Project Documentation: Hospital Employee Management

HospitalEmployeeManager

Technologies and Tools Used:

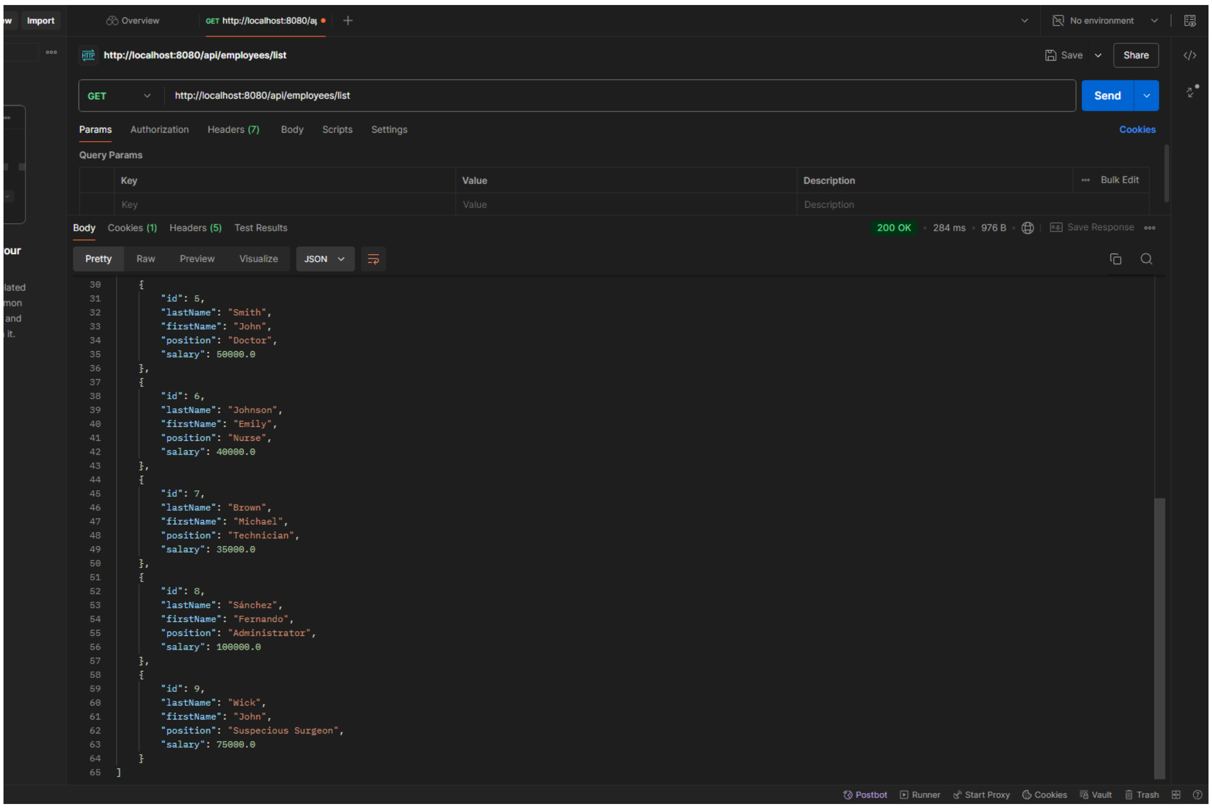
- **Spring Boot 3.2.2:** Main framework for building the application.
- **Spring Data JPA:** For database interaction and CRUD operations.
- **JUnit 5 & Mockito:** Used for unit testing. Mockito is specifically used for mocking in tests.
- **MySQL Connector:** For connecting to a MySQL database.
- **Lombok:** For reducing boilerplate code (e.g., getters, setters).
- **H2 Database:** Used for in-memory database testing.

Introduction: This project is a REST API application designed to manage hospital employees, providing functionalities such as listing, adding, and removing employees. The application is built on Spring Boot 3.2.2, using Spring Data JPA for interaction with a MySQL database and JUnit/Mockito for unit testing. The project follows an MVC (Model-View-Controller) structure and connects to a MySQL database. Below is a description of each project component and its key configurations.

