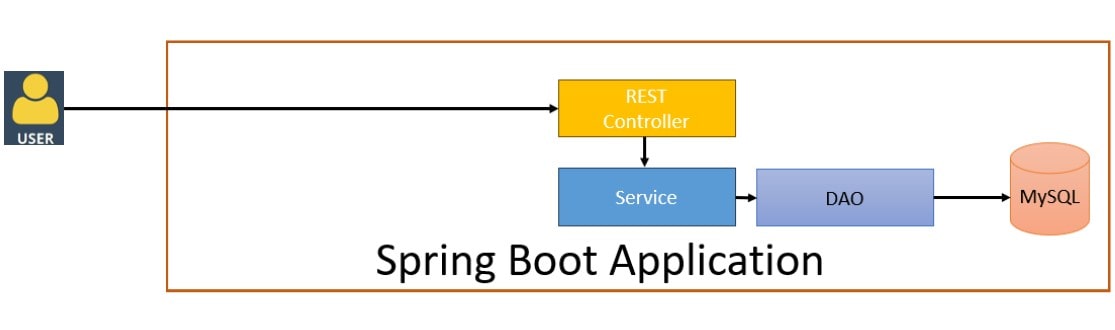
**Employee API:**

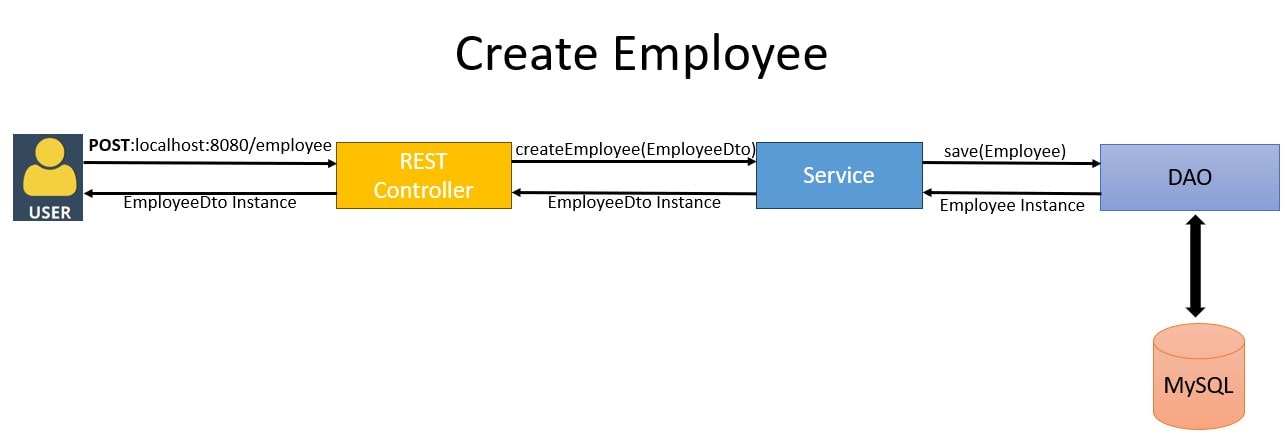
We'll build a Employee API using Spring Boot application to perform CRUD operations with a MySQL database. By the end, you'll have a working API to create, read, update, and delete employee records.

