**OVERVIEW:**

This documentation describes about an Employee Management System Application. The Employee Management System is a Spring Boot-based application designed to manage employee records efficiently. It provides **RESTful APIs** and **Web interfaces** for CRUD operations, ensuring data integrity through robust validation. The project is structured with clear separation of concerns, using DTOs, controllers, and services to handle data flow and business logic.

The following focus on how to set-up, configure and run the same.

Moreover, the application utilizes the following key components:

1. JDK 17
2. Maven
3. Spring boot 3.3.2
4. MySQL
5. Git and Github
6. Thymeleaf template engine (for the Front-end files)
7. HTML
8. Bootstrap-CSS

The complete code for the application is stored in the git hub repository: <https://github.com/Ashm99/employee-management-system-project>

**PREREQUISITES:**

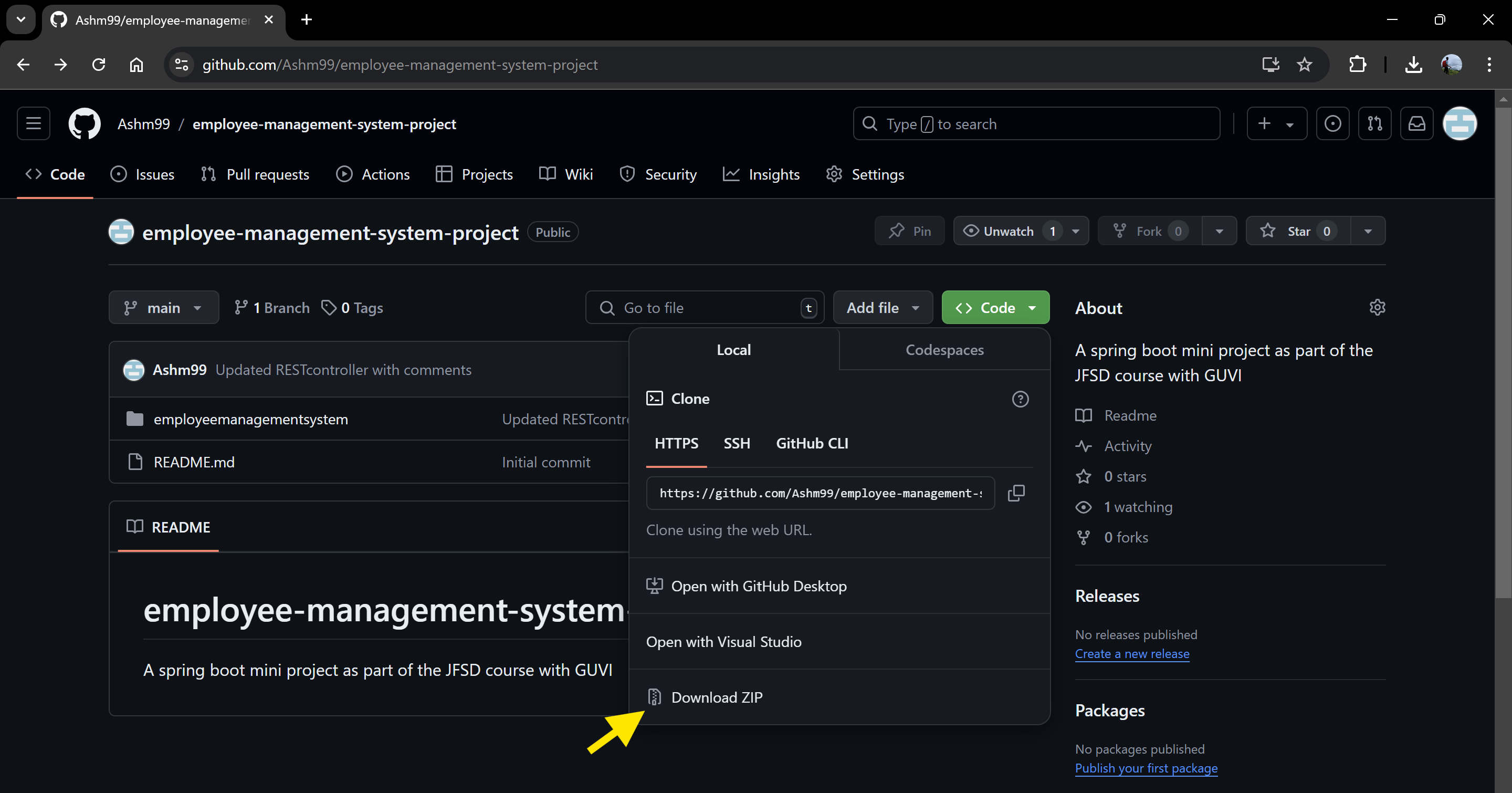
1. An IDE (like, IntelliJ)
2. JDK 17
3. Maven (Build tool)
4. MySQL along with an IDE(like MySQL Workbench)
5. A browser (like Chrome)

Now, let us proceed to the set-up and configuration of the project.

