Exp 31:
Create a web application with simple web page containing login details and create a docker image of the application.(Use Ngnix Web server)

Step 1: Create a Project Directory

bash
mkdir mywebapp
cd mywebapp

Step 2: Create HTML Files

Create two HTML files, `index.html`

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Login Page</title>
</head>
<body>
   <div>
       <h2>Login</h2>
       <form id="loginForm">
           <label for="username">Username:</label>
           <input type="text" id="username" name="username" required>
           <br>
           <label for="password">Password:</label>
           <input type="password" id="password" name="password" required>
           <br>
           <button type="button" onclick="checkLogin()">Login</button>
       </form>

       <div id="resultMessage" style="display: none;">
           <h2>Login Result</h2>
           <p id="successMessage" style="color: green;"></p>
           <p id="errorMessage" style="color: red;"></p>
       </div>
   </div>

   <script>
       function checkLogin() {
           var username = document.getElementById('username').value;
           var password = document.getElementById('password').value;

           // Add your login logic here
           // For simplicity, let's assume a successful login if both
fields are not empty
           if (username && password) {
               document.getElementById('resultMessage').style.display =
'block';
               document.getElementById('successMessage').innerHTML =
'Login successful!';
               document.getElementById('errorMessage').innerHTML = '';
           } else {
               document.getElementById('resultMessage').style.display =
'block';
               document.getElementById('errorMessage').innerHTML =
'Invalid username or password.';
               document.getElementById('successMessage').innerHTML = '';
           }
       }
   </script>
</body>
</html>
```

Step 3: Create Dockerfile

Create a `Dockerfile` with a simplified Nginx configuration.

```
# Use Nginx as the base image
FROM nginx:alpine

# Copy HTML files to the default Nginx public directory
COPY index.html /usr/share/nginx/html/

# Copy Nginx configuration
```

```
COPY default.conf /etc/nginx/conf.d/default.conf

# Expose port 80
EXPOSE 80

# Command to start Nginx when the container runs
CMD ["nginx", "-g", "daemon off;"]
```

Step 4: Create Nginx Configuration

Create a file named `default.conf` with the following content:

```
# default.conf
server {
   listen 80;
   server_name localhost;

   location / {
       root   /usr/share/nginx/html;
       index  index.html index.htm;
       try_files $uri $uri/ /index.html =404;
   }



}
```

Step 5: Build and Run Docker Container

Build the Docker image and run the container:

```
sudo docker build -t simple-login-app .
sudo docker run -p 8080:80 simple-login-app
```

Step 6: Access the Login Page

Open a web browser and go to http://localhost:8080

Exp 41:
Mount any directory of host system to the container.

Step 1: Create the HTML File

Create root dir for example 41

Under that create web-app directory

```
mkdir web-app
```

Inside that create html folder.

```
mkdir html
```

Inside html folder create index.html file

```
Web-app
    Html
        index. Html
```

Index.html

```
<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">

   <meta name="viewport" content="width=device-width,
initial-scale=1.0">

   <title>Contents of host-data/</title>

</head>

<body>

   <h2>Contents of host-data/</h2>
```

```
        <ul>

            <li><a href="file1.txt">file1.txt</a></li>

            <li><a href="file2.txt">file2.txt</a></li>

            <!-- Add more files as needed -->

        </ul>

    </body>

    </html>
```

Under web-app directory for example 41 create Dockerfile

```
# Dockerfile
FROM httpd:2.4

COPY ./html/ /usr/local/apache2/htdocs/

EXPOSE 80
```

Step 3: Build the Docker Image

Open a terminal, navigate to the

project directory, and build the Docker image:

bash

