Spring DAO

Spring=======db

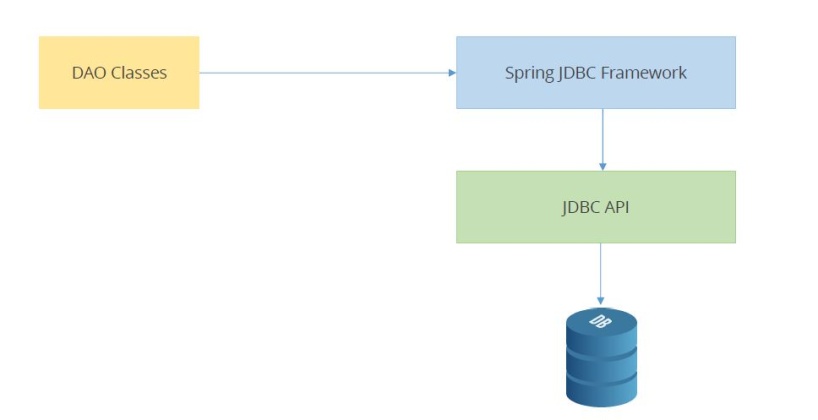
**Spring JdbcTemplate**

* JDBC produces a lot of boiler plate code, such as opening/closing a connection to a database, handling sql exceptions etc. It makes the code extremely cumbersome and difficult to read.
* Implementing JDBC in the [Spring Framework](https://www.journaldev.com/16922/spring-framework) takes care of working with many low-level operations (opening/closing connections, executing SQL queries, etc.).
* Thanks to this, when working with the database in the Spring Framework, we only need to define the connection parameters from the database and register the SQL query, the rest of the work for us is performed by Spring.
* JDBC in Spring has several classes (several approaches) for interacting with the database. The most common of these is using the JdbcTemplatee class. This is the base class that manages the processing of all events and database connections.
* The JdbcTemplate class executes SQL queries, iterates over the  ResultSett, and retrieves the called values, updates the instructions and procedure calls, “catches” the exceptions, and translates them into the exceptions defined in the org.springframework.dao package.
* Instances of the JdbcTemplate class are thread-safe. This means that by configuring a single instance of the JdbcTemplate class, we can then use it for several [DAO](https://www.journaldev.com/16813/dao-design-pattern) objects.
* When using JdbcTemplate, most often, it is configured in the Spring configuration file. After that, it is implemented using bean in DAO classes.

## JDBC vs. Spring JDBC

 the comparison between JDBC and Spring JDBC.

|  |  |  |
| --- | --- | --- |
| **Difference based on** | **JDBC** | **Spring JDBC** |
| |  |  |  | | --- | --- | --- | | Transaction Management Logic:(enable or disable transactions declaratively) | It doesn’t support | Spring JDBC supports | | Distribution based Transaction Management | It doesn’t support | Spring JDBC supports | | Annotation-based transaction management | It doesn’t support | Spring JDBC supports | | Resource Allocating Logic | Connection, Statement, and Prepared statement have to be allocated explicitly by the developer in JDBC. | Spring provides org.springframework.jdbcTemplateobject. It abstracts the Database Resource Allocation Logic. | | Resource Releasing Logic | In JDBC, the developer explicitly writes the logic to release the database resources. | Spring provides org.springframework.jdbc.core.jdbcTemplateobject. | | Exception Handling | JDBC throws java.sql.SQLException. There won’t be specific exceptions for specific problems. | Spring provides Fine Grind Exception Handling mechanism to deal with Database. | |  |  |



The DAO classes are shown connecting to the spring JDBC framework, which is an additional layer which then connects to the JDBC API and finally talks to the database.

## The Advantage of Spring JDBC API

The advantage of using Spring JDBC is that -

* It provides the capability to establish the connection and interact with the database.
* It simplifies the development of JDBC, Hibernate, JPA, and JDO.
* It provides multiple templates to interact with the database including the JdbcTemplate, HibernateTemplate, JDOTemplate, JPATemplate.

