Click here for DevSecOps & Cloud DevOps Course

DevOps Shack

📑 Table of Contents:

.NET Core Web API

- Vue.js Frontend

- Go Binary App

- Multi-language mono-repo scenario

6. 🎯 Pro Tips for Clean, Modular, and Reusable Builds

- Using .dockerignore

- Named stages for reuse

- Using environment variables smartly

- Keeping Dockerfile DRY

7. ᴳ Testing & Debugging Multi-Stage Builds

- Multi-stage friendly Dockerfile linters

- docker build --target

- Troubleshooting build stage errors

8. 📦 Making Dockerfiles CI/CD Ready

- Integrating in GitHub Actions / GitLab CI

- Caching strategies

- Secrets and build-time arguments

9. 🛡 Performance and Security Optimization

- Minimal base images: Alpine vs Distroless

- Trimming unnecessary binaries

- Scanning for vulnerabilities

- Using SBOMs (Software Bill of Materials)

▦ 1. Introduction to Docker & Its Limitations

📦 What is Docker?

Docker is an open-source platform that enables developers to build, package, and run applications as **containers**. Containers encapsulate everything an app needs to run — code, runtime, libraries, environment variables, and configuration files — into a single, lightweight unit.

🗃 Key Benefits of Docker:

- **Consistency:** "It works on my machine" becomes a thing of the past.

- **Isolation:** Each app runs independently.

- **Portability:** Docker runs the same on any infrastructure: local, dev, staging, or production.

- **Scalability:** Easily integrated into microservices and CI/CD pipelines.

🚫 Traditional Dockerfile: What Can Go Wrong?

Most teams start with a **single-stage Dockerfile**, and that's fine — until the container starts ballooning in size or leaking build dependencies.

Here's an example of a traditional Node.js Dockerfile:

```
FROM node:18
```

```
WORKDIR /app
```

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

```
COPY . .
```
```
RUN npm run build
```

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

🔎 Problems with this approach:

Issue	Why It's a Problem
📦 Large Image Size	Node modules, source code, build tools, and final build all exist in the same image.
🔐 Security Risks	The image contains dev tools like npm, node_modules, and even secrets sometimes.
🚀 Slower Deployment	Larger images take longer to transfer, pull, and spin up — affecting your pipeline speed.
🔁 Inefficient Caching	Any small change in code may invalidate all cached layers.
🛠️ No Separation of Concerns	Build and run logic is mixed, making maintenance harder.

🗜️ Why Should You Care?

If you're serious about:

- 🚀 Fast deployments
- 🔐 Secure containers
- 💡 Clean DevOps practices
- 📦 Lightweight, production-ready images

...then **multi-stage builds** are your next step.

☑️ Summary

☑️ Traditional Dockerfile	✖️ Drawbacks
Easy to start with	Not optimized for production
All-in-one image	Bigger attack surface

☑ Traditional Dockerfile	✖ Drawbacks
Includes build tools	Slower CI/CD pipelines
Simple, but inefficient	Hard to maintain at scale

🏗 2. What is a Multi-Stage Docker Build?

🧠 **Definition**

A **Multi-Stage Docker Build** is a feature that allows you to use **multiple FROM statements** in a single Dockerfile, each defining a new build stage. This enables you to:

- Use one image for **building** your application (with all necessary tools),

- And another **minimal image** for **running** it — copying only the final output.

