Rest API and Docker

- Introduction to REST
- Creating RESTful APIs with Express.js
- Handling HTTP methods and status codes
- Deploying Node.js applications
- Docker and Containerization with Docker

What is an API?

- API stands for Application Programming Interface. An API establishes a connection between programs so they can transfer data.
- A program that has an API implies that some parts of its data is exposed for the client to use. The client could be the frontend of the same program or an external program.
- In order to get this data, a structured request has to be sent to the API. If the request meets the desired requirements, a response which contains the data gets sent back to where the request was made. This response usually comes in the form of JSON or XML data.

What is REST?

- REST stands for REpresentational State Transfer.
- It is a standard that guides the design and development of processes which enable us interact with data stored on a web servers.
- We are able to communicate with servers using the HTTP protocol. With these protocols, we can **Create**, **Read**, **Update** and **Delete** data – otherwise known as **CRUD** operations.
- how can we perform these CRUD operations and communicate with data on the server?
- We can do this by sending HTTP requests, and that is where REST comes in. REST simplifies the communication process by providing various HTTP methods/operations/verbs which we can use to send requests to the server.

How to communicate with a server using REST APIs

REST APIs make the communication process with the server easier for us by giving us various HTTP request methods. The most commonly used methods are:

- **GET**: The get method is used to **Read** data on the server.
- **POST**: The post method is used to **Create** data.
- **PATCH/PUT**: The patch method is used to **Update** data.
- **DELETE**: The delete method is used to **Delete** data.

- These methods provided by REST allow us perform CRUD operations easily. That is:
- Create => POST.
  Read => GET.
  Update => PATCH/PUT.
  Delete => DELETE.

- **GET**: Retrieve data from the server.
- **POST**: Create new data on the server.
- **PUT**: Update existing data on the server.
- **PATCH**: Partially update existing data on the server.
- **DELETE**: Delete data from the server

Creating RESTful APIs with Express.js

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware to parse JSON request bodies
app.use(express.json());

// In-memory data for users (this would normally be a database)
let users = [
  { id: 1, name: 'John Doe', email: 'john.doe@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane.smith@example.com' },
];

// GET all users
app.get('/api/users', (req, res) => {
  res.json(users);
});
```

```
// GET a single user by ID
app.get('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id, 10);
  const user = users.find((u) => u.id === userId);

  if (user) {
    res.json(user);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});
```

```
// POST a new user
app.post('/api/users', (req, res) => {
  const { name, email } = req.body;
  const newUser = {
    id: users.length + 1,  // In a real app, ID would be auto-generated by the database
    name,
    email,
  };
  users.push(newUser);
  res.status(201).json(newUser); // Return the new user with a 201 status (created)
});
```

```
// PUT (update) a user by ID
app.put('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id, 10);
  const { name, email } = req.body;
  const userIndex = users.findIndex((u) => u.id === userId);

  if (userIndex !== -1) {
    users[userIndex] = { id: userId, name, email };
    res.json(users[userIndex]);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});
```

```
// DELETE a user by ID
app.delete('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id, 10);
  const userIndex = users.findIndex((u) => u.id === userId);

  if (userIndex !== -1) {
    users.splice(userIndex, 1); // Remove the user from the array
    res.status(204).end(); // No content (successful deletion)
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Handling HTTP methods and status codes

- HTTP status codes are three-digit numbers returned by servers to indicate the status of a client’s request.

- They are categorized into five classes, each indicating a specific type of response. Here's an overview:

➢ Informational responses (100 – 199),

➢ Successful responses (200 – 299),

➢ Redirection messages (300 – 399),

➢ Client error responses (400 – 499), and

➢ Server error responses (500 – 599)1234.

Some common HTTP status codes include:

- **200 (Success/OK)**: The requested page has been fetched successfully.
- **301 (Permanent Redirect)**: The requested resource has been moved permanently.
- **302 (Temporary Redirect)**: The requested resource has been temporarily moved.
- **400 (Bad Request)**: The client's request was malformed.
- **404 (Not Found)**: The requested resource was not found.

Deploying Node.js applications(Heroku and Nodejitsu)

- Github is a storage place or platform allows us to store the projects.
- Github allows multiple developers to work together.

