What is a Dockerfile?

A Dockerfile is a script that contains instructions to build a Docker image. It defines the environment and steps required to set up the application and its dependencies inside a container. Each instruction in the Dockerfile builds upon the previous one, creating a layer in the image.

What is Docker Compose?

docker-compose.yml is a YAML file that defines and manages multi-container Docker applications. It allows you to define multiple services (such as your application and database) and how they interact. You can use it to set up networking, volumes, and other configurations for multiple containers to work together.

Project 1 : Python with PostgreSQL using Docker

In this project, we will create a simple Python web application using Flask that connects to a PostgreSQL database. We'll use Docker to containerize both the Python app and the PostgreSQL database, and Docker Compose to manage them together.

Key Concepts

- **Flask**: A lightweight Python web framework.
- **PostgreSQL**: A relational database to store data.
- **Docker**: A tool to containerize applications for portability.
- **Docker Compose**: A tool to run multi-container applications.

By the end of this project, you'll have a Flask app connected to PostgreSQL, running inside Docker containers.

Step 1: Python Application (app.py)

Create a file app.py with the following code:

python

```
from flask import Flask

import psycopg2

import os


app = Flask(__name__)


DATABASE_URL = os.environ['DATABASE_URL']

conn = psycopg2.connect(DATABASE_URL)

cur = conn.cursor()


@app.route('/')

def hello():

    cur.execute("SELECT 'Hello, World!'")

    result = cur.fetchone()

    return result[0]
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Explanation

- **Flask**: Web framework for the app.
- **psycopg2**: Connects Python to PostgreSQL.
- **DATABASE_URL**: Gets the database URL from the environment variable.

Step 2: Dockerfile

Create a file Dockerfile with the following content:

dockerfile

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["python", "app.py"]
```

Explanation

- **FROM python:3.11-slim**: Uses a lightweight Python image.
- **COPY**: Copies the necessary files into the container.
- **RUN pip install**: Installs the required Python packages.
- **CMD**: Runs the Flask app when the container starts.

The line cur = conn.cursor() in the code creates a **cursor** object, which is used to interact with the PostgreSQL database. Here's a detailed explanation:

What is a Cursor?

A **cursor** is a pointer that allows you to execute SQL queries and retrieve results from the database. It acts as an intermediary between your Python code and the database, enabling you to send SQL commands and fetch data.

How It Works:

- **conn**: This is the connection object, which represents the connection to the PostgreSQL database. It is created by calling psycopg2.connect(DATABASE_URL).
- **cursor()**: This method is called on the connection object (conn) to create a cursor. The cursor is used to execute SQL queries, fetch results, and manage database transactions.

Why Do You Need a Cursor?

You need a cursor to:

- **Execute SQL Queries**: You use the cursor to send SQL commands (like SELECT, INSERT, UPDATE, etc.) to the database.
- **Fetch Data**: After executing a query, the cursor allows you to retrieve the results of that query.

Step 3: Requirements File (requirements.txt)

Create a file requirements.txt with the following content:

plaintext

```
Flask
psycopg2-binary
```

Explanation

- **Flask**: For the web framework.
- **psycopg2-binary**: To connect to PostgreSQL.

Step 4: Docker Compose (docker-compose.yml)

Create a file docker-compose.yml to manage the app and database containers:

yaml

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/mydatabase
    depends_on:
      - db
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

Explanation

- **web**: The Flask app.
  - **build**: Builds the app from the Dockerfile.
  - **ports**: Exposes port 5000.
  - **environment**: Sets the database URL.
- **db**: The PostgreSQL database.
  - **image**: Uses the PostgreSQL image.
  - **volumes**: Stores database data persistently.

Step 5: Running the Application

Build and Start Containers:
Run this command in the project folder:

```
docker-compose up --build
```

1. **Access the App**:
   Open your browser and go to http://localhost:5000. You should see "Hello, World!" fetched from the PostgreSQL database.