The result? **Smaller, cleaner, and more secure containers.**

How Does It Work? Here's the concept in a nutshell:

```
# Stage 1: Build Stage
FROM node:18 AS builder
WORKDIR /app
COPY . .
RUN npm install && npm run build
```

```
# Stage 2: Production Image
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

- The first image (node:18) has everything needed to build the app.

- The final image (nginx:alpine) **only includes the built app**, not the code or dev tools.

Why Use Multi-Stage Builds?

Benefit	Description
Clean Production Image	Keep only what's required to run your app — no dev tools, no source code.
	Images are often **80-90% smaller**, improving CI/CD

Smaller	

Image	

Benefit	Description
Size	speed and reducing attack surface.
🚀 Faster Deployments	Lighter containers = faster transfers = quicker deployments.
🔐 Better Security	Reduces exposure to vulnerabilities by excluding compilers, shells, and unused binaries.
🔁 Improved Caching	Breaks up build stages, allowing Docker to reuse cached layers efficiently.

🧬 When Should You Use It?

Use Case	Should You Use Multi-Stage?
Node.js app with npm build	☑ Yes
.NET Core app with dotnet publish	☑ Absolutely
Go app compiled to binary	☑ Perfect use case
Small static file site	⚠ Not mandatory, but still beneficial
Script-based containers (bash-only)	✖ Probably not needed

⌲ TL;DR

Multi-Stage Builds = One Dockerfile, multiple FROM stages, optimized output.

They help you:

- Split build and production responsibilities

- Use powerful build images (e.g., node, golang, dotnet)

- Deliver minimal runtime images (e.g., alpine, distroless)

🔍 3. Anatomy of a Multi-Stage Dockerfile

🧬 Understanding the Structure

A **multi-stage Dockerfile** is structured like a chain of separate but connected mini-Dockerfiles — each with its own FROM statement. You can think of each

stage as a **checkpoint** where certain work gets done, and only selected outputs are passed to the next.

Here's what it looks like in its simplest

form: # Stage 1: Builder

FROM node:18 AS builder

WORKDIR /app

COPY . .

RUN npm install && npm run build

Stage 2: Runtime

FROM nginx:alpine

COPY --from=builder /app/dist

/usr/share/nginx/html Let's break it down line-by-line.

🛠️ Detailed Breakdown of Each Component

1. FROM with Alias (AS builder)

FROM node:18 AS builder

- The AS builder part names this stage.

- This makes it easier to refer to this stage later (using --from=builder).

- You can define multiple such stages for different steps (e.g., test, lint, compile).

2. WORKDIR and COPY

WORKDIR /app

COPY . .

- WORKDIR sets the current directory inside the container.

3. Installing Dependencies and Building

RUN npm install && npm run build

- This installs project dependencies and compiles the app.

- All build tools, source code, and intermediate files exist **only in this stage**.

4. Final Runtime Stage

FROM nginx:alpine

- This creates a **new, clean image**.

- nginx:alpine is a lightweight base with no development tools — perfect for running production apps.

5. Copying From Previous Stage

COPY --from=builder /app/dist /usr/share/nginx/html

- This copies only the final build output (e.g., /app/dist) into the production image.

- The earlier stage (builder) is **discarded** after build — its tools don't make it to the final image.

🧠 Why This Works So Well

Let's say your full app with all dependencies, dev tools, and source code creates a Docker image of **1.2 GB**.

With multi-stage builds, you can:

- Build in a full Node.js or .NET Core image (including compilers, linters, testing tools)

- Output only the **final, compiled app** into a minimal alpine image

- End up with a **tiny ~80MB container** that:

  - Starts fast

  - Has no unused code

o   Is optimized for Kubernetes or CI/CD deployment

🔄 Multiple Intermediate Stages

You can also use multiple stages for:

- Linting

- Testing

- Generating documentation

- Collecting metrics or coverage

reports Example:

☑ Summary Table

Component	Purpose
FROM ... AS name	Start a new build stage with a label
COPY --from=stage	Copy files between stages
Multiple FROM blocks	Split build and runtime
Final Image	Clean, minimal, ready for production

Q **Real Talk: Why DevOps Teams Love This**

- **Dockerfiles become modular and clean**

- **Security teams love the reduced attack surface**

- **DevOps engineers enjoy faster pipeline runs**

- **Developers can test multiple stages separately**

4. Converting Any Dockerfile into Multi-Stage: A Step-by-Step Guide

⊗ Why Convert?

Before we jump in, here's a reminder of the "why":

Traditional Dockerfile	Multi-Stage Dockerfile
One big image with all tools	Separate build and runtime environments
Dev dependencies stay in the final image	Only the necessary files are carried forward
Higher risk, slower deployments	More secure and faster

Now, let's break down the conversion process.

☐Step-by-Step Conversion Process

☑ Step 1: Identify the Build Phase

Every app has a **build step**, even if it's just installing dependencies.

Examples:

- Node.js: npm install && npm run build

- .NET: dotnet publish

- Go: go build

- Vue/React: npm run build

👉 **Locate where the Dockerfile installs dependencies and builds the source code.**

RUN npm install

RUN npm run build

These commands belong in the **build stage**.

☑ Step 2: Split Build & Runtime Stages

Now, restructure your Dockerfile into at least two stages.

- **Stage 1: Builder**

    o Use a full-featured image (e.g., node, dotnet, golang)

    o Install dependencies

    o Build the application

- o Use a slim/minimal image (nginx:alpine, alpine, distroless)
- o Copy over the built app

only # Builder Stage

FROM node:18 AS builder

WORKDIR /app

RUN npm install && npm run build

Runtime Stage

FROM nginx:alpine

COPY --from=builder /app/dist /usr/share/nginx/html

☑ Step 3: Only Copy the Necessary Files

Don't bring your entire project into the runtime image — just the final output.

☑ Good:

COPY --from=builder /app/dist /usr/share/nginx/html

🚫 Bad:

COPY . . # Brings unnecessary files

Use .dockerignore to exclude junk:

node_modules

.git

Dockerfile

README.md

.env

☑ Step 4: Optimize Layer Caching

Docker caches image layers from top to bottom. If you put COPY . . *before* installing dependencies, even a small code change will force a full rebuild of npm install.

🫠 **Better approach:**

COPY package*.json ./

RUN npm install

COPY . .

This way, if only app code changes (not package.json), Docker reuses the install layer.

☑ **Step 5: Clean Up and Test the Build**

Final image should:

- Not include source code

- Not have dev tools or build logs

- Be minimal in size (use docker image ls to

compare) Test using:

docker build -t my-app .

docker run -p 3000:3000 my-

app Or debug an intermediate

stage:

docker build --target builder -t build-stage .

docker run -it build-stage sh

🔍 **Before & After Comparison**

📄 **Original**

Dockerfile `FROM`

`node:18` `WORKDIR`