Detailed Project Explanation:

1. **Entry Point: HospitalApplication.java** This class contains the main method that starts the Spring Boot application using SpringApplication.run. It marks the application's entry point, loading the Spring context and configuring all components.
2. **Model: Employee.java** The Employee model is a JPA entity representing hospital employees in the database. This class uses JPA annotations to map entity fields to MySQL table columns.
   Important Annotations:
   - `@Entity`: Indicates that this class will be treated as a JPA entity.
   - `@Table(name = "employees")`: Defines the name of the associated table in the database.
   - `@Id` and `@GeneratedValue`: Designate the id field as the primary key and auto-generated.
3. Additionally, Employee uses Lombok to reduce code for getters, setters, and constructors. Lombok is configured with the following annotations:

    - `@Data`: Automatically generates getters, setters, equals(), hashCode(), and toString().
    - `@AllArgsConstructor` and `@NoArgsConstructor`: Generate constructors with and without parameters.
4. **Persistence Layer: EmployeeRepository.java** EmployeeRepository extends JpaRepository, an interface provided by Spring Data JPA. This allows performing CRUD operations on the Employee entity without additional code.
   Automatic Methods: Thanks to Spring Data JPA, methods like findAll(), findById(), save(), and deleteById() are available automatically.
5. **Service Layer: EmployeeService.java and EmployeeServiceImpl.java** The service layer handles the business logic of the application. The EmployeeService interface defines methods for interacting with employees, while the EmployeeServiceImpl class provides the concrete implementation.
   Key Methods:
    - `findAllEmployees()`: Returns the list of employees.
    - `findEmployeeById(int id)`: Searches for an employee by their ID.
    - `saveEmployee(Employee employee)`: Saves a new employee to the database.
    - `deleteEmployeeById(int id)`: Deletes an employee by their ID if they exist.
6. **REST Controller: EmployeeController.java** The controller exposes REST endpoints for managing employees. It uses Spring MVC annotations to define HTTP methods (GET, POST, DELETE) and handle incoming requests.
   Important Endpoints:
    - `GET /api/employees/list`: Returns a list of all employees.
    - `POST /api/employees/add`: Adds a new employee.
    - `DELETE /api/employees/delete/{id}`: Deletes an employee by their ID.
7. **ResponseEntity:** Used to return appropriate HTTP responses, such as HttpStatus.OK, HttpStatus.CREATED, and HttpStatus.NO_CONTENT.



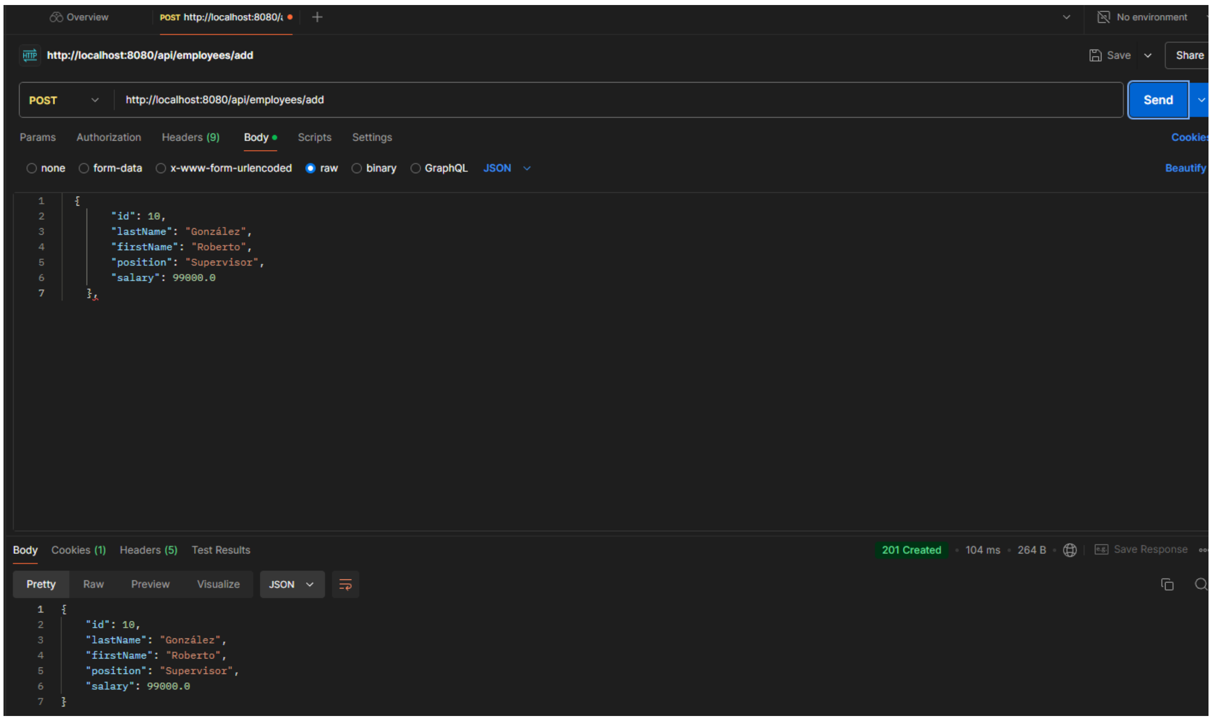
GET http://localhost:8080/api/employees/list

