

# NCC Library Management System CRUD

You need to develop an information system with a backend (MySQL DB and server-side) and a front-end (Client-side; web, Java console app ). You can pick your own domain (a topic) and a platform (Spring Boot, etc.) to develop an application with webservices (both server and client). However, all four CRUD operations need to be implemented at least for one entity. Besides, students need to maintain a GitHub repository with regular submissions and meaningful comments on their development (All the commits need to be error free as it is possible to ask you to pull from any point and run). Example of a bare minimum information system:

- A student information system that manage (Add, edit, view, delete) four attributes such as IndexNo, Name, DoB, and GPA of students.

To implement the described project, here's a plan for developing an information system. The example domain chosen is **Library Management System**, which will manage books in a library.

## Application Features

1. **Domain:** Library Management System
2. **Entities:**
  - **Book:** Attributes include ``BookID``, ``Title``, ``Author``, ``PublishedYear``, and ``ISBN``.
3. **CRUD Operations:** Fully implemented for the **Book** entity.
4. **Technology Stack:**
  - Backend: Spring Boot with MySQL as the database.
  - Frontend:
    - **Web Application:** HTML/CSS/JavaScript with Thymeleaf or ReactJS.
    - **Java Console App:** A simple Java console program to interact with backend services.

## Steps to Develop

### 1. Backend Development

- **Platform:** Spring Boot
- **Key Features:**
  - RESTful APIs for CRUD operations:
    - Create a book: ``POST /books``
    - Read books: ``GET /books`` and ``GET /books/{id}``
    - Update a book: ``PUT /books/{id}``
    - Delete a book: ``DELETE /books/{id}``
  - Database Schema (MySQL):

sql

```
CREATE TABLE Books (
  BookID INT AUTO_INCREMENT PRIMARY KEY,
  Title VARCHAR(255),
  Author VARCHAR(255),
  PublishedYear INT,
  ISBN VARCHAR(13)
);
```

- Example API:

json

```
POST /books
{
  "title": "Clean Code",
  "author": "Robert C. Martin",
  "publishedYear": 2008,
  "isbn": "9780132350884"
}
```

## 2. Frontend Development

- **Web Application:**
  - Form for adding/editing books.
  - Table for viewing the list of books with "Edit" and "Delete" buttons.
- **Java Console App:**
  - Menu-driven program to:
    - Add a book.
    - View all books.
    - Update a book by ID.
    - Delete a book by ID.
  - Example:

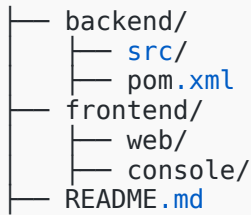
plaintext

```
1. Add a Book
2. View All Books
3. Update a Book
4. Delete a Book
Enter your choice: _
```

## 3. GitHub Repository

- **Structure:**

CSS



- **Requirements:**

- Regular commits with meaningful messages, e.g., `Added Book entity and CRUD APIs`.
- Commit on separate branches for features and merge into the main branch after review.
- Example Commit Messages:
  - Initial setup of Spring Boot project.
  - Implemented Book entity and repository.
  - Developed RESTful endpoints for CRUD operations.
  - Created web form for adding books.
  - Java console app CRUD functionality.

## Example Commit Process

### 1. Create Feature Branch:

bash

```
git checkout -b feature/book-crud
```

### 2. Add and Commit Code:

bash

```
git add .
git commit -m "Implemented CRUD operations for Book entity"
```

### 3. Push Changes:

bash

```
git push origin feature/book-crud
```

### 4. Open Pull Request and merge after code review.

## Deliverables

- Fully functional backend with APIs.

- Frontend with a web interface and Java console app.
- GitHub repository with:
  - Clean commit history.
  - Documentation (``README.md``) explaining the project setup, usage, and APIs.
- Example README content:

markdown

## # Library Management System

### ## Overview

A simple system to manage library books, implemented using Spring Boot, MySQL, and a web/console front-end.

### ## Features

- Add, view, update, and delete books.
- RESTful APIs with JSON responses.
- Web front-end for user interaction.
- Java console app for CLI interaction.

### ## Technologies

- Backend: Spring Boot, MySQL
- Frontend: HTML/JavaScript, Java Console App

Would you like help with any specific code or setup instructions?



**\*\*1\.** Backend Development**\*\***

Give me step by step guidance for this

Here is a step-by-step guide for **backend development** for the Library Management System using Spring Boot with MySQL:

## Step 1: Set Up the Spring Boot Project

### 1. Create a Spring Boot Project:

- Use [Spring Initializr](#) to generate the project.
- Choose the following dependencies:
  - **Spring Web**
  - **Spring Data JPA**
  - **MySQL Driver**
  - **Spring Boot DevTools** (optional, for auto-reloading during development).

### 2. Download and Import:

- Download the project as a ``.zip`` file.
- Import it into your IDE (e.g., IntelliJ IDEA or Eclipse).

### 3. Configure Application Properties: Open ``application.properties`` (or ``application.yml``) and configure the database connection:

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/librarydb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

- Replace `yourpassword` with your MySQL root password.

4. **Create the Database:** Open MySQL Workbench or any SQL client and create the database:

sql

```
CREATE DATABASE librarydb;
```

## Step 2: Create the `Book` Entity

1. **Create the Entity Class:** In the `src/main/java/com/example/library/model` folder, create `Book.java`:

java

```
package com.example.library.model;

import jakarta.persistence.*;

@Entity
@Table(name = "books")
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String author;
    private int publishedYear;
    private String isbn;

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }

    public int getPublishedYear() { return publishedYear; }
    public void setPublishedYear(int publishedYear) { this.publishedYear = publishedYear; }

    public String getIsbn() { return isbn; }
```

```
    public void setIsbn(String isbn) { this.isbn = isbn; }
}
```

## Step 3: Create the Repository

1. **Create Repository Interface:** In `src/main/java/com/example/library/repository`, create `BookRepository.java`:

```
java

package com.example.library.repository;

import com.example.library.model.Book;
import org.springframework.data.jpa.repository.JpaRepository;

public interface BookRepository extends JpaRepository<Book, Long> {
}
```

## Step 4: Create the Service Layer

1. **Create Service Class:** In `src/main/java/com/example/library/service`, create `BookService.java`:

```
java

package com.example.library.service;

import com.example.library.model.Book;
import com.example.library.repository.BookRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }

    public Book getBookById(Long id) {
        return bookRepository.findById(id).orElse(null);
    }

    public Book createBook(Book book) {
        return bookRepository.save(book);
    }

    public Book updateBook(Long id, Book updatedBook) {
        return bookRepository.findById(id).map(book -> {
            book.setTitle(updatedBook.getTitle());
            book.setAuthor(updatedBook.getAuthor());
            book.setPublishedYear(updatedBook.getPublishedYear());
            book.setIsbn(updatedBook.getIsbn());
        }).orElse(null);
    }
}
```

```

        return bookRepository.save(book);
    }).orElse(null);
}

public void deleteBook(Long id) {
    bookRepository.deleteById(id);
}
}

```

## Step 5: Create the Controller

1. **Create Controller Class:** In `src/main/java/com/example/library/controller`, create `BookController.java`:

java

```

package com.example.library.controller;

import com.example.library.model.Book;
import com.example.library.service.BookService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping
    public List<Book> getAllBooks() {
        return bookService.getAllBooks();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable Long id) {
        Book book = bookService.getBookById(id);
        if (book == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(book);
    }

    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookService.createBook(book);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book book) {
        Book updatedBook = bookService.updateBook(id, book);
        if (updatedBook == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(updatedBook);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {

```

```
        bookService.deleteBook(id);  
        return ResponseEntity.noContent().build();  
    }  
}
```

## Step 6: Run and Test the Application

1. **Start the Spring Boot Application:** Run the application using your IDE or by executing:

```
bash  
  
./mvnw spring-boot:run
```

2. **Test the Endpoints:** Use a tool like **Postman** or **cURL** to test the APIs:

- o Create a book:

```
http  
  
POST /api/books  
Content-Type: application/json  
{  
    "title": "Clean Code",  
    "author": "Robert C. Martin",  
    "publishedYear": 2008,  
    "isbn": "9780132350884"  
}
```

- o Get all books:

```
http  
  
GET /api/books
```

- o Update a book:

```
http  
  
PUT /api/books/{id}  
Content-Type: application/json  
{  
    "title": "Updated Title",  
    "author": "Updated Author",  
    "publishedYear": 2020,  
    "isbn": "9780000000000"  
}
```

- o Delete a book:



```
http
```

```
DELETE /api/books/{id}
```

Would you like assistance with a specific step or the frontend development?



**\*\*Create a Spring Boot Project\*\*:**

What is the project type and all . I have Netbeans.

To create a Spring Boot project in **NetBeans**, follow these steps:

## 1. Install Necessary Tools

- **NetBeans IDE:** Ensure you have the latest version of NetBeans with Maven support.
- **JDK:** Install JDK 17 (or later).
- **Spring Boot Plugin for NetBeans:** Install the plugin if it's not already installed. Go to:
  - `Tools > Plugins``
  - Search for **Spring Boot** and install it.

## 2. Create a New Spring Boot Project

1. **Launch NetBeans:** Open the IDE.

2. **Start a New Project:**

- Go to `File > New Project``.
- Select **Maven** under `Java with Maven``.
- Choose **Spring Boot Initializr Project** and click **Next**.

3. **Fill in Spring Boot Project Details:**

- Group: `com.example`` (or your desired package structure).
- Artifact: `library-system``.
- Name: `Library Management System``.
- Description: `A Spring Boot project for managing books in a library``.
- Package: `com.example.library``.
- Java Version: `17`` or compatible.

4. **Add Dependencies:**

- Select the following dependencies:
  - **Spring Web:** For building REST APIs.
  - **Spring Data JPA:** For database operations.
  - **MySQL Driver:** For connecting to MySQL.
  - **Spring Boot DevTools:** For hot-reloading during development.
- Click **Finish**.