```
COPY . .
RUN npm install
RUN npm run build
EXPOSE 3000
CMD ["npm", "start"]
```

🔄 **Multi-Stage Dockerfile**

```
# Build Stage
FROM node:18 AS builder
WORKDIR /app
COPY package*.json
./ RUN npm install
COPY . .
RUN npm run build

# Final Stage
FROM
nginx:alpine
COPY --from=builder /app/dist
/usr/share/nginx/html EXPOSE 80
```

🛠️ **Tools to Help You**

- G docker build --target <stage>: Debug individual stages
- 🧽 dive: Tool to analyze Docker image size
- 🧽 hadolint: Lint your Dockerfile for best practices

📦 **TL;DR Checklist**

☑ Use at least two stages: build and runtime

☑ Use .dockerignore to exclude trash
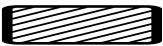
- ☑ Copy only built output to runtime image
- ☑ Use named stages for clarity
- ☑ Use minimal base images in final stage
- ☑ Test your final container — no dev tools, just app

🌐 5. Real-World Examples of Multi-Stage Docker Builds

In this section, we'll cover multi-stage Docker builds for the following tech stacks:

Stack	Language	Framework/Tool	Target
Node.js	JavaScript/TypeScript	Express, React, Vue	Web/API
.NET Core	C#	ASP.NET Core	Web/API

	Go	Native	CLI/API
Go			

Stack	Language	Framework/Tool	Target
Python	Python	Flask, FastAPI	Web/API
Java	Java	Spring Boot	Web/API

Let's go through them one by one 👆

1. Node.js (React / Express / Vue)

Use Case: SPA frontend or REST API with a build step (React/Vue) or npm start (Express)

```
# Stage 1: Build App

FROM node:18 AS builder

WORKDIR /app

COPY package*.json

./ RUN npm install

RUN npm run build    # For frontend apps


# Stage 2: Serve with Nginx

FROM nginx:alpine

COPY --from=builder /app/dist

/usr/share/nginx/html EXPOSE 80
```

For Express.js (no build), just move app code (not node_modules) to production image and re-install with --production.

2. .NET Core (ASP.NET Core)

Use Case: Web API with publish step using dotnet publish

```
# Stage 1: Build

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
```

```
WORKDIR /src

COPY *.csproj .

RUN dotnet

restore COPY . .

RUN dotnet publish -c Release -o /app/publish


# Stage 2: Runtime

FROM mcr.microsoft.com/dotnet/aspnet:8.0

WORKDIR /app

COPY --from=build /app/publish .

ENTRYPOINT ["dotnet", "YourApp.dll"]
```

☑ Clean separation of SDK (build tools) and Runtime
☑ Output is optimized and trimmed

3. Go (Golang)

Use Case: CLI tools or lightweight APIs compiled to binaries

```
# Stage 1: Build

FROM golang:1.21 AS builder

WORKDIR /app


RUN go build -o myapp


# Stage 2: Runtime

(minimal) FROM

alpine:latest WORKDIR /app

COPY --from=builder /app/myapp .

