Multistage Dockerfile

In simple terms, it's a way to streamline your Docker images by using multiple build stages. It helps to keep the final image size minimal, improves security, and accelerates build times. Let's break it down with an example:

Node & Nginx multistage dockerfile example

```
# Stage 1: Build Stage
FROM node:14 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Production Stage
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Explanation of each section:

1. **Build Stage (node:14 AS build):** Here, we use the official Node.js image to build our application. We copy package files, install dependencies, and build the app. The result is stored in the `/app/build` directory.
2. **Production Stage (nginx:alpine):** Now, we switch to a lightweight Nginx image. We copy only the necessary artifacts from the build stage (`/app/build`) into the Nginx web server's HTML directory. This results in a minimal image tailored for production.

Benefits of Multistage Dockerfiles:

1. **Reduced Image Size:** By discarding unnecessary build dependencies, the final image is leaner and more efficient.
2. **Enhanced Security:** Only the final artifacts required for production are included, reducing potential attack vectors.
3. **Faster Builds:** The build process is faster as only the essential steps are repeated in each stage.

Additional Tips:

- 🧹 **Clean Up:** Utilize the `--from=build` option to copy only what's needed. No more unnecessary baggage!
- 🔄 **Reusability:** Each stage can be named (`AS build`), making it easy to reference in subsequent stages.
- 🚀 **Experiment:** Tailor stages to fit your specific project needs. It's all about finding the right balance!

Node multistage Dockerfile along with explanations for each stage:

```
# Stage 1: Build stage
FROM node:14 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Production stage
FROM node:14-alpine
WORKDIR /app
# Copy only necessary files from the build stage
COPY --from=build /app/dist ./dist
COPY --from=build /app/package*.json ./
RUN npm install --production
EXPOSE 3000
CMD ["npm", "start"]
```

Explanation:

1. **Stage 1 (Build stage):**
    - Uses the `node:14` image as the base image for building the application.
    - Sets the working directory to `/app`.
    - Copies only the package.json and package-lock.json files first and installs dependencies.
    - Copies the entire application source code.
    - Builds the application using the specified build command (`npm run build` in this case).
2. **Stage 2 (Production stage):**
    - Switches to a smaller base image (`node:14-alpine`) for the production image.
    - Sets the working directory to `/app`.
    - Copies only the necessary files from the build stage, including the built artifacts from the `dist` directory and the package.json file.
    - Installs only production dependencies using `npm install --production`.

- Exposes the port (in this case, port 3000) that the application runs on.
- Specifies the default command to run the application (`npm start`).

By using multistage builds, you ensure that the final image only contains the necessary artifacts and dependencies for running the application in a production environment. This helps reduce the overall size of the Docker image, making it more lightweight and efficient.

Java Example

```
# Stage 1: Build stage
FROM maven:3.8.4-openjdk-11 AS build
WORKDIR /app

# Copy only the POM file to leverage Docker cache
COPY pom.xml .

# Download dependencies and build the project
RUN mvn dependency:go-offline
RUN mvn package -DskipTests

# Stage 2: Runtime stage
FROM openjdk:11-jre-slim
WORKDIR /app

# Copy the built JAR file from the build stage
COPY --from=build /app/target/your-app.jar .

# Set the entry point for the application
ENTRYPOINT ["java", "-jar", "your-app.jar"]
```

Let's break down the Dockerfile:

1. **Stage 1: Build stage**
    - **`FROM maven:3.8.4-openjdk-11 AS build`**: Use a Maven image with OpenJDK 11 as the base image for building the Java application.
    - **`WORKDIR /app`:** Set the working directory to `/app` within the container.
    - **`COPY pom.xml .`**: Copy only the `pom.xml` file into the container to leverage Docker cache for Maven dependencies.
    - **`RUN mvn dependency:go-offline`**: Download Maven dependencies. This step is separate to take advantage of Docker cache.
    - **`RUN mvn package -DskipTests`**: Build the application. Skipping tests for faster build. The JAR file is generated in the `target` directory.
2. **Stage 2: Runtime stage**
    - **`FROM openjdk:11-jre-slim`**: Use a smaller base image with only the JRE for running the application.
    - **`WORKDIR /app`**: Set the working directory.
    - **`COPY --from=build /app/target/your-app.jar .`**: Copy the JAR file from the build stage to the runtime stage.
    - **`ENTRYPOINT ["java", "-jar", "your-app.jar"]`**: Set the entry point to run the Java application when the container starts.

To build your Docker image, run the following command in the directory containing the Dockerfile:

```
docker build -t your-image-name .
```

Replace `your-app` with your actual application name and adjust the image names accordingly.
This multi-stage Dockerfile efficiently builds a Java application, separating the build environment from the runtime environment, resulting in a smaller final Docker image.