### 3. NetBeans Project Structure

After the project is created, the structure will look like this:

```
bash
```

```
Library Management System
├── src/main/java/com/example/library
│   ├── model           # Entity classes like Book.java
│   ├── repository      # Repository interfaces like BookRepository.java
│   ├── service         # Service layer like BookService.java
│   └── controller      # Controller classes like BookController.java
├── src/main/resources
│   └── application.properties # Configuration file for database and other settings
└── pom.xml              # Project dependencies and build configurations
```

### 4. Configure the Database

Edit the `application.properties` file under `src/main/resources` to include your MySQL database credentials:

```
properties
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/librarydb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

### 5. Run the Project

#### 1. Build the Project:

- Right-click on the project and choose `Clean and Build`.

#### 2. Run the Project:

- Right-click on the project and select `Run`.
- The Spring Boot application will start, and you will see output in the console.

#### 3. Access the Application:

- The application will typically run on `http://localhost:8080`.

### 6. Test the REST APIs

You can test the REST endpoints using tools like:

- Postman**
- cURL**
- Or, build the frontend/web client as the next step.

Let me know if you need help with any specific setup or further steps!



**\*\*1\.** Install Necessary Tools**\*\***

Cant see any Spring Boot Plugins in NetBeans

If you can't find a Spring Boot plugin in NetBeans, don't worry. You can still create and manage a Spring Boot project in NetBeans manually. Here's how:

## 1. Use Spring Initializr to Generate the Project

1. Go to [Spring Initializr](#).
2. Configure the project:
  - **Project:** Maven
  - **Language:** Java
  - **Spring Boot Version:** Latest stable version (e.g., 3.x.x)
  - **Group:** ``com.example``
  - **Artifact:** ``library-system``
  - **Dependencies:**
    - Spring Web
    - Spring Data JPA
    - MySQL Driver
    - Spring Boot DevTools
  - Click **Generate** to download a ZIP file of your Spring Boot project.
3. **Extract the ZIP:**
  - Extract the downloaded ZIP file to a folder.

## 2. Import the Project into NetBeans

1. **Open NetBeans.**
2. **Import the Project:**
  - Go to ``File > Open Project``.
  - Navigate to the folder where you extracted the Spring Boot project.
  - Select the folder and click **Open Project**.
3. NetBeans will load the project as a Maven project.

## 3. Add the Database Configuration

1. Open the ``src/main/resources/application.properties`` file.
2. Add your MySQL database credentials:

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/librarydb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

## 4. Build and Run the Project

### 1. Build the Project:

- Right-click on the project in NetBeans and select `Clean and Build`.

### 2. Run the Project:

- Right-click on the project and select `Run`.
- The Spring Boot application will start, and the default server will run on `http://localhost:8080`.

## 5. Install and Test Endpoints

Once your project is up and running:

- You can create REST endpoints in the `controller` package.
- Test them using Postman or a browser.

If you need help with adding code or troubleshooting, let me know!



**Test the Endpoints:** Use a tool like **Postman** or **cURL** to test the APIs:  
What is the complete url of the api ?

The complete URL of the API depends on your local setup and the endpoint paths defined in the `BookController`. Assuming you're running the Spring Boot application locally, here's how the API URLs will look:

### 1. Base URL:

bash

```
http://localhost:8080/api/books
```

- `http://localhost`: Default local server.
- `8080`: Default Spring Boot port.
- `/api/books`: The base mapping defined in `@RequestMapping("/api/books")` in the `BookController`.

# Complete URLs for CRUD Operations

Operation	HTTP Method	Complete URL	Request/Response Example
Get All Books	<code>`GET`</code>	<code>`http://localhost:8080/api/books`</code>	Returns a list of books.
Get Book by ID	<code>`GET`</code>	<code>`http://localhost:8080/api/books/{id}`</code>	Replace <code>`{id}`</code> with the Book ID. Example: <code>`http://localhost:8080/api/books/1`</code>
Create a Book	<code>`POST`</code>	<code>`http://localhost:8080/api/books`</code>	Send a JSON object in the body to create a new book.
Update a Book	<code>`PUT`</code>	<code>`http://localhost:8080/api/books/{id}`</code>	Replace <code>`{id}`</code> with the Book ID and send updated details in the body.
Delete a Book	<code>`DELETE`</code>	<code>`http://localhost:8080/api/books/{id}`</code>	Replace <code>`{id}`</code> with the Book ID. Deletes the book.

## Request/Response Examples

### 1. Get All Books

- Request:

```
http

GET http://localhost:8080/api/books
```

- Response:

```
json

[
  {
    "id": 1,
    "title": "Clean Code",
    "author": "Robert C. Martin",
    "publishedYear": 2008,
    "isbn": "9780132350884"
  },
  {
    "id": 2,
    "title": "Effective Java",
    "author": "Joshua Bloch",
    "publishedYear": 2018,
    "isbn": "9780134685991"
  }
]
```

### 2. Create a New Book

- Request:

http

POST http://localhost:8080/api/books  
Content-Type: application/json

```
{  
  "title": "Design Patterns",  
  "author": "Erich Gamma",  
  "publishedYear": 1994,  
  "isbn": "9780201633610"  
}
```

- **Response:**

json

```
{  
  "id": 3,  
  "title": "Design Patterns",  
  "author": "Erich Gamma",  
  "publishedYear": 1994,  
  "isbn": "9780201633610"  
}
```

---

### 3. Update a Book

- **Request:**

http

PUT http://localhost:8080/api/books/1  
Content-Type: application/json

```
{  
  "title": "Clean Code: Updated Edition",  
  "author": "Robert C. Martin",  
  "publishedYear": 2023,  
  "isbn": "9780132350884"  
}
```

- **Response:**

json

```
{  
  "id": 1,  
  "title": "Clean Code: Updated Edition",  
  "author": "Robert C. Martin",  
  "publishedYear": 2023,  
  "isbn": "9780132350884"  
}
```

## 4. Delete a Book

- **Request:**

```
http
```

```
DELETE http://localhost:8080/api/books/1
```

- **Response:**

```
http
```

```
Status: 204 No Content
```

If you have modified the `server.port` in the `application.properties`, replace `8080` with your configured port. Let me know if you need further assistance!

getting this error while GET `http://localhost:8080/api/books` in postman

To help resolve your issue, I need more details about the error message you're encountering. However, common issues when making a GET request to `http://localhost:8080/api/books` in Postman and their solutions are provided below:

## Common Errors and Solutions

### 1. Error: 404 Not Found

- **Cause:** The endpoint is not correctly mapped or the server is not running.
- **Solution:**
  - Ensure that the `BookController` is annotated with `@RestController` and the `@RequestMapping("/api/books")` mapping is correct.
  - Verify the application is running without errors in the console (check for the message: `Tomcat started on port(s): 8080`).
  - Confirm you're accessing the correct URL (`http://localhost:8080/api/books`).

```
java
```

```
@RestController
@RequestMapping("/api/books")
public class BookController {
    // Controller code
}
```

### 2. Error: 500 Internal Server Error

- **Cause:** There's a problem in the code or database connection.
- **Solution:**
  - Check the console logs for stack traces and debug the issue.
  - Ensure the database (`librarydb`) is created in MySQL and matches the configuration in `application.properties`.

Example database configuration:

```
properties

spring.datasource.url=jdbc:mysql://localhost:3306/librarydb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
```

---

### 3. Error: 401 Unauthorized or 403 Forbidden

- **Cause:** Security settings are blocking the request.
- **Solution:**
  - If you've added Spring Security to the project, ensure you've configured it to allow unauthenticated access to the `/api/books` endpoint.
  - Temporarily disable security in `application.properties` for testing:

```
properties

spring.security.enabled=false
```

---

### 4. Error: Connection Refused

- **Cause:** The application is not running or listening on port `8080`.
- **Solution:**
  - Start the application and ensure it's running on port `8080` (or the port specified in `application.properties`).
  - Verify that no other application is using port `8080`.

Change the port if necessary:

```
properties

server.port=8081
```

Then access: `http://localhost:8081/api/books`.

---

### 5. Empty Response

- **Cause:** There are no records in the database.
- **Solution:**



- Add some test data to the database using the POST endpoint:

```
http

POST http://localhost:8080/api/books
Content-Type: application/json

{
  "title": "Clean Code",
  "author": "Robert C. Martin",
  "publishedYear": 2008,
  "isbn": "9780132350884"
}
```

- Retry the GET request.

---

## Steps to Debug

1. **Verify Logs:** Check the application logs in your IDE or terminal. Look for any errors or exceptions.
2. **Test with a Browser:** Open `http://localhost:8080/api/books` in a browser to confirm the endpoint is accessible.
3. **Verify Controller Code:** Ensure your `BookController` is correct:

```
java

@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping
    public List<Book> getAllBooks() {
        return bookService.getAllBooks();
    }
}
```

4. **Restart the Application:** Stop and restart the Spring Boot application to ensure all changes are applied.

---

Let me know the exact error or share the stack trace, and I'll assist further!

Give me an sql query to add two sample data to the db

Here's how you can insert sample data into your database. Assuming you have a table called `books` in your MySQL database, use the following SQL queries:

---

## Sample Data Insertion Queries

```
sql
```

```
-- Insert the first sample book
INSERT INTO books (title, author, published_year, isbn)
VALUES ('Clean Code', 'Robert C. Martin', 2008, '9780132350884');

-- Insert the second sample book
INSERT INTO books (title, author, published_year, isbn)
VALUES ('Effective Java', 'Joshua Bloch', 2018, '9780134685991');
```

## Prerequisite: Table Schema

Ensure your `books` table is created with the following schema:

```
sql
```

```
CREATE TABLE books (
  id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  author VARCHAR(255) NOT NULL,
  published_year INT NOT NULL,
  isbn VARCHAR(20) NOT NULL
);
```

## How to Execute the Query

1. Open your MySQL client (MySQL Workbench, phpMyAdmin, or CLI).
2. Select the database (`USE librarydb;`) if it's not already selected.
3. Execute the `INSERT` queries above.

## Verify the Data

After inserting, run the following query to confirm the data was added:

```
sql
```

```
SELECT * FROM books;
```

The output should look like this:

id	title	author	published_year	isbn
1	Clean Code	Robert C. Martin	2008	9780132350884
2	Effective Java	Joshua Bloch	2018	9780134685991

Let me know if you need more help!

how to check this api calls in the browser without postman ?

You can test your API calls directly in the browser for **GET requests** or by using simple tools/extensions for other HTTP methods like POST, PUT, and DELETE. Here's how to proceed:

## 1. For GET Requests

You can test the **GET** API calls in your browser by simply typing the URL into the address bar. For example:

- Open your browser and enter:

```
bash
```

```
http://localhost:8080/api/books
```

- If the API works correctly, you'll see the JSON response containing the list of books in your database.

## 2. For Other HTTP Methods (POST, PUT, DELETE)

Browsers do not support testing other HTTP methods directly, but you can use these options:

### Option 1: Use Browser Extensions

#### 1. REST Client Browser Extensions:

- **Google Chrome:**
  - Install an extension like [Restman](#) or [Advanced REST Client](#).
- **Firefox:**
  - Install [RESTED](#).

After installation:

- Open the extension.
- Set the method (GET, POST, PUT, DELETE).
- Enter the URL (e.g., ``http://localhost:8080/api/books``).
- For POST/PUT, add a JSON body as needed.

### Option 2: Use HTML + JavaScript (Fetch API)

Create a simple HTML page with JavaScript to test the API.

#### Example: Test API with Fetch

html

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>API Tester</title>
</head>
<body>
  <h1>Test API Calls</h1>
  <button id="getBooks">Get Books</button>
  <pre id="response"></pre>

  <script>
    document.getElementById("getBooks").addEventListener("click", () => {
      fetch("http://localhost:8080/api/books")
        .then(response => response.json())
        .then(data => {
          document.getElementById("response").textContent = JSON.stringify(data, null, 2);
        })
        .catch(error => console.error("Error:", error));
    });
  </script>
</body>
</html>

```

### Steps to Test:

1. Save this code in a file (e.g., `test-api.html`).
2. Open the file in your browser.
3. Click the **"Get Books"** button, and the response from the API will appear on the page.

### Option 3: Use Command-Line Tools

If you don't want to use the browser:

#### 1. cURL:

- Open a terminal/command prompt.
- Run the following for a GET request:

```
bash
```

```
curl http://localhost:8080/api/books
```

#### 2. HTTPie:

- Install HTTPie: `pip install httpie` (if you have Python installed).
- Run:

```
bash
```

```
http GET http://localhost:8080/api/books
```

Let me know if you need additional guidance!

Give me step by step guidance for this with complete code

Here's a complete step-by-step guide to test your API using an HTML page with JavaScript. The Fetch API is used to send HTTP requests and display responses on a simple web page.

