

**Lesson 8 – Containerization and Orchestration**

**Topics**

1. What is Containerization?
2. Benefits of Containerization
3. What is Container orchestration?
4. Container Orchestration Tools

* Kubernetes
* Docker

1. **What is Containerization?**

* Containerization is the process of packaging an application and all its dependencies, such as libraries, configuration files, and runtime, into a container.
* A container is a lightweight, standalone, and portable unit that ensures the application runs consistently across different environments, regardless of differences in the underlying infrastructure or operating system.

1. **What is Container orchestration?**

Container orchestration is the automated process of managing, deploying, scaling, and networking containers across clusters of servers. It simplifies handling multiple containers in complex applications, ensuring they work together efficiently and reliably.

1. **Containerization and orchestration technologies**

* **Docker:** For building and running containers.
* **Kubernetes:** For orchestrating and managing containers in a production environment.
* **Podman:** A Docker-compatible, rootless container engine focused on security.
* **Containerd:** A container runtime that powers tools like Docker and Kubernetes.
* **AWS Fargate**:A serverless container platform for running containers in AWS without managing infrastructure.

1. **What is Docker?**

Docker is a containerization tool that enables developers to package an application and its dependencies into a single container.

1. **Why containerize(Dockerize) a React App?**

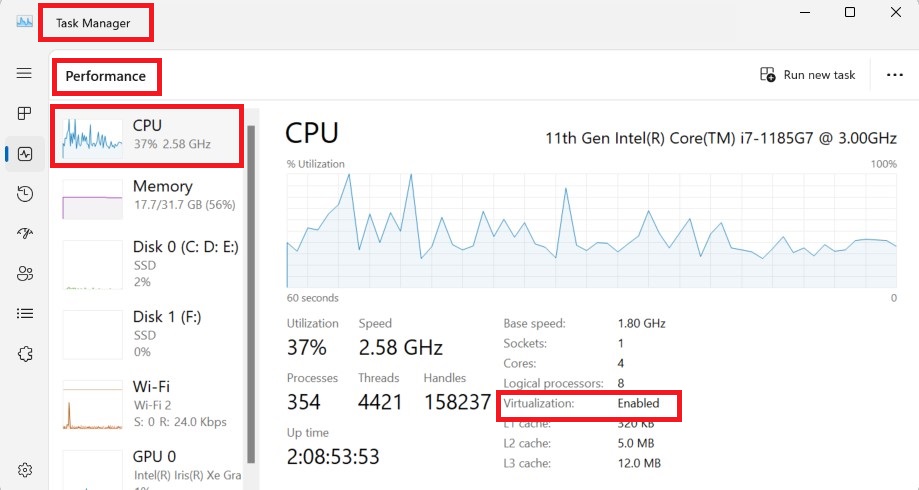
Containerizing a React app with Docker offers several important advantages:

1. **Consistency Across Environments**: Docker ensures the app runs the same way across development, staging, and production environments, minimizing environment-specific bugs.
2. **Easier Deployment**: With Docker, deploying a React app becomes straightforward and flexible, allowing it to be hosted on cloud providers like AWS, Google Cloud, or Azure with minimal configuration.
3. **Dependency Isolation:** Docker containers package all dependencies, making the app independent of the host system's configuration, reducing conflicts, and simplifying dependency management.
4. **Scalability:** Docker enables easy scaling by running multiple instances (containers) in parallel, which improves load management and availability.

**Activity 13 – Containerizing a React App**

**Step 1 – Installing the Docker Desktop on your computer.**

* 1. Verify virtualization is enabled on your computer. If it is not, activate it from the BIOS settings.

****

* 1. Install WSL (Windows Subsystem for Linux). It is a lightweight Linux runtime environment for Docker on Windows. Open PowerShell as Administrator and run the following command.

A screenshot of a computer program

Description automatically generated

* 1. Download the Docker desktop from the website <https://www.docker.com/products/docker-desktop/> and install it. Here, you need to have an user account.

**A screenshot of a computer

Description automatically generated**

**Step 2 - Create a .env file**

Environmental variable file (**.env**) will help to avoid hard-coding sensitive parameters like port numbers, connection strings, and URLs.