**Architecture:**



**Flow & Skeleton:**

**Create Employee API:**



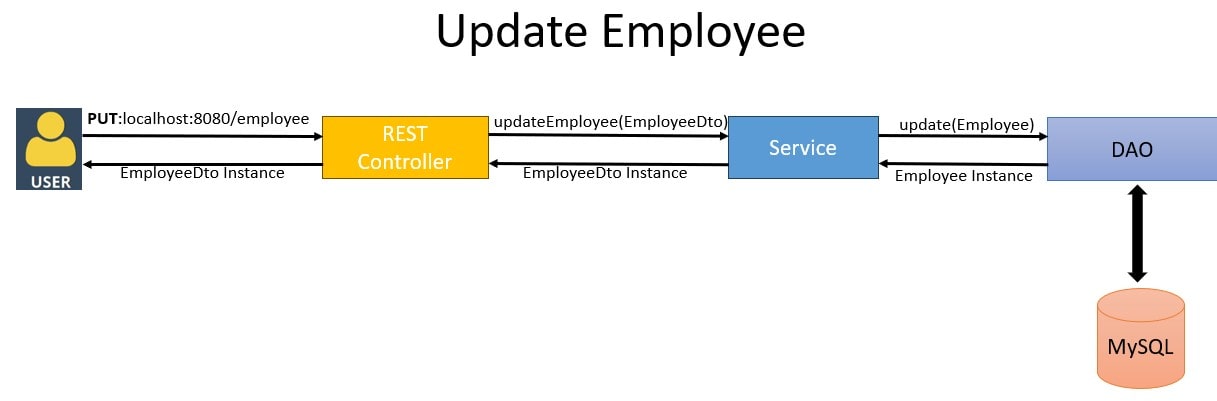
**Get Employee By Id API:**



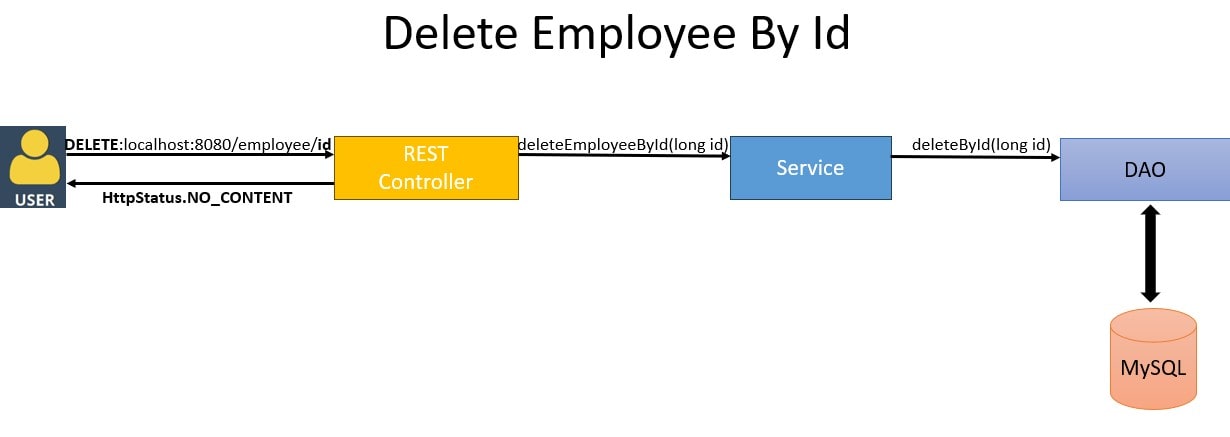
**Get All Employees:**



**Update Employee API:**



**Delete Employee By Id API:**



**1. Setup the Project**

**Software:**

* JDK 21
* MySQL database
* Postman

**1.1. Create a Spring Boot Project**

Use [Spring Initializr](https://start.spring.io/) to create your project with the following settings:

* **Project:** Maven
* **Language:** Java
* **Spring Boot:** 3.x
* **Dependencies:** Spring Web, Spring Data JPA, MySQL Driver

**1.2. Configure pom.xml**

Ensure you have the **spring-boot-starter-data-jpa** dependency:

<dependency>

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

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

</dependency>

**2. Configure MySQL Connection**

Add the following properties to src/main/resources/application.properties:

spring.datasource.url=jdbc:mysql://localhost:3306/empdb?createDatabaseIfNotExist=true&useSSL=false

spring.datasource.username=root

spring.datasource.password=root

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

spring.jpa.hibernate.ddl-auto=create

**3. Create the Employee Entity**

Create a class **Employee** to map to the **employees** table in MySQL:

import javax.persistence.\*;

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "name")

private String name;

