

# **CREATE A DOCKERFILE FOR A WEB APP**

## **Micro Project Report**

**Submitted by**

Giridharan S (22ITR025)

**Course Code & Name :** 22ITF02 – DEVOPS

**Programme & Branch :** B.Tech - IT

**Department :** Information Technology

**Kongu Engineering College,  
Perundurai**

**May 2025**

## **BONAFIDE CERTIFICATE**

Certified that this micro project documentation of “**CREATE A DOCKERFILE FOR A WEB APP**”, the bonafide work of **GIRIDHARAN S (22ITR025)** who carried out the project under my supervision. Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Submitted on \_\_\_\_\_

### **SIGNATURE**

D. VIJAY ANAND M.E.,

### **SIGNATURE**

Dr.S.ANANDAMURUGAN

M.E., Ph.D

### **SUPERVISOR**

Assistant Professor (SLG)

Department of IT,

Kongu Engineering College,

Perundurai

### **HEAD OF THE DEPARTMENT**

Associate Professor & HOD,

Department of IT,

Kongu Engineering College,

Perundurai.

# CREATE A DOCKERFILE FOR A WEB APP

## AIM

To containerize a simple Node.js web application using Docker by writing a Dockerfile and running the container locally to ensure platform-independent deployment.

## ABSTRACT

This project focuses on simplifying the deployment process of web applications by utilizing Docker containerization. A basic Node.js web application is developed and packaged into a Docker container using a custom-written Dockerfile. The Docker image is then built and run locally, ensuring a consistent environment regardless of the host system. This approach eliminates system dependency issues, enhances portability, and prepares the application for scalable cloud or production deployment. The solution highlights the efficiency and reliability that Docker brings to software development and delivery pipelines, even for simple applications.

## SCOPE AND OBJECTIVES

### Scope:

- Containerization of a Node.js web application.
- Local deployment and testing using Docker.
- Platform-independent packaging for easy distribution.
- Foundation for future scalability into CI/CD or Kubernetes environments.

### Objectives:

- Develop a simple Node.js web application.
- Write a Dockerfile to containerize the app.
- Build and run the Docker image locally.
- Ensure consistent performance across different systems.
- Gain practical experience with Docker and image management.

# **HARDWARE & SOFTWARE REQUIREMENTS**

## **Hardware Requirements:**

- A system with at least 4GB RAM
- 2+ GHz Processor
- Internet connectivity

## **Software Requirements:**

- Node.js
- Docker
- Git
- Visual Studio Code
- GitHub Account
- DockerHub Account

# **PROJECT DESIGN**

The project design focuses on containerizing a simple Node.js web application using Docker. The application is built locally and deployed as a Docker container to ensure consistent behavior across environments. The Dockerfile contains the instructions to set up the runtime environment, install dependencies, and start the Node.js server.

The structure of the project includes:

- app.js – The main Node.js application file
- package.json – Project metadata and dependency definitions
- Dockerfile – Instructions to build the Docker image
- .dockerignore – Specifies files to exclude from the Docker build context

Once the Dockerfile is written, the image is built using Docker CLI and then run as a container locally. This allows the web application to be accessed via localhost on a specified port.

## **ADVANTAGES**

- Platform-independent deployment using Docker.
- Simplified application setup and execution.
- Consistent environment across development and production.
- Easy to build, run, and test locally.
- Reduces dependency-related issues and manual configurations.