1. Create a **“.env”**file in the server folder and enter port number and mongodb URI.

**A screenshot of a computer

Description automatically generated**

1. Install a node package called “dotenv” on the server app.

npm install dotenv

1. Import the package and configure it to access environment variables..

import dotenv from "dotenv";

dotenv.config();

1. Access the environment variables using process.env object in Node.js to access environment variables.

………

dotenv.config();

const URI =**`**mongodb+srv://**${process.env.MONGO\_USERNAME}:${process.env.MONGO\_PASSWORD}**@itcluster.quf0h.mongodb.net/**${process.env.MONGO\_DATABASE}**?retryWrites=true&w=majority&appName=itcluster**`**;

mongoose.connect(URI);

app.listen(**process.env.PORT**, () => {

   console.log("You are connected");

});

1. Create a **“.env”**file in the client folder for React environment variables. All variables must start with REACT\_APP\_. Here we stored the base URL of the API.

A screenshot of a computer

Description automatically generated

1. Construct the base API URL dynamically by **process.env.REACT\_APP\_API\_URL** is an environment variable. This approach can easily adopt different environments(e.g. development, staging and production) without hardcoding URLs. Do this for all the HTTP requests in the client app.

const response = await axios.post**(`${process.env.REACT\_APP\_API\_URL}/registerUser`,** {

        name: userData.name,

        email: userData.email,

        password: userData.password

      })

**Step 3 – Creating the Dockerfile for client**

Docker uses a **Dockerfile** to define the environment for the React app. This file contains instructions for setting up the app within a container, such as installing Node.js, copying the app files, installing dependencies, and building the app.

1. Create the **Dockerfile**  at the React app folder and add the following code to the file. Open **postITapp/client** folder and create it. The commands in the file and explanation are given below.

A screenshot of a computer program

Description automatically generated

|  |  |
| --- | --- |
| **Command** | **Explanation** |
| FROM node:alpine | Specifies the base image as Alpine Node.js. |
| WORKDIR /app | Sets the working directory inside the container to /app. |
| COPY package\*.json ./ | Copies package.json and package-lock.json files to the container's /app directory. |
| RUN npm install | Installs the dependencies listed in the package.json file using npm. |
| COPY . . | Copies all the application files from the local directory to the container's working directory. |
| RUN npm run build | Builds the React application for production, generating static files in the build folder. |
| EXPOSE 3000 | Opens port 3000 in the container to allow external access to the app. |
| CMD ["npm", "start"] | Sets the default command to run the client app when the container starts. |

1. Create the **.dockerignore** at the React app folder(**postITapp/client**) and enter **node\_modules**.

**A screenshot of a computer

Description automatically generated**

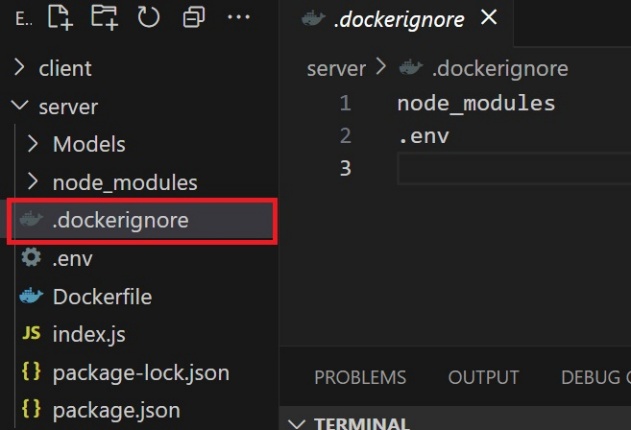
**Step 4 – Creating the Dockerfile for server**

1. Open the folder **postITapp/server** and create the **Dockerfile**

**A screenshot of a computer

Description automatically generated**

1. Open the folder **postITapp/server** and create the **.dockerignore**



**Step 5 – Creating the docker-compose.yml configuration file in the main folder(postITapp).**

A screenshot of a computer

Description automatically generated

The **docker-compose.yml** file is a configuration file used with **Docker Compose**, a tool for defining and running multi-container Docker applications. This file allows you to specify all the services, networks, and volumes required for your application in a single YAML file. Specify the configuration in the YAML file. The following is the YAML file.



