SPRING DATA JPA CRUD EXAMPLE

The Spring boot supports the JPA(Java Persistence API) for the database operations. It provides the starter dependency, which makes development easier and fast. It also provides repositories, which give required support for most of the database operations. Creating the CRUD application with the spring data jpa is a simple process.

## **Creating the spring data JPA CRUD application**

Create a spring boot application with the required starter dependencies. Also, add the web, spring data jpa, and Lombok dependencies.

In this example, we are creating a spring boot application. We are using the **com.cg.springjpaexample** as the base package for the application.

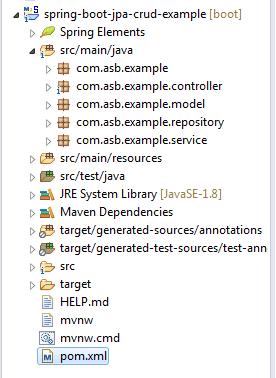
**Choose following dependencies:**

Spring – Web

Spring Data JPA

Mysql/oracle/postgresql

The below image shows the project structure.



The below class is the entry point of the application.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package com.cg.springjpaexample;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  public class SpringBootJpaCrudExampleApplication {      public static void main(String[] args) {          SpringApplication.run(SpringBootJpaCrudExampleApplication.class, args);      }  } |

### **Add database related configurations setup**

Open the application property file under the **resources** folder, and add the below content. Here, we are setting the database configuration with the database username, password, and URL.

Also, we have disabled any SQL DDL operation execution by application by setting spring.jpa.hibernate.ddl-auto property to none. This prevents any table or schema creation or updates from the application side during startup.

|  |  |
| --- | --- |
| 1  2  3  4  5 | spring.datasource.url=  spring.datasource.username=  spring.datasource.password=  spring.jpa.properties.hibernate.jdbc.lob.non\_contextual\_creation=true  spring.jpa.hibernate.ddl-auto=  spring.jpa.show-sql=true |

### **Create an entity class and the repository interface**

Create a new package(model) for the entity and create a java class called **Employee.**Add the below content. Here, we are using the **@Entity** annotation to make the class an entity.

We are also using the **@Table** annotation to specify the mapping of the class with the database table. We have also used Lombok annotations for getters, setters, and constructor creation.

The entity class contains all the database table fields mapped to corresponding entity class fields using the **@Column** annotation. The id field is autogenerated value, which is annotated with the **@Id** and **@GeneratedValue** annotation, with generation strategy type as Identity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package com.cg.springjpaexample.model;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Table;  import lombok.AllArgsConstructor;  import lombok.Getter;  import lombok.NoArgsConstructor;  import lombok.Setter;  @Entity  @AllArgsConstructor  @Getter  @Setter  @NoArgsConstructor  @Table(name="Employee")  public class Employee {      @Id      @GeneratedValue(strategy=GenerationType.IDENTITY)      @Column(name="Id")      private Long id;        @Column(name="EmployeeName")      private String employeeName;        @Column(name="EmployeeCode")      private String employeeCode;        @Column(name="Designation")      private String designation;  } |

Create a package for the repository layer and the **EmployeeRepository** interface and add the below content.

Also, Spring boot provides the JpaRepository interface, which supports the pagination and sorting of the result set, etc.

For our CRUD operation, we do not have to write any extra code, as CRUD operation is already supported by spring boot by default. Also, our repository interface is extending the **JpaRepository**interface to inherit the available features.

|  |  |
| --- | --- |
| 1  2  3  4  5 | package com.cg.springjpaexample.repository;  import org.springframework.data.jpa.repository.JpaRepository;  import com.asb.example.model.Employee;  public interface EmployeeRepository extends JpaRepository<Employee, Long> {  } |

### **Creating the service and controller layers**

It is always good practice to put our business logic inside the service layer, which gives a layer of separation from the controller and the database operations. Even though we are performing only the CRUD operations, we are going to create a service layer in this example.

Create a new package for the service layer. Create an interface called **EmployeeService**, and add the required methods as shown below.

We also have different methods defined for the CRUD operation.

package com.asb.example.service;

import java.util.List;

import com.asb.example.model.Employee;

public interface EmployeeService {

public Employee createEmployee(Employee emp);

public Employee updateEmployee(Employee emp);

public Employee getEmployee(Long empId);

public void deleteEmployee(Long empId);

public List<Employee> getAllEmployee();

}

Create an implementation class for the created EmployeeService interface and annotate the class with the @Service annotation.

