Hands-on Lab: Write a Dockerfile for a Node app deployment

Estimate Time: 25 minutes

Introduction

In this lab, you will explore how generative AI can assist in creating a Dockerfile for a Node.js application. You will utilize IBM's generative AI lab, powered by advanced language models. The goal is to generate the necessary code for the Dockerfile, simplifying the development of the Node.js application in a Docker environment.

Learning Objectives

After completing this exercise, you will be able to:

- Understand and Utilize IBM's generative AI lab to generate appropriate responses for queries related to Dockerfile.
- Develop a Dockerfile using GenAI for a node.js application
- Ensure clarity and precision in providing details when seeking results from ChatGPT

Please note Generative AI is an evolving field. As you attempt the labs, your experience and output might differ from here.

Prologue

What is a Dockerfile, and why use it?

A Dockerfile is like a recipe for your application's environment. It contains instructions that Docker uses to build a Docker image. This image, in turn, is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.

Dockerfiles are crucial in containerization, offering a standardized and reproducible way to package and deploy applications. They allow developers to encapsulate their applications and dependencies, ensuring consistent behavior across different environments. Dockerfiles simplify the development process and facilitates seamless collaboration and deployment.

Start generating Dockerfile

In case you need familiarity with the Interface/classroom please see the This Lab for reference.

Once your AI Classroom is ready, let's proceed and ask Generative AI for assistance writing a Dockerfile.

1. Enter the following question in the message box: `What is Dockerfile ?`

Now, view the AI's response for an explanation of a Dockerfile.

▼ Click here to view the sample response generated

A Dockerfile is a text document that contains a series of instructions on how to build a Docker image. It specifies the base image to use, the application code to include, and the commands to run during the image creation process. Dockerfiles enable developers to automate the process of creating Docker images, ensuring that the environment is consistent across different deployments.

Here are some key components of a Dockerfile:

1. **FROM**: Specifies the base image to use for your application.
2. **COPY**: Copies files from your local filesystem into the image.
3. **RUN**: Executes commands in the shell to install packages or set up the environment.
4. **CMD**: Specifies the default command to run when a container is started from the image.
5. **EXPOSE**: Indicates the ports that the container listens on at runtime.

To build an image from a Dockerfile, you would typically use the command `docker build -t <image-name> .` in the directory containing the Dockerfile.

For more advanced use, you can also use multi-stage builds to reduce the final image size, leverage caching for faster builds, and define environment variables using the `ENV` instruction.

Understanding Dockerfiles is essential for containerization and microservices architecture, as they enable consistent and reproducible application deployments.

2. Enter the following question in the message box: `What are the requirements for a Dockerfile?`

Next, read the AI's response and proceed to ask questions about Dockerfile requirements.

▼ Click here to view the sample response generated

A Dockerfile has specific requirements and best practices to ensure it functions correctly and efficiently. Here are the key requirements for a Dockerfile:

1. **Syntax**: The Dockerfile must follow a specific syntax, using instructions like `FROM`, `COPY`, `RUN`, etc. Each instruction should be on a new line, and comments can be added using the `#` symbol.

2. **Base Image**: The first instruction in a Dockerfile should typically be `FROM`, which specifies the base image. This image can be an official image from Docker Hub or a custom image.

3. **File Structure**: The Dockerfile must be named `Dockerfile` (case-sensitive) and should be located in the directory from which you plan to build the image.

4. **Commands**: Each instruction in the Dockerfile should be valid. For example, `RUN` commands must be valid shell commands, and `COPY` should reference files that exist in the build context (the directory containing the Dockerfile).

5. **Build Context**: When you build a Docker image, you must provide a build context (usually the directory containing the Dockerfile). This context includes all files that are referenced in the Dockerfile.

6. **Permissions**: Ensure that the Docker daemon has appropriate permissions to read the Dockerfile and any files it references.

