Spring JDBC

Introduction to JDBC

What is JDBC?

- JDBC (Java Database Connectivity) is a standard API for interacting with relational databases using Java.
- Challenges with plain JDBC:
  - Boilerplate code for connection management.
  - Complex exception handling.

- Manual resource cleanup.

How Spring JDBC Helps:

- Simplifies database access with templates.
- Handles resource management automatically.
- Provides exception translation.

Introducing JdbcTemplate

- Core class for Spring JDBC.
- Simplifies:
  - Query execution.
  - Parameterized queries.
  - Batch processing.

Example of a plain JDBC query:

```
try (Connection connection = DriverManager.getConnection("u
rl", "username", "password")) {
    PreparedStatement preparedStatement = connection.prepar
eStatement("SELECT * FROM users");
    ResultSet resultSet = preparedStatement.executeQuery();
    while (resultSet.next()) {
        System.out.println(resultSet.getString("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

With JdbcTemplate:

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
List<String> users = jdbcTemplate.queryForList("SELECT name
```

```
FROM users", String.class);
users.forEach(System.out::println);
```

Basic Configuration of Spring JDBC

We'll configure Spring JDBC to interact with the database and perform some basic queries.

---

1. Configure JdbcTemplate

What is JdbcTemplate?

`JdbcTemplate` is the core class in Spring JDBC for executing SQL queries, update statements, and stored procedures.

Configuration Steps

We need to configure `JdbcTemplate` as a bean so that it can be autowired wherever needed.

Code Example: JdbcTemplate Configuration

Create a configuration class in `src/main/java/com/example/demo/config`:

```
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.sql.DataSource;

@Configuration
public class JdbcConfig {

    @Bean
```

```
    public JdbcTemplate jdbcTemplate(DataSource dataSource)
{
        return new JdbcTemplate(dataSource);
    }
}
```

- The `DataSource` bean is automatically provided by Spring Boot based on the `application.properties` file.
- The `JdbcTemplate` bean will now be available throughout the application.

Write a DAO Class

What is a DAO Class?

A DAO (Data Access Object) class encapsulates database access logic.

Code Example: EmployeeDAO

Create a DAO class in `src/main/java/com/example/demo/dao`:

```
package com.example.demo.dao;

import com.example.demo.model.Employee;
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public class EmployeeDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;
```

```
    // Insert a new employee
    public int save(Employee employee) {
        String sql = "INSERT INTO employees (name, departme
nt, salary) VALUES (?, ?, ?)";
        return jdbcTemplate.update(sql, employee.getName(),
employee.getDepartment(), employee.getSalary());
    }

    // Retrieve all employees
    public List<Employee> findAll() {
        String sql = "SELECT * FROM employees";
        return jdbcTemplate.query(sql, (rs, rowNum) ->
                new Employee(rs.getInt("id"), rs.getString
("name"),
                        rs.getString("department"), rs.getD
ouble("salary")));
    }

    // Delete an employee by ID
    public int deleteById(int id) {
        String sql = "DELETE FROM employees WHERE id = ?";
        return jdbcTemplate.update(sql, id);
    }
}
```

Create a Model Class

What is a Model Class?

It represents the data structure of the database entity.

Code Example: Employee Model

Create the `Employee` class in `src/main/java/com/example/demo/model`:

```
package com.example.demo.model;
```

```
public class Employee {

    private int id;
    private String name;
    private String department;
    private double salary;

    public Employee() {
    }

    public Employee(int id, String name, String department,
double salary) {
        this.id = id;
        this.name = name;
        this.department = department;
        this.salary = salary;
    }

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDepartment() {
        return department;
```

```
    }

    public void setDepartment(String department) {
        this.department = department;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

Writing Advanced Queries and Completing Full CRUD Operations

In this step, we will implement advanced queries and complete all CRUD (Create, Read, Update, Delete) operations, incorporating best practices.

---

1. Advanced Queries Using JdbcTemplate

Using RowMapper for Complex Mapping

A `RowMapper` is used to map rows from a database result set to Java objects.

Code Example: Custom RowMapper

Create a `RowMapper` for the `Employee` class in `src/main/java/com/example/demo/mapper`:

```
package com.example.demo.mapper;

import com.example.demo.model.Employee;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
```

```
import java.sql.SQLException;

public class EmployeeRowMapper implements RowMapper<Employe
e> {
    @Override
    public Employee mapRow(ResultSet rs, int rowNum) throws
SQLException {
        Employee employee = new Employee();
        employee.setId(rs.getInt("id"));
        employee.setName(rs.getString("name"));
        employee.setDepartment(rs.getString("department"));
        employee.setSalary(rs.getDouble("salary"));
        return employee;
    }
}
```

Integrating RowMapper in DAO

Update the `EmployeeDAO` class to use the `EmployeeRowMapper`.