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies (1) Headers (5) Test Results — 200 OK · 284 ms · 976 B

```
    {
        "id": 5,
        "lastName": "Smith",
        "firstName": "John",
        "position": "Doctor",
        "salary": 50000.0
    },
    {
        "id": 6,
        "lastName": "Johnson",
        "firstName": "Emily",
        "position": "Nurse",
        "salary": 40000.0
    },
    {
        "id": 7,
        "lastName": "Brown",
        "firstName": "Michael",
        "position": "Technician",
        "salary": 35000.0
    },
    {
        "id": 8,
        "lastName": "Sánchez",
        "firstName": "Fernando",
        "position": "Administrator",
        "salary": 100000.0
    },
    {
        "id": 9,
        "lastName": "Wick",
        "firstName": "John",
        "position": "Suspecious Surgeon",
        "salary": 75000.0
    }
]
```

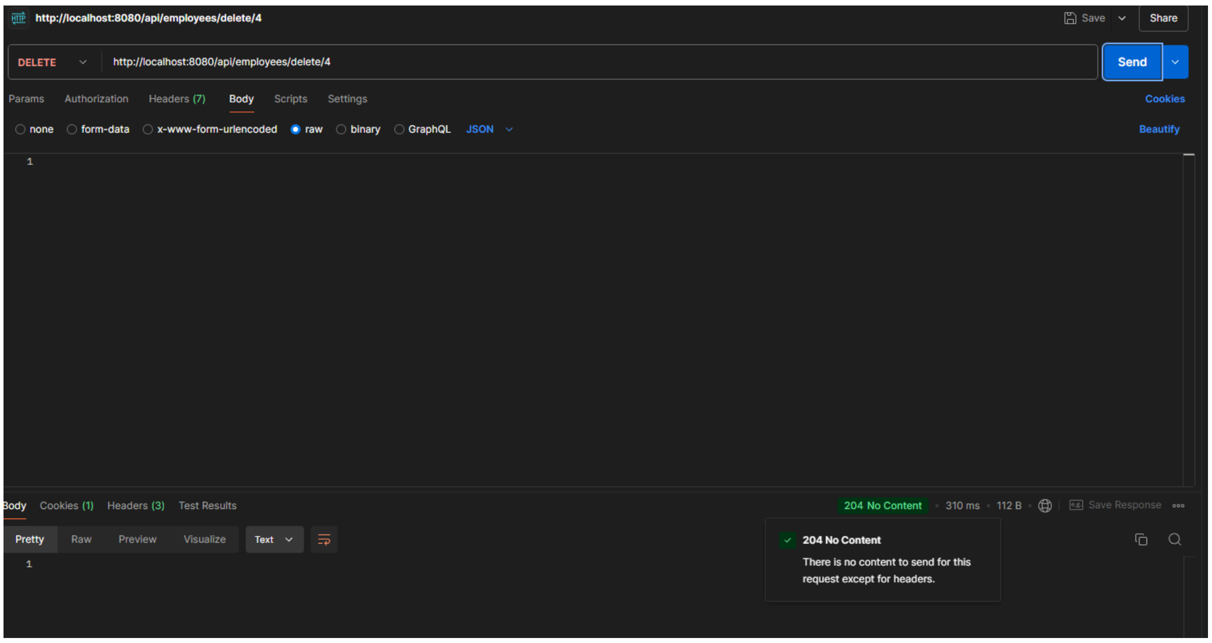


POST http://localhost:8080/api/employees/add

```
{
    "id": 10,
    "lastName": "González",
    "firstName": "Roberto",
    "position": "Supervisor",
    "salary": 99000.0
},
```

Body Cookies (1) Headers (5) Test Results — 201 Created · 104 ms · 264 B

```
{
    "id": 10,
    "lastName": "González",
    "firstName": "Roberto",
    "position": "Supervisor",
    "salary": 99000.0
}
```



DELETE http://localhost:8080/api/employees/delete/4

Body: raw [✓ raw] JSON

204 No Content · 310 ms · 112 B

204 No Content
There is no content to send for this request except for headers.

Key Configurations: application.properties The `application.properties` file is essential for configuring the database connection and Hibernate behavior. It defines MySQL properties and JPA/Hibernate settings.

Database Connection:

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/hospital_db

spring.datasource.username=root

spring.datasource.password=your_password_here

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

These properties connect the application to the local MySQL database. `hospital_db` is the name of the database, and the username and password must be configured correctly.

Hibernate Configuration:

properties

```properties
spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

