Case Study Document: Spring Boot and Spring REST API

Table of Contents

4. Features

- 4.1 Employee Management
- 4.2 Data Storage and Retrieval
- 4.3 CRUD Operations
- 4.4 Centralized Exception Handlers Using Controller Advice

5. Implementation Steps

- 5.1 Setting up Database Server
- 5.2 Creating a Spring Boot Project
- 5.3 Configuring Spring Boot and Database Integration
- 5.4 Implementing User Management
- 5.5 Handling Data Storage and Retrieval
- 5.6 Implementing CRUD Operations
- 5.7 Handling Exception centrally using Spring REST Controller Advice

1. Introduction

1.1 Purpose

The purpose of this case study **is to demonstrate the integration of Spring Boot, a powerful Java-based framework for building web applications, Spring REST – Very efficient implementation for handing REST Web Services and Oracle as Relational Database**. The assignment aims to showcase the implementation of essential features using this technology stack.

1.2 Technologies Used

- Java 8+

- Spring Boot 2.7.14

- Oracle 11gXE

- Maven for dependency management using pom.xml

- Hibernate as an ORM Tool

- IDE of choice - Eclipse

2. Business Scenario

2.1 Background

The Case Study is based on a scenario where a company requires a RESTful Application to manage Employee Data. The data consists of employees' basic information, such as employee id, first name, last name and email. The application should support the CRUD Operations such as...

- Creating a New Employee Entity --- POST /api/employees

- Reading all Employee Entities --- GET /api/employees
- Reading Single Employee Entity --- GET /api/employees/{employee id}
- Updating Employee Entity --- PUT /api/employees
- Deleting Employee Entity --- DELETE /api/employees

2.2 Problem Statement

The company needs an efficient and scalable solution to manage Employee Data. By providing the Spring REST Backend App, the Data Management can be done with CRUD Operations. This Spring REST Backend App would generate the exchange of data in terms of JSON Data Feed. Further this Back End REST Application will be consumed by Spring MVC Front End Client by hitting the REST API Service End Points.

2.3 Objectives

The main objectives of this case study are:

1. Implement a Spring Boot application integrated with Oracle as the backend database.

2. Create RESTful API endpoints to perform CRUD operations on user data.

3. Demonstrate the use of Spring MVC Front End Client to pull up the JSON Data from Spring REST Application.

4. Validate and test the application for functionality and performance.

3. System Architecture

3.1 High-Level Overview

The system will follow a typical three-tier architecture:

1. Presentation Layer: Handles incoming HTTP requests and returns responses to clients in terms of JSON Data Feed.

2. Business Logic Layer: Contains application logic and coordinates data access.

3. Data Access Layer: Manages interaction with the Oracle database.

3.2 Components

The major components of the system include:

- Controller Layer: Intercepts RESTful API Service Endpoints for Employee Data Management.

- Service Layer: Implements the patching layer between Controller and the Data Access Layer for better transaction management.

- Data Access Layer: Encapsulates the Data Access Logic to interact with the Database .

3.3 Data Flow

The data flow within the system is as follows:

1. Clients (web or mobile applications) send HTTP requests to the API endpoints.

2. The Controller Layer intercepts the requests and validates the inputs.

3. The Data Access Layer interacts with the database to store or retrieve data.

5. Responses are sent back through the Controller Layer to the clients as JSON Data Feed.

4. Features

The Spring REST application will offer the following features:

4.1 Employee Management

- Create new Employee records with first name, last name email, and id.

- Retrieve Employee record by ID.

- Update existing Employee information.

- Delete Employee record.

4.2 CRUD Operations

- Implement Create, Read, Update, and Delete operations using Spring Boot REST

- Validate inputs and handle exceptions gracefully.

5. Implementation Steps –

5.1 Setting up Oracle Database

This is done by configuring Database Properties in application.properties file

5.2 Creating a Spring Boot Project

This is done by using Spring Boot Initializer

5.3 Implementing User Management

This is done by writing the DAO Methods and their implementations to translate the HTTP Methods as CRUD Operations