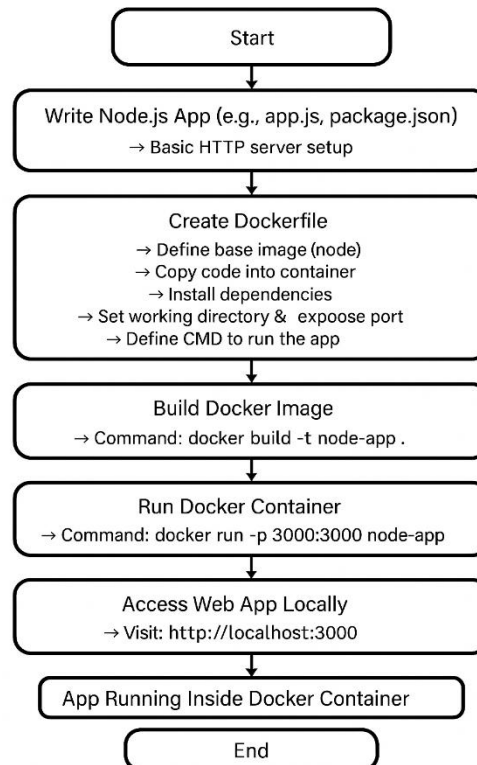
## **PROPOSED STYLE**

The implementation of this project involves containerizing a simple Node.js web application using Docker to enable easy and consistent deployment. Initially, the Node.js app is developed and tested locally to ensure it functions correctly. A Dockerfile is created to define the environment, dependencies, and commands required to build a Docker image of the app. The source code along with the Dockerfile is maintained in a GitHub repository for version control and easy collaboration. The Docker image is built using the Dockerfile and then pushed to DockerHub under the username giridharans1729, securely managing credentials for authentication. The Docker image can be run locally as a container to verify the deployment and functionality of the app within an isolated environment. This setup lays the foundation for further automation and scaling by providing a portable, version-controlled container image of the Node.js application.

## **IMPLEMENTATION OF THE PROJECT**

The implementation of this project involves containerizing a simple Node.js web application using Docker to enable easy and consistent deployment. Initially, the Node.js app is developed and tested locally to ensure it functions correctly. A Dockerfile is created to define the environment, dependencies, and commands required to build a Docker image of the app. The source code along with the Dockerfile is maintained in a GitHub repository for version control and easy collaboration. The Docker image is built using the Dockerfile and then pushed to DockerHub under the username `giridharans1729`, securely managing credentials for authentication. The Docker image can be run locally as a container to verify the deployment and functionality of the app within an isolated environment. This setup lays the foundation for further automation and scaling by providing a portable, version-controlled container image of the Node.js application.

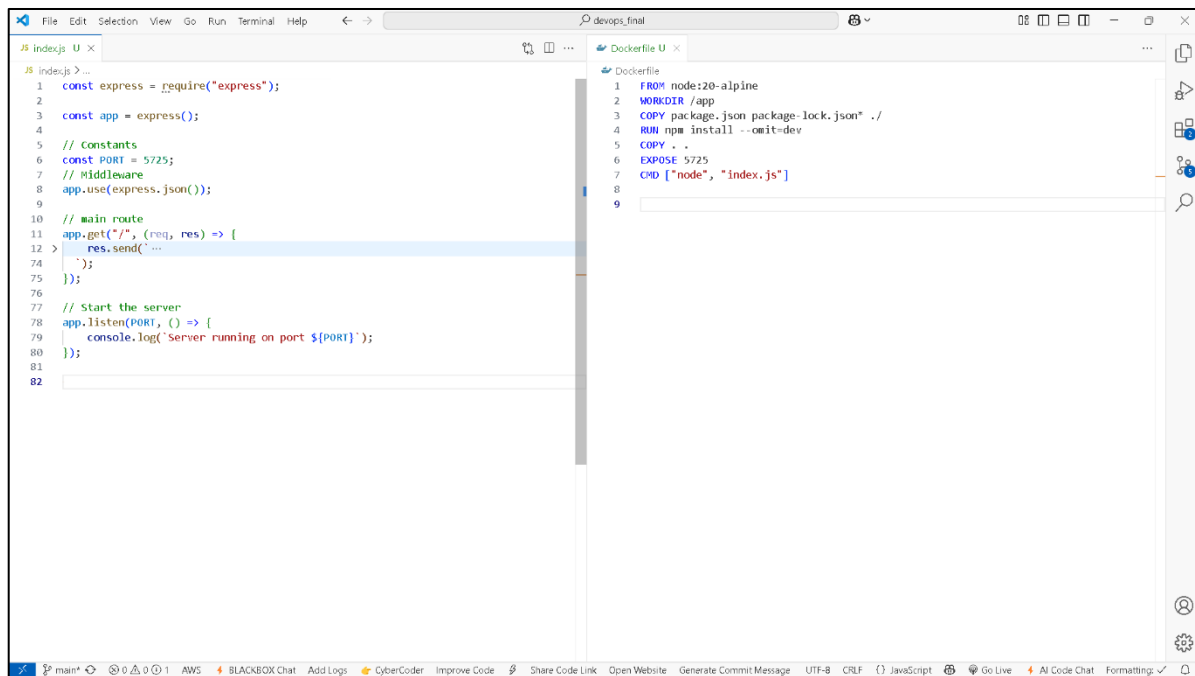
## Flow of Design:



**Flow Chart**

As shown in the flow chart, the process starts when a developer writes and tests the Node.js application locally. Next, a Dockerfile is created to define the container environment. Using Docker commands, the developer builds a Docker image of the application. After the image is successfully built, it is run as a container locally, exposing the app on a specified port. Finally, the application becomes accessible to users through a web browser via localhost, completing the containerization and deployment process on the local machine.

## OUTPUT:

A screenshot of the Visual Studio Code editor interface. The left pane shows the 'index.js' file with the following code:

```
1 const express = require("express");
2
3 const app = express();
4
5 // Constants
6 const PORT = 5725;
7 // Middleware
8 app.use(express.json());
9
10 // main route
11 app.get("/", (req, res) => {
12   res.send("...");
13 });
14
15 // Start the server
16 app.listen(PORT, () => {
17   console.log(`Server running on port ${PORT}`);
18 });
```

The right pane shows the 'Dockerfile' file with the following code:

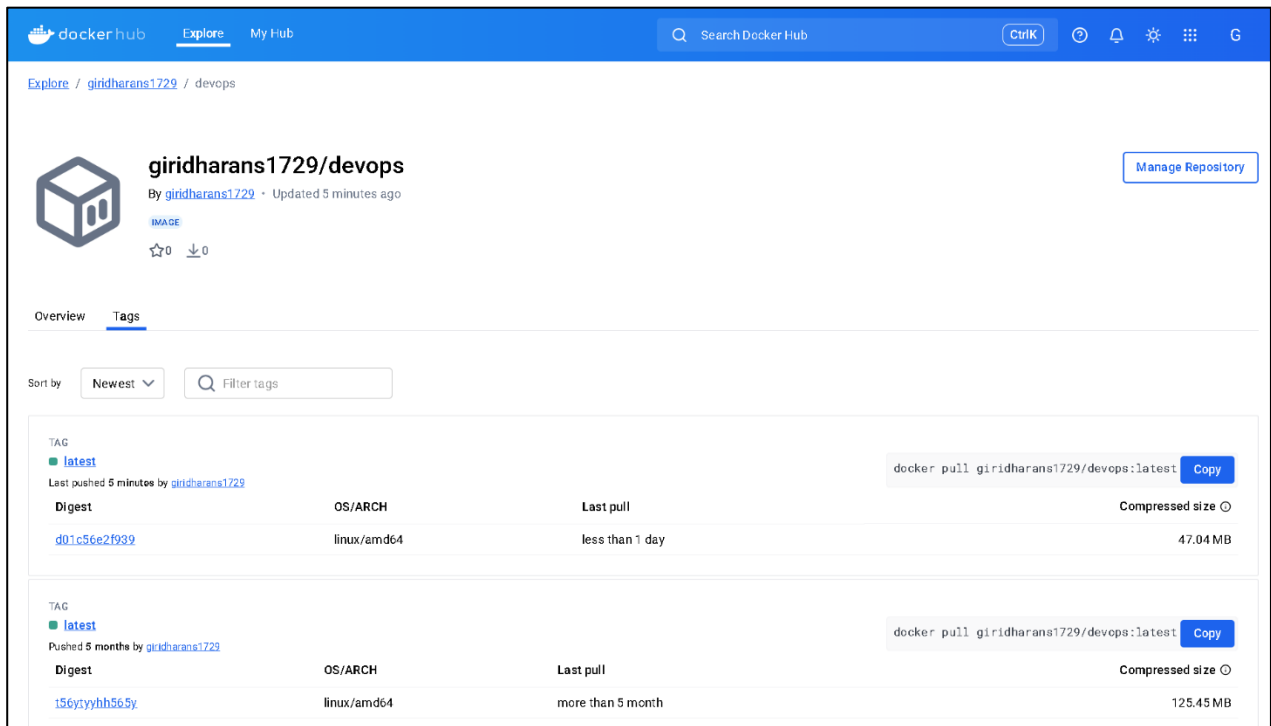
```
1 FROM node:20-alpine
2 WORKDIR /app
3 COPY package.json package-lock.json ./
4 RUN npm install --omit-dev
5 COPY . .
6 EXPOSE 5725
7 CMD ["node", "index.js"]
```

