Docker Multi-Stage Build - Complete Documentation

Table of Contents

What is Docker Multi-Stage Build?

Docker Multi-Stage Build is a powerful feature that allows you to use multiple `FROM` statements in a single Dockerfile[1]. Each `FROM` instruction creates a new stage in the build process, enabling you to **optimize image size** and **improve security** by separating build dependencies from runtime requirements[2].

Key Concepts

- **Multiple Stages**: Each stage has its own base image and purpose
- **Selective Copying**: Copy only necessary artifacts between stages
- **Artifact Exclusion**: Build tools and dependencies are left behind
- **Production-Ready**: Final image contains only runtime requirements

Why Use Multi-Stage Builds?

Multi-stage builds solve several critical problems in containerized application development[3]:

Benefit	Description	Impact
Smaller Image Size	Excludes build tools and dependencies from final image	**Faster deployments**
Enhanced Security	Reduces attack surface by removing unnecessary components	**Lower vulnerability risk**
Better Performance	Lighter images load and start faster	**Improved runtime speed**
Cleaner Workflow	Single Dockerfile for entire build process	**Simplified maintenance**

Benefit	Description	Impact
Cost Optimization	Reduced storage and bandwidth usage	**Lower infrastructure costs**

How Multi-Stage Builds Work

Single-Stage vs Multi-Stage Comparison

```
graph TB
    subgraph "❌ Single-Stage Build Problems"
        SINGLE[📦 Single Stage]
        SINGLE --> BUILD_TOOLS[🔧 Build Tools]
        SINGLE --> SOURCE[📄 Source Code]
        SINGLE --> DEPS[📚 All Dependencies]
        SINGLE --> ARTIFACTS[⚡ Build Artifacts]
        SINGLE --> FINAL1[📦 Final Image: 200MB+]
    end

    subgraph "☑ Multi-Stage Build Solution"
        STAGE1[🏗 Build Stage]
        STAGE2[🚀 Runtime Stage]

        STAGE1 --> BUILD_TOOLS2[🔧 Build Tools]
        STAGE1 --> SOURCE2[📄 Source Code]
        STAGE1 --> BUILD_DEPS[📚 Build Dependencies]
        STAGE1 --> COMPILE[⚙ Compile/Build]

        STAGE2 --> RUNTIME_BASE[🏃 Runtime Base Image]
        STAGE2 --> COPY_ARTIFACTS[📋 Copy Build Artifacts]
        STAGE2 --> FINAL2[📦 Final Image: 50MB]

        COMPILE -.->|Copy Only Artifacts| COPY_ARTIFACTS
    end
```

Build Process Flow

```
sequenceDiagram
    participant User as 👤 Developer
    participant Docker as 🐳 Docker Engine
    participant Stage1 as 🏗 Build Stage (installer)
    participant Stage2 as 🚀 Runtime Stage (deployer)
    participant Registry as 📦 Image Registry

    User->>Docker: docker build -t multistage .
    Docker->>Stage1: FROM node:18-alpine AS installer
    Stage1->>Stage1: WORKDIR /app
    Stage1->>Stage1: COPY package*.json ./
    Stage1->>Stage1: RUN npm install
```

```
    Stage1->>Stage1: COPY . .
    Stage1->>Stage1: RUN npm run build

    Docker->>Stage2: FROM nginx:latest AS deployer
    Stage2->>Stage1: COPY --from=installer /app/build /usr/share/nginx/html

    Docker->>User: 📦 Optimized Image Ready (50MB)
    User->>Registry: docker push multistage
```

📦 Step-by-Step Tutorial

🛠️ Project Setup

1. Clone the Application

```
git clone
cd react-app-docker
ls  # Check project structure
```

Project Structure:

```
react-app-docker/
├── src/
├── public/
├── package.json
├── package-lock.json
└── README.md
```

📝 Creating Multi-Stage Dockerfile

2. Create Dockerfile

```
touch Dockerfile
vi Dockerfile
```

3. Multi-Stage Dockerfile Content

```
# 🏗️ Stage 1: Build Stage (installer)
FROM node:18-alpine AS installer
WORKDIR /app

# Copy package files for dependency installation
COPY package*.json ./

# Install all dependencies (including devDependencies)
```

```
RUN npm install

# Copy source code
COPY . .

# Build the application
RUN npm run build

# 🚀 Stage 2: Runtime Stage (deployer)
FROM nginx:latest AS deployer

# Copy only build artifacts from previous stage
COPY --from=installer /app/build /usr/share/nginx/html

# Nginx will serve the static files
EXPOSE 80
```

