

A Beginner's Guide to Docker

Pruteanu Robert-Andrei

What is Docker and Why is it Useful?

Docker is an open-source platform that enables developers to build, ship, and run applications inside isolated environments called containers. Containers package an application and its dependencies together, ensuring consistent behavior across different environments.

Images and Containers

Docker Images

- A Docker image is a lightweight, standalone package containing everything an application needs to run, such as the code, runtime, libraries, and configurations.
- An image is a blueprint for creating a container.
- Images are immutable; changes require building a new image.

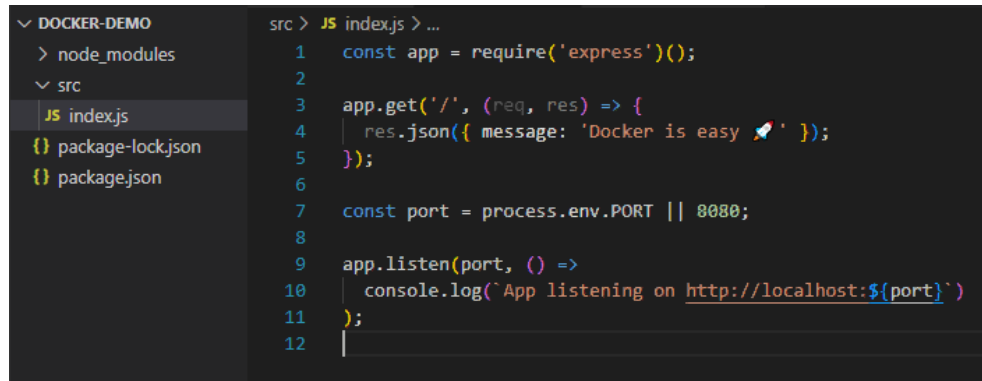
Docker Containers

- A container is a running instance of a Docker image.
- Containers are isolated environments with their own filesystem, processes, and network.
- Containers can be stopped, started, and removed without affecting the image.

How to set up docker

1. Project setup

Here I have a simple index.js file, that exposes an API endpoint, that sends a response “Docker is easy”



```

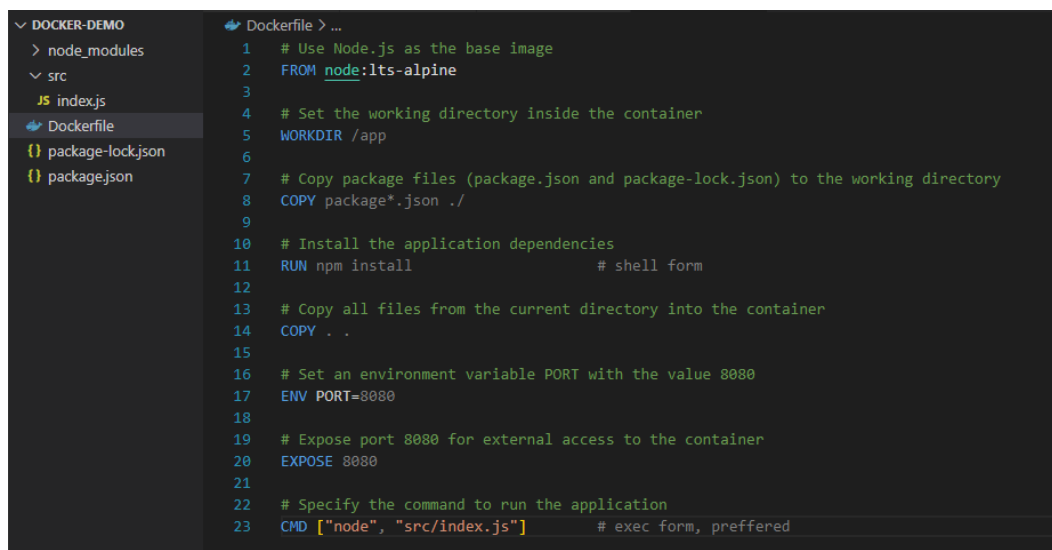
▼ DOCKER-DEMO
  > node_modules
  ▼ src
    JS index.js
    {} package-lock.json
    {} package.json

src > JS index.js > ...
1  const app = require('express')();
2
3  app.get('/', (req, res) => {
4    res.json({ message: 'Docker is easy 🚀' });
5  });
6
7  const port = process.env.PORT || 8080;
8
9  app.listen(port, () =>
10   console.log(`App listening on http://localhost:${port}`)
11 );
12 |
```

We start by creating a “Dockerfile”, in the root of the project. A Dockerfile is a text file that defines the steps required to create a Docker image.

