### Spring JDBC Template

\*\*`JdbcTemplate`\*\* is the core class provided by Spring JDBC to simplify database interaction. It is part of the Spring Framework's abstraction layer for Java Database Connectivity (JDBC), eliminating much of the boilerplate code required when using plain JDBC.

The `JdbcTemplate` class helps developers to:

- Execute SQL queries.

- Perform database updates (insert, update, delete).

- Retrieve data from a result set.

- Manage resources like connections, statements, and result sets automatically.

- Handle exceptions in a consistent manner by converting `SQLException` into `DataAccessException`.

### Key Features of `JdbcTemplate`:

1. \*\*Connection Handling\*\*: Manages the opening and closing of database connections automatically.

2. \*\*Exception Handling\*\*: Converts JDBC exceptions into Spring's `DataAccessException`, simplifying exception management.

3. \*\*Query Execution\*\*: Provides methods for executing SQL queries (`SELECT`, `INSERT`, `UPDATE`, `DELETE`).

4. \*\*Row Mapping\*\*: Maps rows of a `ResultSet` to Java objects via the `RowMapper` interface.

5. \*\*Batch Operations\*\*: Allows batch updates for improving performance in bulk operations.

### Commonly Used Methods in `JdbcTemplate`

- `query()`: Executes SQL queries and returns a result set.

- `queryForObject()`: Executes a query that returns a single result (a scalar value or a single row).

- `update()`: Executes an `INSERT`, `UPDATE`, or `DELETE` statement.

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

- `execute()`: Executes a generic SQL statement, such as DDL commands (`CREATE TABLE`, etc.).

---

### Example: Using `JdbcTemplate`

Let’s look at some practical examples of how to use `JdbcTemplate` in a Spring application.

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

For this example, let's assume we are working with a simple `users` table in a MySQL database:

```sql

CREATE TABLE users (

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(50),

email VARCHAR(50)

);

```

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

To use Spring JDBC, you need to include the necessary dependencies in your `pom.xml` (for Maven projects).

\*\*`pom.xml`\*\*:

```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>

```

Add your database configuration in the `application.properties` file:

\*\*`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

```

---

### 3. \*\*Basic CRUD Operations Using `JdbcTemplate`\*\*

#### \*\*Inserting Data\*\*

The `update()` method is commonly used for inserting, updating, and deleting records. Here’s how you can insert a new record into the `users` table.

```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 int saveUser(String name, String email) {

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

return jdbcTemplate.update(sql, name, email);

}

}

```

In this example, `jdbcTemplate.update()` is used to insert a new user into the `users` table. The placeholders `?` are replaced with the `name` and `email` arguments passed into the method.

---

#### \*\*Querying Data\*\*

To retrieve data, you can use the `query()` method in combination with `RowMapper` to map the `ResultSet` to Java objects.

```java

import org.springframework.jdbc.core.RowMapper;

import org.springframework.stereotype.Repository;

import java.sql.ResultSet;

import java.sql.SQLException;

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, new UserRowMapper());

}

}

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"));

}

}

class User {

private int id;

private String name;

private String email;

public User(int id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters and Setters

}

```

Here, the `query()` method fetches all rows from the `users` table and maps them to the `User` object using the `UserRowMapper`.

---

#### \*\*Retrieving a Single Value\*\*

If you need to retrieve a single value or a single row, `queryForObject()` is used. For example, retrieving a user’s email by ID:

```java

public String findEmailById(int id) {

String sql = "SELECT email FROM users WHERE id = ?";

return jdbcTemplate.queryForObject(sql, String.class, id);

}

```

Here, `queryForObject()` returns a single value (`String` in this case) representing the user’s email. The `id` is used as the query parameter.

---

#### \*\*Updating Data\*\*

To update existing records in the database, use the `update()` method again:

```java

public int updateUserEmail(int id, String newEmail) {

String sql = "UPDATE users SET email = ? WHERE id = ?";

return jdbcTemplate.update(sql, newEmail, id);

}

```

This SQL statement updates the email of a specific user based on the `id` provided.

---

#### \*\*Deleting Data\*\*

Deleting records is also handled with the `update()` method:

```java

public int deleteUserById(int id) {

String sql = "DELETE FROM users WHERE id = ?";

return jdbcTemplate.update(sql, id);

}

```

This method deletes a user from the `users` table based on the user’s ID.

---

### 4. \*\*Batch Processing\*\*

If you need to perform bulk inserts, updates, or deletes, `batchUpdate()` can be used to execute a batch of SQL statements.

```java

public int[] batchInsertUsers(List<User> users) {

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

return jdbcTemplate.batchUpdate(sql, users, users.size(), (ps, user) -> {

ps.setString(1, user.getName());

ps.setString(2, user.getEmail());

});

}

```

In this example, `batchUpdate()` is used to insert multiple users into the `users` table in a single batch.

---

### 5. \*\*Executing SQL DDL Statements\*\*

You can also use `JdbcTemplate` to execute Data Definition Language (DDL) statements, such as creating tables.

```java

public void createTable() {

String sql = "CREATE TABLE IF NOT EXISTS users (id INT PRIMARY KEY AUTO\_INCREMENT, name VARCHAR(50), email VARCHAR(50))";

jdbcTemplate.execute(sql);

}

```

The `execute()` method runs any SQL statement, such as `CREATE`, `DROP`, `ALTER`, etc.

---

### Conclusion

The `JdbcTemplate` class in Spring provides a simplified way to interact with relational databases. By reducing the amount of boilerplate code needed for resource management, exception handling, and SQL execution, `JdbcTemplate` allows developers to focus on writing business logic rather than managing database connections and SQL execution details. Whether you're executing simple queries or performing batch updates, `JdbcTemplate` is a versatile and powerful tool in the Spring ecosystem.