- `ddl-auto=update`: Allows Hibernate to automatically create or update the database schema based on the defined entities.
- `show-sql=true`: Enables the display of SQL queries generated by Hibernate.
- `hibernate.dialect=org.hibernate.dialect.MySQLDialect`: Specifies the MySQL dialect, which is crucial for Hibernate to generate the correct queries for this database.

MySQL Dialect: The MySQL dialect tells Hibernate how to generate SQL statements specific to MySQL. If not specified correctly, Hibernate may not interact properly with the database, causing errors.

Unit Tests: EmployeeControllerTest.java Unit tests in this project use JUnit 5 and Mockito. This file tests the EmployeeController using a Spring MVC test context.

MockMvc: A class that simulates HTTP requests and validates responses, allowing for controller testing without starting a real server. **Mockito:** Used to mock the behavior of the EmployeeService. This allows testing the controller without relying on a real database.

Test Structure: In the test `testGetAllEmployees()`, Mockito.when() is used to define what the mocked service should return. Then, MockMvc is used to simulate a GET request and validate that the JSON response and employees are correct.

java

```java
@Test

public void testGetAllEmployees() throws Exception {


Mockito.when(employeeService.findAllEmployees()).thenReturn(Arrays.a
sList(

        new Employee("Smith", "John", "Doctor", 50000),

        new Employee("Johnson", "Emily", "Nurse", 40000)

    ));



    mockMvc.perform(get("/api/employees/list")

            .contentType(MediaType.APPLICATION_JSON))

            .andExpect(status().isOk())

            .andExpect(jsonPath("$[0].firstName").value("John"))

            .andExpect(jsonPath("$[1].firstName").value("Emily"));

}
```

This test validates that the GET `/api/employees/list` endpoint returns the expected employees.

Conclusion: This project provides a robust REST API for employee management, built with Spring Boot and Spring Data JPA. The MySQL and Hibernate configurations ensure that data is correctly persisted in the database, while unit tests with Mockito ensure the proper functionality of the controller. The clear and modular structure of the application facilitates future maintenance and scalability.