Module 4 : Spring JDBC Integration with MySQL for Data Access

-Prof. Anamika Dhawan

Why?

The problems of JDBC API are as follows:

- We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
- We need to perform exception handling code on the database logic.
- We need to handle transaction.
- Repetition of all these codes from one to another database logic is a time consuming task.

Data access with JDBC

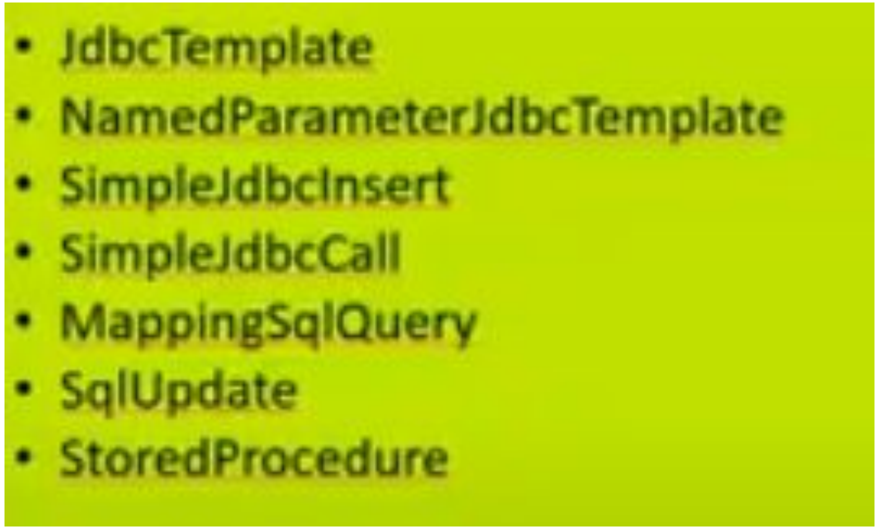
The value-add provided by the Spring Framework JDBC abstraction is perhaps best shown by the sequence of actions outlined in the table below.

The table shows what actions Spring will take care of and which actions are the responsibility of you, the application developer.

The Spring Framework takes care of all the low-level details that can make JDBC such a tedious API to develop with.

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

JDBC database access approach



- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert
- SimpleJdbcCall
- MappingSqlQuery
- SqlUpdate
- StoredProcedure

JDBCTemplate

It is the classic Spring JDBC approach and the most popular.

This "lowest level" approach and all others use a JdbcTemplate under the covers, and all are updated with Java 5 support such as generics and var args.

JDBC template

It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

It handles the exception and provides the informative exception messages by the help of exception classes defined in the **org.springframework.dao** package.

We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

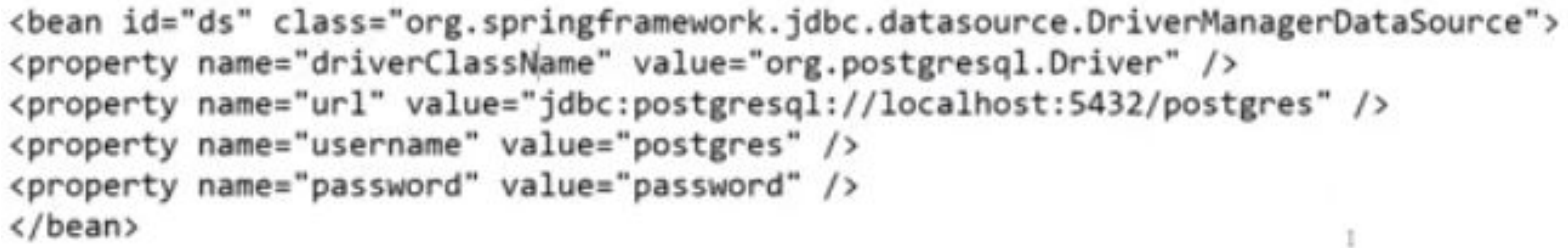
No.	Method	Description
1)	public int update(String query)	is used to insert, update and delete records.
2)	public int update(String query,Object... args)	is used to insert, update and delete records using PreparedStatement using given arguments.
3)	public void execute(String query)	is used to execute DDL query.
4)	public T execute(String sql, PreparedStatementCallback action)	executes the query by using PreparedStatement callback.
5)	public T query(String sql, ResultSetExtractor rse)	is used to fetch records using ResultSetExtractor.
6)	public List query(String sql, RowMapper rse)	is used to fetch records using RowMapper.