🔧 Build Process

4. Build the Multi-Stage Image

```
docker build -t multistage .
```

📊 Build Output Analysis

```
graph LR
    subgraph "🏗️ Build Stage Process"
        A[📄 package.json] --> B[📦 npm install]
        C[📁 Source Code] --> B
        B --> D[⚙️ npm run build]
        D --> E[📁 /app/build]
    end

    subgraph "🚀 Runtime Stage Process"
        F[🌐 nginx:latest] --> G[📋 Copy build artifacts]
        E -.->|COPY --from=installer| G
        G --> H[🎯 Final Image]
    end

    style E fill:#90EE90
    style H fill:#87CEEB
```

🔄 Multi-Stage Build Workflow

🎯 Complete Workflow Diagram

```
flowchart TD
    START([🚀 Start Build Process]) --> STAGE1{🏗️ Build Stage}

    STAGE1 --> NODE[📦 FROM node:18-alpine AS installer]
    NODE --> WORKDIR[📁 WORKDIR /app]
    WORKDIR --> COPY_PKG[📋 COPY package*.json ./]
    COPY_PKG --> NPM_INSTALL[⬇️ RUN npm install]
    NPM_INSTALL --> COPY_SRC[📄 COPY . .]
    COPY_SRC --> NPM_BUILD[⚙️ RUN npm run build]

    NPM_BUILD --> STAGE2{🚀 Runtime Stage}

    STAGE2 --> NGINX[🌐 FROM nginx:latest AS deployer]
    NGINX --> COPY_BUILD[📋 COPY --from=installer /app/build
/usr/share/nginx/html]
    COPY_BUILD --> FINAL[✨ Optimized Final Image]

    FINAL --> SIZE_CHECK{📏 Size Check}
    SIZE_CHECK -->|Before: 200MB+| BEFORE[❌ Single Stage: Bloated]
    SIZE_CHECK -->|After: ~50MB| AFTER[☑️ Multi-Stage: Optimized]

    style STAGE1 fill:#FFE4B5
    style STAGE2 fill:#E0FFFF
    style FINAL fill:#90EE90
    style AFTER fill:#98FB98
```

🔍 Stage Dependency Graph

```
graph TB
    subgraph "📦 Base Images"
        NODE18[🟢 node:18-alpine]
        NGINX[🟢 nginx:latest]
    end

    subgraph "🏗️ Build Stage (installer)"
        INSTALLER[installer stage]
        BUILD_DEPS[📚 Build Dependencies]
        SOURCE_CODE[📄 Source Code]
        BUILD_ARTIFACTS[⚡ Build Artifacts]

        NODE18 --> INSTALLER
        INSTALLER --> BUILD_DEPS
        INSTALLER --> SOURCE_CODE
        BUILD_DEPS --> BUILD_ARTIFACTS
        SOURCE_CODE --> BUILD_ARTIFACTS
    end

    subgraph "🚀 Runtime Stage (deployer)"
        DEPLOYER[deployer stage]
        STATIC_FILES[📁 Static Files Only]
```

```
        NGINX --> DEPLOYER
        BUILD_ARTIFACTS -.->|COPY --from=installer| DEPLOYER
        DEPLOYER --> STATIC_FILES
    end

    subgraph "🗑️ Excluded from Final Image"
        EXCLUDED[❌ node_modules❌ Source code❌ Build tools❌ Dev dependencies]
    end

    style BUILD_ARTIFACTS fill:#90EE90
    style STATIC_FILES fill:#87CEEB
    style EXCLUDED fill:#FFB6C1
```

🛠️ Commands and Best Practices

🔧 Essential Docker Commands

Build Commands

```
# Build multi-stage image
docker build -t multistage .

# Build with specific target stage
docker build --target installer -t build-stage .

# Build with build arguments
docker build --build-arg NODE_VERSION=18 -t multistage .
```

Image Management

```
# List all images
docker images

# Remove specific image
docker image rm multistage

# Remove dangling images
docker image prune

# Check image size
docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"
```

Container Operations

```
# Run container
docker run -d -p 3000:80 --name app-container multistage
```

```
# Check running containers
docker ps

# View container logs
docker logs

# Execute commands in container
docker exec -it  /bin/sh
```

🔍 Debugging and Inspection Commands

```
# Inspect container configuration
docker inspect

# Check container filesystem
docker exec -it  ls -la /usr/share/nginx/html

# Monitor container resource usage
docker stats

# View container port mappings
docker port
```