Stop the Containers:
Run this command to stop and remove the containers:

```
docker-compose down
```

Conclusion
This project shows how to create a Python Flask app, connect it to PostgreSQL, and run both inside Docker containers using Docker Compose. It's a great way to manage and deploy web applications in isolated environments.

Project 2.Node.js with MongoDB using Docker

In this project, we will create a simple **Node.js** application that connects to a **MongoDB** database. Both the Node.js application and the MongoDB database will run inside Docker containers. We'll use **Docker Compose** to manage these containers and ensure seamless integration between them.

Key Concepts Covered:

- **Node.js**: A JavaScript runtime for building server-side applications.
- **MongoDB**: A NoSQL database for storing JSON-like documents.
- **Docker**: A platform to containerize applications for consistency across environments.
- **Docker Compose**: A tool to manage multi-container applications.

By the end of this tutorial, you'll have a running Node.js app connected to MongoDB, both running in Docker containers.

Step 1: Node.js Application

Create a simple Node.js application that connects to MongoDB and provides an API endpoint.

app.js

javascript

```
const express = require('express');
const mongoose = require('mongoose');

const app = express();
const port = 3000;

// MongoDB connection URI from environment variable
const mongoURI = process.env.MONGO_URI;

// Connect to MongoDB
mongoose
  .connect(mongoURI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB'))
  .catch((err) => console.error('MongoDB connection error:', err));

// Define a simple route
app.get('/', (req, res) => {
  res.send('Hello, Node.js with MongoDB!');
});

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

Explanation:

- **express**: A web framework for Node.js.
- **mongoose**: A library to interact with MongoDB from Node.js.
- **process.env.MONGO_URI**: Reads the MongoDB connection URI from an environment variable.
- **app.get('/')**: Defines a route that responds with a message.
- **app.listen(port)**: Starts the server on the specified port.

Step 2: Dockerfile for Node.js

The Dockerfile defines how to containerize the Node.js application.

Dockerfile

dockerfile

```
# Use an official Node.js runtime as a parent image
FROM node:20

# Set the working directory
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
```

```
RUN npm install

# Copy the application code
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Command to run the application
CMD ["npm", "start"]
```

Explanation:

- **FROM node:20**: Specifies the Node.js 20 base image.
- **WORKDIR /usr/src/app**: Sets the working directory inside the container.
- *_COPY package_.json ./**: Copies package.json and package-lock.json to the container.
- **RUN npm install**: Installs the Node.js dependencies.
- **COPY . .**: Copies all application files into the container.
- **EXPOSE 3000**: Exposes port 3000 for the app.
- **CMD ["npm", "start"]**: Runs the application using the npm start command.

Step 3: package.json

Define the dependencies and scripts for the Node.js application.

package.json

json

```
{
  "name": "node-mongo-docker",
  "version": "1.0.0",
  "description": "Node.js app with MongoDB using Docker",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^7.5.1"
  }
}
```

Explanation:

- **express**: A web framework for building APIs.
- **mongoose**: A MongoDB library for Node.js.

Step 4: Docker Compose Configuration

Use Docker Compose to define and manage the Node.js and MongoDB containers.

docker-compose.yml

yaml

```
version: "3.8"

services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - MONGO_URI=mongodb://db:27017/mydatabase
    depends_on:
      - db

  db:
    image: mongo:6
    container_name: mongo
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

Explanation:

- **web**: The Node.js application service.
  - **build**: Builds the image using the Dockerfile.
  - **ports**: Maps port 3000 on the host to port 3000 in the container.
  - **environment**: Sets the MONGO_URI environment variable for MongoDB connection.
  - **depends_on**: Ensures the db service (MongoDB) starts before the web service.
- **db**: The MongoDB service.
  - **image**: Specifies the MongoDB version (mongo:6).
  - **volumes**: Persists MongoDB data in a volume called mongo_data.