- Def : GitHub is a platform which collaborate multiple developers work together to do changes to the application.
- Git is a version control tool which is used to manage the versions of our project.

Git cont..

- Creating a repository: git init is used to initialize empty git repository(local)
- Gitstatus: command is used to check the status of our repository

Note : gitstatus files in red colour-untracked files/files are not in staging area.

Staging Area:

Staging area is a place where we store the changes which should be committed.

Too send the changes to staging area we use a command called Git add.

e.g: git add filename

git add

Git cont..

- If we want to send only single file in staging area instead of all.

  git add style.css

  (filename which we want to add)

If we want to send two files to staging area

  git add index.html style.css

Git commit –m “changes done for index and style file”

Git commit is used to save the changes permanently to the local repo

Git commit –m “message”

Docker and Containerization with Docker

1. The Software Development Life Cycle (SDLC)

- Software development is a journey with different stages: planning, development, testing, deployment, and monitoring.
- Today's focus: Development (writing code) and Deployment (making it accessible to users).
- Challenges exist in both stages.

2. Challenges in a "Dockerless" World

Development Challenges:

- **Inconsistent Environments:** Developers use different operating systems, leading to compatibility issues.
- **Dependency Hell:** Conflicting software versions cause unexpected errors.
- **Example:** A Java application relying on a specific version of OpenSSL might not work correctly on different systems.

Challenges in a "Dockerless" World Cont..

Deployment Challenges:

- **Manual Configuration:** Setting up servers and deploying code is time consuming and error-prone.
- **Scalability Issues:** Handling increased traffic can be complex and expensive.
- **Downtime during Updates:** Deploying new versions can disrupt service for users.

Introducing Docker

Docker: A platform that packages applications and dependencies into containers.

Analogy: Like a shipping container, it carries everything the application needs to run anywhere.

Benefits:

- **Consistency:** Application runs the same way everywhere.
- **Efficiency:** Lightweight and uses fewer resources.
- **Isolation:** Prevents conflicts and improves security.

Unpacking Docker - Key Concepts

Docker Image: A snapshot of your application and its environment.

- Benefits: Speed, consistency, preservation of dependencies, versioning.

Docker Container: A running instance of an image.

Dockerfile: Instructions for building an image.

The Docker Workflow

1. Write a Dockerfile.

   create Dockerfile with file name as “Dockerfile” with no extension.

   Example:

```
FROM node:latest
COPY index.js /home/app/index.js
COPY package.json /home/app/package.json
WORKDIR /home/app
RUN npm install
EXPOSE 4500
CMD ["node", "index"]
```

Dot indicates current file (we have to give the path to Dockerfile)

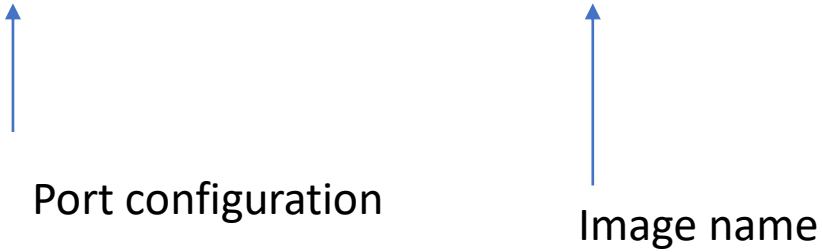
2. Docker builds an image.

   docker build –t myapp1 . //myapp1 is image name given to our application

The Docker Workflow cont..

3. Run the image to create a container.

```
docker run –p  4500:4500  myapp1
```



Explanation of Docker file

```
FROM node:latest
COPY index.js /home/app/index.js
COPY package.json /home/app/package.json
WORKDIR /home/app
RUN npm install
EXPOSE 4500
CMD ["node", "index"]
```

1. FROM node:latest

This line specifies the base image for your Docker container.

- node:latest refers to the latest official Node.js image from Docker Hub. This image contains a pre-configured environment with Node.js and npm installed, making it easy to run Node.js applications in Docker containers.
- The latest tag ensures you get the most recent stable version of Node.js available at the time.

2. COPY index.js /home/app/index.js

This command copies the index.js file from your local machine into the Docker container.