## Step 1: Create an HTML File

1. Open any text editor (e.g., Notepad, VS Code).
2. Create a new file named `test-api.html`.

## Step 2: Add the HTML Code

Copy and paste the following code into the `test-api.html` file:

```
html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>API Tester</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
    }
    button {
      padding: 10px 20px;
      margin: 10px 0;
      background-color: #007BFF;
      color: white;
      border: none;
      border-radius: 5px;
      cursor: pointer;
    }
    button:hover {
      background-color: #0056b3;
    }
    pre {
      background: #f8f9fa;
      padding: 10px;
      border: 1px solid #ccc;
      border-radius: 5px;
      overflow-x: auto;
    }
  </style>
</head>
```

```

<body>
  <h1>Test API Calls</h1>

  <button id="getBooks">GET All Books</button>
  <button id="addBook">POST Create Book</button>
  <button id="updateBook">PUT Update Book</button>
  <button id="deleteBook">DELETE Book</button>

  <h3>API Response:</h3>
  <pre id="response">Click a button to see the API response here.</pre>

  <script>
    const apiUrl = "http://localhost:8080/api/books";
    const responseBox = document.getElementById("response");

    // GET Request
    document.getElementById("getBooks").addEventListener("click", () => {
      fetch(apiUrl)
        .then(response => response.json())
        .then(data => {
          responseBox.textContent = JSON.stringify(data, null, 2);
        })
        .catch(error => {
          responseBox.textContent = "Error: " + error;
        });
    });

    // POST Request
    document.getElementById("addBook").addEventListener("click", () => {
      const newBook = {
        title: "Design Patterns",
        author: "Erich Gamma",
        publishedYear: 1994,
        isbn: "9780201633610"
      };

      fetch(apiUrl, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(newBook)
      })
        .then(response => response.json())
        .then(data => {
          responseBox.textContent = JSON.stringify(data, null, 2);
        })
        .catch(error => {
          responseBox.textContent = "Error: " + error;
        });
    });

    // PUT Request
    document.getElementById("updateBook").addEventListener("click", () => {
      const updatedBook = {
        title: "Clean Code - Updated",
        author: "Robert C. Martin",
        publishedYear: 2010,
        isbn: "9780132350884"
      };

      fetch(`${apiUrl}/1`, { // Replace '1' with the ID of the book to update
        method: "PUT",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(updatedBook)
      })
        .then(response => response.json())
        .then(data => {
          responseBox.textContent = JSON.stringify(data, null, 2);
        })
        .catch(error => {
          responseBox.textContent = "Error: " + error;
        });
    });
  </script>

```

```

});

// DELETE Request
document.getElementById("deleteBook").addEventListener("click", () => {
  fetch(`${apiUrl}/1`, { // Replace '1' with the ID of the book to delete
    method: "DELETE"
  })
  .then(() => {
    responseBox.textContent = "Book deleted successfully.";
  })
  .catch(error => {
    responseBox.textContent = "Error: " + error;
  });
});
</script>
</body>
</html>

```

### Step 3: Open the File in Your Browser

1. Save the file (`test-api.html`) in a folder on your computer.
2. Open the file in your browser (double-click the file or drag it into the browser window).

### Step 4: Test the API

1. **GET All Books:** Click the **GET All Books** button to fetch the list of books.
2. **POST Create Book:** Click the **POST Create Book** button to add a new book to the database.
3. **PUT Update Book:** Click the **PUT Update Book** button to update the book with ID `1`. Replace `1` with the ID you want to update.
4. **DELETE Book:** Click the **DELETE Book** button to delete the book with ID `1`. Replace `1` with the ID you want to delete.

### Expected Output

The response from the API will be displayed in the `

```
` element under "API Response" in JSON format.
```

### Optional Improvements

- Modify the `PUT` and `DELETE` buttons to prompt the user for an ID instead of hardcoding it.
- Add input fields to dynamically test the API.

If you encounter issues, share the error message for further assistance!

The UI not looking nice. I need full crud complete UI for this

For a more visually appealing and complete CRUD UI, we can use **HTML**, **CSS**, and **JavaScript** to build a simple, user-friendly interface. This UI will allow users to perform CRUD operations directly from the browser. Below is a complete solution:

## Step 1: Create `crud-app.html`

Save the following code in a file named `crud-app.html`:

html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Book Management System</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
      background-color: #f4f4f9;
      color: #333;
    }
    h1 {
      text-align: center;
      color: #007BFF;
    }
    .container {
      max-width: 800px;
      margin: 0 auto;
      padding: 20px;
      background: white;
      border-radius: 8px;
      box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    }
    table {
      width: 100%;
      border-collapse: collapse;
      margin: 20px 0;
    }
    table th, table td {
      border: 1px solid #ddd;
      padding: 10px;
      text-align: center;
    }
    table th {
      background-color: #007BFF;
      color: white;
    }
    .form-group {
      margin: 15px 0;
    }
    .form-group label {
      display: block;
      margin-bottom: 5px;
    }
    .form-group input {
      width: 100%;
      padding: 8px;
      box-sizing: border-box;
    }
    button {
      padding: 10px 15px;
      border: none;
      background-color: #007BFF;
      color: white;
      cursor: pointer;
      border-radius: 5px;
      margin-right: 10px;
    }
  </style>
</head>
<body>
  <h1>Book Management System</h1>
  <div class="container">
    <table>
      <tr>
        <th>ID</th>
        <th>Title</th>
        <th>Author</th>
        <th>Year</th>
      </tr>
      <tr>
        <td>1</td>
        <td>The Great Gatsby</td>
        <td>F. Scott Fitzgerald</td>
        <td>1925</td>
      </tr>
      <tr>
        <td>2</td>
        <td>1984</td>
        <td>George Orwell</td>
        <td>1949</td>
      </tr>
    </table>
    <div class="form-group">
      <label>Add New Book</label>
      <input type="text" value="Book Title" />
      <input type="text" value="Author Name" />
      <input type="text" value="Year" />
      <button>Add Book</button>
    </div>
  </div>
</body>
</html>
```



```

        button:hover {
            background-color: #0056b3;
        }
        .buttons {
            text-align: center;
        }
    </style>
</head>
<body>
    <h1>Book Management System</h1>
    <div class="container">
        <h2>Manage Books</h2>

        <!-- Book Form -->
        <form id="bookForm">
            <div class="form-group">
                <label for="title">Title:</label>
                <input type="text" id="title" name="title" required>
            </div>
            <div class="form-group">
                <label for="author">Author:</label>
                <input type="text" id="author" name="author" required>
            </div>
            <div class="form-group">
                <label for="publishedYear">Published Year:</label>
                <input type="number" id="publishedYear" name="publishedYear" required>
            </div>
            <div class="form-group">
                <label for="isbn">ISBN:</label>
                <input type="text" id="isbn" name="isbn" required>
            </div>
            <div class="buttons">
                <button type="submit">Add Book</button>
                <button type="button" id="updateBook" disabled>Update Book</button>
            </div>
        </form>

        <!-- Book Table -->
        <table id="bookTable">
            <thead>
                <tr>
                    <th>ID</th>
                    <th>Title</th>
                    <th>Author</th>
                    <th>Published Year</th>
                    <th>ISBN</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody>
                <!-- Books will be dynamically populated here -->
            </tbody>
        </table>
    </div>

    <script>
        const apiUrl = "http://localhost:8080/api/books";
        const bookTable = document.getElementById("bookTable").querySelector("tbody");
        const bookForm = document.getElementById("bookForm");
        const updateButton = document.getElementById("updateBook");
        let editingBookId = null;

        // Fetch and display books
        function fetchBooks() {
            fetch(apiUrl)
                .then(response => response.json())
                .then(data => {
                    bookTable.innerHTML = ""; // Clear existing rows
                    data.forEach(book => {
                        const row = document.createElement("tr");
                        row.innerHTML = `

```

```

        <td>${book.id}</td>
        <td>${book.title}</td>
        <td>${book.author}</td>
        <td>${book.publishedYear}</td>
        <td>${book.isbn}</td>
        <td>
            <button onclick="editBook(${book.id})">Edit</button>
            <button onclick="deleteBook(${book.id})">Delete</button>
        </td>
    `;
    bookTable.appendChild(row);
  });
}
).catch(error => console.error("Error fetching books:", error));
}

// Add a new book
bookForm.addEventListener("submit", event => {
  event.preventDefault();
  const formData = new FormData(bookForm);
  const bookData = Object.fromEntries(formData);

  if (editingBookId) {
    // Update existing book
    fetch(`${apiUrl}/${editingBookId}`, {
      method: "PUT",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(bookData)
    })
      .then(fetchBooks)
      .catch(error => console.error("Error updating book:", error));
    editingBookId = null;
    updateButton.disabled = true;
  } else {
    // Add new book
    fetch(apiUrl, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(bookData)
    })
      .then(fetchBooks)
      .catch(error => console.error("Error adding book:", error));
  }

  bookForm.reset();
});

// Edit book
window.editBook = function (id) {
  fetch(`${apiUrl}/${id}`)
    .then(response => response.json())
    .then(book => {
      document.getElementById("title").value = book.title;
      document.getElementById("author").value = book.author;
      document.getElementById("publishedYear").value = book.publishedYear;
      document.getElementById("isbn").value = book.isbn;
      editingBookId = id;
      updateButton.disabled = false;
    })
    .catch(error => console.error("Error fetching book:", error));
};

// Delete book
window.deleteBook = function (id) {
  fetch(`${apiUrl}/${id}`, { method: "DELETE" })
    .then(fetchBooks)
    .catch(error => console.error("Error deleting book:", error));
};

// Initial fetch
fetchBooks();

```

```

</script>
</body>
</html>

```

## Step 2: Run the Application

1. Save the file as `crud-app.html`.
2. Open the file in a browser (e.g., Chrome, Firefox).
3. Ensure your Spring Boot application is running.

## Features

- **Add a Book:** Fill in the form and click **Add Book** to create a new book.
- **Edit a Book:** Click **Edit** on a book, update its details in the form, and click **Update Book**.
- **Delete a Book:** Click **Delete** on a book to remove it from the list.
- **List All Books:** All books are displayed in a table.

Let me know if you encounter any issues!

no need of update book botton

Here's the updated code where the **Update Book** button is removed, and editing functionality is streamlined. The form will automatically switch to update mode when a book is selected for editing, and submitting the form will handle both adding and updating books.

## Updated Code: `crud-app.html`

html

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Book Management System</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
      background-color: #f4f4f9;
      color: #333;
    }
    h1 {
      text-align: center;
      color: #007BFF;
    }
    .container {
      max-width: 800px;
      margin: 0 auto;
      padding: 20px;
      background: white;
      border-radius: 8px;
    }
  </style>
</head>
<body>
  <h1>Book Management System</h1>
  <div class="container">
    <div>
      <div>
        <input type="text" value="Book Title" />
        <input type="text" value="Author" />
        <input type="text" value="Genre" />
        <input type="text" value="Year" />
        <input type="button" value="Add Book" />
      </div>
      <div>
        <table>
          <thead>
            <tr>
              <th>Book Title</th>
              <th>Author</th>
              <th>Genre</th>
              <th>Year</th>
              <th>Actions</th>
            </tr>
          </thead>
          <tbody>
            <tr>
              <td>Book 1</td>
              <td>Author 1</td>
              <td>Genre 1</td>
              <td>2020</td>
              <td>
                <input type="button" value="Edit" />
                <input type="button" value="Delete" />
              </td>
            </tr>
          </tbody>
        </table>
      </div>
    </div>
  </div>
</body>
</html>