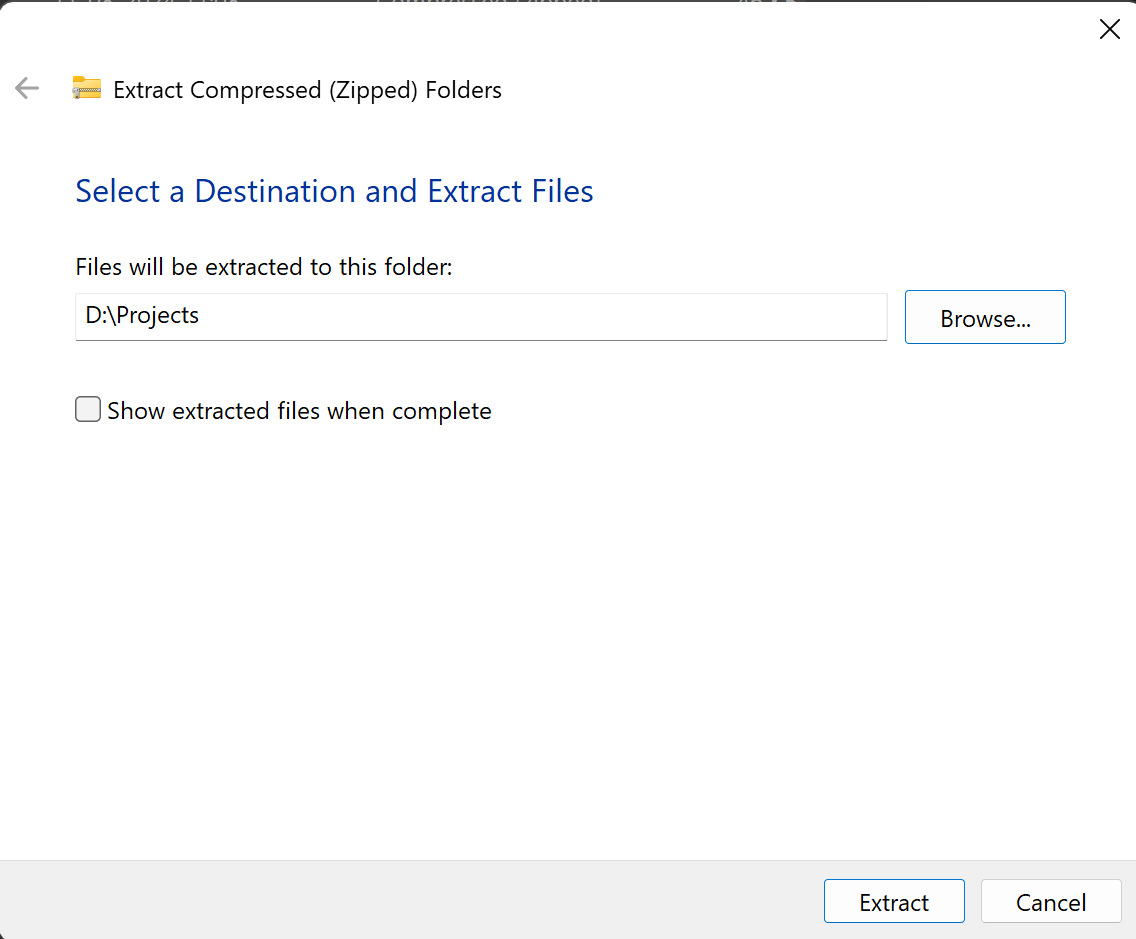
**Getting the code into your local system:**

1. Get the code from the github repository <https://github.com/Ashm99/employee-management-system-project> to your system.

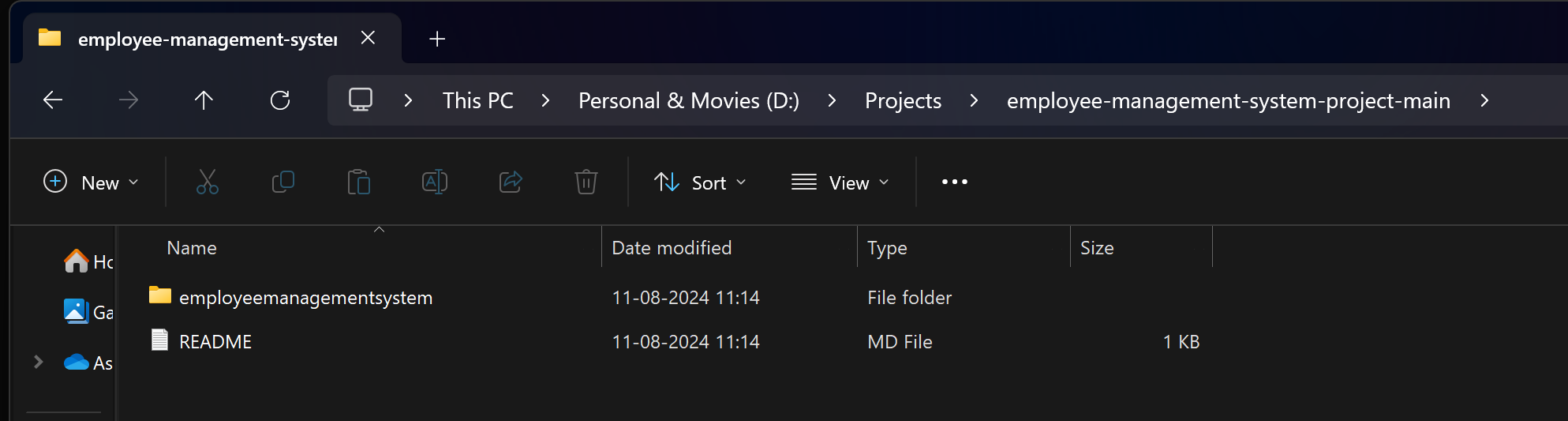
One of the methods you can do is download the file as a zip folder, shown below:



1. Then extract the downloaded zip folder into a folder as per your interest.



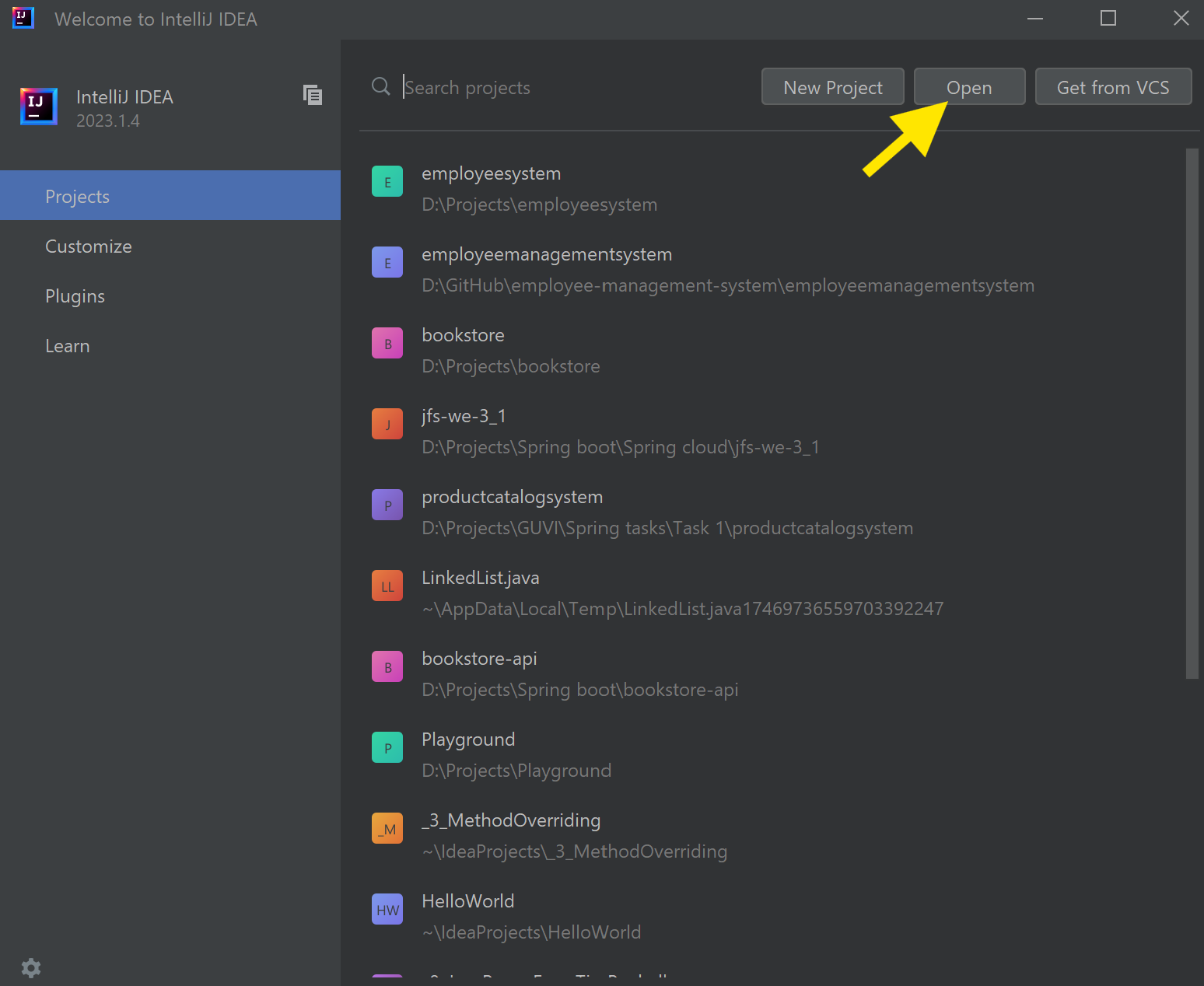
After extracting, ensure that the extracted path is having the package like this:



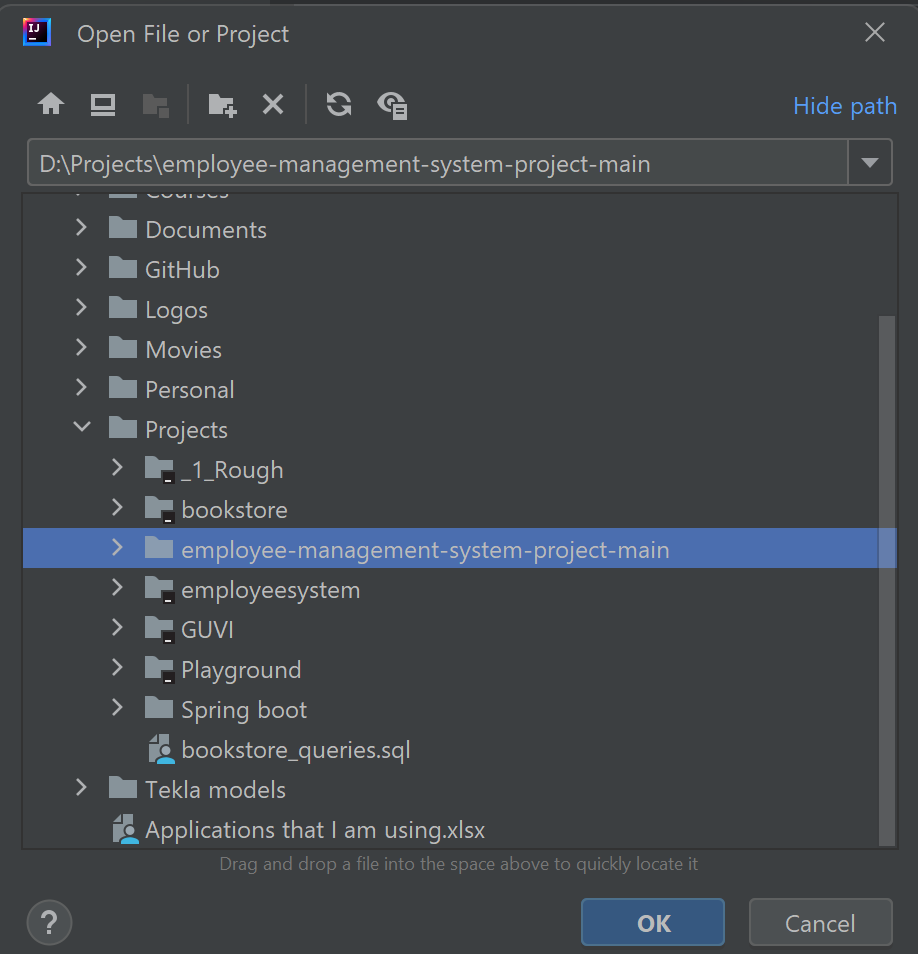
**Opening the project in an IDE:**

In this manual, I will be using IntelliJ as my preferred IDE. Whereas using Eclipse or any other IDE shouldn’t be a problem as the procedure is almost the same in any IDE.

