## **WEEK 9**

Industrial Report

**CRUD operations in Spring Boot:**

**CRUD operations** stand for Create, Read, Update, and Delete, which are fundamental operations used in database and information systems to manage and manipulate data. These operations are typically performed on database records or entities and are fundamental to the functionality of most software applications.

**Create (C):**

**Purpose:** To create new records or entities in the database.

**Database Operation: T**ypically involves inserting a new row of data into a database table.

**Example:** Adding a new user to a user database.

**Read (R):**

**Purpose:** To retrieve or read existing records or entities from the database.

**Database Operation:** Involves querying the database to retrieve information.

**Example:** Retrieving the details of a specific user from a user database.

**Update (U):**

**Purpose:** To modify or update existing records or entities in the database.

**Database Operation:** Typically involves modifying the values of one or more fields in a database record.

**Example:** Changing the email address of a user in a user database.

**Delete (D):**

**Purpose:** To remove or delete existing records or entities from the database.

**Database Operation:** Involves removing a row or record from a database table.

**Example:** Deleting a user from a user database.

CRUD operations are often mapped to HTTP methods:

* **Create:** POST
* **Read:** GET
* **Update:** PUT or PATCH
* **Delete:** DELETE

**Performing CRUD operations in spring boot:-**

1. **Project setup.**

* Open your web browser and go to the <https://start.spring.io/> website.
* Choose the project settings such as language (usually Java), Spring Boot version, and project metadata (like group name, artifact, and package name).
* Select the dependencies you need. **[Spring Web, Spring Data JPA]**
* Click the "Generate" button. This will download a zip file containing your Spring Boot project with the specified settings and dependencies.
* Extract the contents of the downloaded zip file to a location on your computer.

1. **Import the project.**

* Open Eclipse IDE on your computer.
* go to File > Import.
* Choose Existing Maven Project depending on the build tool you selected in the Spring Initializr
* Navigate to the location where you extracted the downloaded project and select the project folder.
* Click "Finish" to complete the import process.
* Eclipse will automatically recognize the project structure and download the necessary dependencies.

1. **Project Settings**

* Add dependency of mysql database in **pom.xml** file.

| <dependencies>  <!-- Other dependencies -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>8.0.23</version> <!-- Use the latest version available -->  </dependency> </dependencies> |
| --- |

* Configure Database connection

| spring.datasource.url=jdbc:mysql://localhost:3306/emp\_db spring.datasource.username=root spring.datasource.password=\*\*\*\*\*\*\*\* spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver spring.jpa.hibernate.ddl-auto=update |
| --- |

1. **Create Entity Class:**

Create a Java class representing your data entity. Annotate it with @Entity to mark it as a JPA entity. Define fields and annotate them with appropriate JPA annotations.

| import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id;  @Entity public class Employee{  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  private int salary;   // generate getters and setters } |
| --- |

1. **Create Repository Interface:**

Create a repository interface by extending the JpaRepository interface provided by Spring Data JPA.

| import org.springframework.data.jpa.repository.JpaRepository;  public interface YourEntityRepository extends JpaRepository<Employee, Long> { } |
| --- |

1. **Create Controller:**

Create a controller class to handle HTTP requests and invoke the service methods.

| import org.springframework.beans.factory.annotation.Autowired; import org.springframework.web.bind.annotation.\*;  import java.util.List; import java.util.Optional;  @RestController @RequestMapping("/employee/") public class RequestHandler{   @Autowired  private Employee emp;   @GetMapping  public List<Employee> getAllEntities() {  return emp.findAll();  }   @GetMapping("/{id}")  public Optional<Employee> getEntityById(@PathVariable Long id) {  return emp.findById(id);  }   @PostMapping  public EmployeesaveEntity(@RequestBody Employee emp) {  return emp.save(entity);  }   @DeleteMapping("/{id}")  public void deleteEntity(@PathVariable Long id) {  emp.deleteById(id);  } } |
| --- |

**Custom Query:-**

In Spring Data JPA, custom queries can be defined using the **@Query** annotation on repository methods. This annotation allows you to provide your own **JPQL** (Java Persistence Query Language) or native SQL query to be executed by the underlying JPA provider. Custom queries are helpful when the standard Spring Data JPA query derivation mechanism or method naming conventions are not sufficient for your needs.

**Example**:-

Assuming you have a project and an entity class Employee:

**//Entity Class**

| @Entity public class Employ{  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;   private String ename;  private int salary;   // getters and setters } |
| --- |

**//Entity Repository**

| import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.jpa.repository.Query; import java.util.List;  public interface EmployRepo extends JpaRepository<Employ, Long> {   // Custom Query using JPQL  @Query("SELECT e FROM Employ e WHERE e.ename =?0")  List<Employee> findByName(String name);   // Custom Query using native SQL  @Query(value = "SELECT \* FROM Employ WHERE salary > ?0", nativeQuery = true)  List<Employ> findBySalary(int salary); } |
| --- |

**//Controller Class**

| import org.springframework.beans.factory.annotation.Autowired; import org.springframework.web.bind.annotation.\*;  import java.util.List;  @RestController @RequestMapping("/api/entities") public class RequestHAndler{   @Autowired  private Employ emp;   @GetMapping("/byname/{name}")  public List<Employ> getEntitiesByName(@PathVariable String name) {  return emp.findByName(name);  }   @GetMapping("/bysalary/{salary}")  public List<Employ> getEntitiesBySalary(@PathVariable int age) {  return emp.findBySalary(salary);  } } |
| --- |

**Testing using PostMan software.**

1. **Retrieve All Employees (GET Request):**
2. Open Postman.
3. Set the method to GET.
4. Enter the URL http://localhost:8080/employee
5. Click "Send."

**This should retrieve all employees from your MySQL database, and you'll see the response in the Postman window.**

**2. Add a New Employee(POST Request):**

Method: POST

URL: http://localhost:8080/employee/

Body (raw JSON):

| {  "ename": "Jhon",  "salary": 150000 } |
| --- |

1. Open Postman.
2. Set the method to POST.
3. Enter the URL http://localhost:8080/employee
4. Go to the "Body" tab, select "raw," and choose "JSON (application/json)."
5. Enter the JSON payload for the new employee.
6. Click "Send."

**This should add a new employee to your MySQL database, and you'll see the response in the Postman window.**

**3. Find Employee by Name(GET Request):**

Method: GET

URL: [http://localhost:8080/EmpByName/{emp.name](http://localhost:8080/EmpByName/%7Bemp.name)}

1. Open Postman.
2. Set the method to GET.
3. Enter the URL <http://localhost:8080/EmpByName/alex> (or any name).
4. Click "Send."

This should retrieve an employee with the name from your MySQL database, and you'll see the response in the Postman window.