We have also injected the EmployeeRepository instance by using the @Autowired annotation to perform database operations. In this class, we have the implementation of the service methods.

package com.asb.example.service;

import java.util.List;

import java.util.Optional;

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

import org.springframework.stereotype.Service;

import com.asb.example.model.Employee;

import com.asb.example.repository.EmployeeRepository;

@Service

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Override

public Employee createEmployee(Employee emp) {

return employeeRepository.save(emp);

}

@Override

public Employee updateEmployee(Employee emp) {

return employeeRepository.save(emp);

}

@Override

public Employee getEmployee(Long empId) {

Optional<Employee> optionalEmp = employeeRepository.findById(empId);

if(optionalEmp.isPresent()) {

return optionalEmp.get();

}

return null;

}

@Override

public void deleteEmployee(Long empId) {

employeeRepository.deleteById(empId);

}

@Override

public List<Employee> getAllEmployee() {

return employeeRepository.findAll();

}

}

Create a package for the controller layer, and create a REST controller class as shown below.

Here, we have different endpoints for CRUD operation.

package com.cg.springjpaexample.controller;

import java.util.List;

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.RestController;

import com.asb.example.model.Employee;

import com.asb.example.service.EmployeeService;

@RestController

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@PostMapping(consumes = "application/json", produces = "application/json", path = "/employee")

public ResponseEntity<Employee> createEmployee(@RequestBody Employee emp) {

return new ResponseEntity<>(employeeService.createEmployee(emp), HttpStatus.CREATED);

}

@PutMapping(consumes = "application/json", produces = "application/json", path = "/employee")

public ResponseEntity<Employee> updateEmployee(@RequestBody Employee emp) {

return new ResponseEntity<>(employeeService.updateEmployee(emp), HttpStatus.CREATED);

}

@DeleteMapping(produces = "application/json", consumes = "text/plain", path = "/employee/{empId}")

public ResponseEntity<String> deleteEmployee(@PathVariable(value="empId") Long empId) {

employeeService.deleteEmployee(empId);

return new ResponseEntity<>("Employee with EmployeeId : " + empId + " deleted successfully", HttpStatus.OK);

}

@GetMapping(path = "/employee/{empId}", produces = "application/json")

public ResponseEntity<Employee> getEmployee(@PathVariable(value = "empId") Long empId) {

return new ResponseEntity<>(employeeService.getEmployee(empId), HttpStatus.OK);

}

@GetMapping(path = "/employees", produces = "application/json")

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

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

}

}

## **Running and testing the application**

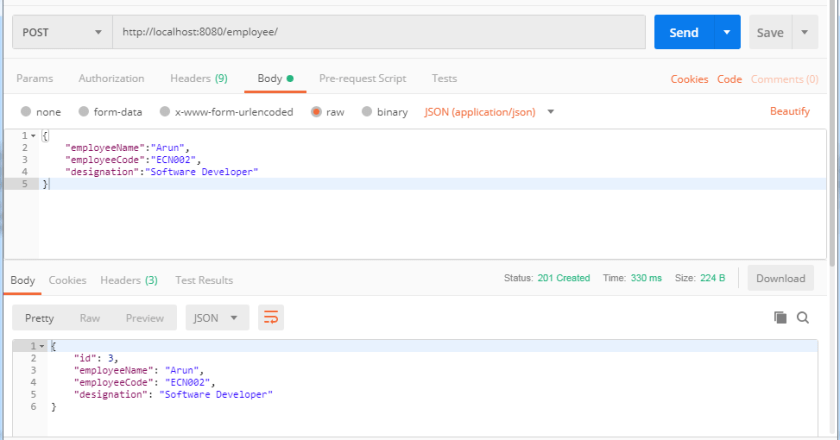
Start the spring boot application. By default, our application will start on the embedded tomcat server on the default port 8080.

Since we have not defined any application context, we can directly access the REST endpoints mentioned in each request mapping.

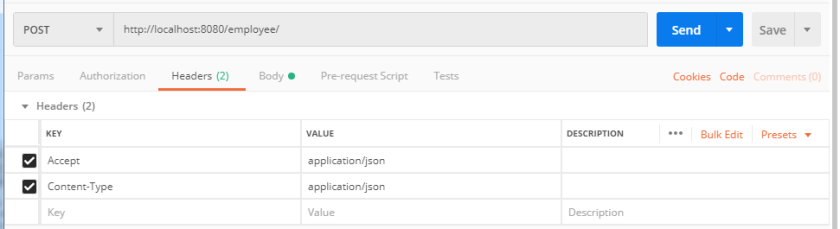
### **Create Example (POST Method)**

Open the postman and Fill the POST URL, to create a new employee record as shown below. Notice that the request type is **POST**and the **URL**is **http:localhost:8080/employee/**.

