Spring DAO/JDBC is one of the module in the spring.

This is used to develop the persistance logic or persistance layer or DAO layer. So using this module we can connect the database to perform database operations.

DAO(Data Access Object)

-----------------------

It is design pattern.

Any java class which seperates the persistance logic from business logic is called DAO class.

or

Any java class which contains persisatce logic is called DAO class.

By using JDBC, SpringJDBC, Spring ORM, Hibernate, ibatis we can develop the DAO layer.

JDBC program

------------

1. Load JDBC driver class.

2. Create the connection object.

3. Create Statement object

4. prepare and execute the SQL query.

5. process the results.

6. Take care of exception handling

7. Close JDBC objects

In the above code except 4 and 5 remaining steps are common for any application.

While developing plain JDBC application programmer has to write application specific logic and common logic.

--

--

Connection connection = null;

try {

Class.forName("driver class");

connection = DriverManager.getConnection("url", "system", "manager");

connection.setAutoCommit(false);

Statement statement = connection.createStatement();

ResultSet rs = statement.executeQuery("");

while (rs.next()) {

int no = rs.getInt(1);

String name = rs.getString(2);

// ......

}

connection.commit();

} catch (Exception e) {

try {

connection.rollback();

} catch (SQLException e1) {

e1.printStackTrace();

}

} finally {

try {

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Above code having application specific logic and common logic.

Problems of JDBC API

--------------------

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 on the database logic.

We need to handle transactions.

we need to write the logic to releasing resources.

We need to spend lot of time to change any logic in the dao layer.

If we want to change the Database its very difficult.

To overcome above problems we will go for Spring DAO/JDBC.

Spring Jdbc is a powerful mechanism to connect the database and execute SQL queries.

It(Spring Jdbc) internally uses JDBC(Adv java) api, but eliminates a lot of problems of JDBC API.

The classes and methods in spring are responsible for the creation and cleaning the resources.

Advantages of Spring Jdbc

--------------------------

-- No need to write the common logic in the application. developer has to consontrate on only application specific logic.

-- Provides abstraction layer on Plain JDBC and simplifies JDBC programming.

-- Converting all Checked to Unchecked exceptions by using Exception Rethrowing

concept(SQLExceptionTranslator).

so working with Exception Handling is optional.

-- provides queryForxxx(-) methods to get select query results in different formats.

-- We can send select query results directly over the network. because queryxxx methods returns collection classes or wraper classes.

Exception Handling

------------------

If we used jdbc directly in the DAO classes all the abnormal situations will throw java.sql.SQLException.

Based on the error code available in the SQLException the developer needs to give proper information to the caller.So working with exception handling is difficult in jdbc.

java.sql.SQLException is checked exception.

Spring provides good exception handling mechanism to work with database. it defines specific exceptions for each and every problem that occurs while dealing with DB.

org.springframework.dao.DataAccessException is the main exception in spring DAO exception hierarchy.

In Spring DAO, we have a support to transfor low level exception to Srping DAO exceptions.

org.springframework.dao.DataAccessException is child class of RuntimeException, so all spring DAO exceptions are unchecked exceptions.

Spring has the capabulity of establishing the connection, integrating with database.

Spring framework simplifies the development of JDBC, Hibernate, JPA, Ibatis, JDO etc.

Spring provides the multiple template classes to connect the DB.

Ex:

JDBCTemplate

NamedParameterJdbcTemplate

SimpleJdbcTemplate

HibernateTemplate

JpaTemplate

JdoTemplate

JmsTemplate -- (for JMS)

Above classes are defined in spring.

First three templates used in spring dao. remaining templates classes are used in different modules.

In Spring JDBC/DAO, to connect the DB we have multiple template classes.

JdbcTemplate

NamedParameterJdbcTemplate

SimpleJdbcTemplate

JdbcTemplate

------------

This is the central class in the Spring JDBC package.

It provides the JDBC support and makes working with JDBC easily.

It simplifies the use of JDBC and helps to avoid common errors.

It provides multiple methods to perform database operations.