1. Open IntelliJ. Select *Open*.

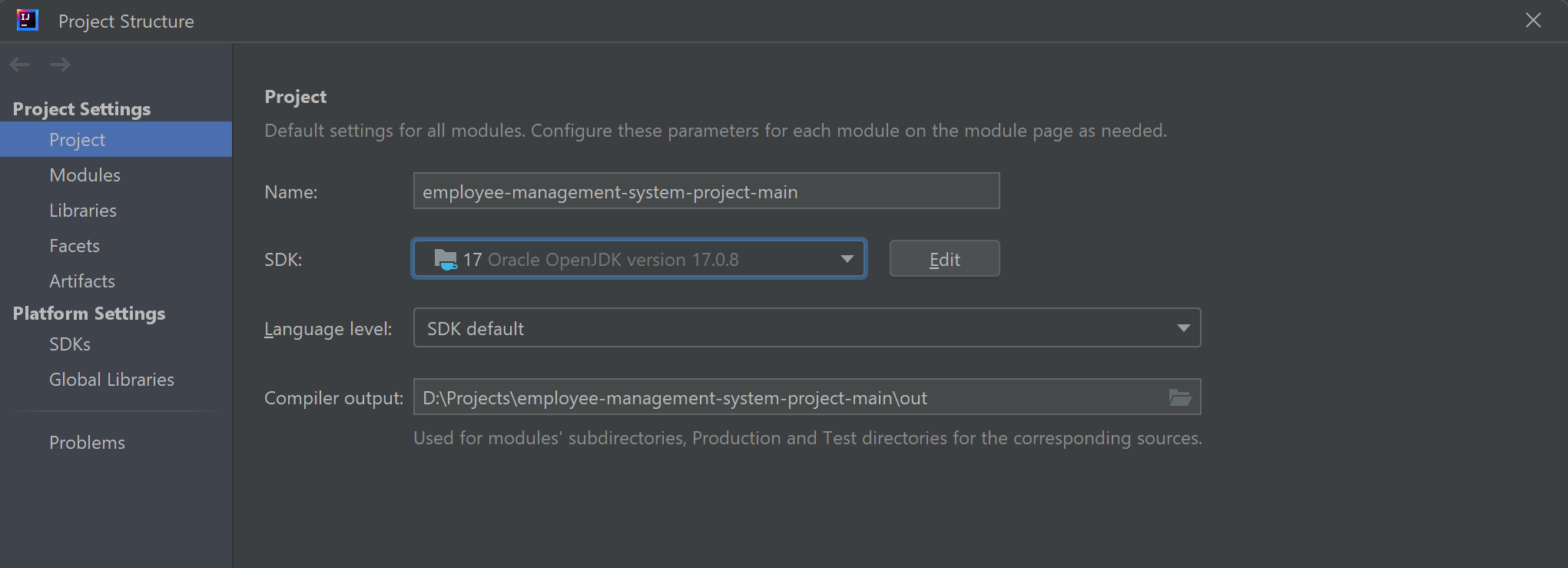


1. Browse to the path where we extracted our project and select *OK*.



Wait for a few seconds as the project will automatically load. During this time, the maven dependencies must also load on its own. You can also manually do this by clicking