The status bar at the bottom shows various icons and text, including 'main', '0', '1', 'AWS', 'BLACKBOX Chat', 'Add Logs', 'CyberCoder', 'Improve Code', 'Share Code Link', 'Open Website', 'Generate Commit Message', 'UTF-8', 'CRLF', 'JavaScript', 'Go Live', 'AI Code Chat', and 'Formatting: ✓'.

**FIGURE 1: Node.js Web App and Dockerfile View in VS Code**

As shown in Figure 1, the Visual Studio Code interface displays the core project files used for containerizing the Node.js web application. The index.js file contains the server logic for the app, while the Dockerfile defines the instructions to build a Docker image from the source code. This setup enables building and running the container locally using Docker, highlighting the foundational steps of containerization without involving a CI/CD pipeline. The use of VS Code ensures streamlined development, editing, and testing of both code and Docker configurations in a single environment.





**FIGURE 2: Docker Hub Repository View After Docker Image Push**

As shown in Figure 2, the Docker Hub dashboard for the user GiridharanS1729 displays the successfully pushed Docker images of the Node.js application. These repositories represent container images built locally and pushed to Docker Hub as part of the project workflow. The timestamps under "Last Pushed" confirm recent activity, demonstrating that the Docker image management process is functioning correctly and the container is available for deployment or distribution.