Key Instructions:

- FROM: Specifies the base image.
- COPY: Copies files into the image.
- RUN: Executes commands during image building.
- CMD: Specifies the default command to run when the container starts.
- ENV: Sets environment variables within the Docker container
- EXPOSE: The container will listen for incoming connections on a specified port at runtime.
- WORKDIR: Sets the working directory for subsequent instructions in the Dockerfile



```

▼ DOCKER-DEMO
  > node_modules
  ▼ src
    JS index.js
    Dockerfile
    {} package-lock.json
    {} package.json

Dockerfile > ...
1  # Use Node.js as the base image
2  FROM node:lts-alpine
3
4  # Set the working directory inside the container
5  WORKDIR /app
6
7  # Copy package files (package.json and package-lock.json) to the working directory
8  COPY package*.json ./
9
10 # Install the application dependencies
11 RUN npm install          # shell form
12
13 # Copy all files from the current directory into the container
14 COPY . .
15
16 # Set an environment variable PORT with the value 8080
17 ENV PORT=8080
18
19 # Expose port 8080 for external access to the container
20 EXPOSE 8080
21
22 # Specify the command to run the application
23 CMD ["node", "src/index.js"]  # exec form, preferred
```

2. Building an Image

We build an image by running the Docker “build” command.

There are a lot off available options we can pass with the command but the one we will use right now is tag, or **-t**. This will give our image a tag so that it’s easier to remember.

```
\DOCKER-DEMO> docker build -t docker-demo .
```

“docker-demo” is the tag, the period (.) is the path to the dockerfile, in our case being the current working directory.

Upon running it, we can notice it goes step-by-step through each step written in the Dockerfile.

```
=> => extracting sha256:8a0c646a09aa4fafc9a6dc812e46c6f3e2753f0f7c63ab019bdae73bc485e106
=> [2/5] WORKDIR /app
=> [3/5] COPY package*.json ./
=> [4/5] RUN npm install          # shell form
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:5a2f349522b32e5d31fc0d3ec36c841fa682940178ced59e6542fd943f8c8afd
=> => naming to docker.io/library/docker-demo
```

Now that we have this image, we can use it to build other images, or use it to run containers. In real life, to use this image, we’d most likely push it to a container registry somewhere (DockerHub), using “docker push”. Then, a developer can use “docker pull” to pull that image back down.

To keep things simple, we’ll run this locally on our system in the next step.

3. Running a container

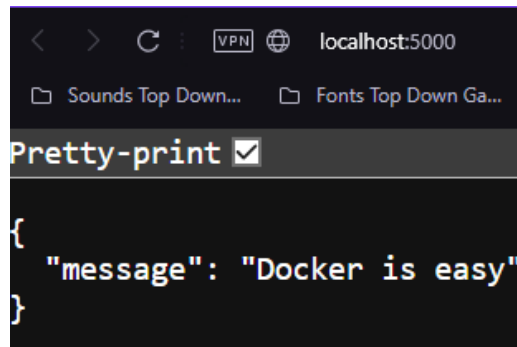
To run the image we can use the “docker run” command, supplied by the image id or tag name (in our case “docker-demo”).

This creates a running process called a Container.