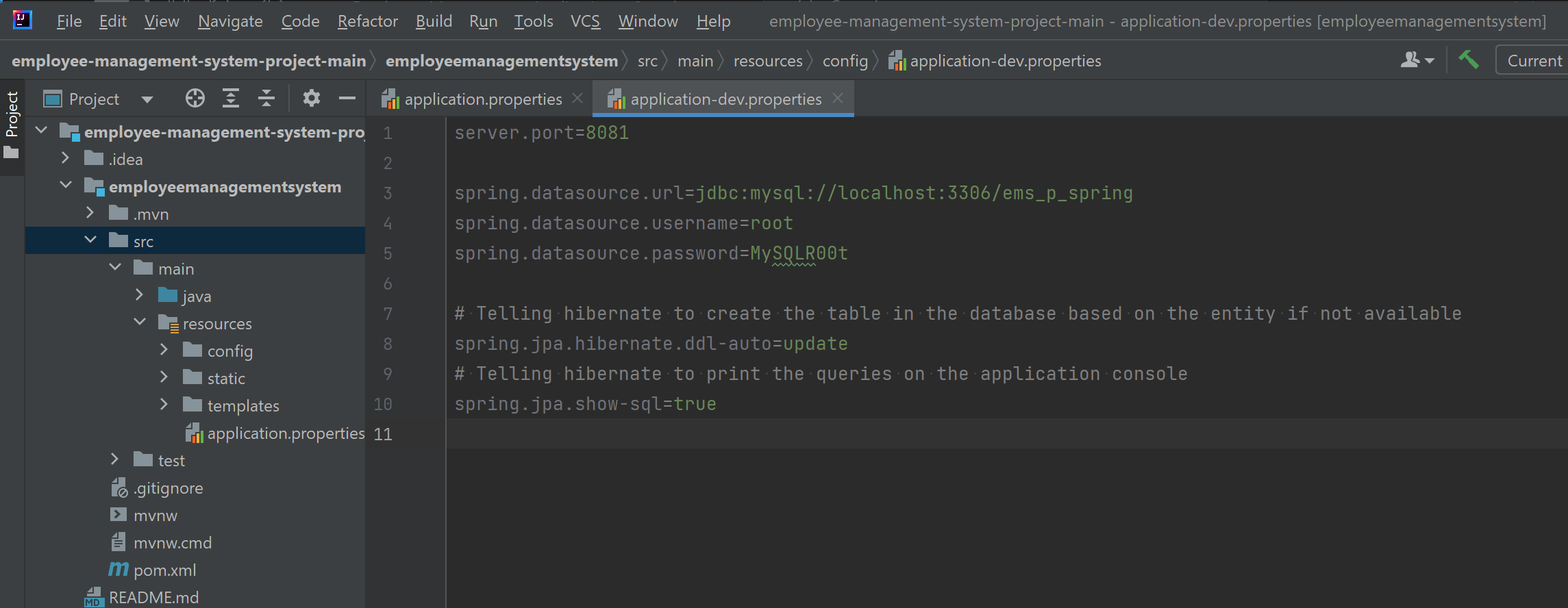
*maven -> Reload all maven projects.*

1. Go to *File -> Project Structure -> Project* and ensure that *JDK 17* is selected as the SDK.

**Setting the database:**

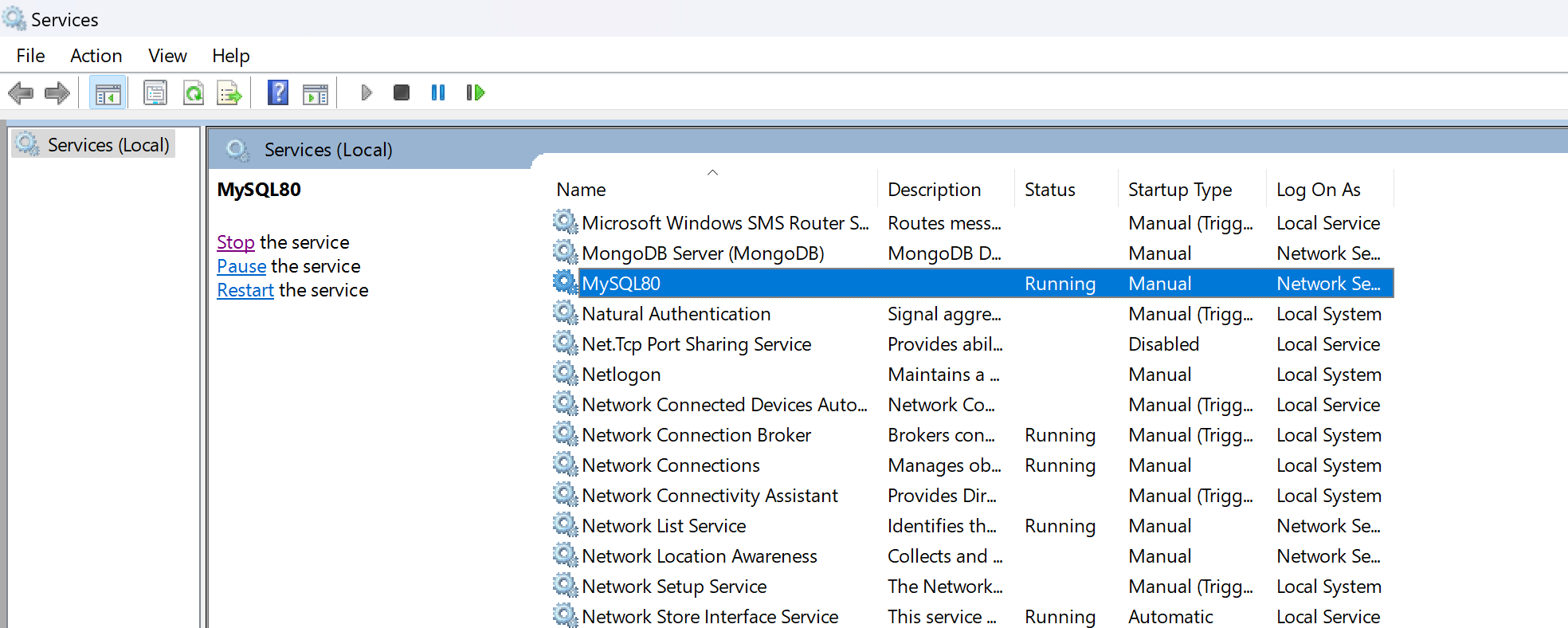
The application is created having MySQL as its database for storing its data. The main configurations of the application are written inside the application-dev.properties file inside

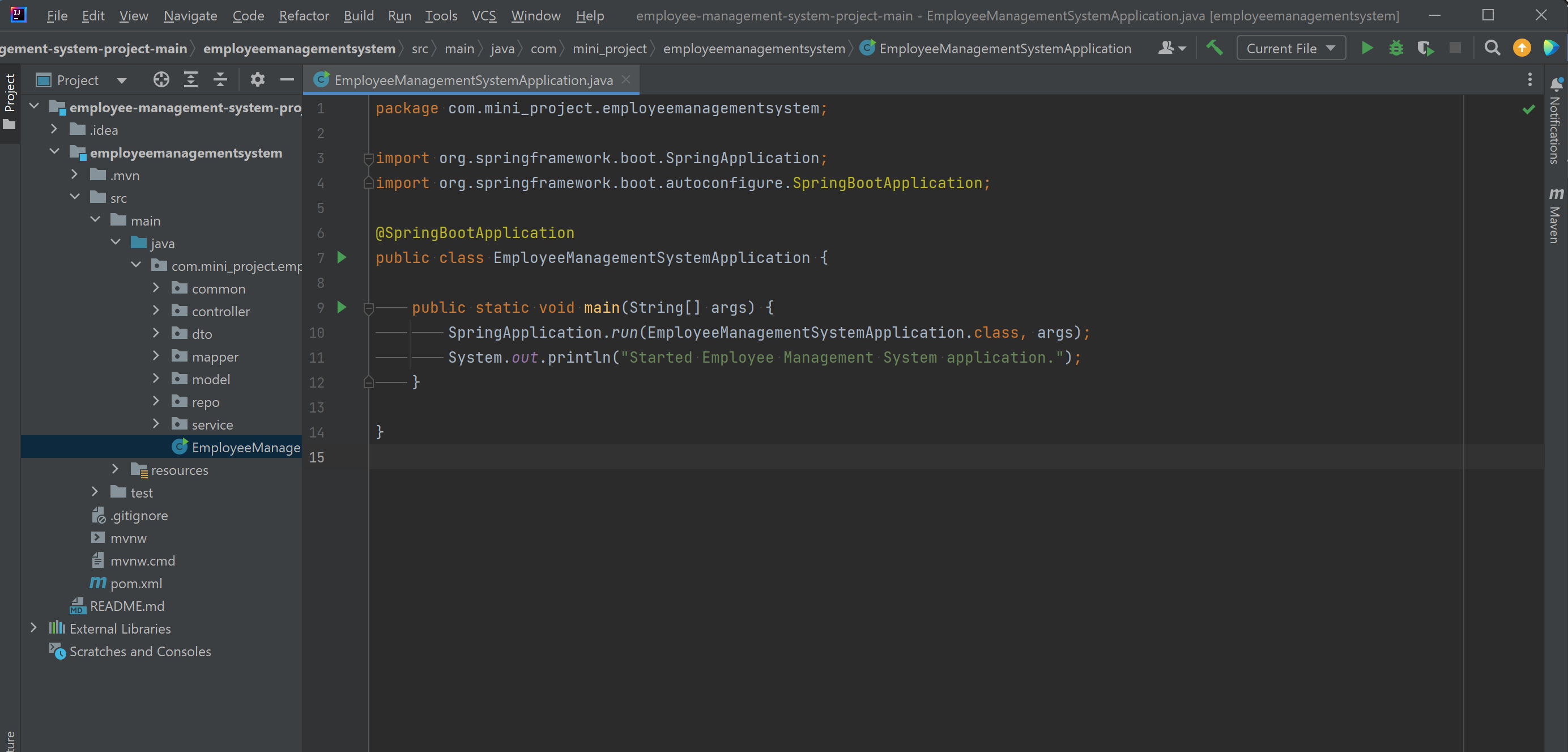
*src->main->resources->config->application-dev.properties*

