🐳 Docker Image Optimization Guide — The Ultimate Cheat Sheet 🚀

Optimize your Docker images for faster builds, smaller size, better caching, and production readiness. Let's go!

🧱 1. Use a Small & Specific Base Image 🐧

✅ Do This:

```
# Use lightweight Alpine variant
FROM node:20-alpine
```

❌ Avoid This:

```
# Heavy image — more layers, longer build times
FROM ubuntu
```

📝 Why?

- Smaller base = smaller image.
- Alpine images are ~5MB vs Ubuntu's ~100MB.
- Smaller size = faster download, upload, deploy.

🪜 2. Order Instructions for Layer Caching 🔁

✅ Do This:

```
# Caches `npm install` unless package.json changes
COPY package*.json ./
RUN npm install

# Copy rest of the source after deps are installed
COPY . .
```

❌ Avoid This:

```
COPY . .      # 👎 invalidates cache if any file changes
RUN npm install
```

📝 Why?

Docker caches layers. Changing a later step invalidates all subsequent layers. Put stable steps early for faster rebuilds.

🖌 3. Remove Unnecessary Files with `.dockerignore` 🚫

☑ Example `.dockerignore`:

```
node_modules
Dockerfile
.dockerignore
.git
npm-debug.log
```

📝 Why?

It prevents Docker from copying unnecessary files into the build context. Less context = faster build and smaller image.

🏗 4. Use Multi-Stage Builds 🧪

☑ Example:

```
# Stage 1: Builder
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# Stage 2: Runtime
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app .
EXPOSE 8000
CMD ["npm", "start"]
```

📝 Why?

- Separate build dependencies from runtime.
- Final image contains only what's needed to run the app.
- Reduces image size by up to 70%.

🧼 5. Remove Unused Dependencies 🖌

✅ Use `--production` for Node.js:

```
RUN npm ci --only=production
```

OR

```
npm prune --production
```

📝 Why?

Avoid bundling dev tools and test libraries into your production image.

📦 6. Combine RUN Commands 🔗

✅ Do This:

```
RUN apk add --no-cache bash curl && \
    rm -rf /var/cache/apk/*
```

❌ Don't Do:

```
RUN apk add bash
RUN apk add curl
```

📝 Why?

Each `RUN` creates a layer. Combining reduces total layers = smaller image size.

🐘 7. Use `--no-cache` for Package Managers 📥

✅ Alpine:

```
RUN apk add --no-cache curl
```

✅ APT (Debian/Ubuntu):

```
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
```

📝 Why?

Avoids unnecessary cache files and reduces image size.

🔐 8. Avoid Root User (Security Best Practice) 🔒

```
# Create a non-root user
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

📝 Why?

Running as root inside containers is risky. Use non-root users for security.

🪄 9. Clean Up Temporary Files 🧼

```
RUN npm install && \
    npm cache clean --force
```

📝 Why?

Removes package cache after install to reduce bloat.

📌 10. Use Specific Tags (Not `latest`) 🎯

```
FROM node:20.11.1-alpine
```

📝 Why?

Using `latest` can break builds if the base image updates and introduces changes. Always pin versions for reliability.

🔒 11. Scan for Vulnerabilities 🧬

```
docker scan my-node-app
```

Or use:

- **Docker Scout**

- **Trivy** (Aqua Security)
- **Snyk**

🧪 12. Analyze Image Size and Layers 🔍

```
docker image inspect my-node-app
docker history my-node-app
```

Or use tools like:

- **Dive**: `dive my-node-app`
- **DockerSlim**: `docker-slim build my-node-app`

🧠 Final Pro Tips 💡

Tip	Benefit
Use `npm ci` instead of `npm install`	Faster and more reliable
Group `COPY` steps wisely	Better cache usage
Avoid adding `.env` or secrets	Security risk
Label images (`LABEL maintainer=...`)	Better documentation
Run production builds with `NODE_ENV=production`	Removes dev dependencies

📌 Sample Optimized Dockerfile for Node.js App

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

# ✨ Stage 2: Runtime
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app .
ENV NODE_ENV=production
RUN npm prune --production
EXPOSE 8000
CMD ["npm", "start"]
```