If want to use the JdbcTemplate class in the application we have to get the JdbcTemplate object in the apllicatin or DAO layer.

Instead of creating jdbctemplate object using new keyword, inject the jdbctemplate object into DAO layer.

But to create JdbcTemplate object, DataSource object is required. so inject datasource object into jdbctemplte object.

To inject datasource object into JdbcTemplate class we have to use either setter injection or constructor injection.

DataSource

----------

it is an interface defined in javax.sql package.

DataSource acts as group of connection objects.

DataSource is an alternate for DriverManager class

Different third party vendors provides the implementation for Datasource.

Ex:

BasicDataSource(from Apache)

DriverManagerDataSource.(from Srping)

ComboPooledDataSource(from C3P0)

Example for JdbcTemplate using DriverManagerDataSource

-----------------------------------------------------

Spring JDBC used to develop DAO layer.

To develop this application following files are required.

1.StudentDAO

2.spring configuration file

3.client

StudentDAO

----------

This class having persistance logic (insert, update, delete and select)

If we want to perfrom CURD operations in DAO we have to get JdbcTemplte object.

Ex:

--

import o.s.jdbc.core.JdbcTemplate;

public class StudentDAO {

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public void insertSudent() {

String insertQuery = "insert into STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) values (101, 'Ramesh','ramesh@gmail.com','22222')";

int noOfRowsInserted = jdbcTemplate.update(insertQuery);

Sop("no of rows inserted: "+noOfRowsInserted);

}

public void updateStudent() {

String updateQuery = "UPDATE STUDENT\_DETAILS SET MOBILE='999999' WHERE NO=9";

int i = jdbcTemplate.update(updateQuery);

Sop("no of rows updated " + i);

}

public void deleteStudent() {

String deleteQuery = "DELETE STUDENT\_DETAILS WHERE NO=100";

int i = jdbcTemplate.update(deleteQuery);

sop("no of rows deleted " + i);

}

}

client applciation.

------------------

--

--

--

ApplicationContext context = -------

StudentDAO studentDAO = context.getBean("studentDAO","StudentDAO.class");

studentDAO.insertStudent();

//studentDAO.updateStudent();

//studentDAO.deleteStudent();

application-context.xml

-----------------------

<beans>

<bean id="studentDAO" class="StudentDAO">

<property name="jdbcTemplate" ref="jdbcTemplateBean"></property>

</bean>

<bean id="jdbcTemplateBean" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

<bean id="dataSourceBean"

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

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>

<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"></property>

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

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

</bean>

/beans>

diagram

-------

NOTE: to execute this application we have to add ojdbc14.jar in the class path.

In above application we discussed how to inject the datasource obect into jdbctemplate and jdbctemplate object into dao class. now we will consontrate about methods in jdbctemplate.

Working with jdbcTemplate class

-------------------------------

The JdbcTemplate has more than 100 methods to perform database operations.

To perform non select operations(insert, update ,delete) jdbctemplate provides different methods.

execute()

update()

execute()

---------

We can perform non select operations using execute()

we can overloaded execute() methods.

sysntax:

public void execute(final String sql) throws DataAccessException

public <T> T execute(ConnectionCallback<T> action) throws DataAccessException

--

--

--

Ex:

--

insert

------

String insertQuery = "insert into STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) values (1001, 'Ramesh', 'ramesh@gmail.com', '22222')";

jdbcTemplate.execute(insertQuery);

update

------

String updateQuery = "UPDATE STUDENT\_DETAILS SET MOBILE='999999' WHERE NO=1001";

jdbcTemplate.execute(updateQuery);

delete

------

String deleteQuery = "DELETE from STUDENT\_DETAILS WHERE NO=1001";

jdbcTemplate.execute(deleteQuery);

so we can perform DML operations using execute() method.

but it is not recomeded to use execute() method to perform insert, update, delete operations. because execute() returns void.

if you want know the how many no of rows effected in the db using execute() is not possible.

we can perform DDL operations using execute().

create table

------------

String createTableQuery = "create table STUDENT\_DETAILS1(NO number(4,0) not null, NAME varchar2(20), EMAIL varchar2(25), MOBILE varchar2(10))";

jdbcTemplate.execute(createTableQuery);

drop table

-----------

String dropTableQuery = "drop table STUDENT\_DETAILS1";

jdbcTemplate.execute(dropTableQuery);

generally to perform DDL operations we will use execute() method. But we won't use this method in the real world applications. because tables are designed by the DBA people.

update()

-------

Generally to perform non select operations we will go for update().