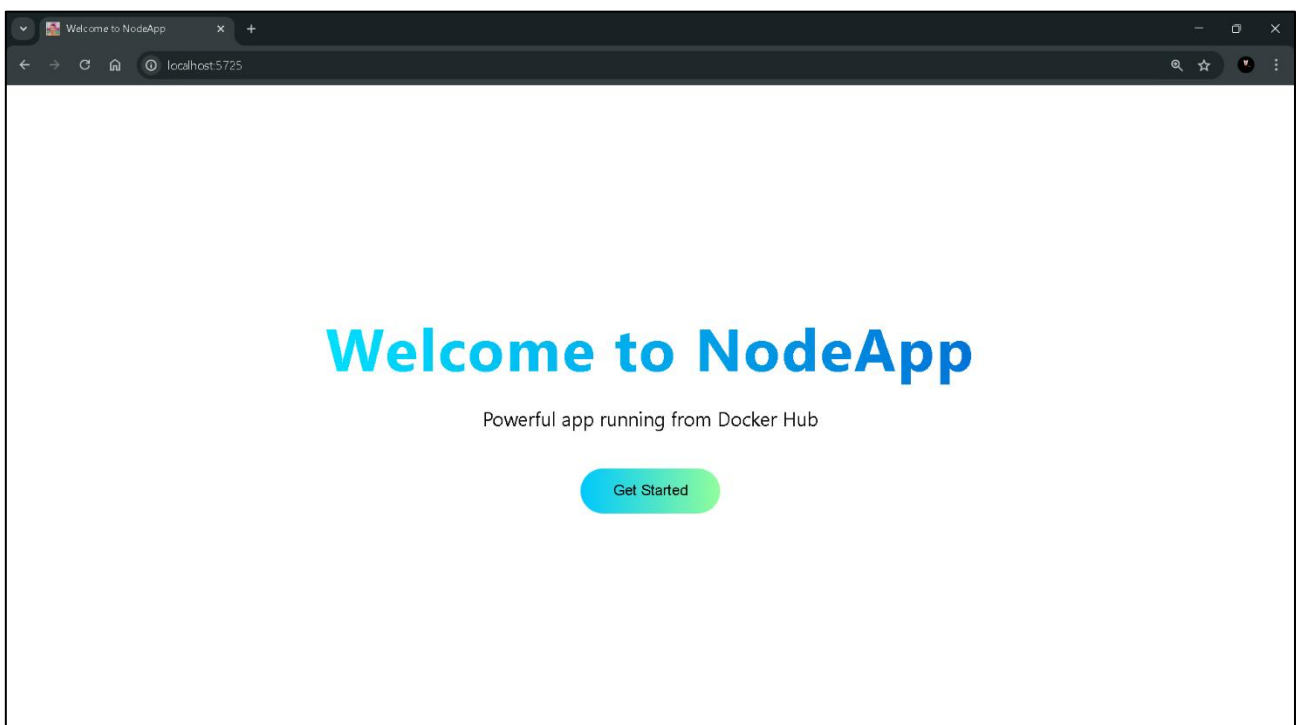
```
CMD Giridharan
docker build -t giridharans1729/devops:latest .
[+] Building 50.7s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 190B
=> [internal] load metadata for docker.io/library/node:20-alpine
=> [auth] library/node:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 79B
=> [1/5] FROM docker.io/library/node:20-alpine@sha256:be56e91681a8ec1bba91e3006039bd228dc797fd984794a3efedab325b36e679
=> => resolve docker.io/library/node:20-alpine@sha256:be56e91681a8ec1bba91e3006039bd228dc797fd984794a3efedab325b36e679
=> => sha256:699b753813e6a832b1187221e7d65c4f04fab95c6dbf8dc51a77376d361e78b5 448B / 448B
=> => sha256:381aeb3d905aca56c1cf878358dbf736546a412ef64c2192098ff0cf4ec8144 1.26MB / 1.26MB
=> => sha256:95c1247b2bae42e663142a67e8e67e0721f7e5626cae79d0e7c51cf0cb9d9db2 42.95MB / 42.95MB
=> => extracting sha256:05c1247b2bae42e663142a67e8e67e0721f7e5626cae79d0e7c51cf0cb9d9db2 1.1s
=> => extracting sha256:381aeb3d905aca56c1cf878358dbf736546a412ef64c2192098ff0cf4ec8144 0.1s
=> => extracting sha256:699b753813e6a832b1187221e7d65c4f04fab95c6dbf8dc51a77376d361e78b5 0.8s
=> [internal] load build context
=> => transferring context: 60.40kB
=> [2/5] WORKDIR /app
=> [3/5] COPY package.json package-lock.json* ./
=> [4/5] RUN npm install --omit=dev
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:d01c56e2f93954338fb23e7b586e7f47430665ce54d73879ee23b3fd28e528e4
=> => exporting config sha256:7882441fbb6eb953a8c799eea0a603acdec34e471b20bbb2b4b617126d721a22
=> => exporting attestation manifest sha256:94a431140438dc6137d5c63e70a446b603140abc0b0f0d4d2bfbad1b910f3837
=> => exporting manifest list sha256:b6f83ba96b2c4224fc14fb059d88a3bfe0e3fffa9abe44d02a301ff665f2feba2
=> => naming to docker.io/giridharans1729/devops:latest
=> => unpacking to docker.io/giridharans1729/devops:latest
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/xb8urj686nvmpky8dfri0001l
What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview
```

