7. Deploy Angular/React/Java/Python or any other Application in Docker Container.

Step 1: Clone the Project Repository

1. **Open a Terminal**: Start a terminal session to enter the commands.
2. **Navigate to a Working Directory**:

```
cd ~
mkdir my_projects
cd my_projects
```

3. **Clone the Repository**: Run the following command to clone the Git repository.

```
git clone <repository_url>
```

Example:

```
git clone https://github.com/sorani123/laptop-shopping-cart-ui.git
```

4. **Navigate into the Project Directory**:

```
cd laptop-shopping-cart-ui
```

Step 2: Create a Dockerfile

1. **Create a New File Named `Dockerfile`**:
   - In the project root directory, create a file named `Dockerfile`:

```
nano Dockerfile
```

2. **Write Dockerfile Instructions**:
   - Depending on the type of application, use one of the following configurations:
   - **For Node.js Applications (Angular/React)**:

```
dockerfile
```

```
# Use the official Node.js image
FROM node:16-alpine

# Set the working directory in the container
WORKDIR /app

# Copy package.json and install dependencies
COPY package*.json ./
RUN npm install

# Copy the rest of the application code
COPY . .

# Build the app
RUN npm run build

# Serve the app using an HTTP server
RUN npm install -g serve
CMD ["serve", "-s", "build", "-l", "3000"]

# Expose the port the app runs on
EXPOSE 3000
```

- **For Java Applications (Spring Boot)**:

```
dockerfile

FROM openjdk:17-jdk-slim
COPY target/*.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
EXPOSE 8080
```

- **For Python Applications (Flask)**:

```
dockerfile

FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
EXPOSE 5000
```

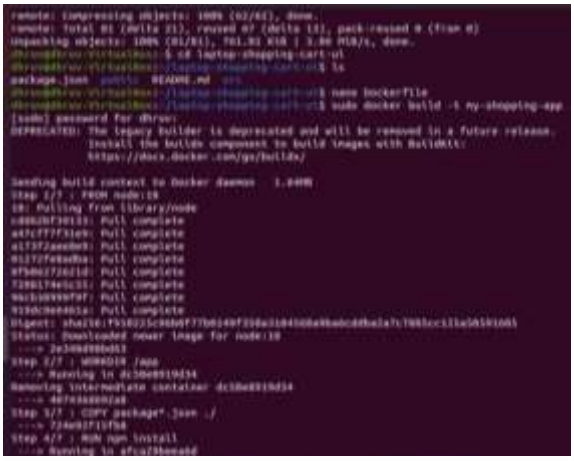
3. **Save and Exit the Dockerfile**:
   - For `nano`, press `CTRL + X`, then `Y`, and `Enter`.

Step 3: Build the Docker Image

1. **Build the Image**:

```
sudo docker build -t my-shopping-app .
```

   - `-t` is used to tag the image with a name (`my-shopping-app`).
2. **Verify Build Output**:
   - During the build, Docker will pull dependencies and prepare the environment as specified in the Dockerfile.



Step 4: Run the Docker Container

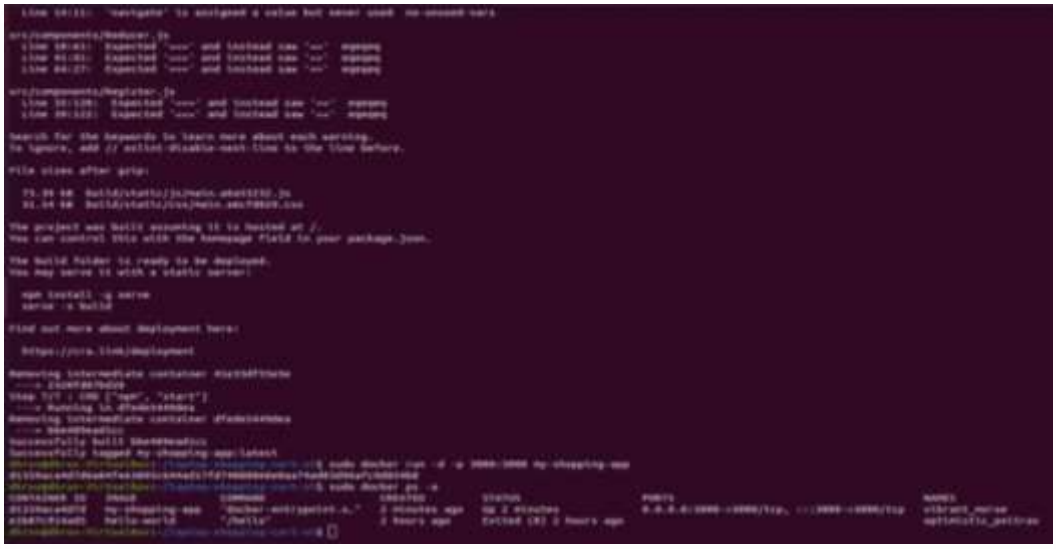
1. **Run the Container**:

```
docker run -p <host_port>:<container_port> my-shopping-app
```