@Column(name = "department")

private String department;

// Getters and Setters

}

**4. Create the Repository**

Define the repository interface EmployeeRepository:

import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

**5. Create DTOs**

Define a Data Transfer Object (DTO) for Employee:

public class EmployeeDto {

private Long id;

private String name;

private String department;

// Getters and Setters

}

**6. Create the Service Layer**

Define the service interface **EmployeeService**:

import .util.List;

public interface EmployeeService {

EmployeeDto createEmployee(EmployeeDto employeeDto);

EmployeeDto getEmployeeById(Long employeeId) throws EmployeeNotFoundException;

List<EmployeeDto> getEmployees();

void deleteEmployee(Long employeeId) throws EmployeeNotFoundException;

EmployeeDto updateEmployee(EmployeeDto employeeDto) throws EmployeeNotFoundException;

}

Implement the service **EmployeeServiceImpl**:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import .util.List;

import .util.stream.Collectors;

@Service

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Override

public EmployeeDto createEmployee(EmployeeDto employeeDto) {

Employee employee = mapToEmployee(employeeDto);

Employee savedEmployee = employeeRepository.save(employee);

return mapToEmployeeDto(savedEmployee);

}

@Override

public EmployeeDto getEmployeeById(Long employeeId)

throws EmployeeNotFoundException {

Employee employee = employeeRepository.findById(employeeId)

.orElseThrow(() -> new EmployeeNotFoundException("Employee not found with id " + employeeId));

return mapToEmployeeDto(employee);

}

@Override

public List<EmployeeDto> getEmployees() {

return employeeRepository.findAll().stream()

.map(this::mapToEmployeeDto)

.collect(Collectors.toList());

}

@Override

public void deleteEmployee(Long employeeId)

throws EmployeeNotFoundException {

if (!employeeRepository.existsById(employeeId)) {

throw new EmployeeNotFoundException("Employee not found with id " + employeeId);

}

employeeRepository.deleteById(employeeId);

}

@Override

public EmployeeDto updateEmployee(EmployeeDto employeeDto)

throws EmployeeNotFoundException {

Employee employee = employeeRepository.findById(employeeDto.getId())

.orElseThrow(() -> new EmployeeNotFoundException("Employee not found with id " + employeeDto.getId()));

employee.setName(employeeDto.getName());

employee.setDepartment(employeeDto.getDepartment());

Employee updatedEmployee = employeeRepository.save(employee);

return mapToEmployeeDto(updatedEmployee);

}

private Employee mapToEmployee(EmployeeDto employeeDto) {

Employee employee = new Employee();

employee.setId(employeeDto.getId());

employee.setName(employeeDto.getName());

employee.setDepartment(employeeDto.getDepartment());

return employee;

}

private EmployeeDto mapToEmployeeDto(Employee employee) {

EmployeeDto dto = new EmployeeDto();

dto.setId(employee.getId());

dto.setName(employee.getName());

dto.setDepartment(employee.getDepartment());

return dto;

}

}

**7. Create the Controller**

Define the REST controller **EmployeeController**:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import .util.List;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@PostMapping

public ResponseEntity<EmployeeDto> createEmployee(@RequestBody EmployeeDto employeeDto) {

EmployeeDto createdEmployee = employeeService.createEmployee(employeeDto);

return new ResponseEntity<>(createdEmployee, HttpStatus.CREATED);

}

@GetMapping("/{id}")

public ResponseEntity<EmployeeDto> getEmployeeById(@PathVariable Long id) {

try {

EmployeeDto employee = employeeService.getEmployeeById(id);

return new ResponseEntity<>(employee, HttpStatus.OK);

} catch (EmployeeNotFoundException e) {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

@GetMapping

public ResponseEntity<List<EmployeeDto>> getAllEmployees() {

List<EmployeeDto> employees = employeeService.getEmployees();

return new ResponseEntity<>(employees, HttpStatus.OK);

