Search...

Spring - JDBC Template

Last Updated : 23 Jul, 2025

In this article, we will discuss the **Spring JDBC Template** and how to configure the **JDBC Template** to execute queries. Spring JDBC Template provides a fluent API that improves code simplicity and readability, and the JDBC Template is used to connect to the database and execute SQL Queries.

What is JDBC?

JDBC (Java Database Connectivity) is an application programming interface **(API)** that defines how a client may access a database. It is a data access technology used for Java database connectivity. It provides methods to query and update data in a database and is oriented toward relational databases. JDBC offers a natural Java interface for working with **SQL**. JDBC is needed to provide a "Pure Java" solution for application development. JDBC API uses JDBC drivers to connect with the database.

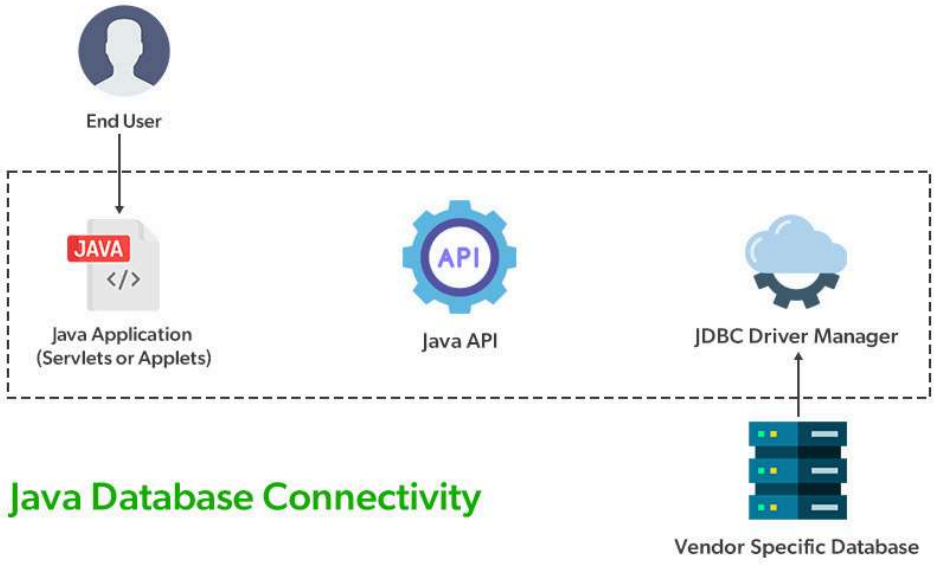
Types of JDBC Drivers

There are 4 types of JDBC Drivers.

1. **JDBC-ODBC Bridge Driver:** Connects to databases via an ODBC driver. It is platform-dependent and not recommended for production use.
2. **Native API Driver:** Uses the database's native API for communication. It is partially written in Java and platform-dependent.
3. **Network Protocol Driver**: A fully Java-based driver that uses middleware to communicate with the database.

4. **Thin Driver:** A fully Java-based driver that communicates directly with the database using its network protocol. It is the most commonly used driver.

The image below demonstrates how a Java application interacts with a database via the Java API and JDBC Driver Manager.



Advantages of JDBC API

- JDBC API helps retrieve and manipulate data from the database efficiently.
- It supports query and stored procedures.
- Almost any database for which **ODBC** driver is installed can be accessed.

Disadvantages of JDBC API

- Writing a lot of codes before and after executing the query, such as creating connection, creating a statement, closing result-set, closing connection, etc.
- Writing exception handling code on the database logic.
- Repetition of these codes from one to another database logic is time-consuming.

These problems of **JDBC API** are eliminated by **Spring JDBC-Template**. It provides methods to write the queries directly that saves a lot of time and effort.

Data Access using Spring JDBC Template

There are a number of options for selecting an approach to form the basis for your **JDBC** database access. Spring framework provides the following approaches for **JDBC** database access:

1. JdbcTemplate
2. NamedParameterJdbcTemplate
3. SimpleJdbcTemplate
4. SimpleJdbcInsert and SimpleJdbcCall

1. JDBC Template

JdbcTemplate is a central class in the **JDBC** core package that simplifies the use of **JDBC** and helps to avoid common errors. It internally uses **JDBC API** and eliminates a lot of problems with **JDBC API**. It executes SQL queries or updates, initiating iteration over ResultSets and catching **JDBC** exceptions and translating them to the generic. It executes core **JDBC** workflow, leaving application code to provide SQL and extract results. It handles the exception and provides the informative exception messages with the help of exception classes defined in the **org.springframework.dao** package.

The common methods of spring JdbcTemplate class:

Methods	Description
public int update(String query)	Used to insert, update and delete records.
public int update(String query, Object... args)	Used to insert, update and delete records using **PreparedStatement** using

Methods	Description
	given arguments.
public T execute(String sql, PreparedStatementCallback action)	Executes the query by using **PreparedStatementCallback.**
public void execute(String query)	Used to execute **DDL** query.
public T query(String sql, ResultSetExtractor result)	Used to fetch records using **ResultSetExtractor**.

JDBC Template Queries

1. Counting Records:

Basic query to count students stored in the database using **JdbcTemplate**.

```
int count = jdbcTemplate.queryForObject(
    "SELECT COUNT(*) FROM STUDENT", Integer.class);
```

2. Inserting a Record:

Basic query to insert elements into the database.

```
public int addStudent(int id, String name, String country) {
    return jdbcTemplate.update(
        "INSERT INTO STUDENT (id, name, country) VALUES (?, ?, ?)",
        id, name, country);
}
```

Note: The standard syntax of providing parameters is using the **"?"** character.