```

```

        box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    }
    table {
        width: 100%;
        border-collapse: collapse;
        margin: 20px 0;
    }
    table th, table td {
        border: 1px solid #ddd;
        padding: 10px;
        text-align: center;
    }
    table th {
        background-color: #007BFF;
        color: white;
    }
    .form-group {
        margin: 15px 0;
    }
    .form-group label {
        display: block;
        margin-bottom: 5px;
    }
    .form-group input {
        width: 100%;
        padding: 8px;
        box-sizing: border-box;
    }
    button {
        padding: 10px 15px;
        border: none;
        background-color: #007BFF;
        color: white;
        cursor: pointer;
        border-radius: 5px;
        margin-right: 10px;
    }
    button:hover {
        background-color: #0056b3;
    }
    .buttons {
        text-align: center;
    }
</style>
</head>
<body>
    <h1>Book Management System</h1>
    <div class="container">
        <h2>Manage Books</h2>

        <!-- Book Form -->
        <form id="bookForm">
            <div class="form-group">
                <label for="title">Title:</label>
                <input type="text" id="title" name="title" required>
            </div>
            <div class="form-group">
                <label for="author">Author:</label>
                <input type="text" id="author" name="author" required>
            </div>
            <div class="form-group">
                <label for="publishedYear">Published Year:</label>
                <input type="number" id="publishedYear" name="publishedYear" required>
            </div>
            <div class="form-group">
                <label for="isbn">ISBN:</label>
                <input type="text" id="isbn" name="isbn" required>
            </div>
            <div class="buttons">
                <button type="submit" id="submitButton">Add Book</button>
            </div>
        </form>
    </div>
</body>
</html>

```

```

</form>

<!-- Book Table -->
<table id="bookTable">
  <thead>
    <tr>
      <th>ID</th>
      <th>Title</th>
      <th>Author</th>
      <th>Published Year</th>
      <th>ISBN</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    <!-- Books will be dynamically populated here -->
  </tbody>
</table>
</div>

<script>
const apiUrl = "http://localhost:8080/api/books";
const bookTable = document.getElementById("bookTable").querySelector("tbody");
const bookForm = document.getElementById("bookForm");
const submitButton = document.getElementById("submitButton");
let editingBookId = null;

// Fetch and display books
function fetchBooks() {
  fetch(apiUrl)
    .then(response => response.json())
    .then(data => {
      bookTable.innerHTML = ""; // Clear existing rows
      data.forEach(book => {
        const row = document.createElement("tr");
        row.innerHTML = `
          <td>${book.id}</td>
          <td>${book.title}</td>
          <td>${book.author}</td>
          <td>${book.publishedYear}</td>
          <td>${book.isbn}</td>
          <td>
            <button onclick="editBook(${book.id})">Edit</button>
            <button onclick="deleteBook(${book.id})">Delete</button>
          </td>
        `;
        bookTable.appendChild(row);
      });
    })
    .catch(error => console.error("Error fetching books:", error));
}

// Add or update a book
bookForm.addEventListener("submit", event => {
  event.preventDefault();
  const formData = new FormData(bookForm);
  const bookData = Object.fromEntries(formData);

  if (editingBookId) {
    // Update existing book
    fetch(`${apiUrl}/${editingBookId}`, {
      method: "PUT",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(bookData)
    })
      .then(fetchBooks)
      .catch(error => console.error("Error updating book:", error));
    editingBookId = null;
    submitButton.textContent = "Add Book";
  } else {
    // Add new book

```

```

        fetch(apiUrl, {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify(bookData)
        })
        .then(fetchBooks)
        .catch(error => console.error("Error adding book:", error));
    }

    bookForm.reset();
});

// Edit book
window.editBook = function (id) {
    fetch(`${apiUrl}/${id}`)
        .then(response => response.json())
        .then(book => {
            document.getElementById("title").value = book.title;
            document.getElementById("author").value = book.author;
            document.getElementById("publishedYear").value = book.publishedYear;
            document.getElementById("isbn").value = book.isbn;
            editingBookId = id;
            submitButton.textContent = "Update Book";
        })
        .catch(error => console.error("Error fetching book:", error));
};

// Delete book
window.deleteBook = function (id) {
    fetch(`${apiUrl}/${id}`, { method: "DELETE" })
        .then(fetchBooks)
        .catch(error => console.error("Error deleting book:", error));
};

// Initial fetch
fetchBooks();
</script>
</body>
</html>

```

## Changes Made

### 1. Removed Update Button:

- The update functionality is integrated into the form itself.
- The form's submit button dynamically switches between "Add Book" and "Update Book" based on context.

### 2. Streamlined Editing:

- Selecting a book for editing pre-fills the form fields.
- Submitting the form updates the book.

### 3. Delete and Fetch remain the same:

- No changes to deleting books or fetching all books.

## How It Works

1. **Add Book:** Fill in the form and click **Add Book**.

2. **Edit Book:**

- Click **Edit** next to the book.

- The form is pre-filled, and the button changes to **Update Book**.
  - After editing, click the same button to save changes.
3. **Delete Book:** Click **Delete** next to the book to remove it.

---

Let me know if you need further assistance or enhancements!

i need to create a report that illustrate all the functionalities of the developed system

Here's a step-by-step guide and template to create a comprehensive report illustrating the functionalities of the developed system:

## Report Title

**"Book Management System: Functionalities and Features"**

## Table of Contents

1. Introduction
2. System Overview
3. Functionalities
  - Add Books
  - View Books
  - Edit Books
  - Delete Books
4. Architecture and Design
  - System Architecture
  - Frontend and Backend Design
5. Technologies Used
6. API Endpoints
7. Testing and Validation
8. Screenshots and Illustrations
9. Conclusion

---

## 1. Introduction

Briefly describe the purpose of the system.

The Book Management System is a web application designed to manage books in a library or inventory system. It supports essential CRUD operations such as adding, viewing, editing, and deleting books. The application is built with Spring Boot for the backend and a user-friendly HTML, CSS, and JavaScript interface for the frontend.

---

## 2. System Overview

Provide a high-level overview of the system's capabilities.

The system allows users to:

- Add new books with details such as title, author, published year, and ISBN.
- View all books in a tabular format.
- Edit existing book details.
- Delete books from the system.

It interacts with a backend API, ensuring seamless data handling and persistence in a MySQL database.

---

### 3. Functionalities

#### Add Books

- Users can fill in the form fields for title, author, published year, and ISBN to add a book.
- Clicking the **Add Book** button sends a POST request to the backend API.

#### API Endpoint:

```
http
```

```
POST /api/books
```

---

#### View Books

- The system retrieves all books and displays them in a table format.
- Each row contains the book's ID, title, author, published year, ISBN, and action buttons.

#### API Endpoint:

```
http
```

```
GET /api/books
```

---

#### Edit Books

- Clicking the **Edit** button next to a book pre-fills the form fields with the book's current details.
- Updating the details and submitting the form sends a PUT request to the backend.

#### API Endpoint:



```
http
```

```
PUT /api/books/{id}
```

---

## Delete Books

- Users can click the **Delete** button to remove a book from the system.
- A DELETE request is sent to the backend, and the list of books is updated.

### API Endpoint:

```
http
```

```
DELETE /api/books/{id}
```

---

## 4. Architecture and Design

### System Architecture

Include a diagram showing the system components:

1. Client-side (Browser with HTML/JavaScript).
2. Backend (Spring Boot REST API).
3. Database (MySQL).

### Frontend Design

- **HTML:** Structure of the user interface.
- **CSS:** Styling for a user-friendly appearance.
- **JavaScript:** Handles API calls and DOM updates.

### Backend Design

- **Spring Boot:** RESTful API implementation with controllers, services, and repositories.

---

## 5. Technologies Used

- **Frontend:** HTML, CSS, JavaScript
- **Backend:** Spring Boot
- **Database:** MySQL
- **Tools:** NetBeans, Postman (for testing APIs)

---

## 6. API Endpoints

Summarize the available endpoints:

HTTP Method	Endpoint	Description
GET	<code>`/api/books`</code>	Fetch all books
GET	<code>`/api/books/{id}`</code>	Fetch a specific book
POST	<code>`/api/books`</code>	Add a new book
PUT	<code>`/api/books/{id}`</code>	Update an existing book
DELETE	<code>`/api/books/{id}`</code>	Delete a book

## 7. Testing and Validation

Explain how the system was tested:

- 1. **Frontend Testing:**
  - Form submission for valid and invalid inputs.
  - UI updates after API responses.
- 2. **API Testing:**
  - Used Postman to test all API endpoints.
  - Verified data persistence in MySQL.
- 3. **Error Handling:**
  - Validation for empty or invalid fields.

## 8. Screenshots and Illustrations

Include screenshots of the following:

- Add Book functionality in action.
- Viewing all books.
- Editing a book's details.
- Deleting a book.

## 9. Conclusion

Summarize the system’s achievements and potential future enhancements.

The Book Management System successfully implements all CRUD operations with a seamless user interface and robust backend API. Future improvements could include adding user authentication, search functionality, and exporting book data.

## Final Touch

Save this as a **PDF report** with a title page and relevant formatting. Let me know if you need help designing a title page or creating diagrams!

I want to convert this backend springboot project as a docker image and need to host on dockerhub . i am using M1 Macbook.

To convert your Spring Boot backend project into a Docker image and host it on Docker Hub, follow these steps:

## 1. Install Required Tools

Ensure the following are installed on your M1 MacBook:

1. **Docker:** Install Docker Desktop for Mac ([Download Here](#)).
2. **JDK:** Ensure you have the correct JDK version installed for your Spring Boot project.

## 2. Create a `Dockerfile`

Add a `Dockerfile` in the root directory of your Spring Boot project.

Here's the content for your `Dockerfile`:

```
dockerfile

# Use an appropriate base image compatible with M1 Mac (arm64 architecture)
FROM openjdk:17-alpine

# Set the working directory inside the container
WORKDIR /app

# Copy the JAR file to the container
COPY target/<your-jar-name>.jar app.jar

# Expose the application port
EXPOSE 8080

# Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Replace `**<your-jar-name>**` with the name of your built JAR file (e.g., `**book-management-system-1.0.0.jar**`).

## 3. Build Your Spring Boot Project

Run the following command to build the JAR file:

```
bash

./mvnw clean package
```

The built JAR file should appear in the `**target/**` directory.

## 4. Build the Docker Image

Run the following command in the terminal from the project directory where the `Dockerfile` is located:

```
bash

docker build -t <dockerhub-username>/book-management-system:latest .
```

Replace `<dockerhub-username>` with your Docker Hub username.

---

## 5. Test the Docker Image Locally

Run the image locally to verify that it works:

```
bash

docker run -p 8080:8080 <dockerhub-username>/book-management-system:latest
```

Open a browser or Postman and visit `http://localhost:8080/api/books` to ensure the API is accessible.

---

## 6. Push the Docker Image to Docker Hub

### 1. Log in to Docker Hub:

```
bash

docker login
```

### 2. Push the image:

```
bash

docker push <dockerhub-username>/book-management-system:latest
```

## 7. Verify the Image on Docker Hub

Log in to your Docker Hub account and confirm that the image has been successfully uploaded.

---

## 8. Run the Docker Image from Docker Hub

To test the image from Docker Hub on any machine:

```
bash
```

```
docker run -p 8080:8080 <dockerhub-username>/book-management-system:latest
```

## M1 MacBook Considerations

Since you are using an M1 MacBook, ensure that:

1. Your base image (`openjdk:17-alpine`) supports **arm64 architecture**. Most modern images do.
2. If compatibility issues arise, use a multi-platform base image or an emulation layer like `--platform linux/amd64` when building the image:

```
bash
```

```
docker build --platform linux/amd64 -t <dockerhub-username>/book-management-system:latest .
```

## Optional: Multi-Platform Build for x86 and arm64

If you need compatibility across different architectures, use Docker Buildx:

1. Enable Buildx:

```
bash
```

```
docker buildx create --use
```

2. Build for multiple platforms:

```
bash
```

```
docker buildx build --platform linux/amd64,linux/arm64 -t <dockerhub-username>/book-management-system:latest --push .
```

This ensures your Docker image works on x86 and arm64 machines.

Let me know if you need help with any of the steps!



**\*\*4\.** Build the Docker Image**\*\***

getting this error:

```
zakee@MacBookAir library-system % docker build -t zakee14910/library-system:latest .
```

```
[+] Building 2.6s (3/3) FINISHED          docker:desktop-linux
=> [internal] load .dockerignore          0.0s
=> => transferring context: 2B            0.0s
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 398B      0.0s
=> ERROR [internal] load metadata for docker.io/library/openjdk:17-alpine 2.6s
```

```
-----
> [internal] load metadata for docker.io/library/openjdk:17-alpine:
-----
```

```
Dockerfile:2
```

```
-----
1 | # Use an appropriate base image compatible with M1 Mac (arm64 architecture)
2 | >>> FROM openjdk:17-alpine
3 |
4 | # Set the working directory inside the container
-----
```

```
ERROR: failed to solve: openjdk:17-alpine: no match for platform in manifest
sha256:4b6abae565492dbe9e7a894137c966a7485154238902f2f25e9dbd9784383d81: not found
```

The error occurs because the `openjdk:17-alpine` image does not have support for the ARM64 architecture (used by M1 MacBooks). To fix this issue, use an alternative base image or specify the platform explicitly.

## Solution 1: Use an ARM64-Compatible Base Image

Replace `openjdk:17-alpine` with an image that supports ARM64, such as `eclipse-temurin`. Update your `Dockerfile` like this:

```
dockerfile
```

```
# Use an ARM64-compatible OpenJDK image
FROM eclipse-temurin:17-jdk-alpine

# Set the working directory inside the container
WORKDIR /app

# Copy the JAR file to the container
COPY target/<your-jar-name>.jar app.jar

# Expose the application port
EXPOSE 8080

# Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

## Solution 2: Use `--platform` to Specify Emulation

If you must use `openjdk:17-alpine`, you can use Docker's emulation feature to build the image for the `amd64` architecture:

1. Add the `--platform linux/amd64` flag when building:

```
bash

docker build --platform linux/amd64 -t zakee14910/library-system:latest .
```

2. Alternatively, use Docker Buildx for cross-platform builds:

```
bash

docker buildx create --use
docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest --push .
```

---

## Solution 3: Check Available Tags

Verify the available tags for `openjdk` by running:

```
bash

docker search openjdk
```

Choose an appropriate image with ARM64 or multi-platform support.

---

## Rebuild and Test

After updating your `Dockerfile` with one of the solutions, rebuild the image:

```
bash

docker build -t zakee14910/library-system:latest .
```

Then test it locally:

```
bash

docker run -p 8080:8080 zakee14910/library-system:latest
```

Let me know if you face any further issues!

```
getting this error:
zakee@MacBookAir library-system % docker buildx create --use
```

```
docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest --push .
```

recurring\_dhawan

```
[+] Building 46.2s (5/5) FINISHED          docker-container:recurring_dhawan
```

```
=> [internal] booting buildkit          42.7s
```

```
=> => pulling image moby/buildkit:buildx-stable-1          42.0s
```

```
=> => creating container buildx_buildkit_recurring_dhawan0          0.7s
```

```
=> [internal] load build definition from Dockerfile          0.0s
```

```
=> => transferring dockerfile: 374B          0.0s
```

```
=> CANCELED [linux/amd64 internal] load metadata for docker.io/library/e          3.3s
```

```
=> ERROR [linux/arm64 internal] load metadata for docker.io/library/ecli          3.3s
```

```
=> [auth] library/eclipse-temurin:pull token for registry-1.docker.io          0.0s
```

-----

```
> [linux/arm64 internal] load metadata for docker.io/library/eclipse-temurin:17-jdk-alpine:
```

-----

Dockerfile:2

-----

```
1 | # Use an ARM64-compatible OpenJDK image
```

```
2 | >>> FROM eclipse-temurin:17-jdk-alpine
```

```
3 |
```

```
4 | # Set the working directory inside the container
```

-----

```
ERROR: failed to solve: eclipse-temurin:17-jdk-alpine: failed to resolve source metadata for
docker.io/library/eclipse-temurin:17-jdk-alpine: no match for platform in manifest: not found
```

The error indicates that the `eclipse-temurin:17-jdk-alpine` image does not support the `linux/arm64` architecture. Here's how to resolve the issue:

## 1. Use a Multi-Platform Image

Switch to a multi-platform-compatible base image. The `azul/zulu-openjdk` image supports both `linux/amd64` and `linux/arm64`. Update your `Dockerfile`:

dockerfile

```
# Use an ARM64-compatible OpenJDK image
FROM azul/zulu-openjdk:17-jdk-alpine
```

```
# Set the working directory inside the container
WORKDIR /app
```

```
# Copy the JAR file to the container
COPY target/<your-jar-name>.jar app.jar
```

```
# Expose the application port
EXPOSE 8080
```

```
# Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```



Replace ``<your-jar-name>`` with the name of your JAR file (e.g., ``library-system.jar``).

---

## 2. Specify Platform for Buildx

Use the following command to build the image for both ``linux/amd64`` and ``linux/arm64``:

```
bash

docker buildx build --platform linux/amd64,linux/arm64 -t zakeel4910/library-system:latest -
-push .
```

---

## 3. Test the Image Locally

Test the image on your local system to ensure it works:

```
bash

docker run -p 8080:8080 zakeel4910/library-system:latest
```

---

## 4. Validate Platform Compatibility

If the problem persists, check available tags and supported platforms for the image by running:

```
bash

docker manifest inspect azul/zulu-openjdk:17-jdk-alpine
```

You can also search for other multi-platform Java images:

```
bash

docker search openjdk
```

---

## 5. Alternative: Use Debian-Based Images

If the ``alpine`` variant continues to cause issues, switch to a ``Debian``-based image, which is generally more compatible:

```
dockerfile
```

```
FROM openjdk:17-jdk-slim
```

The `openjdk:17-jdk-slim` image is lightweight and works across multiple platforms, including ARM64.

## 6. Rebuild and Test Again

Rebuild the image after updating the `Dockerfile`:

```
bash
```

```
docker build -t zakee14910/library-system:latest .
```

Then push to Docker Hub:

```
bash
```

```
docker push zakee14910/library-system:latest
```

Let me know if you encounter further issues!

getting this:

```
zakee@MacBookAir library-system % docker buildx create --use
docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest --push .
```

```
sleepy_ishizaka
```

```
[+] Building 8.6s (5/5) FINISHED          docker-container:sleepy_ishizaka
```

```
=> [internal] booting buildkit          5.4s
```

```
=> => pulling image moby/buildkit:buildx-stable-1          5.0s
```

```
=> => creating container buildx_buildkit_sleepy_ishizaka0    0.4s
```

```
=> [internal] load build definition from Dockerfile          0.0s
```

```
=> => transferring dockerfile: 453B          0.0s
```

```
=> ERROR [linux/arm64 internal] load metadata for docker.io/azul/zulu-op 3.0s
```

```
=> ERROR [linux/amd64 internal] load metadata for docker.io/azul/zulu-op 3.0s
```

```
=> [auth] azul/zulu-openjdk:pull token for registry-1.docker.io    0.0s
```

```
-----
```

```
> [linux/arm64 internal] load metadata for docker.io/azul/zulu-openjdk:17-jdk-alpine:
```

```
-----
```

```
-----
```

```
> [linux/amd64 internal] load metadata for docker.io/azul/zulu-openjdk:17-jdk-alpine:
```

```
-----
```

```
Dockerfile:5
```

```
-----
```

```

3 |
4 |   # Use an ARM64-compatible OpenJDK image
5 | >>> FROM azul/zulu-openjdk:17-jdk-alpine
6 |
7 |   # Set the working directory inside the container

```

```

-----
ERROR: failed to solve: azul/zulu-openjdk:17-jdk-alpine: failed to resolve source metadata for
docker.io/azul/zulu-openjdk:17-jdk-alpine: docker.io/azul/zulu-openjdk:17-jdk-alpine: not found

```

The error indicates that the `azul/zulu-openjdk:17-jdk-alpine` image does not exist or is not available for the requested platform. Let's address this issue.

## Step 1: Use a Different Base Image

Switch to a base image that is compatible with ARM64 and widely available. For instance, the `adoptopenjdk` or `temurin` images are good alternatives.

### Update Your `Dockerfile`:

Replace the base image with:

```

dockerfile

FROM adoptopenjdk:17-jdk-hotspot-bionic

```

Or:

```

dockerfile

FROM eclipse-temurin:17-jdk

```

Both are compatible with ARM64 and do not rely on Alpine Linux.

## Step 2: Rebuild the Image

Run the following commands to build and push the Docker image:

```

bash

docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest -
-push .

```

## Step 3: Verify Available Tags

To ensure you are using a valid base image, check for the available tags for `adoptopenjdk` or `eclipse-temurin`:`

```
bash
```

```
docker search adoptopenjdk
docker search eclipse-temurin
```

Alternatively, view the tags in Docker Hub:

- [AdoptOpenJDK Docker Hub](#)
- [Eclipse Temurin Docker Hub](#)

---

## Step 4: Use a Non-Alpine Image if Necessary

If lightweight images like `*-alpine` are causing issues, use standard Debian-based images (e.g., `*-slim` ) that are more robust and widely supported:`

```
dockerfile
```

```
FROM openjdk:17-jdk-slim
```

---

## Step 5: Test Locally

After updating the `Dockerfile` and successfully building the image, run the image locally:`

```
bash
```

```
docker run -p 8080:8080 zakee14910/library-system:latest
```

Check your application at `http://localhost:8080`.`

If issues persist, let me know so we can explore additional options.

What needs to be executed after this:

```
docker buildx create --use
```

```
docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest --push .
```

After executing the commands:

```
bash
```

```
docker buildx create --use
docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest -
-push .
```

The image will be built for the specified platforms (`linux/amd64` and `linux/arm64`) and pushed to your Docker Hub repository (`zakee14910/library-system:latest`).

Here's what you should do next:

## 1. Verify the Docker Image on Docker Hub

1. Log in to your Docker Hub account ([Docker Hub](#)).
2. Search for your repository (`library-system`) and ensure the image is available with the `latest` tag.

## 2. Test the Image

You can now pull the image from Docker Hub and run it to ensure it works as expected.

### On Local Machine:

```
bash
```

```
docker pull zakee14910/library-system:latest
docker run -p 8080:8080 zakee14910/library-system:latest
```

### Access the Application:

Visit `http://localhost:8080` or perform API calls to endpoints such as `http://localhost:8080/api/books`.

## 3. Deploy the Image (Optional)

If you plan to host your Docker image on a cloud platform or deploy it in a production environment, you can use platforms like:

- **AWS Elastic Beanstalk**
- **Google Kubernetes Engine (GKE)**
- **Azure Kubernetes Service (AKS)**
- **Heroku (with Docker support)**

## 4. Clean Up Unused Resources

To avoid clutter or running unnecessary containers, clean up:

```
bash
```

```
docker ps -a          # List all containers
docker rm <container> # Remove unused containers
docker images         # List all images
docker rmi <image>    # Remove unused images
```

Let me know if you need guidance on testing or deploying the image!

I don't have any MySQL image on my Mac below is my local MySQL details. Give me step by step guidance to create my SQL image similar to this to run my project without any Docker network : my application.properties

```
spring.application.name=library-system
spring.datasource.url=jdbc:mysql://localhost:3306/librarydb
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

To run your Spring Boot application with MySQL locally without using Docker networks, you'll need to set up a local MySQL container. Here's a step-by-step guide to do that:

## Step 1: Pull the Official MySQL Docker Image

You can pull the official MySQL image from Docker Hub. The MySQL Docker image will run on your Mac and expose the MySQL server on a specific port (e.g., `3306`).

```
bash
```

```
docker pull mysql:latest
```

## Step 2: Run MySQL Container

Next, you need to run a MySQL container with the settings that match your `application.properties`. Use the following command to start a MySQL container on your local machine:

```
bash
```

```
docker run --name mysql-container -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=librarydb -p 3306:3306 -d mysql:latest
```

Here's what each option means:

- `--name mysql-container`: Names the container `mysql-container`.
- `-e MYSQL_ROOT_PASSWORD=root`: Sets the root password to `root`.
- `-e MYSQL_DATABASE=librarydb`: Creates a database called `librarydb` on MySQL.
- `-p 3306:3306`: Exposes MySQL on port `3306` (the default port).
- `-d mysql:latest`: Runs the container in detached mode using the latest MySQL image.