JdbcTemplate is providing overloaded update() methods to perform DML operations(insert, update, delete).

public int update(final String sql)

public int update(String sql, Object... args)

public int update(String sql, Object[] args, int[] argTypes)

--

--

--

public int update(final String sql)

-----------------------------------

When the DML query is not having any placeholders or named parameters then we will go for this method.

Examples

-------

insert

------

String insertQuery = "insert into STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) values (101, 'Ramesh', 'ramesh@gmail', '22222')";

int noOfRowsInserted = jdbcTemplate.update(insertQuery);

update

------

String updateQuery = "UPDATE STUDENT\_DETAILS SET MOBILE='999999' WHERE NO=101";

int i = jdbcTemplate.update(updateQuery);

delete

------

String deleteQuery = "DELETE STUDENT\_DETAILS WHERE NO=100";

int i = jdbcTemplate.update(deleteQuery);

-----------------------------------------

public int update(String sql, Object... args)

------------------------------------------

When the DML query is having placeholders we will use this method.

insert

-------

String insertQuery = "insert into STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) values (?, ?, ?, ?)";

int noOfRowsInserted = jdbcTemplate.update(insertQuery, new Object[] {2000, "Mahesh", "mahesh@gmail.com", "12345" });

update

------

String updateQuery = "UPDATE STUDENT\_DETAILS SET MOBILE=? WHERE NO=?";

int i = jdbcTemplate.update(updateQuery, new Object[] { "9440488120", 2000 });

System.out.println("no of rows updated " + i);

delete

------

String deleteQuery = "DELETE STUDENT\_DETAILS WHERE NO=?";

int i =jdbcTemplate.update(deleteQuery, 2000);

System.out.println("no of rows deleted " + i);

public int update(String sql, Object[] args, int[] argTypes)

----------------------------------------------------------

When the DML query having any place holders and if we want to specify the place holder types then we will use this method.

insert

------

String insertQuery = "insert into STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) values (?, ?, ?, ?)";

Object[] params = new Object[] {2010, "Mahesh", "mahesh@gmail.com", "12345" };

int[] types = new int[] { Types.INTEGER,

Types.VARCHAR, Types.VARCHAR, Types.VARCHAR };

int noOfRowsInserted = jdbcTemplate.update(insertQuery, params, types);

update

------

String updateQuery = "UPDATE STUDENT\_DETAILS SET MOBILE=? WHERE NO=?";

int i = jdbcTemplate.update(updateQuery, new Object[] { "9440488120",2000 }, new int[]{Types.VARCHAR, Types.INTEGER});

delete

------

String deleteQuery = "DELETE STUDENT\_DETAILS WHERE NO=?";

int i = jdbcTemplate.update(deleteQuery, new Object[] { 2000 }, new int[] { Types.INTEGER });

public int update(String sql, PreparedStatementSetter pss) throws DataAccessException;

This method also used the set the data into placeholder by using PreparedStatement.