3. Fetching Records:

Basic query to fetch records from the database.

```
public List<Student> getAllStudents() {
    return jdbcTemplate.query(
        "SELECT * FROM STUDENT",
        new BeanPropertyRowMapper<>(Student.class));
}
```

Implementation: Spring JdbcTemplate

Let's walk through the steps to configure and use JdbcTemplate in a Spring application.

Step 1: Configure the Data Source

We start by configuring the DataSource and JdbcTemplate in a Spring configuration class.

SpringJdbcConfig.java:

```
@Configuration
@ComponentScan("com.exploit.jdbc")
public class SpringJdbcConfig {

    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        // Correct MySQL driver class for MySQL 8.x
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

        // Correct URL with default MySQL port and timezone configuration
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc?
serverTimezone=UTC");

        // Use environment variables or externalized configuration for
credentials
        // Replace with actual username
        dataSource.setUsername("user");
        // Replace with actual password
        dataSource.setPassword("password");

        return dataSource;
    }
}
```

Step 2: Create the Model Class

Define a model class to represent the database table.

Student.java:

```
// Java Program to Illustrate Student Class

package com.exploit.org;

import lombok.Data;
import lombok.NoArgsConstructor;

// Lombok annotations to generate getters, setters, toString, equals, and
hashCode
@Data
@NoArgsConstructor
public class Student {

    // Class data members
    private Integer id;
    private String name;
    private Integer age;

    // Parameterized Constructor
    public Student(Integer id, String name, Integer age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
}
```

Step 3: Create the DAO Interface

Define a DAO interface for database operations.

StudentDAO.java:

```
// Java Program to Illustrate StudentDAO Class

package com.exploit.org;

// Importing required classes
import java.util.List;
import org.springframework.dao.DataAccessException;
```

```
// Interface
public interface StudentDAO {

    // Create: Insert a new student record
    void createStudent(Student student) throws DataAccessException;

    // Read: Retrieve a student by ID
    Student getStudentById(Integer id) throws DataAccessException;

    // Read: List all students
    List<Student> listStudents() throws DataAccessException;

    // Update: Update an existing student record
    void updateStudent(Student student) throws DataAccessException;

    // Delete: Delete a student by ID
    void deleteStudent(Integer id) throws DataAccessException;
}
```

Step 4: Implement the DAO Interface

Implement the DAO interface using JdbcTemplate.

```
package com.exploit.org;

// Importing required classes
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

@Repository
public class StudentDAOImpl implements StudentDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    // RowMapper to map result set to Student object
    private final RowMapper<Student> rowMapper = (rs, rowNum) -> {
        Student student = new Student();
        student.setId(rs.getInt("id"));
        student.setName(rs.getString("name"));
        student.setAge(rs.getInt("age"));
        return student;
    };

    @Override
    public void createStudent(Student student) throws DataAccessException {
```

```
        String sql = "INSERT INTO Student (id, name, age) VALUES (?, ?, ?)";
    }

    @Override
    public Student getStudentById(Integer id) throws DataAccessException {
        String sql = "SELECT * FROM Student WHERE id = ?";
        return jdbcTemplate.queryForObject(sql, rowMapper, id);
    }

    @Override
    public List<Student> listStudents() throws DataAccessException {
        String sql = "SELECT * FROM Student";
        return jdbcTemplate.query(sql, rowMapper);
    }

    @Override
    public void updateStudent(Student student) throws DataAccessException {
        String sql = "UPDATE Student SET name = ?, age = ? WHERE id = ?";
        jdbcTemplate.update(sql, student.getName(), student.getAge(),
student.getId());
    }

    @Override
    public void deleteStudent(Integer id) throws DataAccessException {
        String sql = "DELETE FROM Student WHERE id = ?";
        jdbcTemplate.update(sql, id);
    }
}
```

Step 5: Add Maven Dependencies

Add the required dependencies in the pom.xml file.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
</dependencies>
```

In the above pom.xml file we have used "spring-boot-starter-jdbc" dependency for implementing Java Database Connectivity in our

application. Also we have used "mysql-connector-java" dependency to connect to the MySQL database and execute SQL queries.

Step 6: Implementing the StudentDAO Interface with jdbcTemplate

Below is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface **StudentDAO.**

StudentJDBCTemplate.java:

```
package com.exploit.org;

// Importing required classes
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import org.springframework.dao.DataAccessException;

@Repository
public class StudentJDBCTemplate implements StudentDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    // RowMapper to map result set to Student object
    private final StudentMapper studentMapper = new StudentMapper();

    @Override
    public List<Student> listStudents() throws DataAccessException {
        String SQL = "SELECT * FROM Student";
        return jdbcTemplate.query(SQL, studentMapper);
    }

    @Override
    public void createStudent(Student student) throws DataAccessException {
        String SQL = "INSERT INTO Student (id, name, age) VALUES (?, ?, ?)";
        jdbcTemplate.update(SQL, student.getId(), student.getName(),
student.getAge());
    }

    @Override
    public Student getStudentById(Integer id) throws DataAccessException {
        String SQL = "SELECT * FROM Student WHERE id = ?";
        return jdbcTemplate.queryForObject(SQL, studentMapper, id);
    }

    @Override
    public void updateStudent(Student student) throws DataAccessException {
```

```
        String SQL = "UPDATE Student SET name = ?, age = ? WHERE id = ?";
        jdbcTemplate.update(SQL, student.getName(), student.getAge(),
student.getId());
    }

    @Override
    public void deleteStudent(Integer id) throws DataAccessException {
        String SQL = "DELETE FROM Student WHERE id = ?";
        jdbcTemplate.update(SQL, id);
    }
}
```

Note: StudentJDBCTemplate implements **StudentDAO** and provides database operations using **JdbcTemplate**. If you are using **StudentJDBCTemplate**, you do not need a separate **StudentDAOImpl** class.