

#90DaysOfDevOps Challenge - Day 17 - Docker Project for DevOps Engineers

Welcome to Day 17 of the #90DaysOfDevOps challenge. Today, we will dive into Docker and explore a Docker project for DevOps Engineers. In this project, we will learn about Dockerfile, build images, run containers, and push images to repositories.

Dockerfile

A Dockerfile is a text file that contains a set of instructions to build a Docker image. It provides a repeatable and automated way to package your application and its dependencies into a container. Let's go through the essential components of a Dockerfile:

Base Image

The base image is the starting point for your Docker image. It provides the underlying operating system and environment for your application. You can choose from a wide range of pre-built base images available on Docker Hub, such as Alpine Linux, Ubuntu, or specific language runtimes like Node.js or Python.

```
FROM node:14
```

In the example above, we are using the official Node.js base image with version 14. This image includes Node.js and its runtime environment, which is suitable for running Node.js applications.

Working Directory

The working directory is the location inside the container where your application code will be copied. It is a good practice to set the working directory to a specific path.

```
WORKDIR /app
```

In this case, we set the working directory to `/app`.

Copy Files

Next, you need to copy your application code and any necessary files to the working directory in the container. This ensures that the container has all the required files to run your application.

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

The above line copies the `package.json` and `package-lock.json` files from the host directory to the current working directory in the container.

Install Dependencies

After copying the necessary files, you can install your application's dependencies using package managers like npm or pip.

```
RUN npm install
```

This command installs the Node.js dependencies based on the `package.json` file.

Copy Application Code

Once the dependencies are installed, you can copy the rest of your application code to the working directory in the container.

```
COPY . .
```

This line copies all the files and folders from the host directory to the current working directory in the container.

Expose Port

If your application listens on a specific port, you need to expose that port in the Dockerfile.

```
EXPOSE 3000
```

In this example, we expose port 3000, which is the default port for a Node.js web application.

Define Command

Finally, you need to define the command that will be executed when the container starts.

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

This command specifies that the `node` command should be executed, and the `app.js` file should be the entry point for your application.

By combining these instructions in a Dockerfile, you can build an image that encapsulates your application and its dependencies. This image can be run as a container on any system with Docker installed, providing a consistent and isolated environment for your application.

Dockerfile Example

```
# Use the official Node.js base image with version 14
FROM node:14

# Set the working directory to /app
WORKDIR /app

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

# Install the Node.js dependencies
RUN npm install

# Copy the application code to the working directory
COPY . .

# Expose port 3000
EXPOSE 3000

# Define the command to start the application
CMD ["node", "app.js"]
```

In the example above, we start with the official Node.js base image with version 14. We set the working directory to `/app` and copy the `package.json` and `package-lock.json` files to the working directory. Then, we install the Node.js dependencies using `npm install`. Next, we copy the rest of the application code to the working directory. We expose port 3000 to allow external access to the application running inside the container. Finally, we define the command to start the application, specifying `node app.js` as the entry point.

This example covers the essential elements of a Dockerfile, including the base image selection, working directory setup, copying files, installing dependencies, exposing ports, and defining the command. You can modify this example according to your specific application requirements, such as using a different base image, exposing different ports, or changing the entry point command.

Task 1 - Create a Dockerfile for a simple web application (e.g. a Node.js or Python app)

```
# app.py

# Importing the Flask class from the flask module
from flask import Flask

# Creating an instance of the Flask application
app = Flask(__name__)

# Defining a route decorator for the root URL ("/")
@app.route('/')
def hello():
    # Returning the string "Hello, World!" as the response to the request
    return "Hello, World!"
```

```
# Checking if the script is being executed directly (not imported as a
module)
if __name__ == '__main__':
    # Running the Flask application
    # Starts a development server on the specified host (0.0.0.0) and port
    (3000) to handle incoming requests
    app.run(host='0.0.0.0', port=3000)
```

```
# Use the official Python 3 base image
FROM python:3

# Set the working directory inside the container
WORKDIR /app

# Install Flask library using pip
RUN pip install flask

# Copy the app.py file from the local directory to the working directory
inside the container
COPY app.py /app/

# Expose port 3000 to allow external access
EXPOSE 3000

# Set the command to run when the container starts
CMD [ "python", "./app.py" ]
```