ENTRYPOINT ["./myapp"]
```

☑ Final image is just a few MBs
☑ Super fast and perfect for Kubernetes

4. Python (Flask / FastAPI)

Use Case: APIs with pip-based dependencies

```
# Stage 1: Build

FROM python:3.11-slim AS builder

WORKDIR /app

COPY requirements.txt .

RUN pip install --user -r requirements.txt


# Stage 2: Runtime

FROM python:3.11-slim

WORKDIR /app

COPY --from=builder /root/.local /root/.local

ENV PATH=/root/.local/bin:$PATH

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

☑ Separates dependency install
☑ Keeps runtime lean and clean

5. Java (Spring Boot)

Use Case: Spring Boot JAR builds

```
# Stage 1: Build

FROM maven:3.9.6-eclipse-temurin-17 AS builder

WORKDIR /build

COPY pom.xml .
```

```
RUN mvn dependency:go-ofline

RUN mvn clean package -DskipTests

# Stage 2: Runtime

FROM eclipse-temurin:17-jre

WORKDIR /app

COPY --from=builder /build/target/*.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]
```

☑ Final image doesn't include Maven or source
☑ Just the runnable .jar

⬅ Common Patterns You Can Reuse

Concept	Applies To
COPY --from=builder	All stacks
Slim/minimal runtime images	Go, Python, Node, Java
Clean separation of logic	All multi-stage builds
Reduced image size	ALL — up to **80% smaller**
Faster CI/CD deploys	Especially Node, .NET, Java

🛠 6. Tools to Analyze and Optimize Multi-Stage Builds

Multi-stage builds are powerful, but to **fully harness their potential**, you'll want to inspect what your images are doing under the hood. The following tools and techniques help you **analyze image layers**, **reduce size**, **improve caching**, and **ensure security**.

🔍 1. dive — Inspect Image Layers

📦 Inspect each layer of a Docker image and see exactly what files are added, modified, or removed.

Installation:

brew install dive # macOS

scoop install dive # Windows

Usage:

dive my-app:latest

What You'll See:

- Layer-by-layer file system diff

- Compressed/uncompressed sizes

- Warnings for inefficient layers (e.g., COPY . . too early)

- Opportunity to optimize layers

🔎 **Why it helps:**

- Spot bloat like leftover logs, build tools

- Confirm your multi-stage copy is working

- Visualize .dockerignore effectiveness

🔍 **2. hadolint — Dockerfile Linter**

🧰 A linter for Dockerfiles to enforce best practices and standards.

Installation:

brew install hadolint

Usage:

hadolint Dockerfile

Checks For:

- Missing HEALTHCHECK

- Using latest tag (which is bad)

- Not using --no-cache with apk add

- Inefficient copy/add commands

📑 **Sample Warning:**

DL3008: Pin versions in apt get install. Instead of apt-get install <pkg>, use apt-get install <pkg>=<version>`"

🔍 3. docker-slim — Minify Your

🎯 Automatically create a **smaller, secure** version of your image by stripping everything not used at runtime.

Installation:

brew install docker-slim

Usage:

docker-slim build my-app

What It Does:

- Analyzes your container's runtime behavior

- Removes unused binaries, files, libraries

- Outputs a new slimmed-down image

💧 **Typical Results:**

- Original: 600MB

- Slimmed: 50MB 🎉

🔍 4. Docker Bench for Security

🛡 Audits your host and Docker containers for security best practices.

Usage:

git clone https://github.com/docker/docker-bench-

security.git cd docker-bench-security

./docker-bench-security.sh

Checks For:

- Permissions, kernel capabilities

- Image user settings (non-root usage)

- Unused ports, exposed volumes

- Logging and auditing best practices

☑ Especially useful for production-ready pipelines

🧠 5. Tips for Manual Optimization

Problem	Fix
Image too large	Use alpine, or distroless
Dev tools in final image	Use separate build stage
Slow rebuilds	Separate COPY and RUN steps for cache reuse
Junk in image	Use .dockerignore, trim logs/temp
Large node_modules	Use npm prune --production in final stage

⌒ 6. Testing Multi-Stage Targets

Sometimes, you need to debug a specific stage.

Build only a stage:

docker build --target builder -t my-app-builder .

Run a stage interactively:

docker run -it my-app-builder sh

This helps test that build, tests, or assets are created correctly — **before they're passed to the runtime image**.