5.5 Handling Data Storage and Retrieval

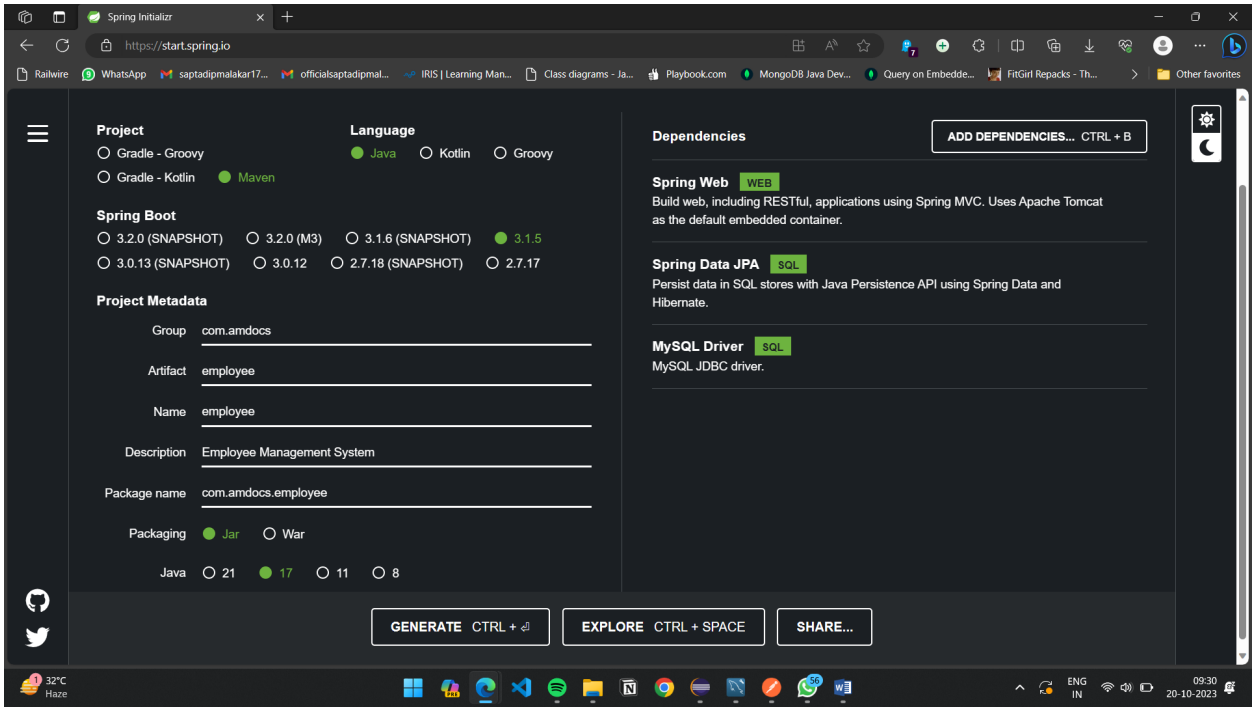
This is done by using ORM Tool like Hibernate

5.6 Implementing CRUD Operations

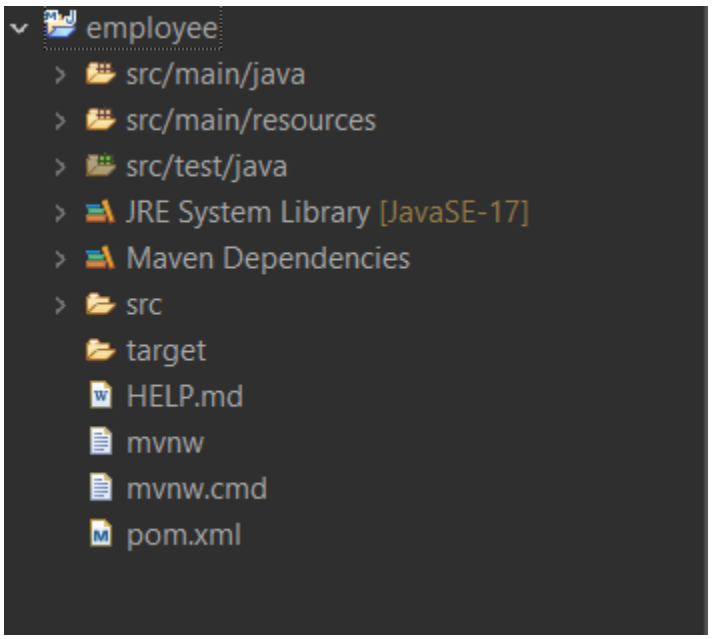
- **C**reating a New Employee Entity --- POST /api/employees
- **R**eading all Employee Entities --- GET /api/employees
- Reading Single Employee Entity --- GET /api/employees/{employee id}
- **U**pdating Employee Entity --- PUT /api/employees
- **D**eleting Employee Entity --- DELETE /api/employees