## Terminologies of Spring JDBC API (Connection Pooling)

The terminologies of Spring JDBC API  when a connection is opened and closed (when required) in an application, the object utilization becomes difficult and the cost of the implementation increases. In such cases, connection pooling is used.

It is a mechanism of pre-creating a group of database connections and keeping them in cache memory for use and reuse. It provides high performance and efficient resource management.

## Terminologies of Spring JDBC API (java.sql.DataSource)

This is an interface which is an object-oriented representation of the connection pooling. It is a connection factory for Java application. It is an alternative for driver manager and provides Database connections to the Java application.

The following connection pooling mechanisms provide its implementation classes -

* >Apache BasicDatasource
* >Spring DriverManagerDatasource and
* >C3pOComboPooleDatasource.

## Spring JDBC Template

Spring JDBC provides a class called “JdbcTemplate” to handle the common logic. JdbcTemplate is an abstraction of the JDBC technology.

The methods provided by JdbcTemplate are -

* For insert(), update(), and delete() methods of JDBC, Spring JdbcTemplate provides update() method
* For findByAccno(), findByBalance(), findAll(), rowCount() methods of JDBC, Spring Template provides query(), queryForInt(), queryObject(), queryForMap(), and queryForList()
* To handle batch updates, JdbcTemplate has convenient methods.

Spring JDBC Template Methods

The table below shows the JDBC template methods.

|  |  |
| --- | --- |
| public int update (String query) | public int update (String query) |
| public int update (String query) | inserts, updates, and deletes records using PreparedStatement using given arguments |
| public void execute (String query) | executes DDL query |
| public T execute (String SQL, PreparedStatementCallback action) | executes the query by using PreparedStatementcallback |
| public T query (String SQL, ResultSetExtractorrse) | fetches records using ResultSetExtractor |
| public List query (String SQL, RowMapperrse) | fetches records using RowMapper |

## Implementing Spring JDBC in an Application

To implement spring JDBC in an application, we have to -

1. Create database and table in the database
2. Create Java projects in eclipse
3. Create the Spring Bean class(POJO class)
4. Create a DAO class
5. Configure DataSource and the spring bean spring configuration XML file
6. Create the Spring Bean or Spring container either by XmlBeanFactory or ApplicationContext

Lets do the example

Data Access Object (DAO)

DAO stands for Data Access Object, which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and JDBC Template object.

Querying for an integer

String SQL = "select count(\*) from Student";

int rowCount = jdbcTemplateObject.queryForInt( SQL );

Querying for a long

String SQL = "select count(\*) from Student";

long rowCount = jdbcTemplateObject.queryForLong( SQL );

A simple query using a bind variable

String SQL = "select age from Student where id = ?";

int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});

Querying for a String

String SQL = "select name from Student where id = ?";

String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class);

Querying and returning an object

String SQL = "select \* from Student where id = ?";

Student student = jdbcTemplateObject.queryForObject(

SQL, new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

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

Student student = new Student();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Querying and returning multiple objects

String SQL = "select \* from Student";

List<Student> students = jdbcTemplateObject.query(

SQL, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

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

Student student = new Student();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Inserting a row into the table

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );

Updating a row into the table

String SQL = "update Student set name = ? where id = ?";

jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );

Deleting a row from the table

String SQL = "delete Student where id = ?";

jdbcTemplateObject.update( SQL, new Object[]{20} );

Executing DDL Statements

You can use the execute(..) method from jdbcTemplate to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table −

String SQL = "CREATE TABLE Student( " +

"ID INT NOT NULL AUTO\_INCREMENT, " +

"NAME VARCHAR(20) NOT NULL, " +

"AGE INT NOT NULL, " +

"PRIMARY KEY (ID));"

jdbcTemplateObject.execute( SQL );

//==================================

SQL Stored Procedure in Spring

The SimpleJdbcCall class can be used to call a stored procedure with IN and OUT parameters.