}

@PutMapping

public ResponseEntity<EmployeeDto> updateEmployee(@RequestBody EmployeeDto employeeDto) {

try {

EmployeeDto updatedEmployee = employeeService.updateEmployee(employeeDto);

return new ResponseEntity<>(updatedEmployee, HttpStatus.OK);

} catch (EmployeeNotFoundException e) {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {

try {

employeeService.deleteEmployee(id);

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

} catch (EmployeeNotFoundException e) {

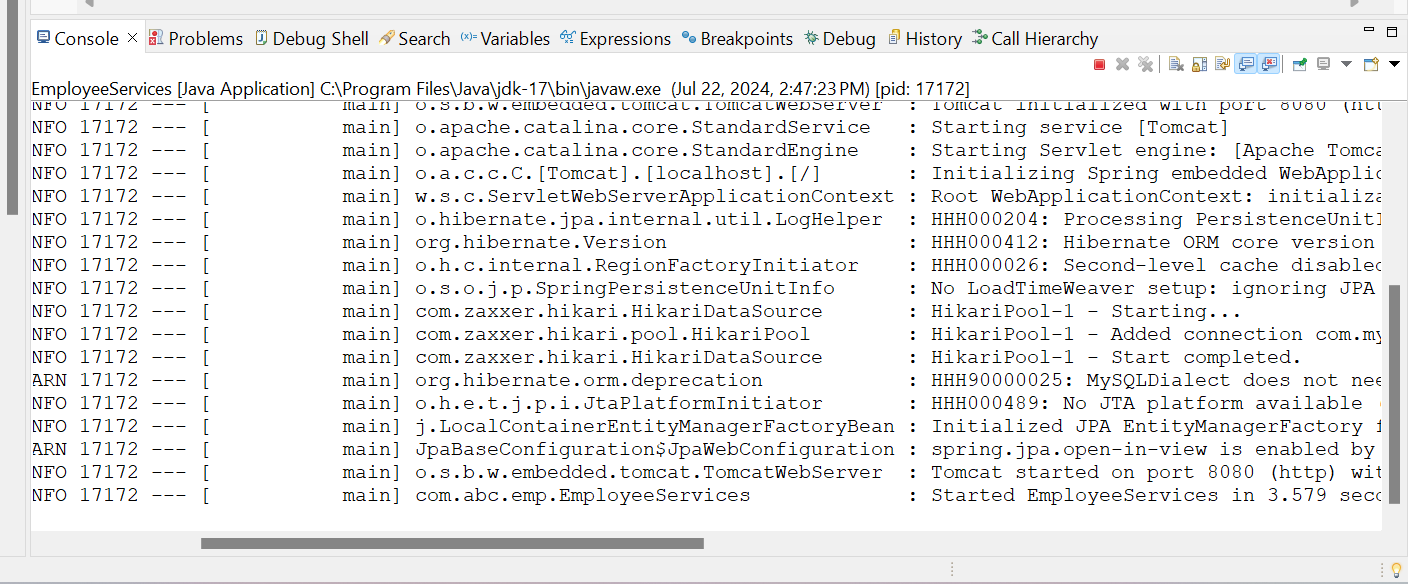
return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