📊 Performance Comparison

📏 Size Comparison

```
graph LR
    subgraph "📊 Image Size Comparison"
        SINGLE[❌ Single-Stage200MB+]
        MULTI[☑ Multi-Stage~50MB]

        SINGLE --> REDUCTION[75% Size Reduction]
        REDUCTION --> MULTI
    end

    subgraph "⚡ Performance Impact"
        FASTER[🚀 3x Faster Pull]
        SECURE[🔒 Lower Attack Surface]
        COST[💰 Reduced Storage Cost]
    end

    MULTI --> FASTER
    MULTI --> SECURE
    MULTI --> COST

    style SINGLE fill:#FFB6C1
```

```
    style MULTI fill:#90EE90
    style REDUCTION fill:#FFD700
```

Benefits Breakdown

Metric	Single-Stage	Multi-Stage	Improvement
Image Size	200MB+	~50MB	**75% reduction**
Pull Time	30 seconds	10 seconds	**3x faster**
Startup Time	15 seconds	8 seconds	**2x faster**
Security Vulnerabilities	High	Low	**60% fewer**
Storage Cost	High	Low	**75% savings**

Debugging and Troubleshooting

Container Investigation Commands

```
# Check container logs
docker logs

# Access container shell
docker exec -it  /bin/sh

# Inspect container details
docker inspect
```

Inside Container Exploration

```
graph TB
    CONTAINER[ Running Container]

    subgraph " Container Filesystem"
        ROOT[/ (root directory)]
        USR[/usr]
        SHARE[/usr/share]
        NGINX[/usr/share/nginx]
        HTML[/usr/share/nginx/html]
        FILES[ Static Files]

        ROOT --> USR
        USR --> SHARE
        SHARE --> NGINX
        NGINX --> HTML
        HTML --> FILES
    end
```

```
    subgraph "🔍 Inspection Commands"
        LS[ls -la]
        CAT[cat index.html]
        PS[ps aux]
        TOP[top]
    end

    CONTAINER --> ROOT
    FILES --> LS
    FILES --> CAT

    style FILES fill:#90EE90
    style HTML fill:#87CEEB
```

🚨 Common Issues and Solutions

❌ Problem	🔍 Symptom	☑ Solution
Build fails	`unknown instruction WORKDIR`	Check Dockerfile syntax
Large image size	Image still 200MB+	Verify multi-stage is working
Container won't start	Exit code 125	Check port conflicts
Files not found	404 errors	Verify COPY paths
Permission issues	Access denied	Use non-root user

✨ Best Practices

🎯 Multi-Stage Build Best Practices

1. 🏷 Use Named Stages

```dockerfile
# ☑ Good: Named stages
FROM node:18-alpine AS installer
FROM nginx:latest AS deployer

# ❌ Bad: Unnamed stages
FROM node:18-alpine
FROM nginx:latest
```

2. 📦 Choose Optimal Base Images

```dockerfile
# ☑ Good: Lightweight base images
FROM node:18-alpine AS installer    # Small Alpine-based
FROM nginx:alpine AS deployer       # Lightweight nginx
```

```
# ❌ Bad: Heavy base images
FROM node:18 AS installer           # Ubuntu-based (larger)
FROM nginx:latest AS deployer       # Full nginx image
```

3. 🎯 Copy Only What's Needed

```
# ☑ Good: Selective copying
COPY --from=installer /app/build /usr/share/nginx/html

# ❌ Bad: Copying everything
COPY --from=installer /app /usr/share/nginx/html
```

4. 📝 Optimize Layer Caching

```
# ☑ Good: Copy package files first
COPY package*.json ./
RUN npm install
COPY . .

# ❌ Bad: Copy everything first
COPY . .
RUN npm install
```

🔒 Security Best Practices

5. 👤 Use Non-Root User

```
# ☑ Good: Non-root user
FROM nginx:alpine AS deployer
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001
USER nextjs

# ❌ Bad: Running as root (default)
FROM nginx:alpine AS deployer
# No user specified - runs as root
```

6. 🗑 Remove Unnecessary Packages

```
# ☑ Good: Clean up after installation
RUN apt-get update && apt-get install -y \
    package1 \
    package2 \
```

```
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# ❌ Bad: Leave package cache
RUN apt-get update && apt-get install -y package1 package2
```

🚀 Performance Best Practices

7. 🎯 Use Specific Targets

```
# Build only specific stage for testing
docker build --target installer -t build-stage .

# Build final production image
docker build -t production-app .
```