Solution:

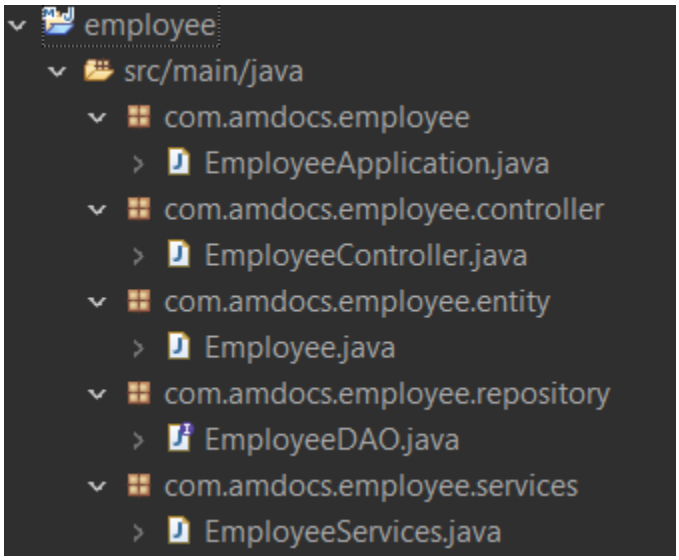
1. Created a spring project using spring initializr.



2. Imported to Eclipse IDE.



3. Created the required packages and classes.



4. EmployeeController Class

```java
package com.amdocs.employee.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.amdocs.employee.entity.Employee;
import com.amdocs.employee.services.EmployeeServices;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {


    @Autowired
    private EmployeeServices employeeService;

    @GetMapping
    public ResponseEntity<?> getAllEmployees() {
        return ResponseEntity.ok (this.employeeService.getAllEmployees());
    }

```

```java
    @GetMapping("/{id}")
    public  ResponseEntity<?>  getEmployeeById(@PathVariable Integer id) {
        return ResponseEntity.ok (employeeService.getEmployeeById(id));
    }

    @PostMapping
    public Employee addEmployee(@RequestBody Employee employee) {
        return this.employeeService.saveEmployee(employee);
    }

    @PutMapping("/{id}")
    public Employee updateEmployee(@PathVariable int id, @RequestBody Employee employee) {
        employee.setEmp_Id(id);
        return employeeService.updateOrCreateEmployee(employee);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<HttpStatus> deleteEmployee(@PathVariable Integer id) {

        try {
        this.employeeService.deleteEmployee(id);
        return new ResponseEntity<>(HttpStatus.OK);
        }catch(Exception e) {
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```

5. Entity Class

```java
package com.amdocs.employee.entity;

import jakarta.persistence.Entity;

@Entity
public class Employee{

    @Id
    private int emp_Id;
    private String Name;
    private String Number;
    private String Role;

    public Employee() {
        super();
    }

    public Employee(int emp_Id, String name, String number, String role) {
        super();
        this.emp_Id = emp_Id;
        Name = name;
        Number = number;
        Role = role;
    }

    public int getEmp_Id() {
        return emp_Id;
    }

    public void setEmp_Id(int emp_Id) {
        this.emp_Id = emp_Id;
    }

    public String getName() {
        return Name;
    }

    public void setName(String name) {
        Name = name;
    }

    public String getNumber() {
        return Number;
    }

    public void setNumber(String number) {
        Number = number;
    }

    public String getRole() {
        return Role;
    }

    public void setRole(String role) {
        Role = role;
    }

    @Override
    public String toString() {
        return "Employee [emp_Id=" + emp_Id + ", Name=" + Name + ", Number=" + Number + ", Role=" + Role + "]";
    }


}
```

6. DAO class

```java
package com.amdocs.employee.repository;

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

public interface EmployeeDAO extends JpaRepository<Employee,Integer>{

}

```

7. Services Class

```java
package com.amdocs.employee.services;