   - Replace `<host_port>` and `<container_port>` with the actual port, e.g., `3000:3000` for Angular/React.
2. **Example Command**:

```
docker run -p 3000:3000 my-shopping-app
```

3. **Check for Application Access**:
   - Open a browser and go to `http://localhost:<host_port>`, e.g., `http://localhost:3000`.



Step 5: Troubleshooting and Logs

1. **View Running Containers**:

   ```
   docker ps
   ```

   - This lists all active containers with their IDs, names, and other details.

   Image Space: Insert a screenshot showing `docker ps` output.

2. **View Container Logs**:

   ```
   docker logs <container_id>
   ```

   - Replace `<container_id>` with the ID of your container.
3. **Access the Container's Shell (Optional)**:

   ```
   docker exec -it <container_id> /bin/sh
   ```

   - This command lets you interact with the container's filesystem for further troubleshooting.

Step 6: Stopping and Removing Containers

1. **Stop the Container**:

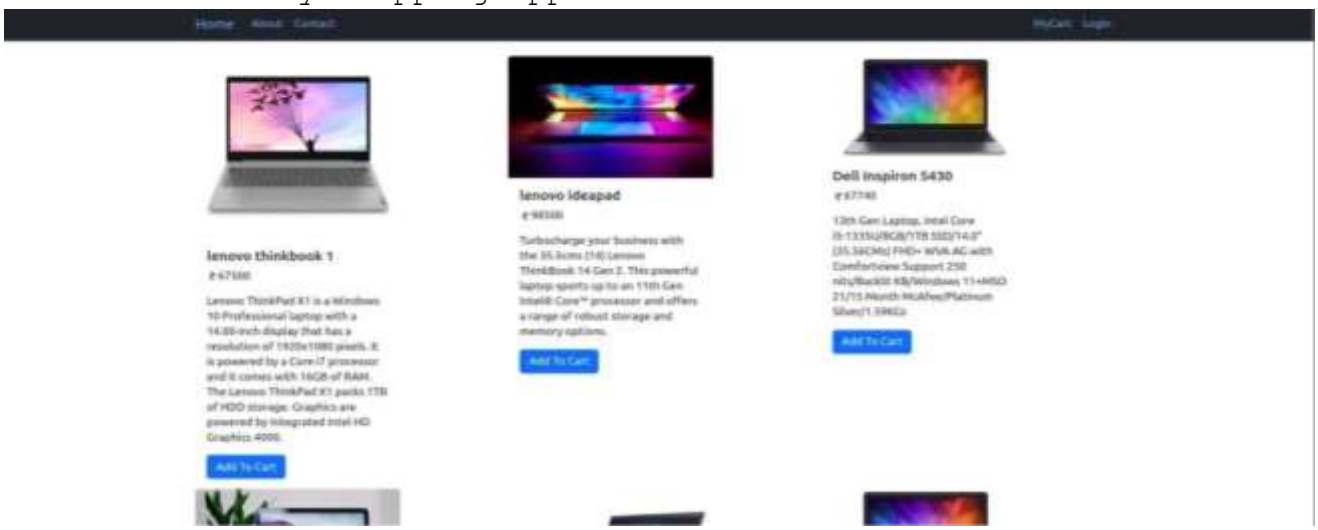
```
docker stop <container_id>
```

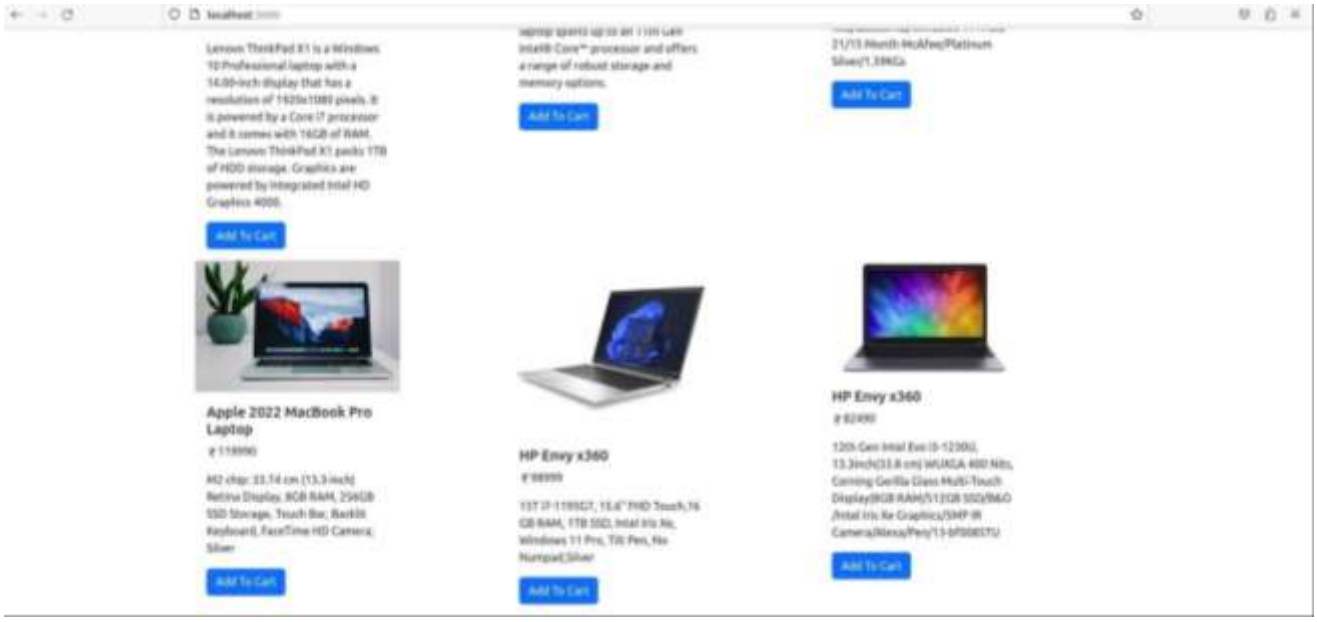
2. **Remove the Container**:

```
docker rm <container_id>
```

3. **Remove the Docker Image** (if no longer needed):

```
docker rmi my-shopping-app
```