This Dockerfile and Python code demonstrate containerizing a simple Flask web application. The Dockerfile sets up the environment and exposes port 3000. The Python code creates a Flask app that responds with "Hello, World!" for the root URL. By building and running the Docker image, the Flask app becomes accessible on port 3000.

```
esteban@Estebans-MacBook-Pro: ~/Library/CloudStorage/GoogleDrive-morenoramirezesteban@gmail.com/My Drive/GitHub/90DaysOfDevOps/2023/day17/dockerfile [master] docker build -t my-web-app .
[+] Building 0.9s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 151B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/python:3
=> [1/4] FROM docker.io/library/python:3@sha256:bc621d7364e291d9c8a8221b936d4f8b7ccea8d2c4d17cc9efdcc8a60f8e31f7
=> [internal] load build context
=> => transferring context: 330B
=> CACHED [2/4] WORKDIR /app
=> CACHED [3/4] RUN pip install flask
=> [4/4] COPY app.py /app/
=> exporting to image
=> => exporting layers
=> => writing image sha256:1de670e8052d8907c21caf27186f3f90f990427aa5847997462e2258d740f7da
=> => naming to docker.io/library/my-web-app
```

Task 2 - Build the image using the Dockerfile and run the container

First, we build the docker image from the Dockerfile

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

Then, we run a container using the image created

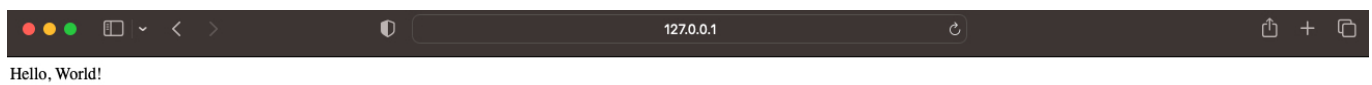
```
docker run -d -p 3000:3000 my-web-app
```

We can check if the docker container is running by running `docker ps`

```
esteban@Estebans-MacBook-Pro ~/Library/CloudStorage/GoogleDrive-morenoramirezesteban@gmail.com/My Drive/GitHub/90DaysOfDevOps/2023/day17/dockerfile [master] docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                    NAMES
1d598de259b3   my-web-app    "python ./app.py"       2 seconds ago Up 1 second    0.0.0.0:3000->3000/tcp   vigorous_kapitsa
```

Task 3 - Verify that the application is working as expected by accessing it in a web browser

We can verify if the application is running by accessing <http://127.0.0.1:3000/> in our preferred browser

A screenshot of a web browser window. The address bar shows '127.0.0.1'. The page content displays 'Hello, World!'.

Task 4 - Push the image to a public or private repository (e.g. Docker Hub)

Log in to Docker Hub using the `docker login` command:

```
docker login
```

Tag your local image with the repository name:


```
docker tag my-web-app estebanmorenoit/my-web-app
```

Push the image to Docker Hub:

```
docker push estebanmorenoit/my-web-app
```

```
esteban@Estebans-MacBook-Pro ~/Library/CloudStorage/GoogleDrive-morenoramirezesteban@gmail.com/My Drive/GitHub/90DaysOfDevOps/2023/day17/dockerfile [master] docker tag my-web-app estebanmorenoit/my-web-app
esteban@Estebans-MacBook-Pro ~/Library/CloudStorage/GoogleDrive-morenoramirezesteban@gmail.com/My Drive/GitHub/90DaysOfDevOps/2023/day17/dockerfile [master] docker push estebanmorenoit/my-web-app
Using default tag: latest
The push refers to repository [docker.io/estebanmorenoit/my-web-app]
b385b086f13b: Pushed
8db7a82d05d4: Pushed
310a6187d092: Pushed
38503bec00e: Mounted from library/python
71e951de0628: Mounted from library/python
0eb817dfc4e1: Mounted from library/python
ea9a66bcf3b5: Mounted from library/python
d148428135e3: Mounted from library/python
b4b4f5c5ff9f: Mounted from library/python
b0df24a95c88: Mounted from library/python
974e52a24adf: Mounted from library/python
latest: digest: sha256:eed8c9b8a7e31479ff130fc372d48b642a5dea568c78a245d51e9daab755ac81 size: 2631
```

Explore > [estebanmorenoit/my-web-app](#)



estebanmorenoit/my-web-app ☆


By [estebanmorenoit](#) • Updated 8 minutes ago

Image

[Manage Repository](#)

↓ Pulls 0

Overview Tags



No overview available

This repository doesn't have an overview

Docker Pull Command

```
docker pull estebanmoreno...
```

[Here](#) you can find '**my-web-app**' repository after following the above steps

Congratulations on completing Day 17 of the #90DaysOfDevOps challenge. Today, we explored Docker and its capabilities for containerization and management. We gained practical experience in building, running, and pushing Docker images.

As we progress to Day 18, we have an exciting Docker project lined up specifically for DevOps Engineers. This project will introduce us to Docker Compose, a powerful tool for orchestrating multi-container applications. We will learn how to define and manage complex application stacks using a simple YAML file. Stay tuned!