DriverManagerDataSource

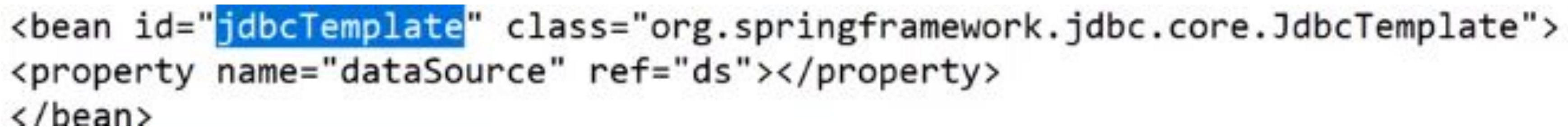
The **DriverManagerDataSource** is used to contain the information about the database such as driver class name, connnection URL, username and password.

There is a property named **datasource** in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.

configuration



```
<bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="org.postgresql.Driver" />
<property name="url" value="jdbc:postgresql://localhost:5432/postgres" />
<property name="username" value="postgres" />
<property name="password" value="password" />
</bean>
```



```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="ds"></property>
</bean>
```

Steps in Spring JDBC

DB

Model

DB Connection

IOC Container

Application

Spring JDBC Application

1.Add Jar files(Core, context in pom)
2.Add external jar files.(postgresql,jdbc,tx)
3.Create table in pgadmin
4.Create a model to map with table
5.Create configuration file for defining connection parameter(DriverManagerDataSource,JdbcTemplate).
6.Create dao file to connect jdbctemplate for dml queries and create a method to perform dml
7.Create spring container
8.create instance of dao
9.Call dao method for dml action.

PreparedStatementCallback

We can execute parameterized query using Spring JdbcTemplate by the help of **execute()** method of JdbcTemplate class. To use parameterized query, we pass the instance of **PreparedStatementCallback** in the execute method.

It processes the input parameters and output results. In such case, you don't need to care about single and double quotes.

It has only one method doInPreparedStatement:

```
1.  public T doInPreparedStatement(PreparedStatement ps)throws SQLException,
    DataAccessException
```

```
public Boolean insEmp(final Employee e) {
        String query = "insert into emp values(?,?,?)";
        return jdbcTemplate.execute(query, new PreparedStatementCallback<Boolean>() {
                public Boolean doInPreparedStatement(PreparedStatement arg0) throws
                SQLException, DataAccessException {
                                arg0.setInt(1, e.getId());
                                arg0.setString(2, e.getName());
                                arg0.setString(3, e.getEmail());
                                return arg0.execute();
                        }
                });
}
```

ResultSetExtractor

We can easily fetch the records from the database using **query()** method of **JdbcTemplate** class where we need to pass the instance of ResultSetExtractor.

ResultSetExtractor interface can be used to fetch records from the database. It accepts a ResultSet and returns the list.

It defines only one method extractData that accepts ResultSet instance as a parameter. Syntax of the method is given below:

```
public T extractData(ResultSet rs)throws SQLException,DataAccessException
public List<Employee> getAllEmployees() {
        String query = "select * from emp";
        return JdbcTemplate.query(query, new
                ResultSetExtractor<List<Employee>>() {
                        public List<Employee> extractData(ResultSet arg0) throws
                                SQLException, DataAccessException {
                                List<Employee> employees =new ArrayList<Employee>();
                                while (arg0.next()) {
                                        Employee e = new Employee();
                                        e.setId(arg0.getInt(1));
                                        e.setName(arg0.getString(2));
                                        e.setEmail(arg0.getString(3));
                                        employees.add(e);
                                }
                        return employees;
                }
        });
```

RowMapper

we can use RowMapper interface to fetch the records from the database using **query()** method of **JdbcTemplate** class. In the execute of we need to pass the instance of RowMapper now.

RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.It defines only one method mapRow that accepts ResultSet instance and int as the parameter list. Syntax of the method is given below:

```
public T mapRow(ResultSet rs, int rowNumber)throws SQLException
public List<Employee> findall() {
        String query = "select * from emp";
        List<Employee> employees = jdbcTemplate.query(query, new
                RowMapper<Employee>()
                {
                        public Employee mapRow(ResultSet arg0, int arg1) throws
                        SQLException {
                                Employee e = new Employee();
                                e.setId(arg0.getInt(1));
                                e.setName(arg0.getString(2));
                                e.setEmail(arg0.getString(3));
                                return e;
                        }
                });
        return employees;
}
```