ Only runtime code in final image
Secure	✅ Non-root user, no dev tools
Faster Builds	✅ Reuses build layers
Clean Code Separation	✅ No TypeScript or build files inside final container
Portable	✅ Can run on any platform with Node 20

🧠 Bonus Tip: View Image Sizes

```
docker images
```

Compare the **multi-stage image (~100MB)** vs a **single-stage image (~400–600MB)** 🤯

🔚 Final Thoughts

> This approach follows **Docker best practices**:
>
> - Multi-stage
> - Production-ready
> - Secure by default
> - Reproducible builds