This will set up a MySQL server running on your local machine with the `librarydb` database.

### Step 3: Modify Your `application.properties`

Make sure your `application.properties` file is correctly configured for the MySQL container. Since we are running MySQL locally, you can leave the connection details the same as you've mentioned.

`properties`

```
spring.application.name=library-system
spring.datasource.url=jdbc:mysql://localhost:3306/librarydb
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

### Key Points:

- `spring.datasource.url`: Refers to `localhost:3306` where your MySQL container is running.
- `spring.datasource.username`: The MySQL root username (`root`).
- `spring.datasource.password`: The root password you set earlier (`root`).
- `spring.jpa.hibernate.ddl-auto=update`: This auto-updates the schema. You can use `create-drop` or `validate` based on your needs.
- `spring.jpa.show-sql=true`: To see SQL queries logged in the console.

### Step 4: Verify MySQL is Running

To verify that your MySQL container is running properly, use the following command to list all running containers:

`bash`

```
docker ps
```

You should see something like this:

```
lua
```

CONTAINER ID NAMES	IMAGE	COMMAND	CREATED	STATUS	PORTS
<container-id> 3306->3306/tcp	mysql:latest mysql-container	"docker-entrypoint.s..."	<time>	Up <time>	0.0.0.0:

If you don't see the container, you can restart it:

```
bash
```

```
docker start mysql-container
```

## Step 5: Access MySQL from Inside the Container (Optional)

If you want to access MySQL directly from the container, you can use the following command:

```
bash
```

```
docker exec -it mysql-container mysql -u root -p
```

Then, enter the password (`root`), and you should have access to the MySQL shell.

## Step 6: Test the Connection with Spring Boot

Now, run your Spring Boot application:

```
bash
```

```
./mvnw spring-boot:run
```

Your application should successfully connect to the MySQL database (`librarydb`) running in the Docker container.

## Step 7: Access MySQL via a GUI (Optional)

If you want to interact with MySQL using a GUI like MySQL Workbench or DBeaver:

- **Host:** `localhost`
- **Port:** `3306`
- **Username:** `root`
- **Password:** `root`



## Step 8: Stopping and Restarting MySQL Container (Optional)

- To stop the MySQL container:

```
bash

docker stop mysql-container
```

- To restart the MySQL container:

```
bash

docker start mysql-container
```

---

## Step 9: Clean Up (Optional)

If you no longer need the MySQL container, you can remove it with:

```
bash

docker rm -f mysql-container
```

This will remove the container, but it will not delete the image or database. If you need to remove the image as well:

```
bash

docker rmi mysql:latest
```

---

## Summary

You've now set up a MySQL container running on your local machine with the database `librarydb`. Your Spring Boot application can connect to this container without using Docker networks, simply by connecting to `localhost:3306`.

Let me know if you encounter any issues!

```
how to stop this run : ./mvnw spring-boot:run
```

To stop a Spring Boot application that was started using the `./mvnw spring-boot:run` command, you can do the following:

## 1. Stop the Process in Terminal

If you started the application in the same terminal window, you can simply stop it by pressing:

```
bash
```

```
Ctrl + C
```

This will gracefully stop the running application.

## 2. If the Application is Running in the Background (Optional)

If you started the application in the background (e.g., with `&` or through a separate terminal session), you can find the process and kill it manually.

### Find the Process ID (PID)

To find the PID of the running application, use the following command:

```
bash
```

```
ps aux | grep 'spring-boot:run'
```

This will show you the running processes that match the search term. For example, you might see something like:

```
yaml
```

```
yourusername  12345  1.2  0.3  34000  3500 pts/0    S+   10:00   0:03 java -jar your-app.jar
```

### Kill the Process

Once you identify the PID (e.g., `12345`), you can stop it by using the `kill` command:

```
bash
```

```
kill 12345
```

If the process doesn't stop, you can forcefully terminate it with:

```
bash
```

```
kill -9 12345
```

### 3. Stopping with `mvn` (If it's a Maven Process)

If you started the application with Maven, you can also stop it by navigating to the project directory and running:

```
bash
```

```
mvn spring-boot:stop
```

However, note that you may need the `spring-boot-maven-plugin` in your `pom.xml` for this command to work.

Let me know if you need more clarification on any of these steps!

my sql is working fine now. now i need to run my backend image along with this bd.

Great! Now that your MySQL database is running successfully, you can run your Spring Boot backend Docker image alongside it. You need to ensure that the backend application connects to the MySQL instance. Here's how you can achieve this:

## Step-by-Step Guide to Run Both MySQL and Spring Boot Backend in Docker

Since you're not using Docker networks, we will run both the MySQL container and the Spring Boot container separately. Your Spring Boot container will connect to the MySQL container on `localhost` (since you're running them locally on your machine).

### 1. Build and Push Your Spring Boot Docker Image (If Not Done Already)

If you haven't built and pushed your Spring Boot Docker image yet, do that first. You can skip this if you already have the image.

```
bash
```

```
docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest -  
-push .
```

This will build the image and push it to Docker Hub.

## 2. Run the MySQL Docker Container (If Not Done Already)

If you haven't already run the MySQL container, use this command:

```
bash
```

```
docker run --name mysql-container -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=librarydb -p 3306:3306 -d mysql:latest
```

Ensure the MySQL container is running, and you can verify it with:

```
bash
```

```
docker ps
```

## 3. Run the Spring Boot Application Docker Container

Now, run your Spring Boot backend Docker container. Make sure it connects to the local MySQL container.

```
bash
```

```
docker run --name library-system-container -p 8080:8080 -e SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/librarydb -e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root -d zakeel4910/library-system:latest
```

Here's the breakdown of the above command:

- `--name library-system-container`: Names the container `library-system-container`.
- `-p 8080:8080`: Exposes port `8080` from the container to your local machine. This allows you to access the Spring Boot application at `http://localhost:8080`.
- `-e SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/librarydb`: Sets the MySQL connection URL to `localhost:3306` where your MySQL container is running.
- `-e SPRING_DATASOURCE_USERNAME=root`: Sets the username to `root` for the MySQL database.
- `-e SPRING_DATASOURCE_PASSWORD=root`: Sets the password to `root` for the MySQL database.
- `-d`: Runs the container in detached mode (in the background).

## 4. Verify Both Containers are Running

You can verify that both containers are running by using the following command:

```
bash
```

```
docker ps
```

You should see both containers (`mysql-container` and `library-system-container`) listed.

Example output:

```
perl
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
TUS	PORTS	NAMES		
<container-id>	mysql:latest	"docker-entrypoint.s..."	<time>	Up
<time>	0.0.0.0:3306->3306/tcp	mysql-container		
<container-id>	zakeel4910/library-system:latest	"java -jar /app.jar"	<time>	Up
<time>	0.0.0.0:8080->8080/tcp	library-system-container		

## 5. Access Your Spring Boot Application

Now, open your browser or use Postman to test your application. It should be running at:

```
arduino
```

```
http://localhost:8080
```

You should be able to interact with the backend (e.g., perform CRUD operations on your `books` API).

## 6. Optional: Access MySQL Database Directly

If you want to access MySQL directly, you can use MySQL Workbench or another MySQL client and connect to:

- **Host:** `localhost`
- **Port:** `3306`
- **Username:** `root`
- **Password:** `root`

Alternatively, you can access the MySQL shell via Docker:

```
bash
```

```
docker exec -it mysql-container mysql -u root -p
```

Then enter the password (`root`).

## 7. Clean Up (Optional)

To stop and remove the containers when you're done, you can run:

```
bash

docker stop mysql-container library-system-container
docker rm mysql-container library-system-container
```

If you no longer need the images:

```
bash

docker rmi mysql:latest zakee14910/library-system:latest
```

## Summary

- You've set up MySQL to run in a Docker container.
- You've run your Spring Boot application in a separate Docker container that connects to MySQL.
- You can access both the backend application at `http://localhost:8080` and interact with the MySQL database.

Let me know if you run into any issues or need further clarification!



**\*\*3\ Run the Spring Boot Application Docker Container\*\***

```
zakee@MacBookAir library-system % docker run --name library-system-container -p 8080:8080 -e
SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/librarydb -e
SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root -d
zakee14910/library-system:latest
```

```
docker: Error response from daemon: Conflict. The container name "/library-system-container" is
already in use by container
"1563259161c0ba90bc6719ea50348e76e7beab759843ad1e08d7319d80e5add0". You have to
remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
```

The error you're encountering happens because you are trying to create a container with the name `library-system-container`, but a container with that name already exists. You have two options to resolve this issue:

### Option 1: Remove the Existing Container

If the previous container with the name `library-system-container` is not needed, you can stop and remove it.

### 1. Stop the container:

```
bash

docker stop library-system-container
```

### 2. Remove the container:

```
bash

docker rm library-system-container
```

After removing the old container, you can run the `docker run` command again to create and run the new container.

```
bash

docker run --name library-system-container -p 8080:8080 -e SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/librarydb -e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root -d zakeel4910/library-system:latest
```

## Option 2: Rename the New Container

If you want to keep the existing container and run a new one with a different name, you can simply change the container name in your `docker run` command. For example:

```
bash

docker run --name library-system-container-v2 -p 8080:8080 -e SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/librarydb -e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root -d zakeel4910/library-system:latest
```

This will create a new container with the name `library-system-container-v2`, and it will not conflict with the existing container.

Let me know if you need further assistance!



```
<container-id> zakeel4910/library-system:latest "java -jar /app.jar" <time\> Up <time\>
0.0.0.0:8080->8080/tcp library-system\container
```

Cant see this

It seems you're referring to the output of the ``docker ps`` command, which shows the details of running containers, including the ``library-system-container``. If you're not seeing the expected output, here's what you can check and do:

## 1. Verify if the Container is Running

Run the following command to list all the running containers:

```
bash

docker ps
```

This will show a list of active containers. You should see an entry similar to this:

```
perl

CONTAINER ID   IMAGE                                COMMAND                                  CREATED        STA
TUS           PORTS          NAMES                                     library-system-container
<container-id> zakee14910/library-system:latest "java -jar /app.jar"    <time>         Up
<time>        0.0.0.0:8080->8080/tcp
```

If you don't see the container listed, it could mean the container isn't running. You can check for all containers (including stopped ones) with:

```
bash

docker ps -a
```

## 2. Check the Logs of the Container

If the container is not showing up or if there is an issue, you can check the logs to see if there are any errors preventing it from starting:

```
bash

docker logs library-system-container
```

This will show the logs of the container. If your application is failing to start, the logs should give you more details about the error.

## 3. Check if the Port is Occupied



If the container is running but you're unable to access it via `http://localhost:8080`, it might be due to the port being already occupied by another process. You can check if port `8080` is in use by running:

```
bash

lsof -i :8080
```

If it shows any processes using port 8080, you may need to stop them. You can kill a process by using the `kill` command with the appropriate PID from the output:

```
bash

kill -9 <PID>
```

Then, try running the container again.