**Step 6 - Running YAML file to build the container.**

Open the root directory of the app in your terminal and execute the following command to run the **docker-compose.yml** file. The --build flag tells Docker Compose to **rebuild** the images before starting the containers.

docker-compose up --build

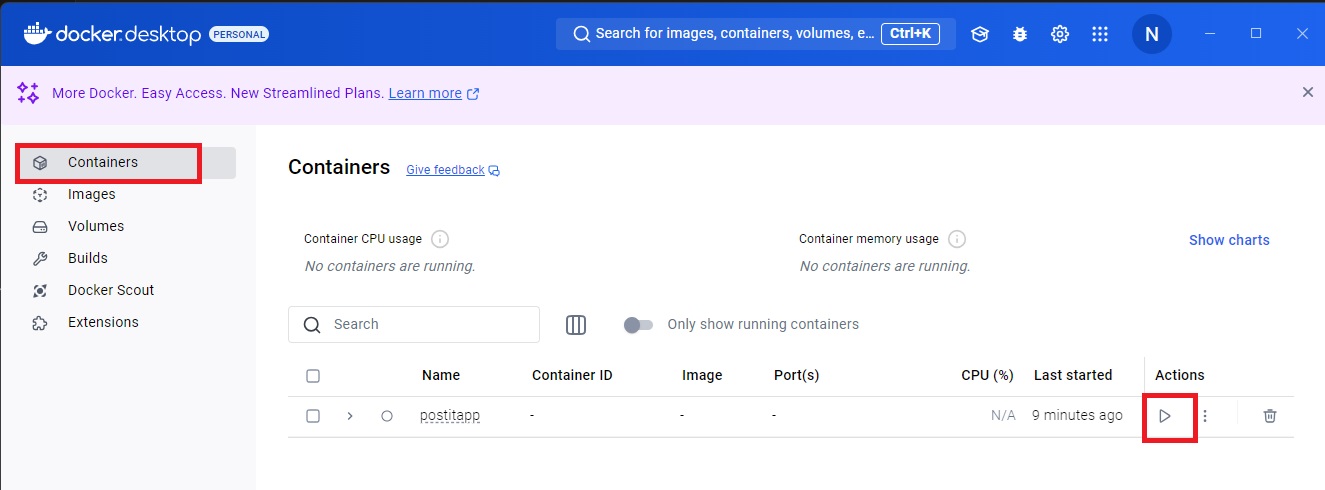
****

Once the containers are up and running, you can access the application by visiting <http://localhost:3000/> in your web browser. Check the Docker images for client and server on the Docker Desktop.

A screenshot of a computer

Description automatically generated

Check the Docker container for the app on the Docker Desktop. Here, you can start and stop the app.



**Step 7 – Push the docker images to Docker hub.**

The docker push command is used to upload a Docker image from your local Docker environment to a container registry (Docker Hub) for deploying the app to the cloud hosting platform.

1. Make sure you are logged into Docker Desktop on your local machine.

**A screenshot of a computer

Description automatically generated**

1. Tag the Docker Image from the terminal.

**Syntax**

**docker tag <local-image-name>:<tag> <dockerhub-username>/<repository-name>:<tag>**

**docker tag postitapp-server:latest nmamdali/postitapp-server:latest**

**docker tag postitapp-client:latest nmamdali/postitapp-client:latest**

1. Push the Docker Image from the terminal. Use the **docker push** command to upload the image to Docker Hub.

**docker push nmamdali/postitapp-server:latest**

**docker push nmamdali/postitapp-client:latest**

1. Open dockerhub and see the repositories where you can find the images that you pushed from your machine.( https://hub.docker.com/)

**A screenshot of a computer

Description automatically generated**

Once your Docker image is pushed to Docker Hub, you can:

* **Share & Collaborate**: Share the image for team use or public access.
* **Deploy**: Use the image for deployment on servers, Kubernetes, Docker Compose, or cloud platforms.
* **Run Anywhere:** Pull and run the image on any machine with Docker installed.
* **Automate CI/CD**: Integrate the image into CI/CD pipelines for testing and deployment.
* **Backup**: Keep the image as a reliable backup of your application.
* **Showcase Public Projects**: Host public repositories to share or demonstrate projects.
* **Integrate with Tools:** Link Docker Hub to platforms like Kubernetes, Jenkins, or monitoring tools.

This makes the image accessible, portable, and ready for use in multiple environments.