8. 📦 Multi-Architecture Support

```
# Support multiple architectures
FROM --platform=$BUILDPLATFORM node:18-alpine AS installer
# Build process...

FROM --platform=$TARGETPLATFORM nginx:alpine AS deployer
# Runtime setup...
```

🔄 Advanced Multi-Stage Patterns

9. 🧪 Testing Stage

```
# Build stage
FROM node:18-alpine AS installer
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Test stage
FROM installer AS tester
RUN npm test

# Production stage
FROM nginx:alpine AS deployer
COPY --from=installer /app/build /usr/share/nginx/html
```

10. 🔀 Parallel Builds

```
# Base dependencies
FROM node:18-alpine AS base
WORKDIR /app
COPY package*.json ./
RUN npm install

# Frontend build
FROM base AS frontend
COPY frontend/ ./
RUN npm run build:frontend

# Backend build
FROM base AS backend
COPY backend/ ./
RUN npm run build:backend

# Final stage
FROM nginx:alpine AS final
COPY --from=frontend /app/dist /usr/share/nginx/html
COPY --from=backend /app/build /app/api
```

📊 Additional Best Practices

```
mindmap
  root((🎯 Multi-StageBest Practices))
    🏗️ Build Optimization
      📦 Use Alpine images
      🎯 Named stages
      📋 Layer caching
      🗂️ .dockerignore file
    🔒 Security
      👤 Non-root user
      🧹 Clean package cache
      🔍 Minimal attack surface
      🚫 No secrets in layers
    ⚡ Performance
      🔄 Parallel builds
      📏 Smaller final image
      🚀 Faster deployments
      💾 Reduced storage
    🛠️ Maintenance
      📝 Clear documentation
      🏷️ Consistent naming
      🧪 Testing stages
      🔄 CI/CD integration
```

🎓 Summary

Docker Multi-Stage Builds are a **game-changing feature** that revolutionizes container image optimization[1] [2]. By separating build and runtime environments, you can achieve:

🔑 Key Achievements

- 🗜️ **75% smaller images** - From 200MB+ to ~50MB
- 🔒 **Enhanced security** - Reduced attack surface
- ⚡ **Faster deployments** - 3x faster pull times
- 💰 **Cost savings** - Lower storage and bandwidth costs
- 🧹 **Cleaner workflow** - Single Dockerfile for entire process

🚀 Implementation Steps

1. 🏗️ **Design stages** - Separate build and runtime concerns
2. 📦 **Choose base images** - Use lightweight Alpine variants
3. 🎯 **Copy selectively** - Only production artifacts
4. 🔒 **Apply security** - Non-root users, clean packages
5. 📊 **Monitor results** - Measure size and performance improvements

Multi-stage builds represent a **fundamental shift** from monolithic container images to **optimized, production-ready deployments**. They embody the principle of "build fat, ship thin" - using all necessary tools during build time while delivering minimal, secure runtime images[3].

Start implementing multi-stage builds in your projects today to unlock **significant performance gains** and **security improvements** in your containerized applications! 🚀

[1] https://docs.docker.com/build/building/multi-stage/ [2] https://docs.docker.com/get-started/docker-concepts/building-images/multi-stage-builds/ [3] https://docs.docker.com/build/building/best-practices/ [4] https://docs.docker.com/guides/cpp/multistage/ [5] https://dev.to/raunakgurud09/mastering-docker-multistage-builds-1e0m [6] https://dev.to/abhay_yt_52a8e72b213be229/streamline-your-docker-images-with-multi-stage-builds-340c [7] https://depot.dev/blog/docker-multi-stage-builds [8] https://dev.to/citrux-digital/understanding-docker-multistage-builds-3fm7 [9] https://labs.iximiuz.com/tutorials/docker-multi-stage-builds [10] https://dev.to/kalkwst/multi-stage-dockerfiles-3e90 [11] https://ruan.dev/blog/2022/07/31/docker-multistage-builds-for-hugo [12] https://www.cherryservers.com/blog/docker-multistage-build [13] https://github.com/patrickhoefler/dockerfilegraph [14] https://docs.docker.com/build/building/multi-platform/ [15] https://overcast.blog/building-efficient-multi-stage-dockerfiles-for-production-055f34c4baed [16] https://learn.microsoft.com/en-us/dotnet/architecture/microservices/docker-application-development-process/docker-app-development-workflow [17] https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/ [18] https://earthly.dev/blog/docker-multistage/ [19] https://blog.devgenius.io/docker-multi-stage-build-in-detail-3d7da2948797?gi=265baab36942 [20] https://www.youtube.com/watch?v=ajetvJmBvFo