Step 5: Running the Application

Build and Start the Containers: Open a terminal in the project directory and run:

```
docker-compose up --build
```

1. This command:
   - Builds the Docker image for the Node.js app.

    - Starts both the Node.js app and MongoDB containers.
    - Maps ports 3000 (Node.js) and 27017 (MongoDB) for external access.
2. **Access the Application**: Open your browser and navigate to http://localhost:3000. You should see the message: Hello, Node.js with MongoDB!.

Stop the Containers: To stop and remove the containers, run:

```
docker-compose down
```

Conclusion

This guide demonstrates how to set up a Node.js application with MongoDB using Docker and Docker Compose. The containers ensure a consistent and portable development environment, making it easy to deploy the application across different systems.

Project 3: Java with MySQL Using Docker

This project demonstrates how to build a containerized Java application using Spring Boot and MySQL with Docker and Docker Compose. The steps include creating the Java application, writing a Dockerfile, setting up the docker-compose.yml, and running the containers.

---

1. Java Spring Boot Application

Main Application Code

Create a Spring Boot application in the src/main/java directory. Below is an example of a simple application:

src/main/java/com/example/demo/DemoApplication.java

java

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Application Configuration

Configure the database connection and application properties in src/main/resources/application.properties.

src/main/resources/application.properties

properties

```
spring.datasource.url=jdbc:mysql://db:3306/mydatabase

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

server.port=8080
```

---

2. Maven Configuration

Add the necessary dependencies and configurations in the pom.xml file.

pom.xml

xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>

    <artifactId>demo</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <packaging>jar</packaging>

    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <properties>

        <java.version>17</java.version>

    </properties>

    <dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </dependency>

        <dependency>

            <groupId>mysql</groupId>

            <artifactId>mysql-connector-java</artifactId>

            <scope>runtime</scope>

        </dependency>

    </dependencies>

</project>
```

---

3. Dockerfile

Create a Dockerfile to containerize the Spring Boot application.

Dockerfile

dockerfile

```
# Use OpenJDK 17 as the base image

FROM openjdk:17


# Set the working directory

WORKDIR /app


# Copy Maven wrapper and project files

COPY mvnw pom.xml ./

COPY .mvn .mvn

COPY src ./src


# Build the project

RUN ./mvnw clean package -DskipTests


# Expose the application port
```

```
EXPOSE 8080

# Command to run the application

CMD ["java", "-jar", "target/demo-0.0.1-SNAPSHOT.jar"]
```

---

4. Docker Compose Configuration

Create a docker-compose.yml file to define and run multi-container Docker applications. This file will set up the Spring Boot application and MySQL database.

docker-compose.yml

yaml

```
version: "3.8"

services:

  app:

    build:

      context: .

      dockerfile: Dockerfile

    ports:

      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://db:3306/mydatabase
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=root
    depends_on:
      - db

  db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: mydatabase
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```

---

5. Maven Wrapper

Ensure the Maven wrapper files (mvnw, .mvn/) are included in the project for building the application inside the container.

---

6. Build and Run

Step 1: Build and Run Containers

Run the following command in the project directory to build and start the containers:

```
docker-compose up --build
```

This command will build the application and start the containers.

Step 2: Access the Application

Once the containers are running, you can access the application at:

- **Application URL**: http://localhost:8080
- **MySQL Database**: Accessible on localhost:3306

---

7. Verify the Setup

- **Application Logs**: Check the logs to ensure the application connects to the database successfully.
- **Database Connection**: Use a database client (e.g., MySQL Workbench) to verify the MySQL database.

Conclusion

This project demonstrates the integration of a Java Spring Boot application with a MySQL database in a containerized environment. It provides a scalable and portable setup for modern Java development, ensuring consistency across development, testing, and production environments.

Project 4: Rust To-Do App with PostgreSQL

Overview: This project is a simple Rust-based To-Do application that uses PostgreSQL for persistent storage. The application supports basic CRUD (Create, Read, Update, Delete) operations for managing tasks. It is also Dockerized, allowing for easy deployment and scaling.

Key Features:

- **Full CRUD operations** for managing tasks.
- **PostgreSQL** as the database for storing tasks.
- **Dockerized setup** for easy deployment and scalability.