7. **Best Practices**:

    - **Use Specific Tags**: Instead of using `latest` for base images, specify a version tag to avoid unexpected changes.
    - **Minimize Layers**: Combine commands where possible to reduce the number of layers in the final image, improving build performance and image size.
    - **Clean Up**: Remove unnecessary files or dependencies in the same `RUN` command to keep the image size down.

- **Use .dockerignore**: Create a `.dockerignore` file to exclude files and directories from the build context that are not needed in the image, which can help reduce build time and image size.

By adhering to these requirements and best practices, you can create efficient and functional Dockerfiles that facilitate the containerization of applications.

3. Enter the following question in the message box: `What's the first crucial step when creating a Dockerfile for a Node.js application?`

It is essential to understand the first step after selecting the base image for a Node.js Dockerfile.

▼ Click here to view the sample response generated

The first crucial step when creating a Dockerfile for a Node.js application is to specify the base image using the `FROM` instruction. For Node.js applications, you would typically choose an official Node.js image from Docker Hub as your base image.

Here's an example of how to start your Dockerfile for a Node.js application:

```
FROM node:16
```

In this example, `node:16` specifies that you want to use the official Node.js image with version 16. This ensures that your application runs in an environment that has Node.js pre-installed.

After specifying the base image, the next steps usually include:

1. **Setting the Working Directory**: Use the `WORKDIR` instruction to set the directory where your application code will reside.

   ```
   WORKDIR /app
   ```

2. **Copying Package Files**: Copy your `package.json` and `package-lock.json` files to the working directory to ensure that you can install your application's dependencies.

   ```
   COPY package*.json ./
   ```

3. **Installing Dependencies**: Use the `RUN` instruction to install your application dependencies.

   ```
   RUN npm install
   ```

4. **Copying Application Code**: Copy the rest of your application code into the image.

   ```
   COPY . .
   ```

5. **Exposing the Port**: If your application listens on a specific port, use the `EXPOSE` instruction to document this.

   ```
   EXPOSE 3000
   ```

6. **Specifying the Command**: Finally, use the `CMD` instruction to specify the command that should run when the container starts.

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

By following these steps, you can create a well-structured Dockerfile that effectively sets up your Node.js application in a Docker container.

4. Enter the following instructions in the message box: `Following the base image selection, Set the working directory inside the container to /app`

Then, you will learn about the next step to define the working directory inside the container.

▼ Click here to view the sample response generated

To set the working directory inside the Docker container to `/app`, you would use the `WORKDIR` instruction in your Dockerfile. This instruction establishes the directory where subsequent commands will be executed and where your application code will reside.

Here's how you can add it to your Dockerfile after selecting the base image:

```
FROM node:16
# Set the working directory inside the container to /app
WORKDIR /app
```

With this setup, any subsequent commands, such as `COPY`, `RUN`, or `CMD`, will be executed in the `/app` directory within the container. This helps to keep your application organized and ensures that all files and dependencies are correctly placed in the designated working directory.

After setting the working directory, you can continue to copy your application files and install dependencies as previously described. Here's an example of what the Dockerfile might look like with the working directory set:

```
FROM node:16
# Set the working directory inside the container to /app
WORKDIR /app
# Copy package.json and package-lock.json
COPY package*.json ./
# Install dependencies
RUN npm install
# Copy the rest of the application code
COPY . .
# Expose the port the app runs on
EXPOSE 3000
# Command to run the application
CMD ["node", "app.js"]
```

This structure ensures that your Node.js application is set up correctly within the Docker container.

5. Enter the following instructions in the message box: `As selected working directory /app, bring the package.json and package-lock.json to the working directory`

After that, you will understand the process of bringing package files into the working directory.

▼ Click here to view the sample response generated

To bring the `package.json` and `package-lock.json` files into the working directory (`/app`) in your Dockerfile, you would use the `COPY` instruction. This instruction copies files from your local filesystem into the specified location in the Docker image.

