Container Network - Multi-stage Docker Build

Multi-stage Builds with a Simple Node.js App

Multi-stage Builds with a Simple Node.js App
In the world of making programs run smoothly inside containers, we've found that using just one step to build them isn't always the best idea. When we stick to single steps, our containers can end up being really big and slow. But fear not! There's a better way: multi-stage builds. These help us make our containers smaller and faster without losing any important stuff.

image-1

Today, we're going on an adventure to discover how multi-stage builds work. We'll break down the complicated bits into easy-to-understand pieces.

Create a Simple Node.js App
Create a directory for our project and navigate into it:

```
mkdir node-app
cd node-app
```

Now, create a file named app.js and add the following code:

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello from Docker! Welcome to our application.\n');
});

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}/`);
});
```

This simple Node.js application creates an HTTP server that listens on port 3000 and responds with "Hello, world!" to all requests.

Create a Dockerfile
Next, let's create a Dockerfile to containerize our Node.js application using two-stage builds:

```
# Stage 1: Build the Node.js application
FROM node:latest AS build

WORKDIR /app

COPY app.js .
```

```
# Stage 2: Create the production image
FROM node:slim

WORKDIR /app

COPY --from=build /app .

EXPOSE 3000

CMD ["node", "app.js"]
```

Stage 1 (build):
Purpose: Build the Node.js application.
Functionality:
Uses node:latest image as the base, providing necessary build tools.
Sets the working directory to /app where application code will be copied.
Copies index.js into the image, preparing for application build.
Stage 2:
Purpose: Create the production image.
Functionality:
Uses node:slim image as the base, providing a minimal runtime environment.
Sets the working directory to /app, ensuring consistency with the build stage.
Copies the built application from the build stage into this stage, excluding unnecessary build tools.
Exposes port 3000, allowing external communication with the application.
Defines the command to run the application (node index.js), starting the HTTP server.
Build and Run the Docker Image
Now, let's build and run the Docker image:

```
docker build -t node-app .
docker run -d -p 3000:3000 node-app
```

Check that the container is running:

```
docker ps
```

This will build the Docker image based on the Dockerfile and run a container based on that image. The application will be accessible at http://localhost:3000.

Testing the App
To test the app, we can use curl to send a request to the running container:

```
curl http://localhost:3000
```

This should return:

Conclusion
In this guide, we've demonstrated how to use two-stage builds in Docker to containerize a simple Node.js application. Each stage has a specific purpose, allowing us to create smaller and more efficient Docker images by separating the build process into distinct steps.

explain these for a simple child in a very easiest manner in details with basics also correct any error

Making a Super Small and Fast Node.js App with Docker 🚀

Hey there! 👋 Today, let's go on an adventure and learn how to make a simple app using **Node.js** and put it inside a **Docker** container in the best way possible! Don't worry; I'll explain everything in the **easiest way** so that even if you are new to this, you can follow along! 😃

🏗️ What is Docker? Why Do We Use It?

Imagine you have a lunchbox 🍱, and you want to take your favorite snacks (your app) with you wherever you go. But sometimes, if you don't pack it properly, it gets too big and heavy. Docker helps us **pack** our app in a very neat and small way so it runs **fast** and **smooth** everywhere. 🚀

But there's a problem... ❌ If we pack everything in one step, our lunchbox (Docker image) becomes **too big and slow**. So, we use **multi-stage builds** to **first prepare** the food and **then pack only the needed parts**! 🍱✨

🏗️ Step 1: Creating a Simple Node.js App

First, we need to make a folder where our app will live. Open your terminal and type:

```sh
mkdir node-app  # Create a folder named 'node-app'
cd node-app     # Move inside the folder
```

Now, let's create a new file inside this folder called **app.js** and write this simple code inside:

```javascript
const http = require('http'); // Import the built-in HTTP module

// Create a simple server that sends a response when someone visits the app
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'}); // Set response type
  res.end('Hello from Docker! Welcome to our application.\n'); // Send message
});

// Set the port where our app will run (default is 3000)
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}/`);
});
```

📌 What this does:

- This creates a **simple server** that listens for people visiting it.
- When they visit http://localhost:3000, it **sends back a message**:
  "Hello from Docker! Welcome to our application."
- It runs on **port 3000**.

🧱 Step 2: Writing the Dockerfile (Our Magic Recipe 📜)

Now, we need to tell **Docker** how to **pack** our app properly. We do this using a file called **Dockerfile**.

Inside the `node-app` folder, create a file named **Dockerfile** and add this:

```
dockerfile

# Stage 1: Build the Node.js application
FROM node:latest AS build  # Use the latest Node.js version

WORKDIR /app  # Create and move to the app directory inside the container

COPY app.js .  # Copy our app.js file into the container

# Stage 2: Create the production image
FROM node:slim  # Use a smaller Node.js image to make it lightweight

WORKDIR /app  # Again, move to the app directory

COPY --from=build /app .  # Copy only the necessary files from Stage 1

EXPOSE 3000  # Open port 3000 so people can visit our app

CMD ["node", "app.js"]  # Run our app when the container starts
```

🧱 Step 3: What is Happening in This Dockerfile? 🤔

🚀 Stage 1: Build

- We use `node:latest`, which has **all** the tools we need to prepare our app.
- We **copy** our `app.js` file inside the container.

📦 Stage 2: Create the Final Version

- We use `node:slim`, which is a **smaller** and **faster** version of Node.js.
- We only **copy the important parts** from Stage 1.
- We tell **Docker** to open **port 3000** so the app is accessible.
- We **run the app** when the container starts.

🧱 Step 4: Building and Running Our App in Docker 🏃‍♂️

Now, let's **build** and **run** our app inside a container!

1️⃣ Build the Docker image:

```
sh

docker build -t node-app .
```

(This takes our code and prepares everything inside a Docker image named `node-app`.)

2️⃣ Run the app in a container:

```
sh

docker run -d -p 3000:3000 node-app
```

(This starts our app in the background and connects it to port 3000.)

3️⃣ **Check if our container is running**:

```
sh

docker ps
```

(This shows a list of running containers.)

4️⃣ **Test the app**:

```
sh

curl http://localhost:3000
```

(This should return: `"Hello from Docker! Welcome to our application."`)

🎉 Conclusion 🎉

- We created a **simple Node.js app**.
- We **packed** it inside a **Docker container**.
- We **used multi-stage builds** to make the final image **smaller** and **faster**. 🚀

By using **multi-stage builds**, we only keep the **important parts**, so our app runs **smoothly** without any extra weight! 🏎️💨

Now, go and try it out! 🛠️ Happy coding! 💻🚀