**

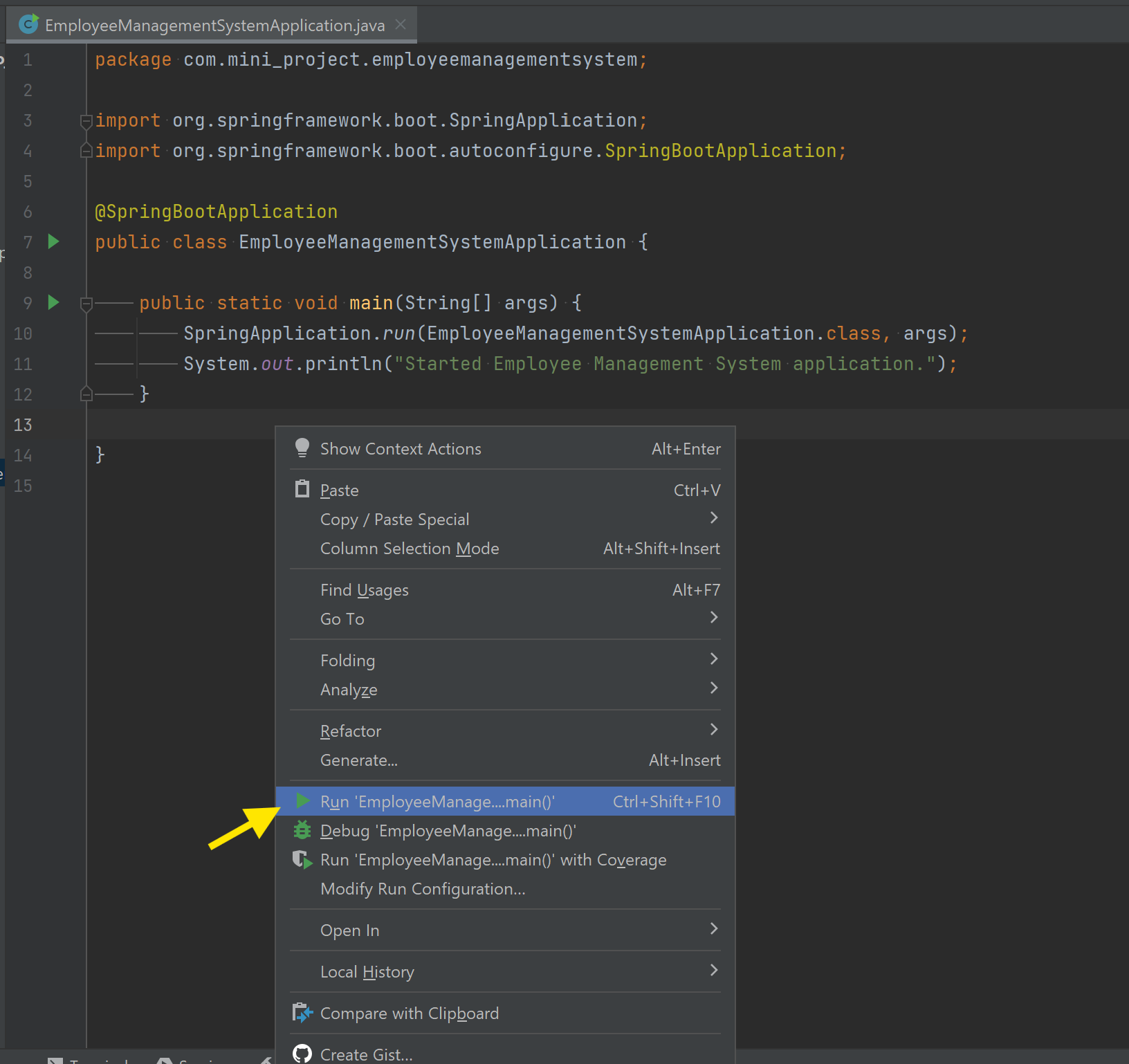
* As shown, change the *url, username* and *password* properties as per your system configurations.
* Create a database in MySQL for this application.
* **Do ensure that the database name and the name present in the url (in this case: ems\_p\_spring) are the exact same.** Any difference in the name will cause error while running the application.

**Starting the application:**

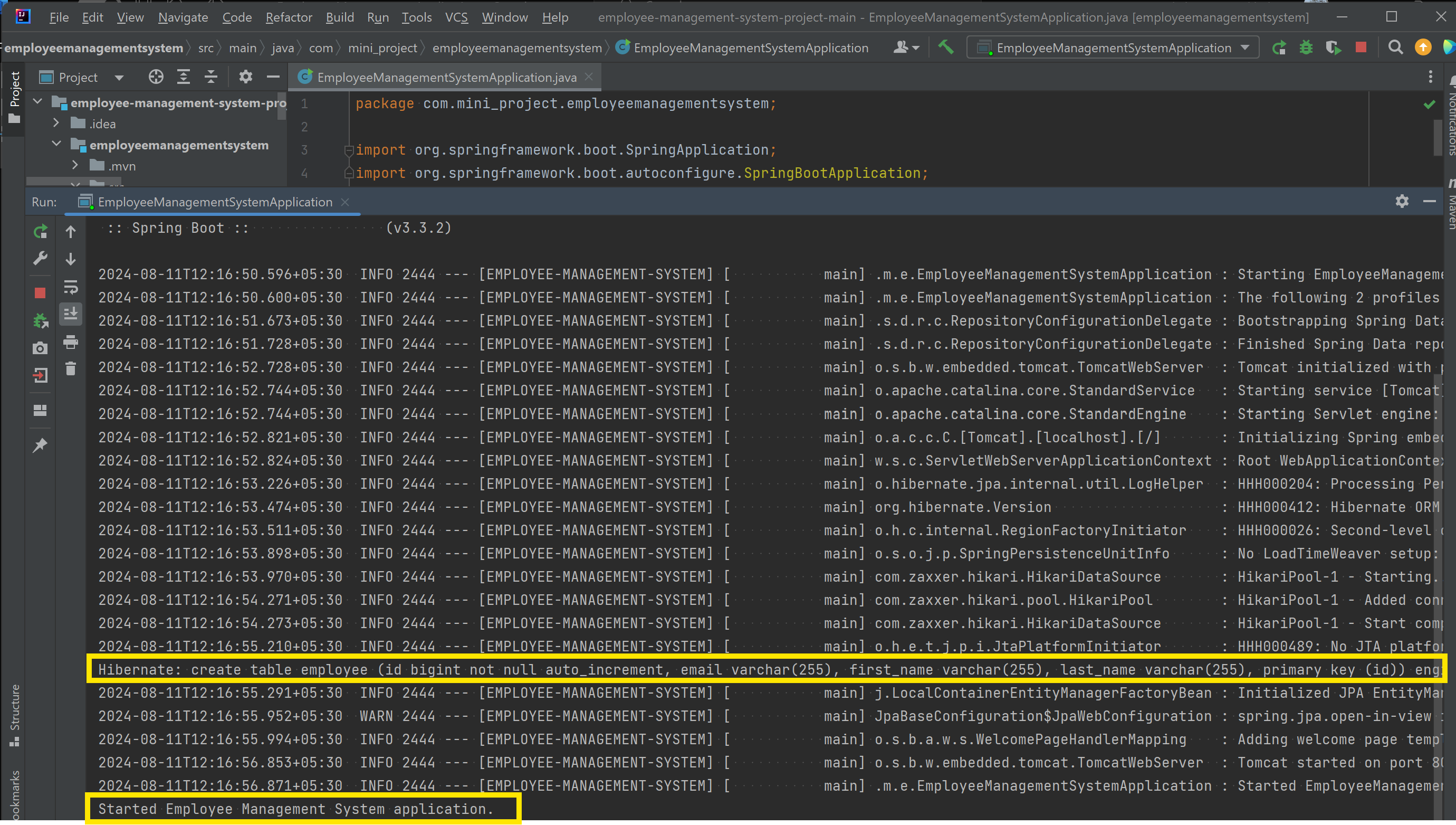
1. Before starting the application, ensure MySQL service is up and running.
2. Go to *employeemanagementsystem -> src -> main -> java -> com.mini\_project.employeemanagementsystem -> EmployeeManagementSystemApplication*



