### Introduction to Spring JDBC

Spring JDBC is a part of the Spring Framework that provides a simpler approach to interact with relational databases using Java's JDBC (Java Database Connectivity) API. While JDBC provides a way to connect and interact with databases, it requires developers to handle a lot of boilerplate code for tasks like managing connections, handling exceptions, and cleaning up resources. Spring JDBC reduces this complexity by providing abstractions that handle these repetitive tasks, allowing developers to focus on business logic.

### Key Features of Spring JDBC:

1. \*\*Simplified Exception Handling\*\*: Converts SQL exceptions (like `SQLException`) into a consistent hierarchy of unchecked exceptions (`DataAccessException`), making error handling more manageable.

2. \*\*Resource Management\*\*: Automatically manages the lifecycle of database resources such as connections, statements, and result sets, ensuring they are closed properly.

3. \*\*JdbcTemplate\*\*: The core API that simplifies database interactions by reducing boilerplate code for common tasks such as querying, updating, and deleting data.

4. \*\*RowMapper\*\*: Simplifies the conversion of rows in a `ResultSet` into Java objects.

5. \*\*Named Parameters\*\*: Provides `NamedParameterJdbcTemplate` to improve the readability and maintainability of SQL queries by using named parameters instead of positional placeholders (`?`).

### Basic Components of Spring JDBC

#### 1. \*\*JdbcTemplate\*\*

`JdbcTemplate` is the central class in Spring JDBC. It provides various methods to execute SQL queries and manage the database lifecycle, including query execution, updates, and batch operations.

- \*\*Common Methods\*\*:

- `query()`: For executing SQL queries that return a list of objects.

- `queryForObject()`: To retrieve a single result from a query.

- `update()`: For executing SQL `INSERT`, `UPDATE`, or `DELETE` operations.

- `batchUpdate()`: To execute multiple SQL statements in a batch.

\*\*Example\*\*:

```java

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Repository;

@Repository

public class UserRepository {

private final JdbcTemplate jdbcTemplate;

public UserRepository(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public List<User> findAllUsers() {

String sql = "SELECT \* FROM users";

return jdbcTemplate.query(sql, (rs, rowNum) ->

new User(rs.getInt("id"), rs.getString("name"), rs.getString("email"))

);

}

public int insertUser(User user) {

String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

return jdbcTemplate.update(sql, user.getName(), user.getEmail());

}

}

```

- `query()`: Retrieves all rows from the `users` table.

- `update()`: Inserts a new user into the `users` table.

---

#### 2. \*\*RowMapper\*\*

`RowMapper` is used to map a row from a `ResultSet` to a Java object. It abstracts away the need to manually retrieve values from the `ResultSet`.

\*\*Example\*\*:

```java

import org.springframework.jdbc.core.RowMapper;

public class UserRowMapper implements RowMapper<User> {

@Override

public User mapRow(ResultSet rs, int rowNum) throws SQLException {

return new User(rs.getInt("id"), rs.getString("name"), rs.getString("email"));

}

}

```

Here, the `UserRowMapper` maps each row of the result set to a `User` object.

---

#### 3. \*\*NamedParameterJdbcTemplate\*\*

`NamedParameterJdbcTemplate` allows the use of named parameters in SQL queries, rather than positional placeholders (`?`). This enhances readability and maintainability, especially when dealing with multiple parameters.

\*\*Example\*\*:

```java

import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

import org.springframework.stereotype.Repository;

import java.util.HashMap;

import java.util.Map;

@Repository

public class UserRepository {

private final NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public UserRepository(NamedParameterJdbcTemplate namedParameterJdbcTemplate) {

this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;

}

public int createUser(User user) {

String sql = "INSERT INTO users (name, email) VALUES (:name, :email)";

Map<String, Object> params = new HashMap<>();

params.put("name", user.getName());

params.put("email", user.getEmail());

return namedParameterJdbcTemplate.update(sql, params);

}

}

```

In this example, the named parameters `:name` and `:email` make the query easier to understand than using positional parameters.

---

### Advantages of Spring JDBC

1. \*\*Reduced Boilerplate Code\*\*: Developers don’t need to manually manage database resources (connections, statements, result sets), reducing the chances of resource leaks.

2. \*\*Simplified Exception Handling\*\*: Spring converts JDBC's checked exceptions into unchecked exceptions using the `DataAccessException` hierarchy, simplifying error handling.

3. \*\*Flexibility with SQL\*\*: Spring JDBC allows you to write raw SQL queries, giving you full control over the database interactions.

4. \*\*Reusable Components\*\*: You can reuse components like `JdbcTemplate` and `RowMapper` across different parts of the application.

5. \*\*Transaction Management\*\*: Spring provides support for managing transactions declaratively or programmatically.

### Example: Full Spring JDBC Application

#### 1. \*\*Database Setup\*\*

Let’s create a simple `users` table for demonstration purposes:

```sql

CREATE TABLE users (

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(50),

email VARCHAR(50)

);

```

#### 2. \*\*Spring Configuration\*\*

- \*\*`application.properties`\*\*:

```properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

```

- \*\*`pom.xml`\*\* (Maven dependencies):

```xml

<dependency>

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

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

```

#### 3. \*\*User Model\*\*:

```java

public class User {

private int id;

private String name;

private String email;

// Constructor, Getters, Setters

}

```

#### 4. \*\*User Repository\*\*:

```java

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public class UserRepository {

private final JdbcTemplate jdbcTemplate;

public UserRepository(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public List<User> findAllUsers() {

String sql = "SELECT \* FROM users";

return jdbcTemplate.query(sql, (rs, rowNum) ->

new User(rs.getInt("id"), rs.getString("name"), rs.getString("email"))

);

}

public int insertUser(User user) {

String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

return jdbcTemplate.update(sql, user.getName(), user.getEmail());

}

}

```

#### 5. \*\*Controller\*\*:

```java

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

import java.util.List;

@RestController

@RequestMapping("/users")

public class UserController {

private final UserRepository userRepository;

public UserController(UserRepository userRepository) {

this.userRepository = userRepository;

}

@GetMapping

public List<User> getAllUsers() {

return userRepository.findAllUsers();

}

@PostMapping

public String createUser(@RequestBody User user) {

userRepository.insertUser(user);

return "User created successfully";

}

}

```

### Conclusion

Spring JDBC provides a lightweight framework for interacting with relational databases, reducing the boilerplate code involved in resource management and exception handling. By using `JdbcTemplate`, `RowMapper`, and `NamedParameterJdbcTemplate`, developers can perform complex database operations with minimal effort while still leveraging the full power of SQL.