```
docker build -t my-web-app .
```

Step 4: Create a Directory on Host

Create a directory on your host machine that you want to mount into the container. For example, let's create a directory named `host-data`:

bash
mkdir host-data

Inside host-data folder create file1.txt files
This is file 1

If file permissions are not allowed to write
Ensure that your user has the necessary permissions to write to the `host-data` directory. You can change the ownership of the directory to your user:
bash
sudo chown -R $(whoami) host-data

Step 5: Run the Docker Container with Volume Mounting

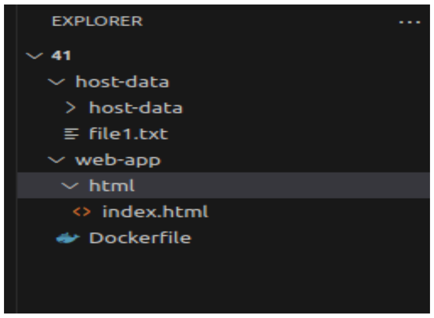
Run the Docker container, and mount the `host-data` directory into the container at the path `/usr/local/apache2/htdocs/`:

bash
docker run -p 8888:80 -v $(pwd)/host-data:/usr/local/apache2/htdocs/ my-web-app

Step 6: Access the Web Application

Open your web browser and go to http://localhost:8080.

Directory structure


```
EXPLORER
∨ 41
  ∨ host-data
    > host-data
    file1.txt
  ∨ web-app
    ∨ html
      index.html
    Dockerfile
```

Exp 44:
Write a C program to create singly linked list and containerize it.

Docker Installation for ubuntu 👍
https://docs.docker.com/engine/install/ubuntu/

Step1: create folder.
Step 2: create linek_list.c file inside that folder.
Program:

```
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node {
   int data;
   struct Node* next;
};

// Function to insert a new node at the beginning of the list
void insertAtBeginning(struct Node** head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = *head;
   *head = newNode;
}

// Function to print the linked list
void printList(struct Node* head) {
   struct Node* current = head;
   while (current != NULL) {
       printf("%d -> ", current->data);
       current = current->next;
   }
   printf("NULL\n");
}

int main() {
   // Initialize an empty linked list
   struct Node* head = NULL;

   // Insert some nodes at the beginning
```

```
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 7);
    insertAtBeginning(&head, 9);

    // Print the linked list
    printf("Linked List: ");
    printList(head);

    return 0;
}
```

2. Create a Dockerfile for containerization:

Dockerfile

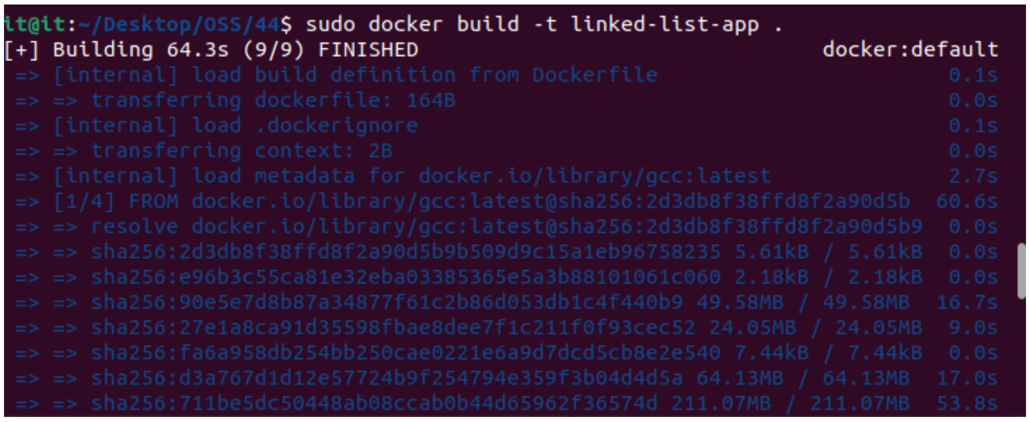
```
# Dockerfile
FROM gcc:latest

WORKDIR /app

COPY linked_list.c .