📋 Summary: Optimization Toolkit

Tool	Purpose
dive	Visual image inspection
hadolint	Dockerfile best practices checker
docker-slim	Auto-minify images
docker-bench	Security audit
--target	Test build stages individually

🚫 7. Common Pitfalls & How to Avoid Them in Multi-Stage Builds

While multi-stage builds help simplify and optimize Docker images, many developers still fall into traps that lead to bloated images, broken builds, or poor CI/CD performance. Here's a list of **gotchas** you'll want to watch out for — with clear guidance on how to fix them.

✖ 1. Copying Everything Instead of Just What's Needed

Problem:

COPY . .

☑ Why it's bad:

- Brings unnecessary files into the build context — .git, .env, README.md, test data, etc.

- Breaks Docker layer caching on every change.

🛠 Fix:

- Use a .dockerignore file to exclude clutter.

- Only copy package*.json for dependency

install. COPY package*.json ./

```
RUN npm install
```

✖ 2. Missing .dockerignore

Problem:

If your .dockerignore is missing, Docker will include **everything** in the build context — including logs, Git history, node_modules, etc.

🗁 Bad Example:

```
# .dockerignore missing or empty
```

⚒ Fix: Add .dockerignore

```
.git node_modules

.env

    Dockerfile

*.log
```

☑ Results in smaller, faster builds and safer images.

✖ 3. Using latest Tag for Base Images

Problem:

```
FROM node:latest
```

☑ Why it's bad:

- Builds can break if the base image updates unexpectedly.

- Introduces non-deterministic behavior in CI/CD.

⚒ Fix: Always **pin your image version**.

```
FROM node:18.18.0
```

✖ 4. Installing Dev Tools in the Runtime

Problem:

Installing things like compilers, test frameworks, or linters in the final image:

RUN npm install eslint

☑ Why it's bad:

- Bloats the image size

- Exposes tools unnecessarily in production

- Breaks minimalism of multi-stage builds

⚒ Fix:

Only install dev tools in the **build stage**.
Then **copy only the final build artifacts** into the runtime stage.

✖ 5. Not Using Named Build Stages

Problem:

COPY --from=0 /app/dist /usr/share/nginx/html

☑ Why it's bad:
Using stage index (0, 1, etc.) makes it hard to read and maintain.

⚒ Fix:
Use **named stages** instead.

FROM node:18 AS builder

COPY --from=builder /app/dist

/usr/share/nginx/html Much cleaner and

understandable.

✖ 6. Overusing Layers

Problem:

RUN apt update

RUN apt install -y curl

✖ 4. Installing Dev Tools in the Runtime

RUN apt install -y git

☑ Why it's bad:

Each RUN creates a new layer. Multiple package manager calls = unnecessary layers.

⚒ **Fix:**

Chain installation commands:

RUN apt update && \

   apt install -y curl git && \

   rm -rf /var/lib/apt/lists/*

Also cleans up cache!

✖ **7. Missing Health Checks**

Problem:

No HEALTHCHECK = No automated way for Docker/K8s to know if the container is working.

⚒ **Fix:**

HEALTHCHECK CMD curl --fail http://localhost:3000/health || exit 1

☑ Makes your containers more production-ready and robust in orchestrators like Kubernetes.

✖ **8. Exposing Secrets**

Problem:

Copying .env or injecting secrets in image layers.

⚒ **Fix:**

- Never COPY .env into the container.

- Use environment variables via docker run -e or CI/CD pipeline secrets.

- Consider tools like **Docker Secrets**, **Vault**, or **AWS Parameter Store**.

☑ **Summary: Pitfall → Fix**

Pitfall	Fix
COPY . .	Copy only needed files + use .dockerignore
No .dockerignore	Create one immediately
FROM base:latest	Pin to specific version
Dev tools in runtime	Use multi-stage to keep runtime clean
Using numeric build stage refs	Use named stages like AS builder
Too many RUNs	Chain them with &&
Missing health checks	Add HEALTHCHECK command
Including secrets	Use runtime injection or external secret stores

🔑 8. Final Thoughts & Best Practices Roundup

This section will tie everything together and provide you with **actionable tips and best practices** that you can apply immediately to your own multi-stage Docker builds. These practices will help you ensure that your builds are both **optimized for speed and security**, and **easy to maintain** over time.