1. Right-clicking inside the *EmployeeManagementSystemApplication.java* file and click Run.

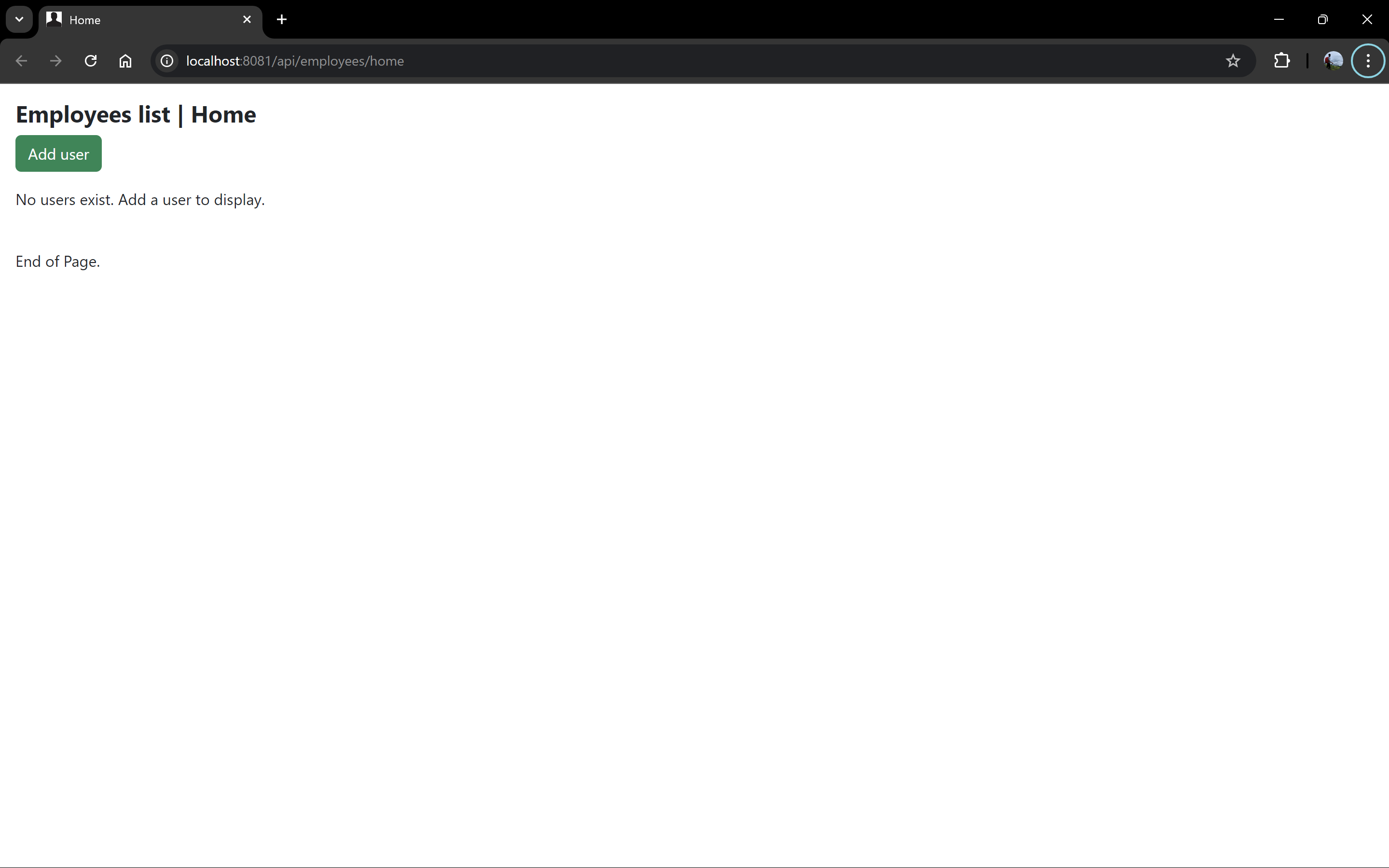


1. On clicking Run, the application will start and you will see the below screen:



If the database name in the *application-dev.properties* and in the MySQL database match, Hibernate will automatically create a table as required and the system will get started.

1. In *application-dev.properties*, *8081* is the port number that is configured for this application. But ensure that the no other service is using that port and it is correctly mentioned in *server.port* in *application-dev.properties*.
2. Now, go to your browser and hit [http://localhost:<port-number>/api/employees/home](http://localhost:%3cport-number%3e/api/employees/home). For port 8081: <http://localhost:8081/api/employees/home>. This should render the homepage of the application.

Once this is done, you will be able to see the home page of the application. 

From here, you can communicate with the application through the browser for adding, viewing, updating and deleting an employee from the database. (Do checkout the video present in the repository for a project walk-through)

**API DOCUMENTATION:**

**ENDPOINTS:**

1. **API DOCUMENTATION FOR EMPLOYEE-RESTCONTROLLER:**

**Overview:**

The EmployeeRestController provides a set of RESTful endpoints for managing employees within the system. This includes operations to add, retrieve, update, and delete employee records.

