



Docker Image Optimization for Node.js Applications



Unveiling the Art of Docker Image Refinement

Introduction

Welcome to a guide on how to make Docker images smaller and more efficient for Node.js applications. We'll break it down step by step, using simple node-app and practical examples. 🛠️

Table of Contents

1. 📝 Creating a Dockerfile for Node.js.
2. 🛡️ Using Multi-Stage Builds for Efficiency.
3. 📊 Comparing Image Sizes.
4. 🏁 Conclusion.

1. Creating a Dockerfile for Node.js

Let's start by creating a Dockerfile. Think of it as a set of instructions for Docker to build your application. 🧱

```
# Use Node.js version 20 as the base image
FROM node:20

# Set the working directory inside the container to /app
WORKDIR /app

# Copy package.json and package-lock.json to the working directory
COPY package*.json .

# Install dependencies based on package.json
RUN npm install

# Copy all the files from the current directory (where Dockerfile is
# located) to /app
COPY . .

# Expose port 3000 to allow incoming connections
EXPOSE 3000

# Define the default command to run the application
CMD ["node", "index.js"]
```

```
• sakib@The-Silly-Sultan:~/Desktop/Docker-Assignment/docker-roadmap/part-1/1-simple-app$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
node-img      latest    1e3d40f53551   10 seconds ago 1.1GB
```

Original Dockerfile Image: Around 1.1 GB

2. Using Multi-Stage Builds for Efficiency.

Multi-stage builds are a way to make our Docker images smaller. It's like using only the tools you need for a job, nothing more. 🛠️🧰

```
# Stage 1: Build Stage
FROM node:20 AS builder

# Set the working directory inside the container to /app
WORKDIR /app

# Copy package.json and package-lock.json to the working directory
COPY package*.json .

# Install dependencies based on package.json
RUN npm install

# Copy all the files from the current directory (where Dockerfile is
# located) to /app
COPY . .

# Stage 2: Final Stage
FROM node:20-alpine

# Set the working directory inside the container to /app
WORKDIR /app

# Copy the built files from the builder stage to the final image
COPY --from=builder /app /app


# Expose port 3000 to allow incoming connections
EXPOSE 3000

# Define the default command to run the application
CMD ["node", "index.js"]
```

```
• sakib@The-Silly-Sultan:~/Desktop/Docker-Assignment/docker-roadmap/part-1/1-simple-app$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
node-multi-stage-img latest             97c66c96c538       8 seconds ago      184MB
node-img             latest             1e3d40f53551       About an hour ago  1.1GB
```

Multi-Stage Dockerfile Image: Approximately 184 MB.

3. Comparing Image Sizes:


After following the steps, we'll see the difference in image sizes: 

- **Original Dockerfile Image:** Around 1.1 GB
- **Multi-Stage Dockerfile Image:** Approximately 184 MB

 **Percentage Reduction of image:** 83.27%

4. Conclusion:

By using multi-stage builds, we've made our Docker images much smaller. This means faster deployments and less strain on resources. . This means that by using a multi-stage Dockerfile, we have achieved an impressive  83.27% reduction in image size compared to the original Dockerfile.

This guide aims to make Docker image optimization easy to understand. It's designed for anyone looking to make their Node.js applications run more efficiently in Docker. 

Thanks for reading 