**FIGURE 3: Docker Build and Deployment Summary**

As shown in Figure 3, the Docker workflow begins with manually building the application image using the Dockerfile provided in the project directory. The image is created by running the docker build command, which packages the production-ready React application into a container image. Once the build is successful, the image is tagged appropriately and pushed to Docker Hub using secure credentials. This manual pipeline highlights the essential containerization steps — building, tagging, and uploading the image — ensuring the application is portable and ready for deployment in any environment. The Docker Hub repository confirms the successful push operation by displaying the latest image with relevant metadata such as the image tag and push timestamp.

```
CA CMD Giridharan x + v
docker push giridharans1729/devops:latest
The push refers to repository [docker.io/giridharans1729/devops]
f18232174bc9: Layer already exists
09c1247b2bae: Pushed
da7743a39d4f: Layer already exists
a2dca174ca88: Layer already exists
11b4aee1f6ab: Layer already exists
a3415f9d8d9a: Layer already exists
6378c7359824: Already exists
381aeb3d985: Layer already exists
699b753813e6: Layer already exists
latest: digest: sha256:b6f83ba96b2c4224fc14fb059d88a3bfe0e3fffa9abe44d02a381ff65f2feba2 size: 856
```

**FIGURE 4: Docker Push Command Execution**



**FIGURE 5 : Deployed Web Application Interface**

Figure 5 shows the Node.js web application running locally in a browser window. This simple app serves as the core project for containerization and deployment. The screenshot demonstrates the app's homepage, confirming that the server is running successfully and serving the expected content. This visual proof verifies that the Docker containerization and deployment processes preserve the application's functionality, making it accessible via a web browser as intended.

## COMMANDS:

- ✓ **git clone repo\_link**  
→ Clone the source code from the GitHub repository.
- ✓ **docker build -t GiridharanS1729/node-app:latest ./app**  
→ Build a Docker image for the Node.js application.
- ✓ **docker login**  
→ Log in to DockerHub to allow pushing images.
- ✓ **docker push GiridharanS1729/node-app:latest**  
→ Push the Docker image to DockerHub.
- ✓ **docker run -p 3000:3000 GiridharanS1729/node-app:latest**  
→ Run the Docker container locally, exposing port 3000.
- ✓ **docker ps**  
→ Check running Docker containers.
- ✓ **docker stop <container\_id>**  
→ Stop the running container.
- ✓ **docker rm <container\_id>**  
→ Remove the stopped container.

## **CONCLUSION:**

This project successfully demonstrates the containerization of a simple Node.js web application using Docker. By creating a Dockerfile and building the application into a Docker image, the project ensures consistent and portable deployment across different environments. Running the application as a Docker container locally eliminates environment setup issues and manual configuration, making development and testing faster and more reliable. This approach provides a solid foundation for scaling the application deployment in the future, including integration with cloud services or CI/CD pipelines. Overall, the project highlights the effectiveness of Docker in simplifying application deployment and improving developer productivity.

## **FUTURE WORK:**

To further improve the CI/CD pipeline implemented in this project, several enhancements can be explored. Incorporating advanced testing strategies such as automated unit, integration, and end-to-end tests within the Jenkins pipeline will strengthen code reliability and catch issues earlier in the development cycle. Integrating real-time notifications through Slack or email for build and deployment statuses will boost team collaboration and response time. Additionally, implementing progressive deployment techniques such as Blue-Green or Canary deployments will reduce downtime and provide safer rollouts. Automating infrastructure provisioning using tools like Terraform can enable scalable and reproducible environments, and adopting GitOps practices with tools like ArgoCD can promote declarative, version-controlled, and continuously synchronized deployments, further enhancing stability and operational efficiency.

GITHUB LINK : [https://github.com/GiridharanS1729/devops\\_final](https://github.com/GiridharanS1729/devops_final)

DOKCER HUB : <https://hub.docker.com/r/giridharans1729/devops/>