- COPY is the Dockerfile command to copy files or directories from the host machine (your local environment) into the container's filesystem.
- The first part (index.js) is the local file or directory you want to copy.
- The second part (/home/app/index.js) is the destination path in the container. This means that the index.js file will be copied to /home/app/index.js in the Docker container.

3. COPY package.json /home/app/package.json

This is similar to the previous COPY command, but it copies the package.json file.

- package.json contains metadata about the Node.js application, such as dependencies, scripts, and other configuration settings.
- By copying this file, you ensure that when the application is built inside the Docker container, npm knows what dependencies to install.

4. WORKDIR /home/app

- This sets the working directory for any subsequent commands in the Dockerfile.
- WORKDIR is used to specify the directory inside the container where the application will be executed.
- By setting /home/app as the working directory, you ensure that the npm install and the application (node index.js) will be executed from this directory, which now contains the copied files (index.js and package.json).

5. RUN npm install

- This command installs the Node.js dependencies listed in package.json.
- RUN is a Dockerfile command used to execute a command in the container during the image build process.
- npm install will install the dependencies defined in package.json. These dependencies will be installed inside the /home/app directory in the container.
- This is a crucial step to ensure that your Node.js application has access to all the required packages (e.g., Express, body-parser) to run correctly.

6. EXPOSE 4500

This line tells Docker to expose port 4500 on the container.

- EXPOSE is a Dockerfile instruction that informs Docker that the container will listen on the specified port at runtime.
- In this case, port 4500 is being exposed, meaning your Node.js application is set to run on this port.
- Note that EXPOSE does not publish the port to the host machine; it simply informs Docker that the container will use this port. You would need to map the container port to a host port when running the container using the -p option (e.g., docker run -p 4500:4500).

7. CMD ["node", "index"]

This is the command that will be executed when a container based on this image is run.

- CMD specifies the default command to run when the container starts.
- In this case, it will run node index, which will start your Node.js application by executing the index.js file.
- CMD can be overridden when running the container if you want to run a different command.