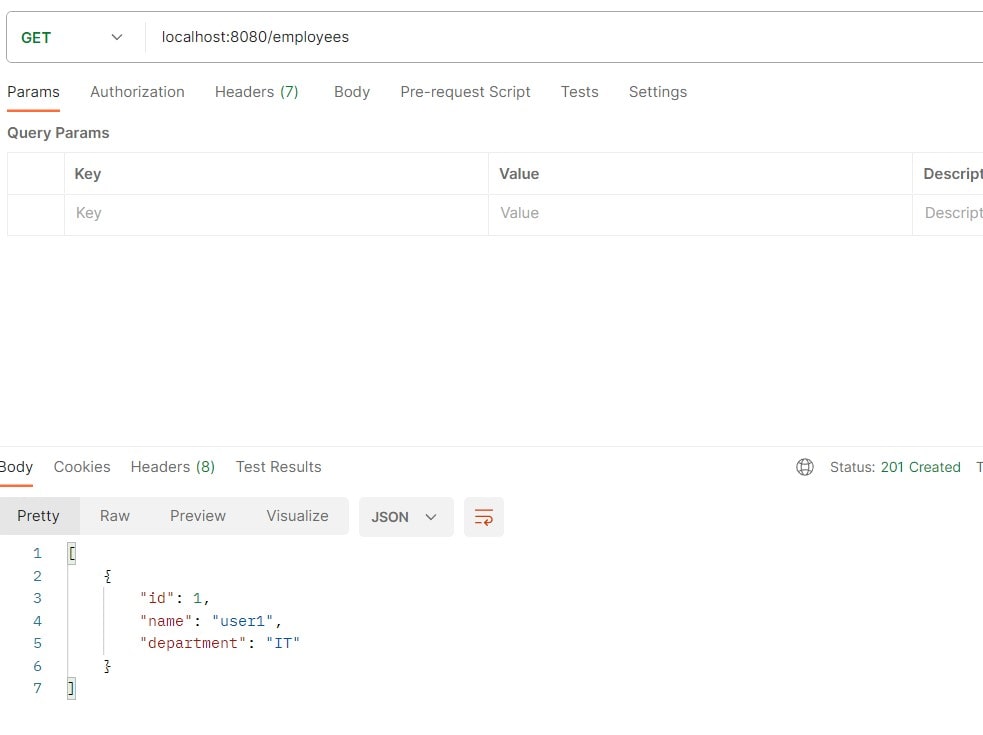
}

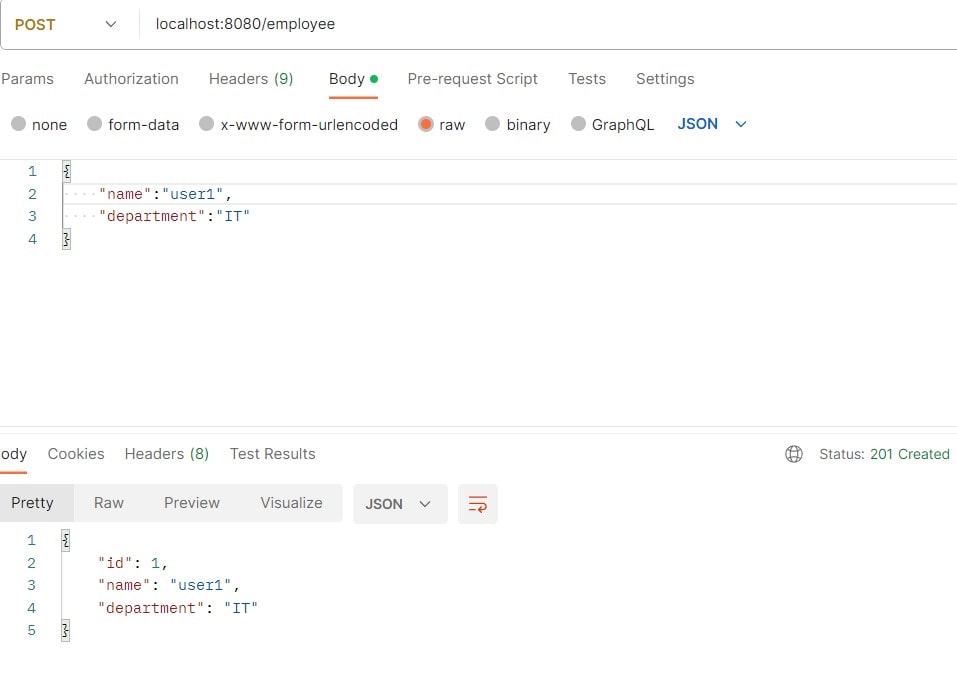
If we now start the Spring Boot application, it does so successfully and connects to the MySQL database. A database named javainusedb gets created.