Project Structure:

The project is organized as follows:

css

```
rust-todo-app/
├── Dockerfile
├── docker-compose.yml
├── Cargo.toml
├── src/
│   ├── main.rs
│   ├── db.rs
│   ├── models.rs
│   └── routes.rs
```

Step-by-Step Implementation

1. Cargo.toml: Define Dependencies

This file manages the dependencies for the Rust project. Add the necessary dependencies for PostgreSQL and Actix-web (a web framework).

toml

```
[package]
name = "rust-todo-app"
version = "0.1.0"
edition = "2021"

[dependencies]
actix-web = "4.0"  # Web framework for handling HTTP requests
actix-rt = "2.6"   # Runtime for Actix
serde = { version = "1.0", features = ["derive"] }  # For serializing and
deserializing data
serde_json = "1.0"  # For working with JSON
tokio = { version = "1", features = ["full"] }  # Async runtime
deadpool-postgres = "0.9"  # Connection pool for PostgreSQL
tokio-postgres = "0.7"  # PostgreSQL client
```

2. src/main.rs: Set up the Web Server and Routes

This file sets up the Actix-web server and configures the routes.

rust

```
mod db;
mod models;
mod routes;

use actix_web::{web, App, HttpServer};
use dotenv::dotenv;
use std::env;

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    dotenv().ok();  // Load environment variables from .env file

    let db_pool = db::create_pool().expect("Failed to create pool");

    HttpServer::new(move || {
        App::new()
            .app_data(web::Data::new(db_pool.clone()))  // Share database pool across requests
            .configure(routes::init)  // Set up routes
    })
    .bind(("0.0.0.0", 8080))?  // Listen on all interfaces, port 8080
    .run()
    .await
}
```

3. src/db.rs: Set up Database Connection Pool

This file manages the connection to the PostgreSQL database using the deadpool-postgres library.

rust

```
use deadpool_postgres::{Config, Pool};
use tokio_postgres::NoTls;

pub fn create_pool() -> Result<Pool, deadpool_postgres::ConfigError> {
    let mut cfg = Config::new();
    cfg.host = Some("db".to_string());  // Database container name in Docker
    cfg.user = Some("user".to_string());
    cfg.password = Some("password".to_string());
    cfg.dbname = Some("mydatabase".to_string());
    cfg.create_pool(NoTls)
}
```

4. src/models.rs: Define the Task Model

The Task struct represents the data for each task.

rust

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize)]
pub struct Task {
    pub id: i32,
    pub title: String,
    pub completed: bool,
}
```

5. src/routes.rs: Define API Routes

This file defines the routes for getting and creating tasks.

rust

```
use actix_web::{get, post, web, HttpResponse, Responder};
use deadpool_postgres::Pool;
use crate::models::Task;
```

```
#[get("/tasks")]
async fn get_tasks(db_pool: web::Data<Pool>) -> impl Responder {
    HttpResponse::Ok().json(vec![])  // Placeholder response
}

#[post("/tasks")]
async fn create_task(db_pool: web::Data<Pool>, task: web::Json<Task>) -> impl
Responder {
    HttpResponse::Created().finish()  // Placeholder response
}

pub fn init(cfg: &mut web::ServiceConfig) {
    cfg.service(get_tasks).service(create_task);
}
```

Running the Application Locally

To run the application locally, follow these steps:

1. Install Rust if you haven't already: https://www.rust-lang.org/learn/get-started

Install dependencies and run the application:

```
cargo run
```

The application will be available at http://localhost:8080.

Dockerizing the Application

To make the app easy to deploy and run in any environment, we’ll use Docker.

Dockerfile: Create a Containerized Rust Application

This Dockerfile builds and runs the Rust app inside a container.

dockerfile

```
# Use the official Rust image as the base
FROM rust:1.72 AS builder

WORKDIR /app

COPY Cargo.toml Cargo.lock ./
RUN mkdir src && echo "fn main() {}" > src/main.rs  # Cache dependencies
RUN cargo build --release && rm -rf src

COPY . .  # Copy the rest of the source code
RUN cargo build --release

# Use a minimal base image for the final stage
FROM debian:bullseye-slim

RUN apt-get update && apt-get install -y libssl-dev ca-certificates && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY --from=builder /app/target/release/rust-todo-app .