---

## 4. Re-run the Container

If the container is stopped or not listed, you can remove the old container and re-run the command:

### 1. Stop and remove the old container (if needed):

```
bash

docker stop library-system-container
docker rm library-system-container
```

### 2. Run the container again:

```
bash

docker run --name library-system-container -p 8080:8080 -e SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/librarydb -e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root -d zakeel4910/library-system:latest
```

---

## 5. Accessing the Application

Once the container is running, you should be able to access the application at:

```
arduino

http://localhost:8080
```

If you're still having issues, feel free to share the logs or more details so I can assist you further!

getting this error. my mysql container working perfectly in local but for docker image getting this error :

```

      _ _ _ _ _
     / _ _ _ _ \
    ( ( \ _ _ | ' _ | ' _ | ' _ V _ ' | \ \ \ \ \
     `W _ _ | | _ | | | | | | ( | | ) ) )
      ' | _ _ | . _ | | _ | | _ _ | \ _ _ | / / / /
     =====|_|=====|_|_/ _ / _ / _ /

```

:: Spring Boot :: (v3.4.1)

```

2024-12-30T08:09:41.214Z INFO 1 --- [library-system] [      main] c.e.l.LibrarySystemApplication
: Starting LibrarySystemApplication v0.0.1-SNAPSHOT using Java 17.0.13 with PID 1 (/app/app.jar
started by root in /app)
2024-12-30T08:09:41.216Z INFO 1 --- [library-system] [      main] c.e.l.LibrarySystemApplication
: No active profile set, falling back to 1 default profile: "default"
2024-12-30T08:09:41.617Z INFO 1 --- [library-system] [      main]
.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT
mode.
2024-12-30T08:09:41.648Z INFO 1 --- [library-system] [      main]
.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 25 ms. Found
1 JPA repository interface.
2024-12-30T08:09:41.907Z INFO 1 --- [library-system] [      main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-12-30T08:09:41.915Z INFO 1 --- [library-system] [      main]
o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-12-30T08:09:41.915Z INFO 1 --- [library-system] [      main]
o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.34]
2024-12-30T08:09:41.933Z INFO 1 --- [library-system] [      main] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring embedded WebApplicationContext
2024-12-30T08:09:41.934Z INFO 1 --- [library-system] [      main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in
690 ms
2024-12-30T08:09:42.024Z INFO 1 --- [library-system] [      main]
o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name:
default]
2024-12-30T08:09:42.063Z INFO 1 --- [library-system] [      main] org.hibernate.Version :
HHH000412: Hibernate ORM core version 6.6.4.Final
2024-12-30T08:09:42.081Z INFO 1 --- [library-system] [      main]
o.h.c.internal.RegionFactoryInitiator : HHH000026: Second-level cache disabled
2024-12-30T08:09:42.223Z INFO 1 --- [library-system] [      main]
o.s.o.j.p.SpringPersistenceUnitInfo : No LoadTimeWeaver setup: ignoring JPA class transformer
2024-12-30T08:09:42.246Z INFO 1 --- [library-system] [      main]
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2024-12-30T08:09:43.313Z WARN 1 --- [library-system] [      main]

```

```
o.h.engine.jdbc.spi.SqlExceptionHelper : SQL Error: 0, SQLState: 08S01
2024-12-30T08:09:43.314Z ERROR 1 --- [library-system] [      main]
o.h.engine.jdbc.spi.SqlExceptionHelper : Communications link failure
```

The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.

```
2024-12-30T08:09:43.316Z WARN 1 --- [library-system] [      main]
o.h.e.j.e.i.JdbcEnvironmentInitiator   : HHH000342: Could not obtain connection to query metadata
```

```
org.hibernate.exception.JDBCConnectionException: unable to obtain isolated JDBC connection
[Communications link failure
```

The error message you're encountering, `Communications link failure`, suggests that the Spring Boot application in your Docker container is unable to connect to the MySQL database, even though the MySQL container is working perfectly on your local machine. This is a common issue when trying to connect a Docker container (your Spring Boot application) to a MySQL service running locally (on the host machine).

Here's a detailed step-by-step guide on how to resolve this issue and establish the connection correctly:

## Steps to Resolve the Issue:

### 1. Check Docker Container Networking

By default, Docker containers cannot access services running on the host machine (i.e., your local MySQL) using `localhost`. This happens because `localhost` inside a Docker container refers to the container itself, not the host machine.

To connect the Spring Boot application container to a MySQL instance running on your host machine, you need to use the special DNS name `host.docker.internal` which resolves to your host machine's IP address.

### 2. Update Database Connection URL

You need to change the `SPRING_DATASOURCE_URL` in your Docker container's environment variables to use `host.docker.internal` instead of `localhost`.

In your `docker run` command, update the `SPRING_DATASOURCE_URL` environment variable like this:

```
bash
```

```
docker run --name library-system-container \
-p 8080:8080 \
-e SPRING_DATASOURCE_URL=jdbc:mysql://host.docker.internal:3306/librarydb \
-e SPRING_DATASOURCE_USERNAME=root \
-e SPRING_DATASOURCE_PASSWORD=root \
-d zakeel4910/library-system:latest
```

Here, `host.docker.internal` will point to your host machine's IP address, allowing your Docker container to access MySQL running on your local machine.

### 3. Verify MySQL is Accessible from the Host

Ensure your MySQL instance on your host is accessible and not blocked by firewall rules. You can do this by testing the connection with the following command (or using a MySQL client):

```
bash

mysql -h 127.0.0.1 -P 3306 -u root -p
```

Make sure you can connect to the `librarydb` database successfully with the correct credentials.

### 4. Retry Running the Container

After updating the database connection URL, try running the Spring Boot container again with the updated connection details.

## Alternative Solution (Docker Network)

If you want to avoid using `host.docker.internal` and prefer a Docker network for communication between containers, follow these steps:

#### 1. Create a Docker Network:

First, create a custom Docker network.

```
bash

docker network create library-network
```

#### 2. Run the MySQL Container in the Network:

Run the MySQL container and attach it to the network.

```
bash

docker run --name mysql-container --network library-network -e MYSQL_ROOT_PASSWORD=root -d mysql:8
```

#### 3. Run the Spring Boot Application Container in the Same Network:

Then, run the Spring Boot container in the same network and use the container name (`mysql-container`) to connect to MySQL.

bash

```
docker run --name library-system-container \
  --network library-network \
  -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:mysql://mysql-container:3306/librarydb \
  -e SPRING_DATASOURCE_USERNAME=root \
  -e SPRING_DATASOURCE_PASSWORD=root \
  -d zakee14910/library-system:latest
```

In this setup, the Spring Boot application can access MySQL by connecting to the container name ``mysql-container`` instead of ``localhost``.

## Summary

- **Option 1:** Use ``host.docker.internal`` to connect from Docker to your local MySQL instance.
- **Option 2:** Set up a custom Docker network to allow the Spring Boot container and MySQL container to communicate using container names.

Try updating your ``docker run`` command with ``host.docker.internal``, and this should resolve the ``Communications link failure`` issue you're seeing.

Let me know if this works or if you need further assistance!

I need to rout my front end file to the as a initail page in the docker. What should I do ? also need to push that to docker hub

To route your front-end file as the initial page served by the Spring Boot application and deploy everything in Docker, follow these steps:

## 1. Embed the Front-End Files in the Backend Application

The simplest approach is to serve your front-end files (HTML, CSS, JS) from the Spring Boot application itself.

### 1. Add Front-End Files to Spring Boot:

- Place your front-end files in the ``src/main/resources/static`` directory.
- For example:

css

```
src/main/resources/static/index.html
src/main/resources/static/css/
src/main/resources/static/js/
```

- Spring Boot will automatically serve the ``index.html`` file when you access the root URL (``http://localhost:8080``).

2. **Test Locally:** Run the Spring Boot application locally using:

```
bash

./mvnw spring-boot:run
```

Verify that accessing `http://localhost:8080` shows your front-end.

---

## 2. Update Dockerfile to Include the Front-End

Modify your `Dockerfile` to ensure the front-end files are bundled in the application JAR.

Here's an updated `Dockerfile`:

```
dockerfile

# Use an ARM64-compatible OpenJDK base image
FROM eclipse-temurin:17-jdk-alpine

# Set the working directory
WORKDIR /app

# Copy the built JAR file to the container
COPY target/library-system-0.0.1-SNAPSHOT.jar app.jar

# Expose the port Spring Boot runs on
EXPOSE 8080

# Run the application
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Ensure your front-end files are part of the compiled JAR by building the project:

```
bash

./mvnw clean package
```

---

## 3. Build and Push Docker Image

1. **Build the Docker Image:**

```
bash

docker buildx build --platform linux/amd64,linux/arm64 -t zakee14910/library-system:latest .
```

2. **Run Locally for Testing:** Run the Docker container to verify the front-end is served properly:

```
bash
```

```
docker run --name library-system-container -p 8080:8080 -d zakee14910/library-system:latest
```

Access `http://localhost:8080` and ensure the front-end is displayed.

### 3. Push to Docker Hub:

```
bash
```

```
docker push zakee14910/library-system:latest
```

---

## 4. Optional: Serve Front-End and Back-End Separately

If you want the front-end and back-end to be independent (e.g., front-end served by Nginx or Apache), you need:

### 1. Front-End Docker Image:

- Create a separate Dockerfile for the front-end.

```
dockerfile
```

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
EXPOSE 80
```

- Build and push the front-end Docker image:

```
bash
```

```
docker build -t zakee14910/library-frontend:latest .
docker push zakee14910/library-frontend:latest
```

### 2. Run Both Containers:

- Run the front-end:

```
bash
```

```
docker run --name library-frontend-container -p 80:80 -d zakee14910/library-frontend:latest
```

- Run the back-end:

```
bash
```

```
docker run --name library-backend-container -p 8080:8080 -d zakee14910/library-system:latest
```

The front-end can now make API calls to the back-end via `http://localhost:8080`.

## 5. Verify Deployment

- If everything works as expected, push both images to Docker Hub.
- Run them on a production server or cloud environment as needed.

Let me know if you need further clarification!

I don't know anything about Jenkins. But as per request I need to put this CI/CD using Jenkins. Give me step by step guidance from Jenkins installation.

To implement a CI/CD pipeline for your Spring Boot and Dockerized project using Jenkins, here's a step-by-step guide starting from Jenkins installation:

## 1. Install Jenkins

### On macOS

1. Install Jenkins via Homebrew:

```
bash
```

```
brew install jenkins-lts
```

2. Start Jenkins:

```
bash
```

```
brew services start jenkins-lts
```

3. Access Jenkins: Open your browser and go to `http://localhost:8080`.

4. Unlock Jenkins:

- Navigate to the terminal and locate the password:



```
bash
```

```
cat /Users/yourusername/.jenkins/secrets/initialAdminPassword
```

- Enter this password on the Jenkins setup page.
5. Install Suggested Plugins:
- Select "Install Suggested Plugins" when prompted.
6. Create an Admin User:
- Set up your Jenkins admin username and password.

## 2. Configure Jenkins

### Install Required Plugins

1. Go to `Manage Jenkins` → `Manage Plugins`.
2. Search for and install the following plugins:
  - **Docker Pipeline:** To handle Docker-related operations.
  - **Pipeline:** For pipeline scripting.
  - **Git:** To pull your code from GitHub.
  - **Maven Integration:** For Maven builds.