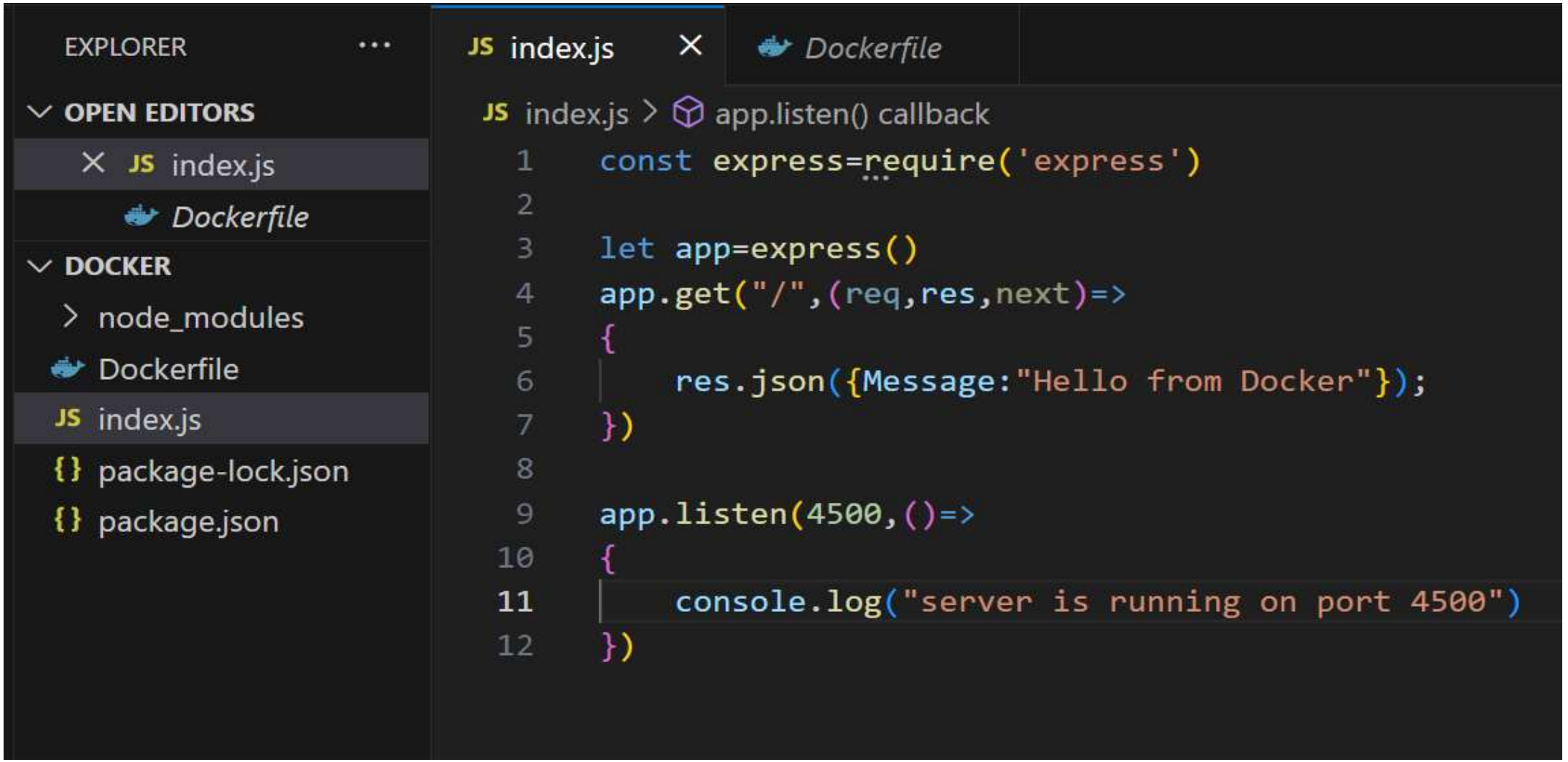
Hands-on with Docker

- **Basic Commands:** docker run , docker ps , docker images , docker stop

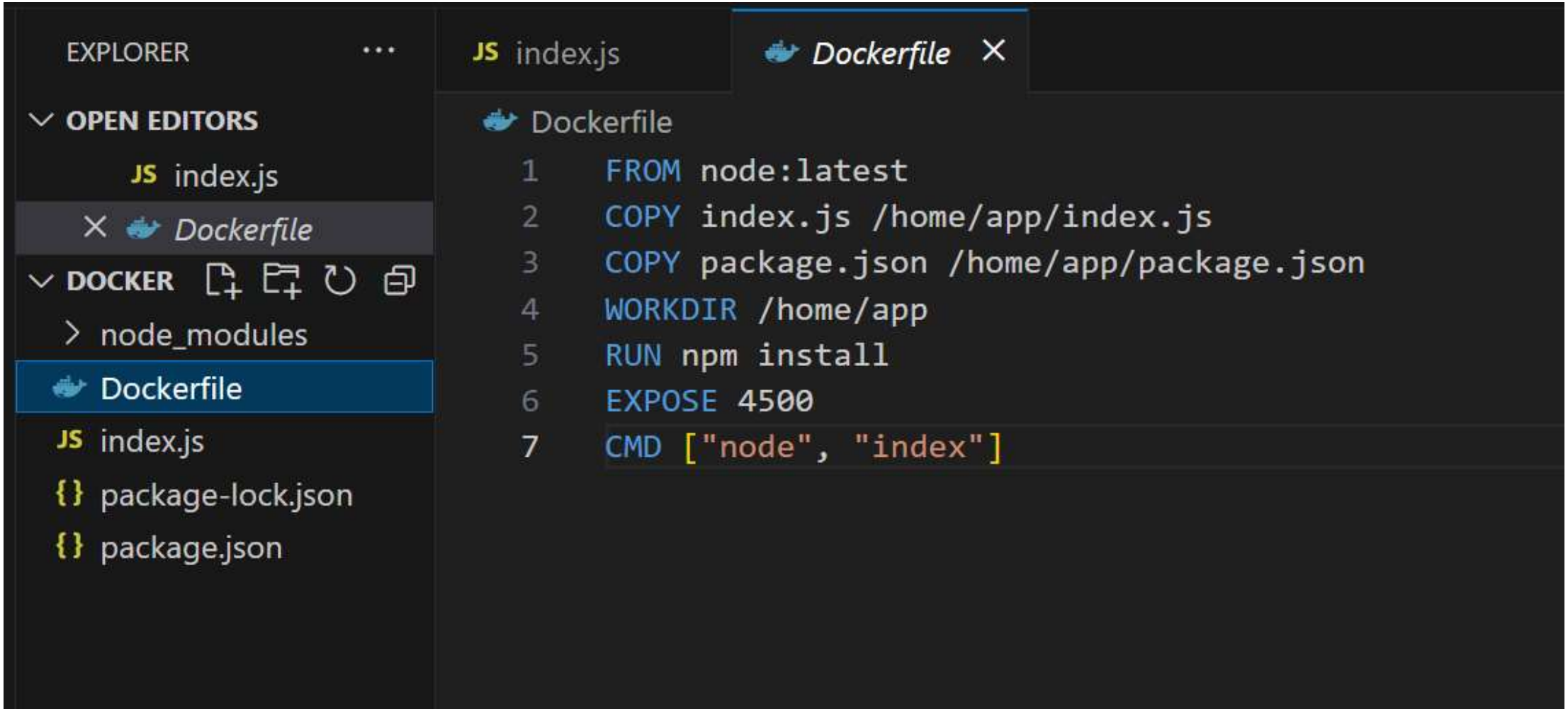
Dockerfiles:

- **FROM instruction:** Specifies the base image, avoiding starting from scratch. (Example: FROM ubuntu:latest)
- COPY , RUN , CMD instructions (explained with examples)
- **Example:** Create a simple Node.js application and Dockerfile, build and run.
- **Open-source Dockerfile example:** Show a real-world Dockerfile and highlight the FROM instruction.

Simple Node.js application:



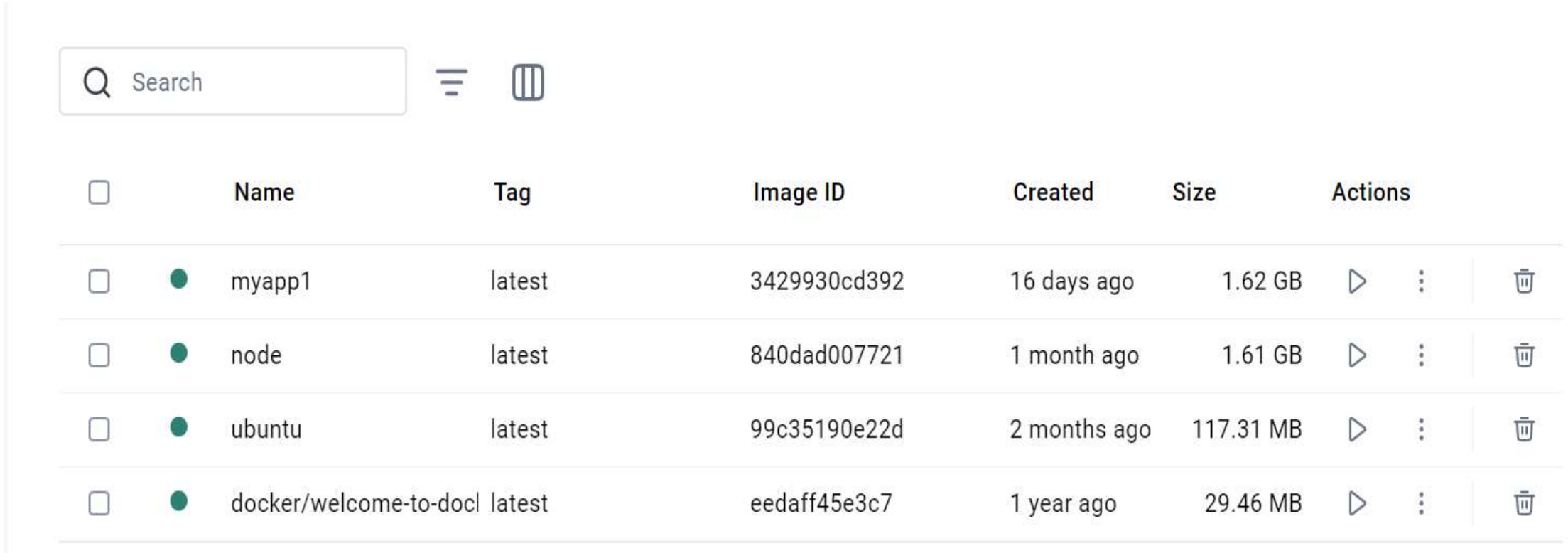
Dockerfile


EXPLORER
OPEN EDITORS
index.js
Dockerfile
DOCKER
node_modules
Dockerfile
index.js
package-lock.json
package.json
index.js
Dockerfile
Dockerfile
1 FROM node:latest
2 COPY index.js /home/app/index.js
3 COPY package.json /home/app/package.json
4 WORKDIR /home/app
5 RUN npm install
6 EXPOSE 4500
7 CMD ["node", "index"]

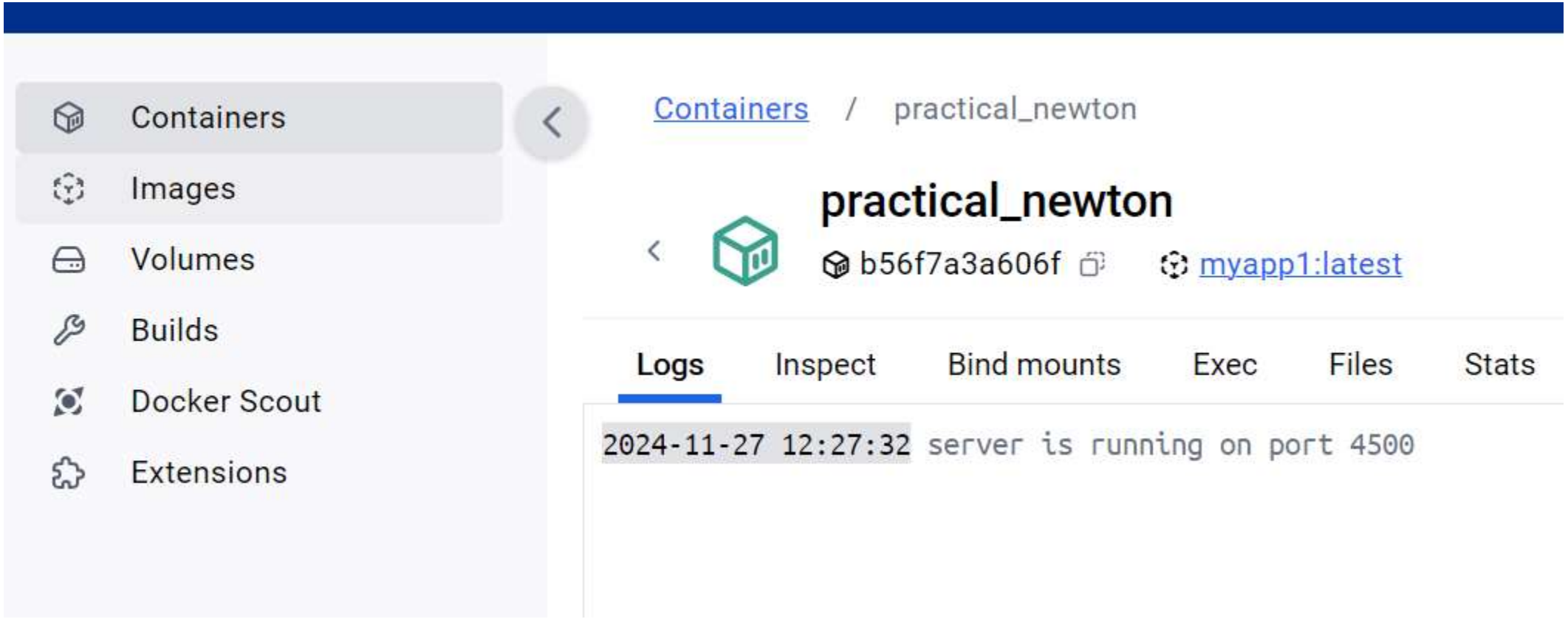
Docker Desktop

Image: myapp1 (Node.js application) on

docker build –t myapp1 .



Run image(myapp1)



To run image myapp1 in your local system