RUN gcc -o linked_list linked_list.c

CMD ["./linked_list"]
```

This Dockerfile uses the official GCC image to compile the C program and run it. It copies the linked_list.c file into the container, compiles it, and sets the compiled program as the default command.

3. Build the Docker image:

bash

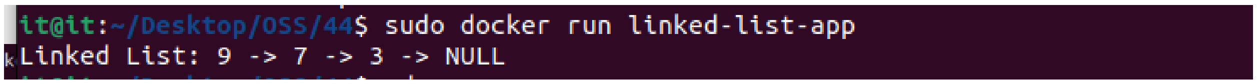
sudo docker build -t linked-list-app .



```
it@it:~/Desktop/OSS/44$ sudo docker build -t linked-list-app .
[+] Building 64.3s (9/9) FINISHED                                   docker:default
 => [internal] load build definition from Dockerfile                          0.1s
 => => transferring dockerfile: 164B                                          0.0s
 => [internal] load .dockerignore                                             0.1s
 => => transferring context: 2B                                               0.0s
 => [internal] load metadata for docker.io/library/gcc:latest                 2.7s
 => [1/4] FROM docker.io/library/gcc:latest@sha256:2d3db8f38ffd8f2a90d5b  60.6s
 => => resolve docker.io/library/gcc:latest@sha256:2d3db8f38ffd8f2a90d5b9  0.0s
 => => sha256:2d3db8f38ffd8f2a90d5b9b509d9c15a1eb96758235 5.61kB / 5.61kB  0.0s
 => => sha256:e96b3c55ca81e32eba03385365e5a3b88101061c060 2.18kB / 2.18kB  0.0s
 => => sha256:90e5e7d8b87a34877f61c2b86d053db1c4f440b9 49.58MB / 49.58MB  16.7s
 => => sha256:27e1a8ca91d35598fbae8dee7f1c211f0f93cec52 24.05MB / 24.05MB  9.0s
 => => sha256:fa6a958db254bb250cae0221e6a9d7dcd5cb8e2e540 7.44kB / 7.44kB  0.0s
 => => sha256:d3a767d1d12e57724b9f254794e359f3b04d4d5a 64.13MB / 64.13MB  17.0s
 => => sha256:711be5dc50448ab08ccab0b44d65962f36574d 211.07MB / 211.07MB  53.8s
```
s

4. Run the Docker container:

bash
sudo docker run linked-list-app



```
it@it:~/Desktop/OSS/44$ sudo docker run linked-list-app
Linked List: 9 -> 7 -> 3 -> NULL
```

This will build the Docker image and execute the C program within a Docker container, creating and printing a simple singly linked list.

Exp 45:
Create a LAMP Stack container and host a web application of your own.

Step 1: Create a Directory

Create a new directory for your project. For example:

bash
mkdir mylampstack
cd mylampstack

Step 2: Create a Dockerfile

Create a file named `Dockerfile` (without any file extension) in the project directory and add the following content:

```
# Use an official PHP runtime as a parent image
FROM php:7.4-apache
```

```
# Set the working directory to /var/www/html
WORKDIR /var/www/html

# Copy the current directory contents into the container at /var/www/html
COPY . /var/www/html

# Enable Apache modules
RUN a2enmod rewrite

# Set the ServerName to suppress the warning
RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf

# Expose port 80
EXPOSE 80

# Start Apache
CMD ["apache2-foreground"]
```

Step 3: Create a PHP Application

Create a simple PHP application. For example, create a file named `index.php` in the project directory with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Simple Web Form</title>
</head>
<body>
   <h1>Simple Web Form</h1>

   <?php
   // Check if the form is submitted
   if ($_SERVER["REQUEST_METHOD"] == "POST") {
       // Retrieve form data
       $name = htmlspecialchars($_POST["name"]);
       $email = htmlspecialchars($_POST["email"]);
       // Display submitted data
       echo "<p>Thank you, $name, for submitting the form!</p>";
       echo "<p>Your email address is: $email</p>";
   }
   ?>

   <form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]); ?>">
       <label for="name">Name:</label>
       <input type="text" id="name" name="name" required>

       <br>

       <label for="email">Email:</label>
       <input type="email" id="email" name="email" required>

       <br>

       <input type="submit" value="Submit">
   </form>
</body>
</html>
```

Step 4: Build the Docker Image

Open a terminal, navigate to your project directory, and run the following command to build the Docker image:

bash
docker build -t mylampstack .