## 3. Prepare Your Project Repository

1. Ensure your project repository has the following:
  - `Dockerfile` for Docker builds.
  - `Jenkinsfile` to define the pipeline.
2. Create a `Jenkinsfile` in the root of your project:

```
groovy

pipeline {
    agent any

    environment {
        DOCKER_IMAGE = "zakeel4910/library-system"
        DOCKER_TAG = "latest"
    }

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/your-username/your-repo.git'
            }
        }

        stage('Build') {
            steps {
                sh './mvnw clean package'
            }
        }
    }
}
```

```

stage('Docker Build & Push') {
    steps {
        script {
            sh "docker build -t ${DOCKER_IMAGE}:${DOCKER_TAG} ."
            sh "docker login -u your-dockerhub-username -p your-dockerhub-passw
ord"
            sh "docker push ${DOCKER_IMAGE}:${DOCKER_TAG}"
        }
    }
}

stage('Deploy') {
    steps {
        sh "docker run --name library-system-container -p 8080:8080 -d ${DOCKER
_IMAGE}:${DOCKER_TAG}"
    }
}
}

```

Replace placeholders like `your-username`, `your-repo`, and `your-dockerhub-username/password` with your actual details.

## 4. Create a Jenkins Job

1. In Jenkins, click on `New Item`.
2. Enter a name for your job and select `Pipeline`.
3. Configure the pipeline:
  - Under `Pipeline` → `Definition`, select `Pipeline script from SCM`.
  - Set `SCM` to `Git` and provide the repository URL.
  - Specify the branch and `Jenkinsfile` path (default is the root of the repo).

## 5. Test the Pipeline

1. Click `Build Now` to run the pipeline.
2. Monitor the build logs to ensure the stages (`Checkout`, `Build`, `Docker Build & Push`, `Deploy`) complete successfully.

## 6. Access Your Application

- Open a browser and navigate to `http://localhost:8080` to confirm the application is running.

### Additional Notes

- **Jenkins Agent Configuration:**
  - If running Jenkins on a remote machine, ensure Docker is installed and accessible.
  - Add the Jenkins user to the `docker` group to avoid permission issues:

```
bash
```

```
sudo usermod -aG docker jenkins
```

- **Environment Variables:**

- Use Jenkins credentials for secure Docker Hub login. Store Docker Hub credentials in Jenkins (``Manage Jenkins`` → ``Credentials``).

- **Improvements:**

- Add error handling in the pipeline.
- Integrate unit tests in the ``Build`` stage.

Let me know if you need help with any of these steps!



Access Jenkins: Open your browser and go to ``http://localhost:8080``.

How to change the running port number of jenkins

To change the running port number of Jenkins, follow these steps:

## Step 1: Locate the Jenkins Configuration File

The Jenkins configuration file is typically located at:

- **macOS (installed via Homebrew):** ``/usr/local/opt/jenkins-lts/homebrew.mxcl.jenkins-lts.plist``
- **Linux:** ``/etc/default/jenkins``
- **Windows:** Configured in the ``jenkins.xml`` file in the Jenkins installation directory.

For macOS (installed via Homebrew), you may need to manually configure the plist.

## Step 2: Edit the Jenkins Configuration

### Linux and Other Unix-like Systems

1. Open the Jenkins configuration file:

```
bash
```

```
sudo nano /etc/default/jenkins
```

2. Look for the line:

```
bash

HTTP_PORT=8080
```

3. Change `8080` to your desired port (e.g., `9090`):

```
bash

HTTP_PORT=9090
```

4. Save the file and exit.

## macOS (Homebrew Installation)

1. Open the Homebrew service plist file:

```
bash

sudo nano /usr/local/opt/jenkins-lts/homebrew.mxcl.jenkins-lts.plist
```

2. Search for the line containing `--httpPort=8080` (it might be wrapped inside a `<string>` tag if XML).

3. Change `8080` to your desired port number.

4. Save and exit.

## Windows

1. Open `jenkins.xml` in the Jenkins installation directory.

2. Look for:

```
xml

--httpPort=8080
```

3. Change `8080` to your desired port.

4. Save the file.

## Step 3: Restart Jenkins

After updating the configuration file, restart Jenkins to apply the changes:

## Linux and macOS

```
bash
```

```
sudo service jenkins restart
```

Or for macOS Homebrew:

```
bash
```

```
brew services restart jenkins-lts
```

## Windows

- Stop the Jenkins service from the Services Manager (``services.msc``).
- Restart the service.

## Step 4: Verify the New Port

- Open a browser and navigate to:

```
vbnet
```

```
http://localhost:<new-port-number>
```

For example, if you set the port to ``9090``, visit:

```
arduino
```

```
http://localhost:9090
```

Let me know if you face any issues!



Create a ``Jenkinsfile`` in the root of your project:

this is my main project folder in git hub. Do i need to change anything ?

<https://github.com/zakee14910/GSCOMP126NCC/tree/main/Backend/library-system>

To integrate Jenkins for CI/CD using the repository you linked, you need to add a ``Jenkinsfile`` to the root of your repository. This file defines the pipeline steps Jenkins will execute. Here's what you need to do:

## Step 1: Create a Jenkinsfile

1. Navigate to the ``Backend/library-system`` directory of your project repository.
2. Create a new file named ``Jenkinsfile``.

## Step 2: Add the Jenkinsfile Content

Here's a sample ``Jenkinsfile`` tailored for your Spring Boot project with Docker integration:

```
groovy

pipeline {
    agent any

    environment {
        DOCKER_HUB_CREDENTIALS = credentials('docker-hub-credentials') // Set this in Jenkins
    }
    s credentials
        DOCKER_IMAGE = 'zakeel4910/library-system'
        DOCKER_TAG = 'latest'
    }

    stages {
        stage('Checkout Code') {
            steps {
                // Clone the code from GitHub
                checkout scm
            }
        }

        stage('Build Project') {
            steps {
                script {
                    // Clean and build the project
                    sh './mvnw clean package -DskipTests'
                }
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    // Build the Docker image
                    sh 'docker build -t $DOCKER_IMAGE:$DOCKER_TAG .'
                }
            }
        }

        stage('Push Docker Image') {
            steps {
                script {
                    // Log in to Docker Hub
                    sh "echo $DOCKER_HUB_CREDENTIALS_PSW | docker login -u $DOCKER_HUB_CREDENTIALS_USR --password-stdin"

                    // Push the Docker image to Docker Hub
                    sh "docker push $DOCKER_IMAGE:$DOCKER_TAG"
                }
            }
        }

        stage('Deploy Application') {
            steps {
                script {
                    // Stop any running container with the same name
                    sh 'docker rm -f library-system-container || true'

                    // Run the new container
                }
            }
        }
    }
}
```

```

        sh 'docker run --name library-system-container -d -p 8080:8080 -e SPRING
_DATASOURCE_URL=jdbc:mysql://host.docker.internal:3306/librarydb -e SPRING_DATASOURCE_USERNA
ME=root -e SPRING_DATASOURCE_PASSWORD=root $DOCKER_IMAGE:$DOCKER_TAG'
    }
}
}
post {
    always {
        // Clean up workspace
        cleanWs()
    }
}
}

```

### Step 3: Set Up Docker Hub Credentials in Jenkins

1. Go to Jenkins Dashboard.
2. Navigate to **Manage Jenkins > Manage Credentials**.
3. Add credentials:
  - **Kind:** Username with password.
  - **ID:** ``docker-hub-credentials`` (or match the ID in the Jenkinsfile).
  - **Username:** Your Docker Hub username.
  - **Password:** Your Docker Hub password.

### Step 4: Configure Jenkins Pipeline

1. Create a new pipeline job in Jenkins.
2. Link your GitHub repository in the job configuration:
  - **Pipeline script from SCM > Git.**
  - Add your repository URL and branch.

### Step 5: Trigger the Pipeline

1. Run the pipeline from Jenkins.
2. Monitor the logs to ensure each stage (build, test, Docker image creation, and deployment) completes successfully.

### Step 6: Test the Deployed Application

- Access the application at ``http://localhost:8080``.
- If necessary, update the ``SPRING_DATASOURCE_URL``, username, or password for MySQL connectivity in the ``Jenkinsfile``.

Let me know if you need further assistance!

Tomorrow I don't have access to you. I need to setup this project in a new windows PC and need to to some code modification. Give me the necessary code for Git hub and docker and so on. I can only bring those things.

To set up your project on a new Windows PC, here's what you need to bring and do:

## 1. Bring Necessary Files

- **Complete Project Code:** Clone the repository or download the files from [your GitHub repo](#).
- **Dockerfile:** Ensure this file is in the root directory of your project.
- **Jenkinsfile:** If using Jenkins, ensure this is also added to the repository.
- **MySQL Backup (Optional):** If you need a specific database state, export your MySQL database as a `.sql` file.`

## 2. Install Required Software on the New PC

- **Java JDK:** Install Java 17.
- **Maven:** Download and install Apache Maven.
- **Git:** Install Git for version control.
- **Docker Desktop:** Install Docker and ensure it is running.
- **MySQL:** Install MySQL if you are not using a Dockerized MySQL setup.
- **IDE:** Install an IDE like IntelliJ IDEA, VS Code, or NetBeans.

## 3. Setting Up GitHub

1. Clone your GitHub repository:

```
bash
```

```
git clone https://github.com/zakee14910/GSCOMP126NCC.git
```

2. Navigate to the project directory:

```
bash
```

```
cd GSCOMP126NCC/Backend/library-system
```

## 4. Build and Run the Project

1. Build the project with Maven:



```
bash
```

```
./mvnw clean package
```

2. Run the project locally:

```
bash
```

```
./mvnw spring-boot:run
```

## 5. Run MySQL (Local or Dockerized)

### Option 1: Local MySQL

- Create the database manually using:

```
sql
```

```
CREATE DATABASE librarydb;
```

- Update `application.properties` if needed.

### Option 2: MySQL in Docker

Use this command to start MySQL in Docker:

```
bash
```

```
docker run --name mysql-container -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=librarydb -p 3306:3306 -d mysql:8.0
```

## 6. Run Docker for Backend

### Dockerfile

Ensure the following `Dockerfile` is in your project directory:

```
dockerfile
```

```
# Use an ARM64-compatible OpenJDK image for M1 or x86 for Windows
FROM eclipse-temurin:17-jdk-alpine
```

```
# Set the working directory
WORKDIR /app
```

```
# Copy the JAR file into the container
COPY target/library-system-0.0.1-SNAPSHOT.jar app.jar

# Expose the application port
EXPOSE 8080

# Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

## Build Docker Image

### 1. Build the Docker image:

```
bash

docker build -t zakeel4910/library-system:latest .
```

### 2. Run the backend container:

```
bash

docker run --name library-system-container -p 8080:8080 -e SPRING_DATASOURCE_URL=jdbc:mysql://host.docker.internal:3306/librarydb -e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root -d zakeel4910/library-system:latest
```

---

## 7. Push Changes to GitHub

1. Make code modifications.
2. Add and commit changes:

```
bash

git add .
git commit -m "Updated code with necessary changes"
```

### 3. Push to GitHub:

```
bash

git push origin main
```

---

## 8. Necessary Docker Commands

- Stop a container:

```
bash
```

```
docker stop <container-name>
```

- Remove a container:

```
bash
```

```
docker rm <container-name>
```

- Remove an image:

```
bash
```

```
docker rmi <image-name>
```

---

## 9. Bring Database Backup Script

If you need the existing database, export it on the current system using:

```
bash
```

```
mysqldump -u root -p librarydb > librarydb_backup.sql
```

Import it into the new system:

```
bash
```

```
mysql -u root -p librarydb < librarydb_backup.sql
```

---

## 10. Additional Notes

- Ensure ``docker-compose.yml`` is present if using Docker Compose.
- Verify Docker Daemon is running before executing Docker commands.
- Bring your ``.env`` file if using environment variables.

Let me know if you need any of the scripts or files!