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


@Service
public class EmployeeServices {

        @Autowired
        private EmployeeDAO employeeRepository;


        public List<Employee> getAllEmployees() {
            return employeeRepository.findAll();
        }

        public Employee getEmployeeById(Integer id) {
            return employeeRepository.findById(id).orElse(null);
        }

        public Employee saveEmployee(Employee employee) {
            |
            employeeRepository.save(employee);
            return employee;
        }
```

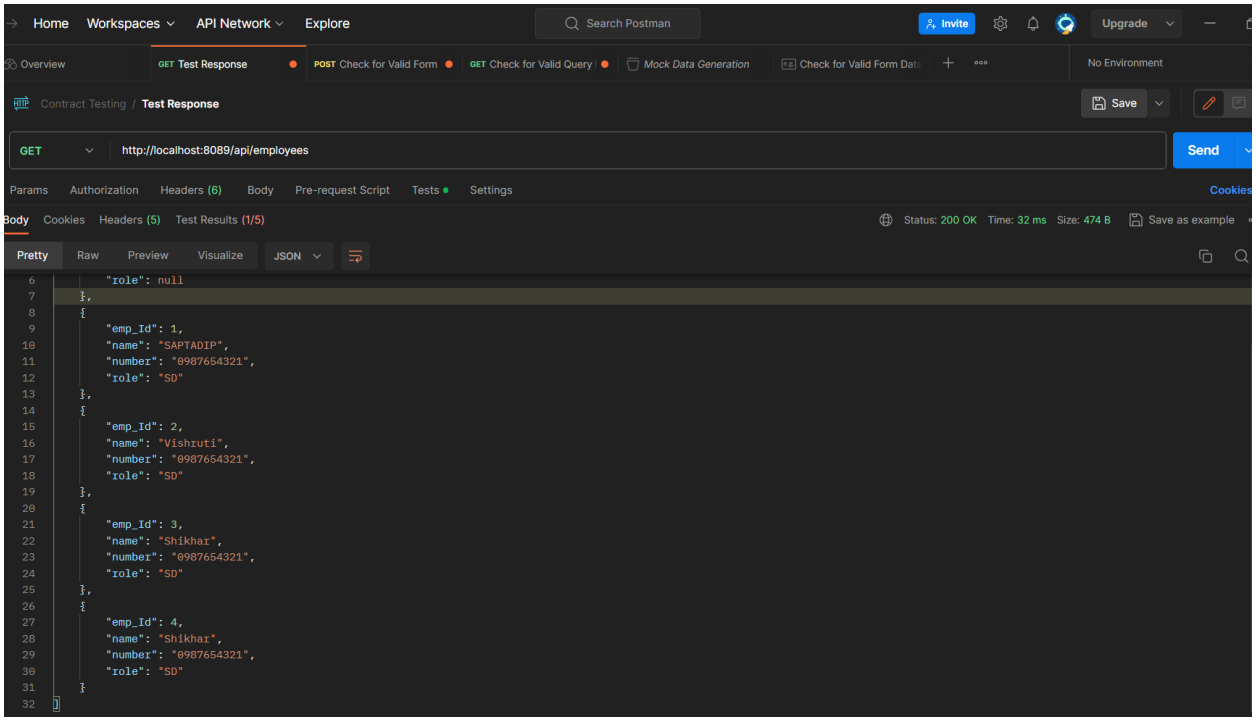
```java

        public Employee updateOrCreateEmployee(Employee employee) {
            Optional<Employee> existingEmployee = employeeRepository.findById(employee.getEmp_Id());

            if (existingEmployee.isPresent()) {
                Employee updatedEmployee = existingEmployee.get();
                updatedEmployee.setName(employee.getName());
                updatedEmployee.setNumber(employee.getNumber());
                updatedEmployee.setRole(employee.getRole());
                return employeeRepository.save(updatedEmployee);
            } else {
                return employeeRepository.save(employee);
            }
        }