Step 5: Run the Docker Container

Run the Docker container using the following command:

bash
```
docker run -p 8080:80 mylampstack
```

New doc
Exp 35

***Docker :With the help of Docker-compose deploy the 'Wordpress' and 'Mysql' container and access the front end of 'Wordpress'**

Certainly! Below is a step-by-step guide on how to create the necessary files and folders for a Dockerized WordPress setup using Docker Compose.

### Step 1: Create a Project Directory

Start by creating a directory for your project. This is where you will store your `docker-compose.yml` file and other related files.

```bash
mkdir my_wordpress_project
cd my_wordpress_project
```

### Step 2: Create the Docker Compose File

Inside your project directory, create a file named `docker-compose.yml`. You can use a text editor like `nano`, `vim`, or `code` to create and edit the file.

```bash
nano docker-compose.yml
```

Copy and paste the following Docker Compose configuration into the file:

```
version: '3'

services:
 wordpress:
   image: wordpress
   ports:
     - "8080:80"
   depends_on:
     - mysql
   environment:
     WORDPRESS_DB_HOST: mysql
     WORDPRESS_DB_USER: wordpress
     WORDPRESS_DB_PASSWORD: wordpress
     WORDPRESS_DB_NAME: wordpress
   volumes:
     - ./wp-content:/var/www/html/wp-content

 mysql:
   image: mysql:5.7
   environment:
     MYSQL_ROOT_PASSWORD: root_password
     MYSQL_DATABASE: wordpress
     MYSQL_USER: wordpress
     MYSQL_PASSWORD: wordpress
   volumes:
     - ./db-data:/var/lib/mysql

 phpmyadmin:
   image: phpmyadmin/phpmyadmin
   depends_on:
     - mysql
   ports:
     - "8081:80"
   environment:
     PMA_HOST: mysql
     PMA_USER: wordpress
     PMA_PASSWORD: wordpress
     MYSQL_ROOT_PASSWORD: root_password
```

### Step 3: Create Directories for Data

Now, create two directories for data persistence: one for WordPress content and one for the MySQL database.

```bash
mkdir wp-content
mkdir db-data
```

### Step 4: Adjust Permissions (Optional)

Depending on your operating system and user permissions, you might need to adjust file permissions. In many cases, Docker manages this for you. However, if you encounter permission issues, you can run the following commands:

```bash
chmod -R 777 wp-content db-data
```

This command grants read, write, and execute permissions to everyone for the `wp-content` and `db-data` directories. Use this cautiously, especially in production environments.

### Step 5: Run Docker Compose

Now, you are ready to run Docker Compose and deploy your WordPress and MySQL containers.

```bash
docker-compose up -d
```

This command pulls the necessary Docker images, starts the containers in the background, and maps port 8080 on your host to the default WordPress port (80).

### Step 6: Access WordPress Front End

Visit `http://localhost:8080` in your web browser to access the WordPress front end. Follow the on-screen instructions to set up your WordPress site.
Visit `http://localhost:8081` in your web browser to access the phpmyAdmin .

### Step 7: Stop and Remove Containers (Optional)

When you're done, you can stop and remove the containers with the following command:

```bash
```

```
docker-compose down --remove-orphans
```

This command stops and removes the containers and cleans up any orphan containers.

That's it! You've created the necessary files and folders and deployed a Dockerized WordPress setup using Docker Compose.