Here's how you can add the `COPY` command to your Dockerfile after setting the working directory:

```
FROM node:16
# 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 ./
```

In this example, `COPY package*.json ./` copies both `package.json` and `package-lock.json` from your local directory to the `/app` directory inside the container. The `*` wildcard ensures that both files are included, assuming they are in the same directory as your Dockerfile.

After copying the package files, you can proceed to install the dependencies using the `RUN` instruction:

```
# Install dependencies
RUN npm install
```

Here's the complete Dockerfile with these steps:

```
FROM node:16
# 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
RUN npm install
# Copy the rest of the application code
COPY . .
# Expose the port the app runs on
EXPOSE 3000
# Command to run the application
CMD ["node", "app.js"]
```

This setup ensures that your Node.js application has access to its dependencies when running in the Docker container.

6. Enter the following questions in the message box: `Now, how do I install dependencies? And copy the remaining application code to the working directory?`

You will then get guidance on installing dependencies and copying the remaining application code.

▼ Click here to view the sample response generated

To install dependencies and copy the remaining application code to the working directory in your Dockerfile, you would use the `RUN` instruction to install the dependencies and the `COPY` instruction to copy your application code.

Here's how you can do it, following the previous steps in your Dockerfile:

1. **Install Dependencies**: Use the `RUN` instruction to execute the command that installs your application's dependencies. This typically involves running `npm install`.
2. **Copy Remaining Application Code**: After installing the dependencies, use the `COPY` instruction to copy the rest of your application code into the working directory.

Here's how the updated Dockerfile would look:

```
FROM node:16
# 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
RUN npm install
# Copy the rest of the application code to the working directory
COPY . .
# Expose the port the app runs on
EXPOSE 3000
# Command to run the application
CMD ["node", "app.js"]
```

Explanation of Each Step:

- **`RUN npm install`**: This command installs all the dependencies listed in your `package.json` file. It creates a `node_modules` directory in the `/app` directory of your container, where all the installed packages will reside.
- **`COPY . .`**: This command copies all files and directories from your local directory (where the Dockerfile is located) into the current working directory in the container, which is `/app`. This includes your application code, assets, and any other necessary files.

After these steps, your Docker container will have all the necessary dependencies installed and the application code copied, making it ready to run. You can then build your Docker image and run the container as needed.

7. Enter the following question in the message box: `Node.js app needs to be accessible on a specific port. What Dockerfile instruction do you include for this purpose?`

Then, you will learn the instructions to include for exposing a specific port.

▼ Click here to view the sample response generated

To make a Node.js application accessible on a specific port in a Docker container, you would use the `EXPOSE` instruction in your Dockerfile. This instruction informs Docker that the container will listen on the specified network port at runtime.

For example, if your Node.js application listens on port 3000, you would include the following line in your Dockerfile:

```
EXPOSE 3000
```

Here's how it fits into the complete Dockerfile:

```
FROM node:16
# 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
RUN npm install
# Copy the rest of the application code to the working directory
COPY . .
# Expose the port the app runs on
EXPOSE 3000
# Command to run the application
CMD ["node", "app.js"]
```

Explanation:

- **`EXPOSE 3000`**: This line tells Docker that the application will be using port 3000. While it does not actually publish the port to the host machine, it serves as documentation for anyone reading the Dockerfile and allows you to use the `-p` flag when running the container to map the container's port to a port on the host.

Running the Container:

When you run your Docker container, you can map the exposed port to a port on your host machine using the `-p` option. For example:

```
docker run -p 3000:3000 <image-name>
```

This command maps port 3000 of your host to port 3000 of the container, allowing you to access your Node.js application via `http://localhost:3000`.

8. Enter the following question in the message box: `How to specify the command to start the Node.js application using CMD?`

After that, you will understand the final step in specifying the command to start the Node.js application.