        public void deleteEmployee(Integer id) {
            employeeRepository.deleteById(id);
        }}
```

6. Application.properties

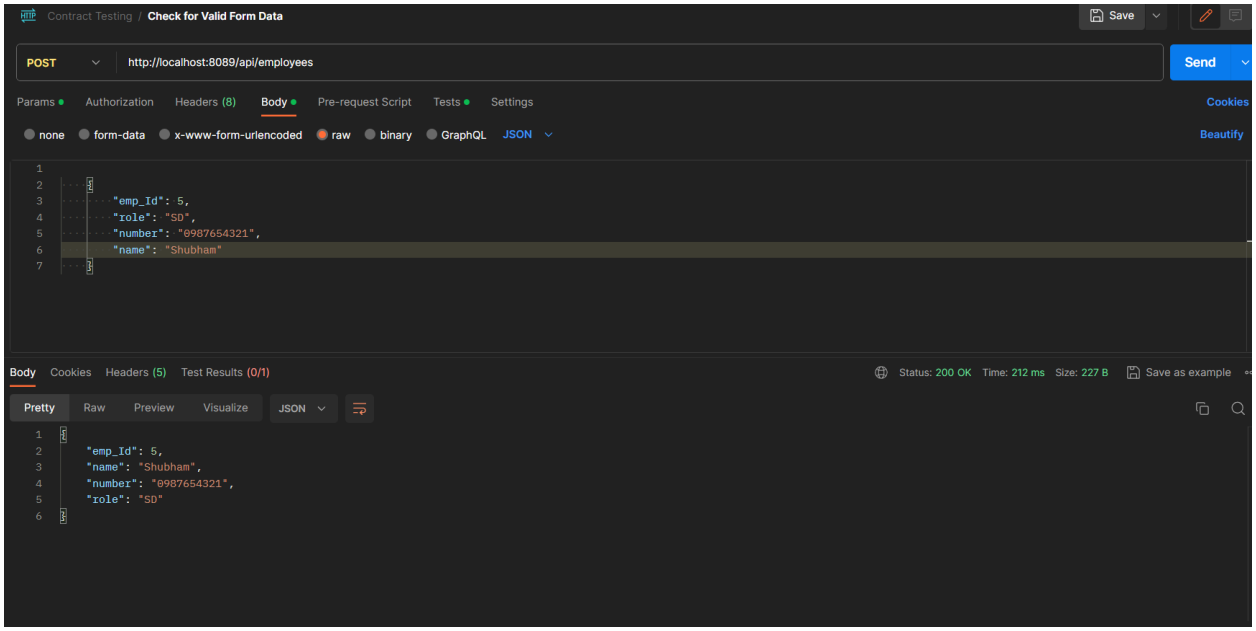
```
server.port=8089

#databaseConfiguration
spring.datasource.url=jdbc:mysql://localhost:3306/employee
spring.datasource.username =root
spring.datasource.password =root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

#hibernateconfig

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Output:

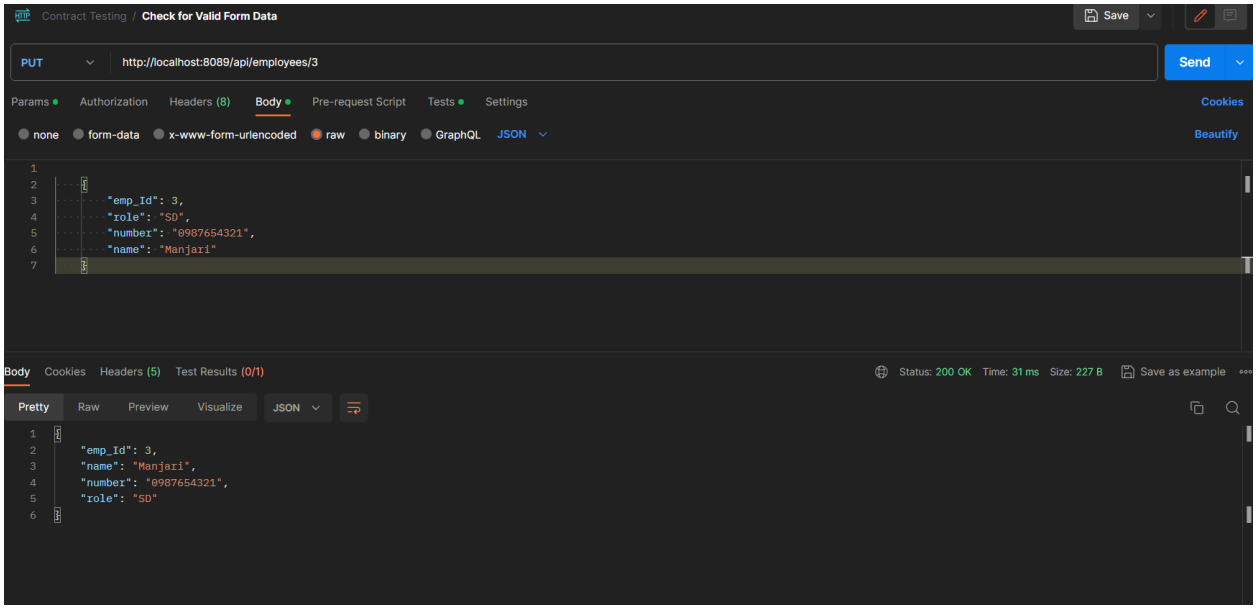
1. GET



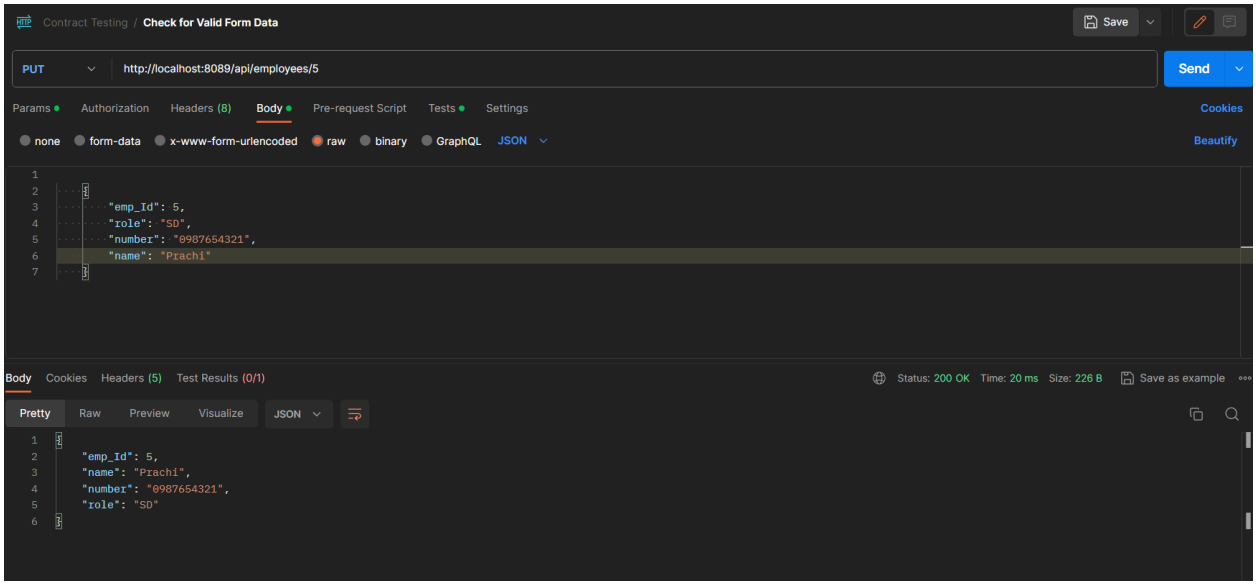
2. POST



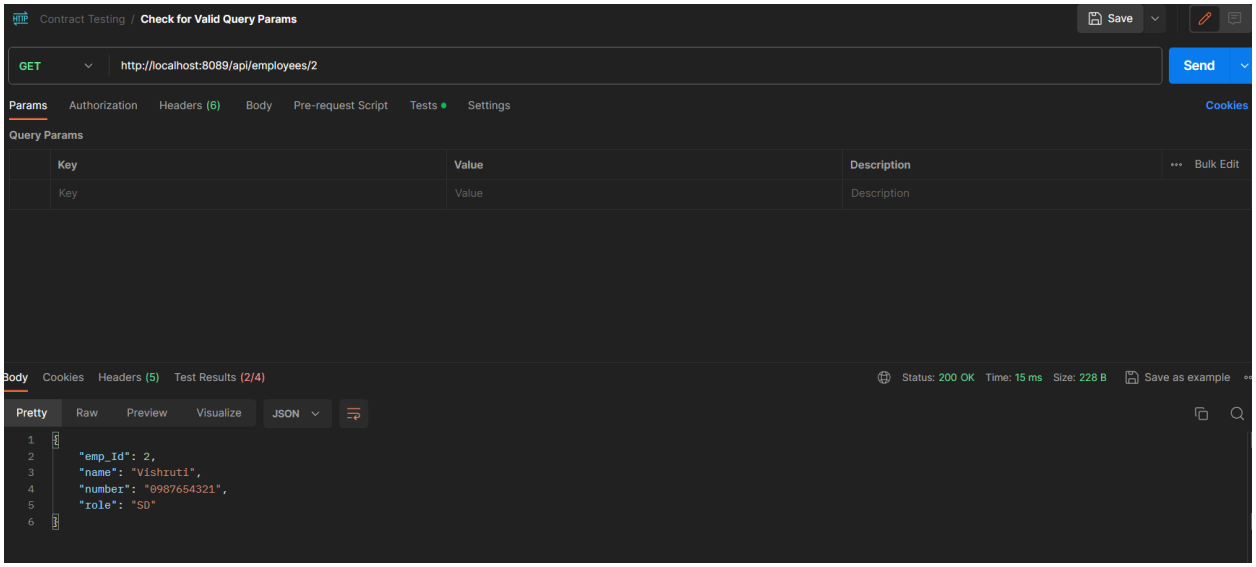
3. If the ID exists the values of the present entry will be updated.