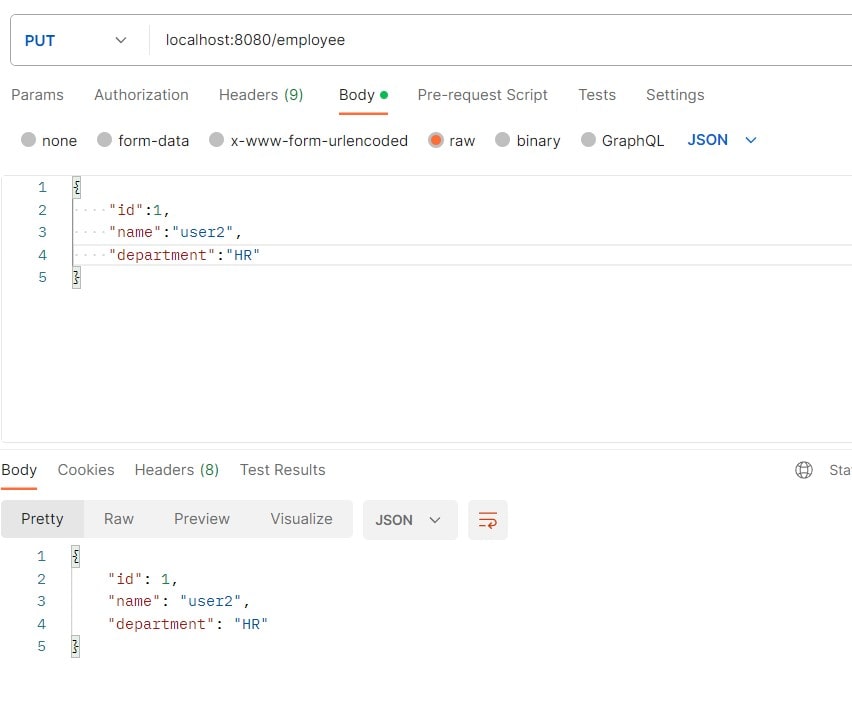


**8. Test the Application**

* **Create Employee:** POST /employees with body { "name": "John Doe", "department": "HR" }
* **Get Employee By ID:** GET /employees/{id}
* **Get All Employees:** GET /employees
* **Update Employee:** PUT /employees with body { "id": 1, "name": "John Smith", "department": "Finance" }
* **Delete Employee:** DELETE /employees/{id}







**Problem Statement:**

Add a search functionality to the existing Employee API to filter employee records by department.

**Requirements - 1:**

**Search Functionality:**

* Implement a new endpoint to search for employees based on their department.
* The endpoint should accept a department name as a query parameter and return a list of employees matching that department.

**Endpoint Changes:**

* Create a new GET endpoint (e.g., /employees/search) that takes a department name as a parameter.
* Ensure that the endpoint returns a list of employees whose department matches the provided value.

**Exception Handling:**

* Handle cases where no employees are found for the given department by returning an appropriate response (e.g., an empty list).

**Acceptance Criteria:**

* The new endpoint should correctly filter employees based on the provided department name.
* The API should return a list of employees in the specified department or an empty list if no matches are found.
* The API documentation should be updated to include details on the new search functionality.

**Requirements - 2:**

**Objective:**

Enhance the existing Employee API to support pagination and sorting. Implement functionality that allows clients to request a specific page of employee records along with sorting options.

**Solution:**

1. **Service Layer Enhancement:** Modify the service layer to support pagination and sorting of employee records. Implement a method in the EmployeeService interface to retrieve a paginated and sorted list of employees.
2. **Controller Update:** Update the controller to handle pagination and sorting parameters. The API should accept query parameters for page number, page size, and sorting criteria.
3. **Testing:** Ensure that the new pagination and sorting features are thoroughly tested to verify correct functionality and performance improvements.

**Acceptance Criteria:**

* The Employee API should provide endpoints that support pagination (e.g., GET /employees?page=0&size=10)