▼ Click here to view the sample response generated

To specify the command to start your Node.js application in the Dockerfile, you would use the `CMD` instruction. This instruction defines the default command that runs when a container is started from the image.

For a Node.js application, you typically want to run your main application file (commonly `app.js`, `server.js`, or similar). Here's how you would include the `CMD` instruction in your Dockerfile:

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

This line indicates that when the container starts, it should execute the command `node app.js`.

Here's how the complete Dockerfile would look, including the `CMD` instruction:

```
FROM node:16
# 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
RUN npm install
# Copy the rest of the application code to the working directory
COPY . .
# Expose the port the app runs on
EXPOSE 3000
# Command to run the application
CMD ["node", "app.js"]
```

Explanation of `CMD`:

- **`CMD ["node", "app.js"]`**: This syntax uses the exec form of CMD, which is the preferred method. It allows you to specify the command and its arguments as a JSON array. This form avoids issues with shell interpretation and ensures that signals are correctly forwarded to the application.

Alternative Syntax:

You can also use the shell form of CMD, which looks like this:

```
CMD node app.js
```

However, the exec form is generally recommended for better handling of signals and arguments.

Running the Container:

When you build and run your Docker container, it will automatically execute the command specified in the `CMD` instruction, starting your Node.js application. For example:

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

This will build your Docker image and run the container, making your Node.js application accessible on port 3000.

9. Enter the following instructions in the message box: `Combine all the steps to form a complete Dockerfile for a Node.js application`

Now, you have completed all of the steps so you can view the generated Dockerfile.

▼ Click here to view the sample response generated

Here's a complete Dockerfile for a Node.js application that incorporates all the steps we've discussed:

```
# Use the official Node.js image as the base image
FROM node:16
# 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
RUN npm install
# Copy the rest of the application code to the working directory
COPY . .
# Expose the port the app runs on
EXPOSE 3000
# Command to run the application
CMD ["node", "app.js"]
```

Explanation of Each Section:

1. **Base Image**:
   - `FROM node:16`: This line specifies that the Docker image will be based on the official Node.js image, version 16.
2. **Working Directory**:
   - `WORKDIR /app`: Sets the working directory inside the container to `/app`, where all subsequent commands will be executed.
3. **Copy Package Files**:
   - `COPY package*.json ./`: Copies both `package.json` and `package-lock.json` into the working directory to ensure that dependencies can be installed.
4. **Install Dependencies**:
   - `RUN npm install`: Installs the application dependencies listed in `package.json`.
5. **Copy Application Code**:
   - `COPY . .`: Copies the rest of the application code from your local directory into the `/app` directory in the container.
6. **Expose Port**:
   - `EXPOSE 3000`: Informs Docker that the application will listen on port 3000 at runtime.

7. **Command to Start the Application**:
    - `CMD ["node", "app.js"]`: Specifies the command to run when the container starts, which in this case is to execute `app.js` using Node.js.

Building and Running the Docker Container:

To build and run your Docker container using this Dockerfile, you would execute the following commands in your terminal:

1. Build the Docker image:

```
docker build -t my-node-app .
```

2. Run the Docker container:

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

This setup will create a Docker container that runs your Node.js application, making it accessible at `http://localhost:3000`.

Summary

In this lab, you leveraged generative AI to simplify the creation of a Dockerfile for a Node.js application. You did this by interacting with the AI in the designated classroom and addressing key questions and steps involved in Dockerfile development. The generated Dockerfile serves as a foundation for deploying Node.js applications in a containerized environment. This experience showcases the potential of generative AI in enhancing the efficiency of Docker-based development workflows. Through guided interactions, you and othere can gain valuable insights into the Dockerfile creation process, making it more accessible and practical for real-world application deployments.

Congratulations!

You have leveraged generative AI and developed a complete Dockerfile for a Node.js application.

Author(s)

Nikesh Kumar
Pallavi Rai

Skills Network

IBM Corporation 2023. All rights reserved.