You can use this approach while working with either of the RDBMS like Apache Derby,

DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase.

To understand the approach, let us take our Student table which can be created in MySQL TEST database with the following DDL −

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Next, consider the following MySQL stored procedure, which takes student Id and returns the corresponding student's name and age using OUT parameters. So let us create this stored procedure in your TEST database using MySQL command prompt −

DELIMITER $$

DROP PROCEDURE IF EXISTS `TEST`.`getRecord` $$

CREATE PROCEDURE `TEST`.`getRecord` (

IN in\_id INTEGER,

OUT out\_name VARCHAR(20),

OUT out\_age INTEGER)

BEGIN

SELECT name, age

INTO out\_name, out\_age

FROM Student where id = in\_id;

END $$

DELIMITER ;

import java.util.List;

import javax.sql.DataSource;

public interface StudentDAO {

/\*\*

\* This is the method to be used to initialize

\* database resources ie. connection.

\*/

public void setDataSource(DataSource ds);

/\*\*

\* This is the method to be used to create

\* a record in the Student table.

\*/

public void create(String name, Integer age);

/\*\*

\* This is the method to be used to list down

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public Student getStudent(Integer id);

/\*\*

\* This is the method to be used to list down

\* all the records from the Student table.

\*/

public List<Student> listStudents();

}

Following is the content of the Student.java file

public class Student {

private Integer age;

private String name;

private Integer id;

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;

}

}

Following is the content of the StudentMapper.java file

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {

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

Student student = new Student();

student.setId(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Following is the implementation class file StudentJDBCTemplate.java for the defined DAO interface StudentDAO −

import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

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

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

import org.springframework.jdbc.core.simple.SimpleJdbcCall;

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private SimpleJdbcCall jdbcCall;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcCall = new SimpleJdbcCall(dataSource).withProcedureName("getRecord");

}

public void create(String name, Integer age) {

JdbcTemplate jdbcTemplateObject = new JdbcTemplate(dataSource);

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, name, age);

System.out.println("Created Record Name = " + name + " Age = " + age);

return;

}

public Student getStudent(Integer id) {

SqlParameterSource in = new MapSqlParameterSource().addValue("in\_id", id);

Map<String, Object> out = jdbcCall.execute(in);

Student student = new Student();

student.setId(id);

student.setName((String) out.get("out\_name"));

student.setAge((Integer) out.get("out\_age"));

return student;

}

public List<Student> listStudents() {

String SQL = "select \* from Student";

List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());

return students;

}

}

Few words about the above program: The code you write for the execution of the call involves creating an SqlParameterSource containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The execute method takes the IN parameters and returns a Map containing any out parameters keyed by the name as specified in the stored procedure. Now let us move with the main application file MainApp.java, which is as follows −

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.StudentJDBCTemplate;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

StudentJDBCTemplate studentJDBCTemplate =

(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

System.out.println("------Records Creation--------" );

studentJDBCTemplate.create("Zara", 11);

studentJDBCTemplate.create("Nuha", 2);

studentJDBCTemplate.create("Ayan", 15);

System.out.println("------Listing Multiple Records--------" );

List<Student> students = studentJDBCTemplate.listStudents();

for (Student record : students) {

System.out.print("ID : " + record.getId() );

System.out.print(", Name : " + record.getName() );

System.out.println(", Age : " + record.getAge());

}

System.out.println("----Listing Record with ID = 2 -----" );

Student student = studentJDBCTemplate.getStudent(2);

System.out.print("ID : " + student.getId() );

System.out.print(", Name : " + student.getName() );

System.out.println(", Age : " + student.getAge());

}

}

Following is the configuration file Beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

<!-- Initialization for data source -->

<bean id = "dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>

<property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>

<property name = "username" value = "root"/>

<property name = "password" value = "password"/>

</bean>

<!-- Definition for studentJDBCTemplate bean -->

<bean id = "studentJDBCTemplate"

class = "StudentJDBCTemplate">

<property name = "dataSource" ref = "dataSource" />

</bean>

</beans>

# NamedParameterJdbcTemplate Class

The org.springframework.jdbc.core.NamedParameterJdbcTemplate class is a template class with a basic set of JDBC operations, allowing the use of named parameters rather than traditional '?' placeholders. This class delegates to a wrapped JdbcTemplate once the substitution from named parameters to JDBC style '?' placeholders is done at execution time. It also allows to expand a list of values to the appropriate number of placeholders.