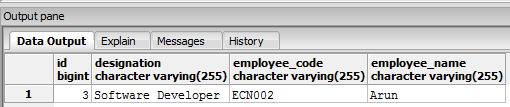
The request body contains the JSON content with employee details.



Make sure that request headers **Accept**and **Content-Type**are set as shown below.

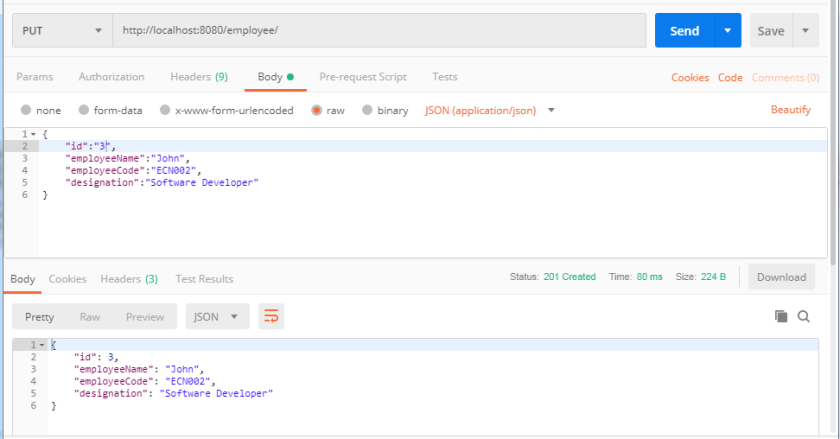


The values are added in the database table as shown below.

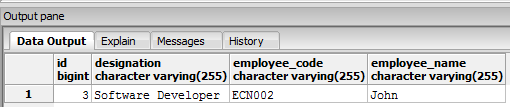


### **Update Example (PUT Method)**

Request and response of update operation are shown below. Here, we have to pass the id column as well in the request, as the Jpa Repository’s save method updates the employee details based on Id input. In this example, we are updating the employee name with the value ‘John’.



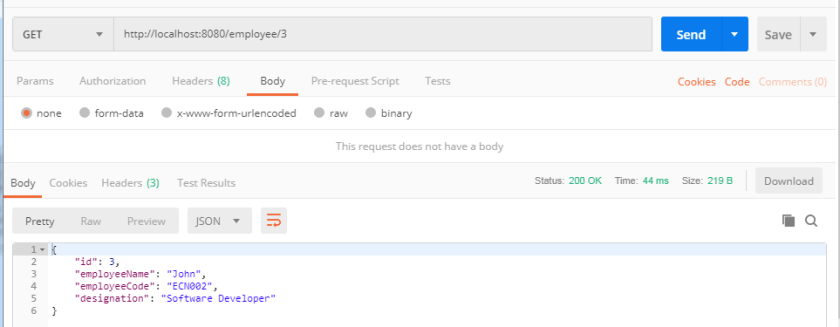
The updated employee name in the database table is shown below.



### **Read By Id Example (GET Method)**

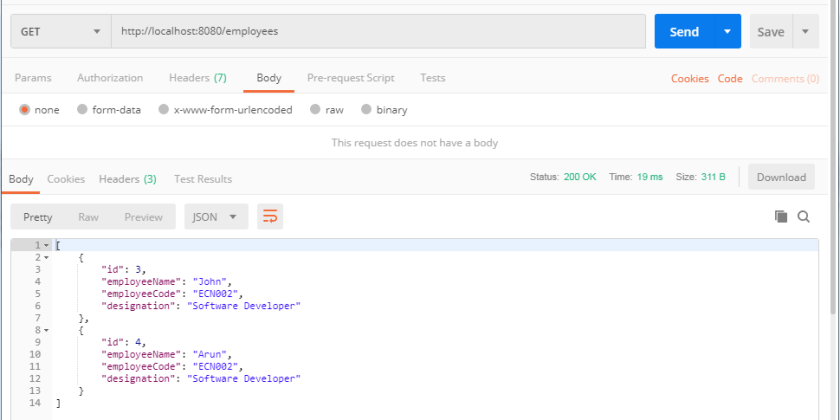
To fetch the stored employee details, we should use the HTTP **GET**method with **URL http:localhost:8080/employee/{empId}**.

Here, we are fetching the record of the employee with Id 3.



### **Read All Employee Details (GET Method)**

Also, the below API retrieves all available records from the database table.



### **Delete Example (DELETE Method)**

For deleting the employee record, we are using the HTTP DELETE method. We are passing the employee Id to delete the record from the database.