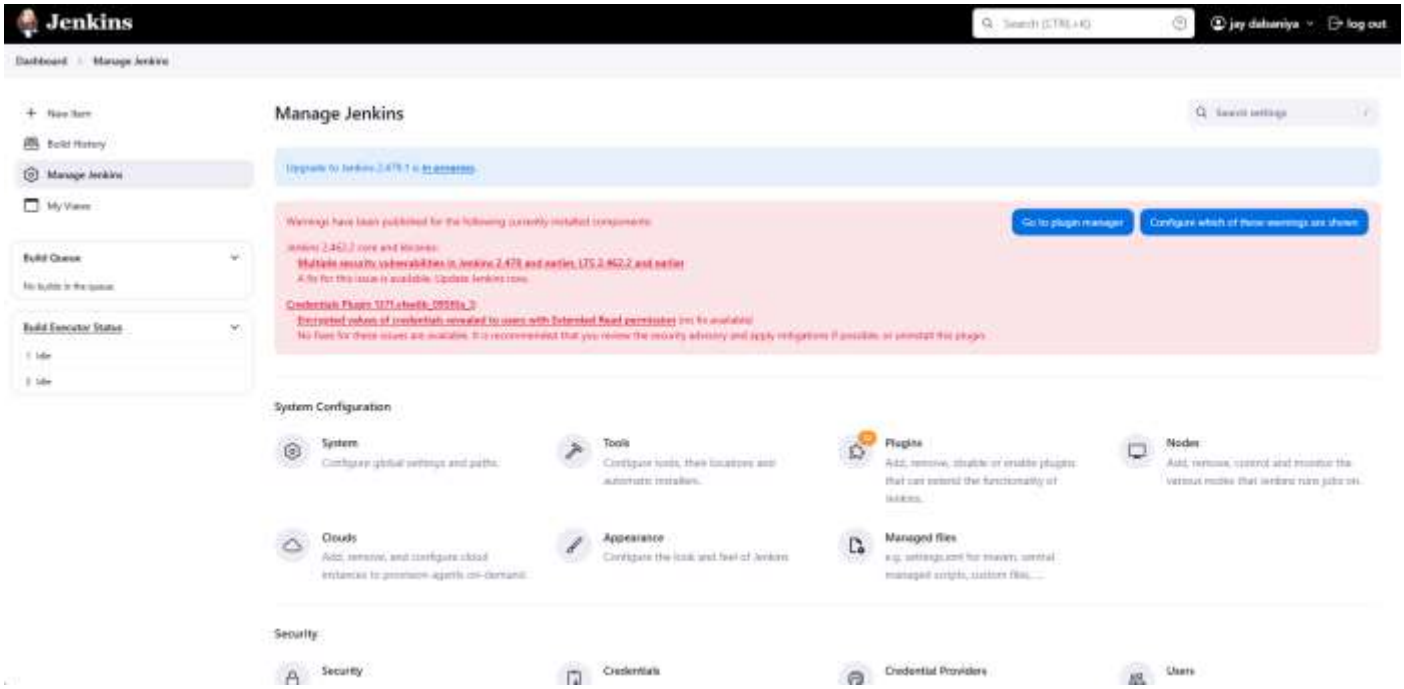


8. Use Jenkins to set up a distributed pipeline that will compile and test a Maven project on two different slave nodes respectively.

Step 1: Open Jenkins

1. Open a web browser.
2. Go to `http://localhost:8000` to access the Jenkins dashboard.

Step 2: Configure Slave Nodes