Interface Declaration

Following is the declaration for

org.springframework.jdbc.core.NamedParameterJdbcTemplate class −

public class NamedParameterJdbcTemplate extends Object implements NamedParameterJdbcOperations

Syntax

MapSqlParameterSource in = new MapSqlParameterSource();

in.addValue("id", id);

in.addValue("description", new SqlLobValue(description, new DefaultLobHandler()), Types.CLOB);

String SQL = "update Student set description = :description where id = :id";

NamedParameterJdbcTemplate jdbcTemplateObject = new NamedParameterJdbcTemplate(dataSource);

jdbcTemplateObject.update(SQL, in);

Where,

in − SqlParameterSource object to pass a parameter to update a query.

SqlLobValue − Object to represent an SQL BLOB/CLOB value parameter.

jdbcTemplateObject − NamedParameterJdbcTemplate object to update student object in the database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example

which will update a query. To write our example, let us have a working Eclipse IDE in place

and use the following steps to create a Spring application.

Step Description

1 Update the project Student created under chapter Spring JDBC - First Application.

2 Update the bean configuration and run the application as explained below.

Following is the content of the Data Access Object interface file StudentDAO.java.

package com;

import java.util.List;

import javax.sql.DataSource;

public interface StudentDAO {

/\*\*

\* This is the method to be used to initialize

\* database resources ie. connection.

\*/

public void setDataSource(DataSource ds);

/\*\*

\* This is the method to be used to update

\* a record into the Student table.

\*/

public void updateDescription(Integer id, String description);

}

Following is the content of the Student.java file.

package com;

public class Student {

private Integer age;

private String name;

private Integer id;

private String description;

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;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

}

Following is the content of the StudentMapper.java file.

package com;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {

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

Student student = new Student();

student.setId(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

student.setDescription(rs.getString("description"));

return student;

}

}

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

package com;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

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

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

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

import org.springframework.jdbc.core.simple.SimpleJdbcCall;

import org.springframework.jdbc.core.support.SqlLobValue;

import org.springframework.jdbc.support.lob.DefaultLobHandler;

import java.io.ByteArrayInputStream;

import java.sql.Types;

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplateObject;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

public void updateDescription(Integer id, String description) {

MapSqlParameterSource in = new MapSqlParameterSource();

in.addValue("id", id);

in.addValue("description", new SqlLobValue(

description, new DefaultLobHandler()), Types.CLOB);

String SQL = "update Student set description = :description where id = :id";

NamedParameterJdbcTemplate jdbcTemplateObject =

new NamedParameterJdbcTemplate(dataSource);

jdbcTemplateObject.update(SQL, in);

System.out.println("Updated Record with ID = " + id );

}

}

Following is the content of the MainApp.java file.

package com;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.StudentJDBCTemplate;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

StudentJDBCTemplate studentJDBCTemplate =

(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

studentJDBCTemplate.updateDescription(1,

"This can be a very long text upto 4 GB of size.");

}

}

Following is the configuration file Beans.xml.

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

<!-- Initialization for data source -->

<bean id = "dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>

<property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>

<property name = "username" value = "root"/>

<property name = "password" value = "admin"/>

</bean>

<!-- Definition for studentJDBCTemplate bean -->

<bean id = "studentJDBCTemplate"

class = "com.StudentJDBCTemplate">

<property name = "dataSource" ref = "dataSource" />

</bean>

</beans>