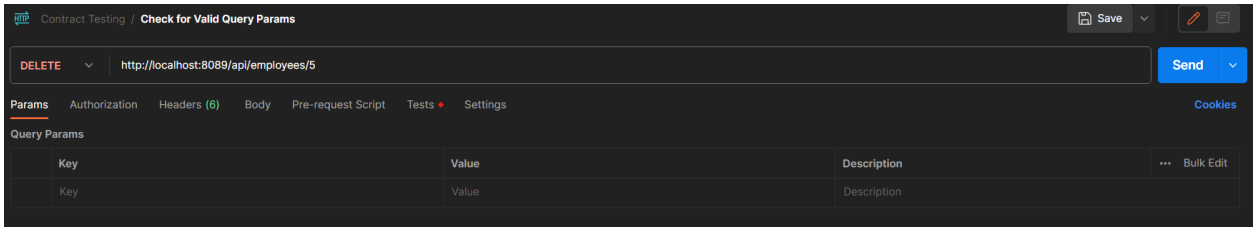
If the ID doesn't exist a new entry will be made to the database.

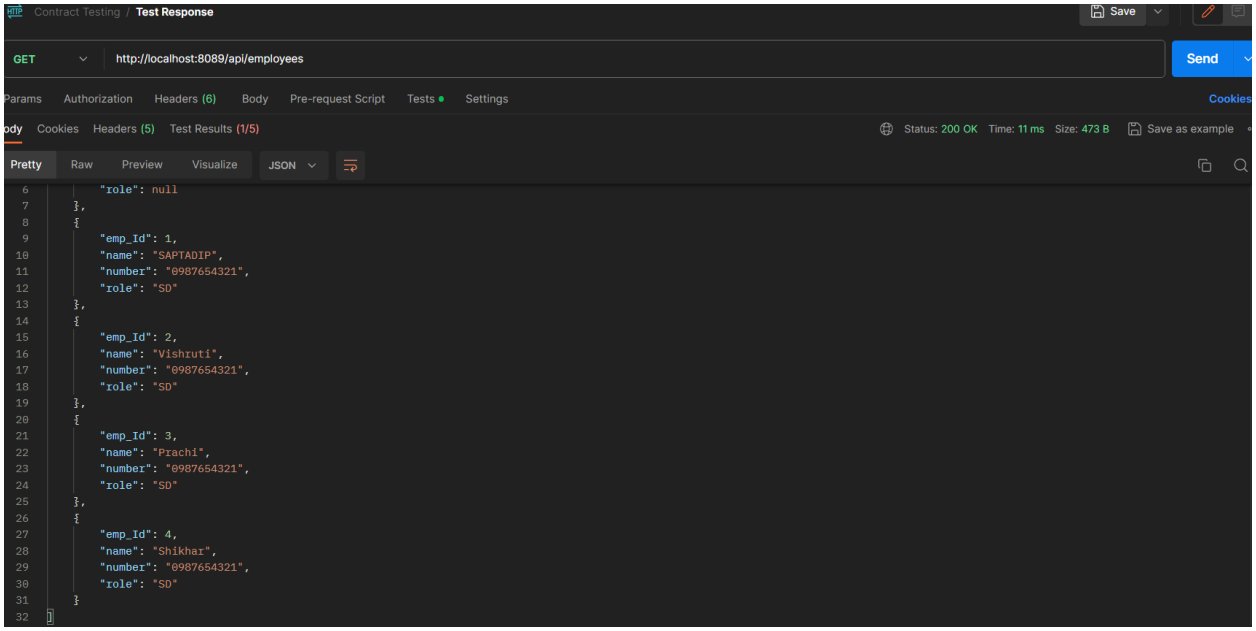


4. GETBYID



5. DELETE





Deleted entry with Id equal to 5.