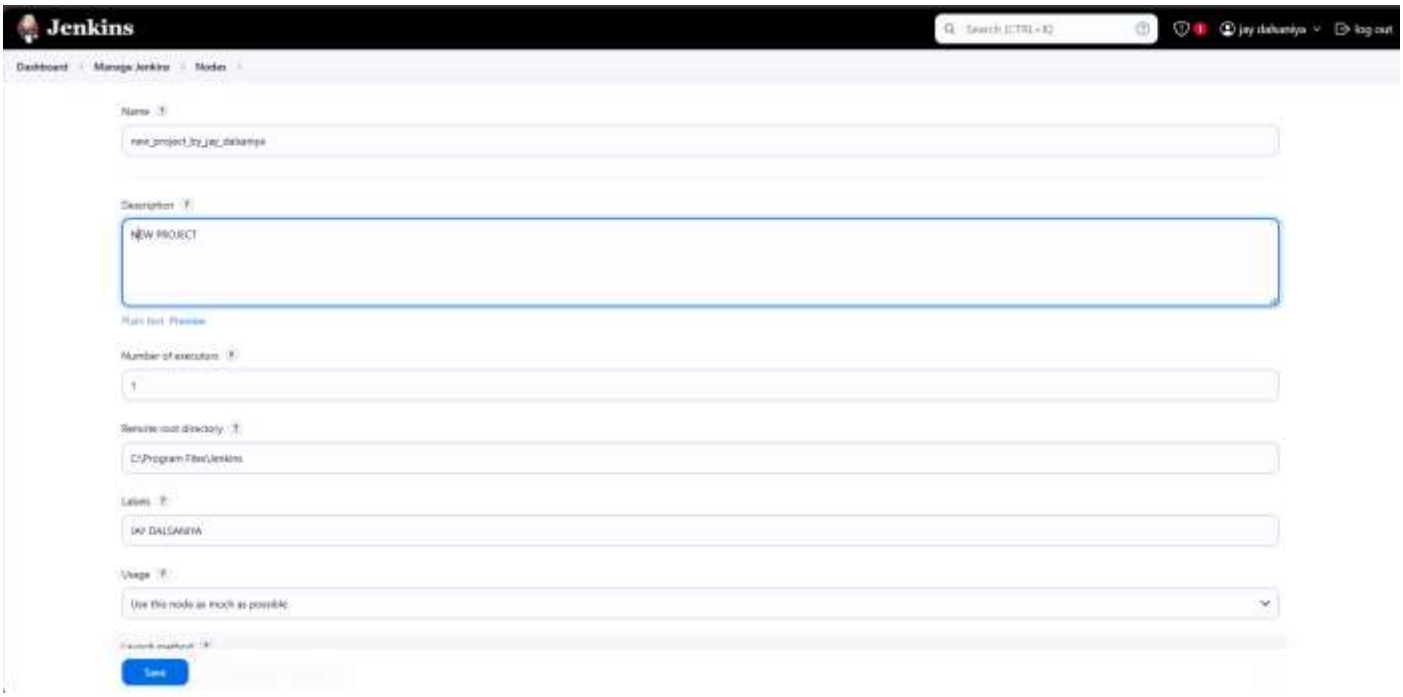


1. From the Jenkins dashboard, navigate to **Manage Jenkins**.
2. Select **Manage Nodes and Clouds**.

3. Click on **New Node** to create a new agent.
   - **Node Name**: Enter a unique name for the agent (e.g., `new_project_by_jay_dalsaniya`).
   - **Number of Executors**: Set to `1`.