```
package com.example.demo.dao;

import com.example.demo.mapper.EmployeeRowMapper;
import com.example.demo.model.Employee;
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public class EmployeeDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;
```

```
    // Insert
    public int save(Employee employee) {
        String sql = "INSERT INTO employees (name, departme
nt, salary) VALUES (?, ?, ?)";
        return jdbcTemplate.update(sql, employee.getName(),
employee.getDepartment(), employee.getSalary());
    }

    // Find by ID
    public Employee findById(int id) {
        String sql = "SELECT * FROM employees WHERE id =
?";
        return jdbcTemplate.queryForObject(sql, new Employe
eRowMapper(), id);
    }

    // Find all
    public List<Employee> findAll() {
        String sql = "SELECT * FROM employees";
        return jdbcTemplate.query(sql, new EmployeeRowMappe
r());
    }

    // Update
    public int update(Employee employee) {
        String sql = "UPDATE employees SET name = ?, depart
ment = ?, salary = ? WHERE id = ?";
        return jdbcTemplate.update(sql, employee.getName(),
employee.getDepartment(), employee.getSalary(), employee.ge
tId());
    }

    // Delete by ID
    public int deleteById(int id) {
        String sql = "DELETE FROM employees WHERE id = ?";
        return jdbcTemplate.update(sql, id);
    }
```

```
    // Find by Department
    public List<Employee> findByDepartment(String departmen
t) {
        String sql = "SELECT * FROM employees WHERE departm
ent = ?";
        return jdbcTemplate.query(sql, new EmployeeRowMappe
r(), department);
    }

    // Count total employees
    public int count() {
        String sql = "SELECT COUNT(*) FROM employees";
        return jdbcTemplate.queryForObject(sql, Integer.cla
ss);
    }
}
```

2. Testing Full CRUD Operations

Code Example: Service Layer for Testing

Create a service to test the complete CRUD operations.

```
package com.example.demo.service;

import com.example.demo.dao.EmployeeDAO;
import com.example.demo.model.Employee;
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import java.util.List;

@Service
```

```
public class EmployeeService {

    @Autowired
    private EmployeeDAO employeeDAO;

    @PostConstruct
    public void testCrudOperations() {
        // 1. Insert a new employee
        Employee emp1 = new Employee(0, "Diana", "Finance",
60000);
        employeeDAO.save(emp1);
        System.out.println("Employee inserted: " + emp1.get
Name());

        // 2. Retrieve all employees
        List<Employee> employees = employeeDAO.findAll();
        System.out.println("All employees: ");
        employees.forEach(System.out::println);

        // 3. Retrieve an employee by ID
        Employee employee = employeeDAO.findById(2);
        System.out.println("Employee with ID 2: " + employe
e);

        // 4. Update an employee
        employee.setSalary(70000);
        employeeDAO.update(employee);
        System.out.println("Updated employee: " + employe
e);

        // 5. Find employees by department
        List<Employee> financeEmployees = employeeDAO.findB
yDepartment("Finance");
        System.out.println("Employees in Finance: ");
        financeEmployees.forEach(System.out::println);

        // 6. Count total employees
        int count = employeeDAO.count();
```

```
        System.out.println("Total employees: " + count);

        // 7. Delete an employee
        employeeDAO.deleteById(1);
        System.out.println("Employee with ID 1 deleted.");
    }
}
```

Explanation of Advanced Features

Parameterized Queries

- Protects against SQL injection.
- Ensures safe handling of user input.

RowMapper Benefits

- Simplifies mapping of SQL result sets to Java objects.
- Encourages code reuse.

Batch Processing (Advanced)

For inserting or updating multiple records efficiently, we use batch processing.

Code Example: Batch Insert

Add a method in `EmployeeDAO` for batch insert:

```
public int[] batchInsert(List<Employee> employees) {
    String sql = "INSERT INTO employees (name, department,
salary) VALUES (?, ?, ?)";
    return jdbcTemplate.batchUpdate(sql, employees, employe
es.size(),
        (ps, employee) -> {
            ps.setString(1, employee.getName());
            ps.setString(2, employee.getDepartment());
            ps.setDouble(3, employee.getSalary());
```

```
        });
}
```

Transaction Management (Advanced)

Spring handles transactions with the `@Transactional` annotation.

Code Example: Transactional Service

Update the service to demonstrate transactional operations:

```
package com.example.demo.service;

import com.example.demo.dao.EmployeeDAO;
import com.example.demo.model.Employee;
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transacti
onal;

@Service
public class EmployeeTransactionalService {

    @Autowired
    private EmployeeDAO employeeDAO;

    @Transactional
    public void performTransaction(Employee emp1, Employee
emp2) {
        employeeDAO.save(emp1);
        // Simulate an error
        if (emp2.getSalary() > 100000) {
            throw new RuntimeException("Salary too high");
        }
        employeeDAO.save(emp2);
    }
```

```
}
```