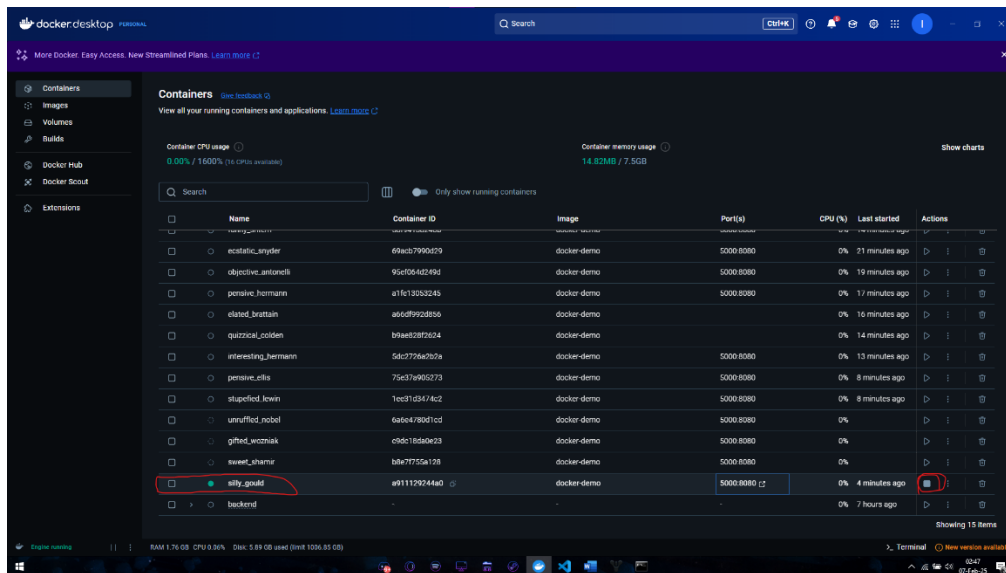
We will also use need to use tort forwarding flag (-p <host_port>:<container_port>). It is necessary when you want to make a service running inside a Docker container accessible from your host system or network.

```
\DOCKER-DEMO> docker run -p 5000:8080
```

Now, if we open the browser and go to localhost:5000:

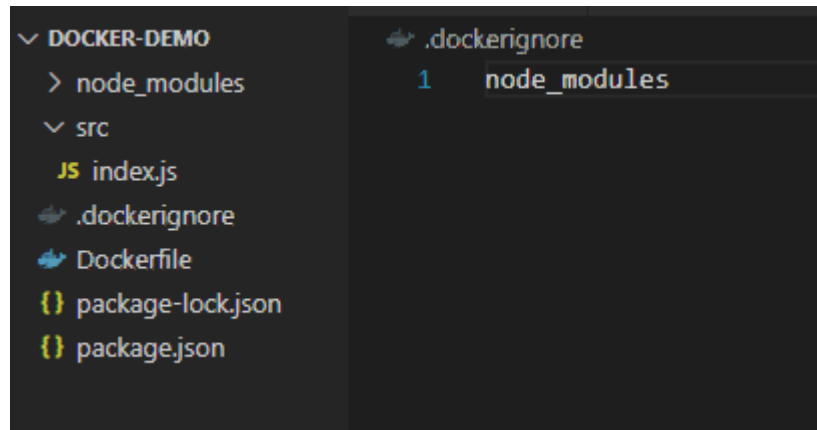


One thing to note is that the docker container will still be running, even after closing the terminal window. To stop it, we can open the Docker dashboard window and close it from there:



One last useful thing we can do is to create a `.dockerignore` file, that works in a similar way to a `.gitignore` file.

For instance, we do not want the `node_modules` folder included in docker, so we do this:



The image shows a code editor interface. On the left is a file explorer for a project named 'DOCKER-DEMO'. It contains a 'node_modules' folder, a 'src' folder with an 'index.js' file, and several other files: '.dockerignore', 'Dockerfile', 'package-lock.json', and 'package.json'. On the right, the '.dockerignore' file is open and contains a single line of text: 'node_modules'.

```
✓ DOCKER-DEMO
  > node_modules
  ✓ src
    JS index.js
    .dockerignore
    Dockerfile
    {} package-lock.json
    {} package.json
```

```
.dockerignore
1  node_modules
```