EXPOSE 8080

CMD ["./rust-todo-app"]
```

docker-compose.yml: Set Up the Application and Database

This file defines the services (Rust app and PostgreSQL database) and their configurations.

yaml

```
version: '3.8'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    depends_on:
      - db
    environment:
      DATABASE_URL: "postgres://user:password@db:5432/mydatabase"
    volumes:
      - ./src:/app/src
    command: ["./rust-todo-app"]

  db:
    image: postgres:15
    container_name: postgres_db
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Running the Application with Docker

To build and start the application using Docker, run the following command:

```
docker-compose up --build
```

Once the containers are up and running, you can access the API at http://localhost:8080.

This project provides a basic structure for a Rust web app with PostgreSQL, allowing users to manage tasks with full CRUD functionality. It is also containerized with Docker for easy deployment.

Project 5: PHP CRUD Application with MySQL

This project demonstrates how to create a simple PHP CRUD (Create, Read, Update, Delete) application that interacts with a MySQL database. The application allows users to perform basic operations such as adding, viewing, editing, and deleting records stored in a MySQL database. The project is containerized using Docker, making it easy to deploy and manage in isolated environments.

Key Features:

- **Create**: Allows users to add new records to the database.
- **Read**: Displays the records stored in the database.
- **Update**: Lets users update existing records.
- **Delete**: Allows users to delete records.

- **Dockerized Application**: Both the PHP application and the MySQL database run in separate Docker containers.

Step-by-Step Guide for PHP CRUD Application with MySQL

1. Create Project Structure

First, create the following directory structure for the project:

arduino

```
php-crud-app/
├── Dockerfile
├── docker-compose.yml
├── index.php
├── create.php
├── update.php
├── delete.php
├── db.php
├── config.php
├── .env
└── sql/
    └── init.sql
```

---

2. Create HTML and PHP for CRUD Operations

Create (create.php)

This page allows users to add new records to the database.

php

```
<?php
include('db.php');
```

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $name = $_POST['name'];
    $email = $_POST['email'];

    // Insert user data into the database
    $sql = "INSERT INTO users (name, email) VALUES ('$name', '$email')";
    if (mysqli_query($conn, $sql)) {
        echo "Record created successfully!";
    } else {
        echo "Error: " . $sql . "<br>" . mysqli_error($conn);
    }

    mysqli_close($conn);
}
?>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Create User</title>
</head>
<body>
    <h1>Create User</h1>
    <form action="create.php" method="POST">
        <label for="name">Name:</label>
        <input type="text" name="name" id="name" required><br><br>

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

        <input type="submit" value="Create">
    </form>
    <a href="index.php">Back to Home</a>
</body>
</html>
```

Read (index.php)

This page displays all the records from the database.

php

```
<?php
include('db.php');

$sql = "SELECT * FROM users";
$result = mysqli_query($conn, $sql);
?>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>View Users</title>
</head>
<body>
    <h1>Users List</h1>
    <table border="1">
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Email</th>
            <th>Actions</th>
        </tr>
        <?php while ($row = mysqli_fetch_assoc($result)) { ?>
            <tr>
```

```
                <td><?php echo $row['id']; ?></td>
                <td><?php echo $row['name']; ?></td>
                <td><?php echo $row['email']; ?></td>
                <td>
                    <a href="update.php?id=<?php echo $row['id']; ?>">Edit</a> |
                    <a href="delete.php?id=<?php echo $row['id']; ?>">Delete</a>
                </td>
            </tr>
        <?php } ?>
    </table>
    <a href="create.php">Create New User</a>
</body>
</html>
```

Update (update.php)

This page allows users to update existing records.

php