- **Remote Root Directory**: Specify the directory where the agent will be located (e.g., `C:\Program Files\Jenkins\slave1`).
- **Labels**: Add a label (e.g., `JAY DALSANIYA`) to identify the node.
- **Usage**: Choose **Use this node as much as possible**.
- **Launch Method**: Set to **Launch agent by connecting it to the controller**.
- **Availability**: Choose **Keep this agent online as much as possible**.

4. Click **Save** to create the node.
5. Repeat steps 1-4 to create a second slave node, if not already configured.



Step 3: Launch and Connect the Slave Node

1. After configuring the slave node, open a command prompt on the machine where the slave node is set up.
2. Run the command to launch the Jenkins agent. The output should indicate the connection status.

   Example output:

```
makefile
Copy code
INFO: Setting up agent: node 2
INFO: Agent discovery successful
INFO: Agent address localhost
INFO: Handshaking
INFO: Connected
```

   This indicates that the slave node has successfully connected to the Jenkins master.

```
C:\Windows\System32\cmd.e
Oct 16, 2024 8:27:49 AM hudson.remoting.Launcher createEngine
INFO: Setting up agent: node 2
Oct 16, 2024 8:27:49 AM hudson.remoting.Engine startEngine
INFO: Using Remoting version: 3261.v9c670a_4748a_9
Oct 16, 2024 8:27:49 AM org.jenkinsci.remoting.engine.WorkDirManager initializeWorkDir
INFO: Using C:\Program Files\Jenkins\slave 2\remoting as a remoting work directory
Oct 16, 2024 8:27:50 AM hudson.remoting.Launcher$CuiListener status
INFO: Locating server among [http://localhost:8080/]
Oct 16, 2024 8:27:50 AM org.jenkinsci.remoting.engine.JnlpAgentEndpointResolver resolve
INFO: Remoting server accepts the following protocols: [JNLP4-connect, Ping]
Oct 16, 2024 8:27:50 AM hudson.remoting.Launcher$CuiListener status
INFO: Agent discovery successful
  Agent address: localhost
  Agent port:    5000
  Identity:      9c:9a:a1:9b:d0:4e:10:44:f7:e7:98:8c:b7:28:40:8a
Oct 16, 2024 8:27:50 AM hudson.remoting.Launcher$CuiListener status
INFO: Handshaking
Oct 16, 2024 8:27:50 AM hudson.remoting.Launcher$CuiListener status
INFO: Connecting to localhost:5000
Oct 16, 2024 8:27:50 AM hudson.remoting.Launcher$CuiListener status
INFO: Server reports protocol JNLP4-connect-proxy not supported, skipping
Oct 16, 2024 8:27:50 AM hudson.remoting.Launcher$CuiListener status
INFO: Trying protocol: JNLP4-connect
Oct 16, 2024 8:27:50 AM org.jenkinsci.remoting.protocol.impl.BIONetworkLayer$Reader run
INFO: Waiting for ProtocolStack to start.
Oct 16, 2024 8:27:51 AM hudson.remoting.Launcher$CuiListener status
INFO: Remote identity confirmed: 9c:9a:a1:9b:d0:4e:10:44:f7:e7:98:8c:b7:28:40:8a
Oct 16, 2024 8:27:51 AM hudson.remoting.Launcher$CuiListener status
INFO: Connected
```

Step 4: Set Up a Distributed Pipeline

1. Go back to the Jenkins dashboard.
2. Select your Maven project or create a new one by clicking on **New Item**.
3. Configure the project as a **Pipeline** job.
   - **Pipeline Script**: Define a pipeline script with stages for `Build` and `Test`.
   - Assign stages to different nodes using `node('node_label')` for each slave node. Example:

```
groovy
Copy code
pipeline {
    agent none
    stages {
        stage('Build and Test') {
            agent { label 'JAY DALSANIYA' }
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Windows Agent') {
            agent { label 'windows-agent1-pipeline' }
            steps {
                sh 'mvn test'
            }
        }
        stage('Another Windows Agent') {
            agent { label 'windows-agent2-pipeline' }
            steps {
                sh 'mvn test'
            }
        }
```

```
            }
        }
```

4. Click **Save** to apply your changes.

Step 5: Run and Monitor the Pipeline

1. Run the pipeline by clicking **Build Now** on the project page.
2. Go to the **Status** page to monitor the pipeline progress.
3. The **Stage View** will display each stage's execution on different nodes.

Observing Results:

- The **Stage View** shows each stage's build and test time, indicating successful distribution across slave nodes.
- Each stage executed on its designated node, ensuring the Maven project was built and tested as intended.