1. **Get All Employees**

* **Endpoint:** /getAll
* **Method:** GET
* **Description:** A RESTful method to get all the employees from the database
* **Request Parameters:** None
* **Response:**
  + **200 OK:** Returns a list of EmployeeDTO objects.
  + **204 No Content:** Returns an empty list if no employees are found.
* **Sample Response(Json):**

[

{

"id": 1,

"firstName": "John",

"lastName": "Doe",

"email": "john.doe@example.com"

},

{

"id": 2,

"firstName": "Johanna",

"lastName": "smith",

"email": "jsmith@example.com"

},

]

1. **Add a New Employee**

* **Endpoint:** /addEmployee
* **Method:** POST
* **Description:** A RESTful method to add a new employee into the database
* **Request Body:**
  + **Content-Type:** application/json
  + **Sample Body(Json):**

{

"firstName": "John",

"lastName": "Doe",

"email": "john.doe@example.com"

}

* **Response:**
  + **201 Created:** Returns the newly created EmployeeDTO object.
* **Sample Response(Json):**

{

"id": 1,

"firstName": "John",

"lastName": "Doe",

"email": "john.doe@example.com"

}

1. **Get Employee by ID**

* **Endpoint:** /getEmployee/{id}
* **Method:** GET
* **Description:** A RESTful method to get an employee from the database through its id
* **Path Parameters:**
  + id (Long): The ID of the employee to retrieve.
* **Response:**
  + **200 OK:** Returns the EmployeeDTO object of the requested employee.
  + **404 Not Found:** If the employee with the specified ID does not exist.
* **Sample Response(json):**

{

"id": 1,

"firstName": "John",

"lastName": "Doe",

"email": "john.doe@example.com"

}

1. **Update an Employee**

* **Endpoint:** /updateEmployee
* **Method:** PUT
* **Description:** A RESTful method to update an employee in the database
* **Request Body:**
  + **Content-Type:** application/json
  + **Sample Body(json):**

{

"id": 1,

"firstName": "John",

"lastName": "Doe",

"email": "john.doe@example.com"

}

* **Response:**
  + **200 OK:** Returns the updated EmployeeDTO object.
* **Sample Response(json):**

{

"id": 1,

"firstName": "John",

"lastName": "Doe",

"email": "john.doe@example.com"

}

1. **Delete an Employee**

* **Endpoint:** /delete/{id}
* **Method:** DELETE
* **Description:** A RESTful method to delete an employee from the database
* **Path Parameters:**
  + id (Long): The ID of the employee to delete.
* **Response:**
  + **200 OK:** Returns a message indicating whether the deletion was successful.
* **Sample Response(json):**
  + "Successfully deleted employee with ID: 1"

1. **API DOCUMENTATION FOR WEBCONTROLLER**

**Overview:**

The WebController handles web-based requests for managing employee records via HTML forms. It provides the endpoints for rendering pages to view, add, update, and delete employees.

1. **Render Home Page**

* **Endpoint:** /api/employees/home
* **Method:** GET
* **Description:** Renders the home page.
* **Response:** HTML page with employee data if available.

1. **View Employee by ID**

* **Endpoint:** /api/employees/view/{id}
* **Method:** GET
* **Description:** Renders a page with details of a specific employee.
* **Path Parameters:**
  + id (Long): The ID of the employee to view.
* **Response:** HTML page displaying the employee details.

1. **Render Add Employee Page**

* **Endpoint:** /api/employees/add-employee
* **Method:** GET
* **Description:** Renders a form for adding a new employee.
* **Response:** HTML form for employee creation.

1. **Save New Employee**

* **Endpoint:** /api/employees/add
* **Method:** POST
* **Description:** Processes the form submission and adds a new employee to the system.
* **Request Body:** Form data (mapped to CreateEmployeeDTO).
* **Response:** HTML page showing success or error based on the outcome.

1. **Delete Employee**

* **Endpoint:** /api/employees/delete/{id}
* **Method:** GET
* **Description:** Deletes an employee and renders a success or error page.
* **Path Parameters:**
  + id (Long): The ID of the employee to delete.
* **Response:** HTML page indicating success or error based on the outcome.

1. **Render Update Employee Page**

* **Endpoint:** /api/employees/update-employee/{id}
* **Method:** GET
* **Description:** Renders a form for updating an existing employee.
* **Path Parameters:**
  + id (Long): The ID of the employee to update.
* **Response:** HTML form for updating the employee.

1. **Save Updated Employee**

* **Endpoint:** /api/employees/update/{id}
* **Method:** POST
* **Description:** Processes the form submission and updates the employee record in to the database.
* **Request Body:** Form data (mapped to CreateEmployeeDTO).
* **Response:** HTML page showing success or error based on the outcome.

**DATA VALIDATION RULES**

In the application, data validation is implemented to ensure that the data provided by users conforms to the expected format and constraints. Here are the key validation rules:

1. **CreateEmployeeDTO Validation Rules**

The CreateEmployeeDTO is used when creating a new employee. The validation rules used for this DTO ensure that the necessary fields are provided and are in the correct format.

* **First Name**
  + Required: Yes
  + Validation: Must not be null or empty.
  + Example Constraint: @NotBlank
* **Last Name**
  + Required: Yes
  + Validation: Must not be null or empty.
  + Example Constraint: @NotBlank
* **Email**
  + Required: Yes
  + Validation: Must be a well-formed email address.
  + Example Constraint: @Email, @NotBlank
  + Format Example: example@domain.com

1. **EmployeeDTO Validation Rules**

The EmployeeDTO is used when retrieving, updating, or displaying employee information. It contains the same validation rules of CreateEmployeeDTO as shown above.

And also includes an additional validation rule for the id field, that is:

* **ID**
  + Required: Yes (For updates)
  + Validation: Must be a positive non-null value.
  + Example Constraint: @NotNull

1. **EmployeeRestController Validation Rules**

The EmployeeRestController uses validation annotations to enforce the below rules on the incoming requests:

* **Request Body Validation:**
  + The @Valid annotation is used in the controller methods to enforce validation on the DTOs being passed in the request body. If validation fails, the request will not proceed, and an appropriate error message will be returned.
* **Path Variables:**
  + id (used in methods like getEmployeeById, deleteEmployee, etc.) must be a valid Long and is validated through standard Spring mechanisms to ensure it matches the required type.

**SCHEMAS:**

**Employee table schema(Database):**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Datatype** | **Remarks** |
| id | BIGINT | Primary Key, Not-Null, Auto-incremental |
| first\_name | VARCHAR(255) | Default Null |
| last\_name | VARCHAR(255) | Default Null |
| email | VARCHAR(255) | Default Null |

**Employee Entity schema(Spring Boot):**

|  |  |
| --- | --- |
| **Field Name** | **Datatype** |
| id | Long |
| firstName | String |
| lastName | String |
| Email | String |

**DTO Objects schema:**