```
<?php
include('db.php');

if (isset($_GET['id'])) {
    $id = $_GET['id'];
    $sql = "SELECT * FROM users WHERE id = $id";
    $result = mysqli_query($conn, $sql);
    $row = mysqli_fetch_assoc($result);

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        $name = $_POST['name'];
        $email = $_POST['email'];

        $sql = "UPDATE users SET name = '$name', email = '$email' WHERE id =
$id";
```

```
        if (mysqli_query($conn, $sql)) {
            echo "Record updated successfully!";
        } else {
            echo "Error: " . $sql . "<br>" . mysqli_error($conn);
        }
    }
}

mysqli_close($conn);
?>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Update User</title>
</head>
<body>
    <h1>Update User</h1>
    <form action="update.php?id=<?php echo $row['id']; ?>" method="POST">
        <label for="name">Name:</label>
        <input type="text" name="name" id="name" value="<?php echo
$row['name']; ?>" required><br><br>

        <label for="email">Email:</label>
        <input type="email" name="email" id="email" value="<?php echo
$row['email']; ?>" required><br><br>

        <input type="submit" value="Update">
    </form>
    <a href="index.php">Back to Home</a>
</body>
</html>
```

Delete (delete.php)

This page allows users to delete a record from the database.

php

```php
<?php
include('db.php');

if (isset($_GET['id'])) {
    $id = $_GET['id'];

    $sql = "DELETE FROM users WHERE id = $id";
    if (mysqli_query($conn, $sql)) {
        echo "Record deleted successfully!";
    } else {
        echo "Error: " . $sql . "<br>" . mysqli_error($conn);
    }
}

mysqli_close($conn);
?>

<a href="index.php">Back to Home</a>
```

---

3. Create Database Connection (db.php)

This file contains the code to connect to the MySQL database.

php

```php
<?php
$servername = getenv('MYSQL_HOST');
$username = getenv('MYSQL_USER');
```

```
$password = getenv('MYSQL_PASSWORD');
$dbname = getenv('MYSQL_DB');

// Create connection
$conn = mysqli_connect($servername, $username, $password, $dbname);

// Check connection
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}
?>
```

---

4. Create Dockerfile

This Dockerfile sets up the PHP environment with Apache and MySQL support.

dockerfile

```
# Use the official PHP image with Apache
FROM php:8.2-apache

# Install necessary PHP extensions
RUN docker-php-ext-install mysqli pdo pdo_mysql

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

# Copy the PHP application code into the container
COPY . .

# Expose the port the app runs on
EXPOSE 80

# Start Apache service
```

```
CMD ["apache2-foreground"]
```

---

5. Create Docker Compose File (docker-compose.yml)

This Docker Compose file sets up both the PHP web application and the MySQL database.

yaml

```yaml
version: "3.8"

services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "80:80"
    environment:
      - MYSQL_HOST=db
      - MYSQL_USER=root
      - MYSQL_PASSWORD=root
      - MYSQL_DB=mydatabase
    depends_on:
      - db

  db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: mydatabase
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```

---

6. Create SQL Initialization Script (sql/init.sql)

This file initializes the MySQL database with a users table.

sql

```sql
CREATE DATABASE IF NOT EXISTS mydatabase;

USE mydatabase;

CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE
);
```

How to Run the Project

Build and Start the Docker Containers: Run the following command to build and start the containers using Docker Compose:

```
docker-compose up --build
```

1. **Access the Application:** Once the containers are up and running, open your browser and go to http://localhost. You should see the list of users (which will be empty initially).

2. **Create a New User:** Click on "Create New User" to add a new user to the database.
3. **Edit or Delete a User:** You can edit or delete users by clicking the respective links next to each user in the list.

Check the Database: You can verify that the data has been inserted into the database by accessing the MySQL container and running the following query:
sql

```
SELECT * FROM users;
```

This project provides a complete CRUD application with PHP and MySQL, demonstrating how to use Docker to containerize both the PHP application and the MySQL database.