🚀 Best Practices for Multi-Stage Docker Builds

1. Start with a Clean Base Image

- **Use minimal base images** (like alpine or distroless) in the runtime stage to keep your container lean.

- **Avoid unnecessary package installations** — only include what's required for the application to run.

Example:

```
FROM node:18-alpine AS builder
```

2. Limit the Number of Layers

- Combine multiple RUN statements into a **single command** to reduce layers and improve caching efficiency.

- Avoid creating unnecessary layers when copying files — only copy what's absolutely necessary.

Bad Example:

```
RUN apt-get update

RUN apt-get install -y curl

RUN apt-get install -y git
```

Good Example:

```
RUN apt-get update && \

    apt-get install -y curl git && \

    rm -rf /var/lib/apt/lists/*
```

3. Optimize Caching with Order of Commands

- Docker caches layers and will reuse them unless a layer's content changes. To maximize caching:

    o Place **frequent changes** (like source code) **lower** in the Dockerfile.

    o Place **infrequent changes** (like dependencies) **near the top**.

Example:

```
# COPY package files early to take advantage of caching for dependencies

COPY package*.json ./

RUN npm install
```

```
# Then copy the rest of the app code
```

This way, npm install will be cached unless package.json or package-lock.json changes.

4. Keep Sensitive Data Out

- Never include sensitive information in the Dockerfile or in layers (e.g., .env files or API keys).

- Use **Docker secrets** or environment variables injected at runtime to manage secrets.

Bad Example:

```
COPY .env .env
```

Good Example:

- Use .dockerignore to prevent .env from being copied.

- Inject secrets at runtime or use secure storage (e.g., Docker Secrets, AWS SSM).

5. Minimize Final Image Size

- Multi-stage builds are a great way to reduce the size of the final image.

- Make sure the **runtime image** only contains what's required to run the app (no build tools or unnecessary dependencies).

Example:

```
FROM node:18-alpine AS builder

WORKDIR /app


RUN npm install && npm run build


# Final stage: Copy only built assets

FROM nginx:alpine

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

By doing this, the final image only contains the compiled assets, and the build tools like node_modules are left behind.

6. Use Versioned Base Images

- Pin the version of your base image to ensure **consistent behavior** across environments.

- Avoid using latest as it may lead to **unpredictable results** if the base image updates unexpectedly.

Bad Example:

```
FROM node:latest
```

Good Example:

FROM node:18.18.0

7. Use Build Arguments for Flexibility

- For configuration options that change based on environment or deployment, use **build arguments** (ARG).

- This makes your Dockerfile more flexible and reusable.

Example:

ARG NODE_ENV=production

ENV NODE_ENV $NODE_ENV

This allows you to specify the environment at build time, so the image can be customized for dev, staging, or production.

8. Ensure Production Readiness with Health Checks

- Always add a **health check** to your Dockerfile to ensure your application is running correctly after deployment.

- This allows orchestrators like **Kubernetes** to automatically restart containers that are unhealthy.

Example:

HEALTHCHECK CMD curl --fail http://localhost:3000/health || exit 1

9. Use Multi-Stage for Specific Build Phases

- **Use named build stages** to clarify the purpose of each stage (e.g., builder, runtime, test).

- This also allows you to target specific stages during the build process if needed.

Example:

FROM node:18 AS builder

...

FROM node:18 AS runtime

...

COPY --from=builder /app /app

This approach improves clarity and makes it easier to test or rebuild individual stages.

10. Monitor Image Vulnerabilities

- **Regularly scan** your Docker images for vulnerabilities. Use tools like **Anchore** or **Trivy** to automate vulnerability scanning.

- Consider using **distroless images** (such as gcr.io/distroless/base) to minimize potential attack vectors in your images.

🧠 Final Thoughts

Multi-stage builds are one of the best ways to optimize Dockerfiles for performance, security, and maintainability. With the practices outlined above, you can ensure that your Docker images are lean, fast, and easy to deploy in production.

By using **named stages**, **pinning base images**, and focusing on **reducing image size** and **security**, you'll be setting yourself up for success in your Docker-based projects.

Now that we've covered everything, here's a quick recap:

🔑 Key Takeaways:

1. Use **minimal base images** and **lean final images**.

2. Maximize **cache usage** and minimize layers.

3. Keep **sensitive data out** and use runtime configuration.

4. **Pin versions** and use build arguments to make your images flexible.

5. Add **health checks** and **scan for vulnerabilities** regularly.