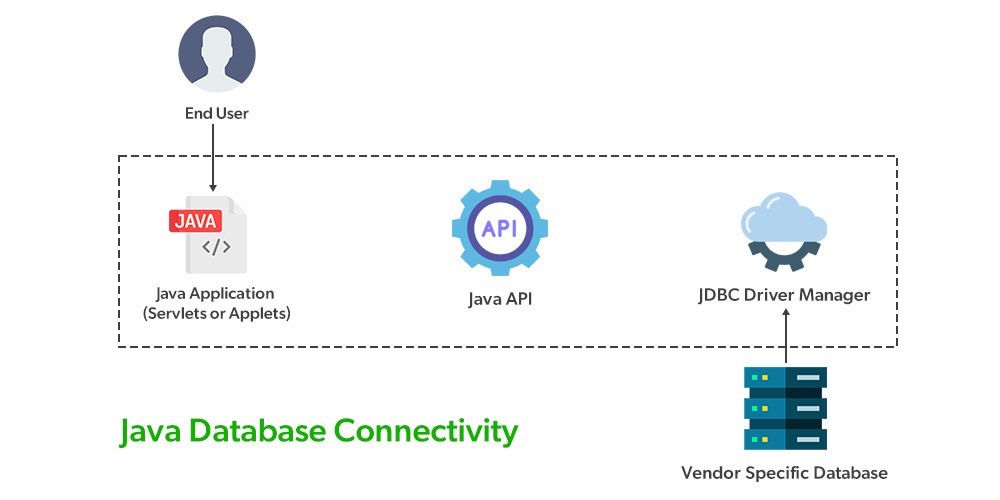
In this article, we will discuss about **Spring JDBC Template**and how to configure the **JDBC Template**to execute queries. Spring JDBC Template provides a fluent API that improves code simplicity and readability and the JDBC Template is used to connect to the database and execute SQL Queries.

[JDBC](https://www.geeksforgeeks.org/introduction-to-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
2. Native API Driver (partially Java driver)
3. Network Protocol Driver (fully Java driver)
4. Thin Driver (fully java driver)



**Advantages of JDBC API**

* Automatically creates the **XML**format of data from the database.
* 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 acentral 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 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**

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

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

And here’s a simple **INSERT**:

public int addStudent(int id)   
{  
 return jdbcTemplate.update("INSERT INTO STUDENT VALUES (?, ?, ?)", id, "megan", "India");  
}

The standard syntax of providing parameters is using the **“?”**character.

**Implementation:** Spring JdbcTemplate

*We start with some simple configurations of the data source. We’ll use a*[*MySQL database*](https://www.geeksforgeeks.org/mysql-database-files/)

**Example:**

* Java

|  |
| --- |
| //Java program for spring datasource configuration  @Configuration  @ComponentScan("com.exploit.jdbc")  **public** **class** SpringJdbcConfig {      @Bean **public** DataSource mysqlDataSource()      {          DriverManagerDataSource dataSource              = **new** DriverManagerDataSource();          dataSource.setDriverClassName(              "com.mysql.jdbc.Driver");          dataSource.setUrl(              "jdbc:<mysql://localhost:8800/springjdbc>");          dataSource.setUsername("user");          dataSource.setPassword("password");    **return** dataSource;      }  } |

In the above code, we have used @Configuration annotation which means for Spring context, the class file is configured in the class path. The @ComponentScan(“com.exploit.jdbc”) annotation will scan the package (inside the parameter) for classes.

**A. File: Student.java (Model Class)**

* Java

|  |
| --- |
| // Java Program to Illustrate Student Class    **package** com.exploit.org;    // Class  **public** **class** Student {        // Class data members  **private** Integer age;  **private** String name;  **private** Integer id;        // Constructor  **public** Student() {}        // Setters and Getters  **public** **void** setAge(Integer age) { **this**.age = age; }  **public** Integer getAge() { **return** age; }  **public** **void** setName(String name) { **this**.name = name; }  **public** String getName() { **return** name; }  **public** **void** setId(Integer id) { **this**.id = id; }  **public** Integer getId() { **return** id; }  } |

**B. File: StudentDAO.java (DAO Class)**

Below is the implementation of the Data Access Object interface file StudentDAO.java.

**Example:**

* Java

|  |
| --- |
| // Java Program to Illustrate StudentDAO Class    **package** com.exploit.org;    // importing required classes  **import** java.util.List;  **import** javax.sql.DataSource;    // Class  **public** **interface** StudentDAO {        // Used to initialize database resources      // ie. connection  **public** **void** setDataSource(DataSource ds);        // Used to list down all the records      // from the Student table  **public** List<Student> listStudents();  } |

In the above code, we created DAO Class for mapping the model class (Student.Java) with the database table using SQL queries.

**C.** File: [Maven Dependency](https://www.geeksforgeeks.org/introduction-apache-maven-build-automation-tool-java-projects/)

Dependency is used in the **pom.xml** file.

**Example:**

* 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**> |

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.

**D. File: StudentJDBCTemplate.java**

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

**Example:**

* Java

|  |
| --- |
| // Java Program Illustrating Implementation  // of StudentDAO Class    **package** com.exploit.org;    // Importing required classes  **import** java.util.List;  **import** javax.sql.DataSource;  **import** org.springframework.jdbc.core.JdbcTemplate;    // Class  // Implementing StudentDAO Class  **public** **class** StudentJDBCTemp **implements** StudentDAO {        // Class data members  **private** DataSource dataSource;  **private** JdbcTemplate jdbcTemplateObject;        // Method 1  **public** **void** setDataSource(DataSource dataSource)      {          // This keyword refers to current instance itself  **this**.dataSource = dataSource;  **this**.jdbcTemplateObject              = **new** JdbcTemplate(dataSource);      }        // Method 2  **public** List<Student> listStudents()      {          // Custom SQL query          String SQL = "select \* from Student";          List<Student> students = jdbcTemplateObject.query(              SQL, **new** StudentMapper());    **return** students;      }  } |