String insertQuery = "insert into STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) values (?, ?, ?, ?)";

int noOfRowsInserted = jdbcTemplate.update(insertQuery, new PreparedStatementSetter() {

@Override

public void setValues(PreparedStatement ps) throws SQLException {

System.out.println("setValues method called..");

ps.setInt(1, student.getSno());

ps.setString(2, student.getSname());

ps.setString(3, student.getSemail());

ps.setString(4, student.getSmobile());

}

});

working with select operations

------------------------------

To perform select operations we have lot of methods in spring JdbcTemplate.

queryForXxx()

-------------

Ex:

int queryForInt(String sql)

int queryForInt(String sql, Object... args)

int queryForInt(String sql, Object[] args, int[] argTypes)

long queryForLong(String sql)

long queryForLong(String sql, Object... args)

long queryForLong(String sql, Object[] args, int[] argTypes)

Map<String, Object> queryForMap(String sql)

Map<String, Object> queryForMap(String sql, Object... args)

Map<String, Object> queryForMap(String sql, Object[] args, int[] argTypes)

<T> T queryForObject(String sql, Class<T> requiredType)

<T> T queryForObject(String sql, Object[] args, Class<T> requiredType)

<T> T queryForObject(String sql, Object[] args, int[] argTypes, Class<T> requiredType)

<T> T queryForObject(String sql, RowMapper<T> rowMapper)

<T> T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)

<T> List<T> queryForList(String sql, Class<T> elementType)

<T> List<T> queryForList(String sql, Object[] args, Class<T> elementType)

and overloaded query(---)

--

--

--

refer JdbcTemplate class or JdbcOperations interface to know all the queryForXxx() methods and query() methods.

public int queryForInt(String sql)

--------------------------

this method executes the select query, and it returns int value as result.

the query, that is passed to this method returns a single row and single column.

If the query doesnt returning exactly one row it throws IncorrectResultSizeDataAccessException.

If the query doesn't returning exactly one column it throws IncorrectResultSetColumnCountException.

if the query doesn't return value(it returns empty result) it throw EmptyResultDataAccessException

If the query returns otherthan int(String), it throws UncategorizedSQLException, message is "Fail to convert to internal representation"

queryForInt(String sql) method internally uses Statement object.

public int queryForInt(String sql, Object... args)

---------------------------------

This method similar to above method, but it takes second paramter, that is used to give the values for place holders

public int queryForInt(String sql, Object[] args, int[] argTypes)

-----------------------------------

This method similar to above method, but third paramter takes SQL types of place holders.

Example

-------

StudentDAO

---------

public void getStudentCount() {

int studentCount = jdbcTemplate.queryForInt("select count(\*) from student\_details");

Sop("student count: " + studentCount);

}

---------------------

int i = jdbcTemplate.queryForInt("select max(no) from student\_details");

---------------------

int i = jdbcTemplate.queryForInt("select min(no) from student\_details");

--------------------------

int i = jdbcTemplate.queryForInt("select min(no), max(no) from student\_details");

If we run above code we will get the exception.

queryForLong

------------

public long queryForLong(String sql)

public long queryForLong(String sql, Object... args)

public long queryForLong(String sql, Object[] args, int[] argTypes)

Above three methods similar to queryForInt.

Ex:

similar to queryForInt only.

public void getStudentCount() {

long l = jdbcTemplate.queryForLong("select count(\*) from student\_details");

System.out.println("student count: " + l);

}

public void getStudentMaxId() {

long l = jdbcTemplate.queryForLong("select max(no) from student\_details");

System.out.println("student max id: " + l);

}

queryForObject(-)

----------------

<T> T queryForObject(String sql, Class<T> requiredType)

It returns any type of object as result.(primitive data type or wrapper classes or String class)

This method also expected single row and single column

This method takes 2 parameters

i.e sql and required type.

if the requiredType is String this method return String as result.

if the requiredType is Double this method return Double as result.

If the query doesn't returning exactly one row it throws IncorrectResultSizeDataAccessException.

If the query doesnt returning exactly one column it throws IncorrectResultSetColumnCountException.

Note:

queryForInt method returns int as result and queryForLong mehtod returns long. but there is no method to return String, Float, Double and so on.

so if we want to get any type of object like int, long, float, double, String then we will go for queryForObject().

NOTE: queryForInt(-) queryForLong(-) internally uses only queryForObject()

<T> T queryForObject(String sql, Object[] args, Class<T> requiredType)

----------------------------------------

above method is used when the query having any place holder.

<T> T queryForObject(String sql, Object[] args, int[] argTypes, Class<T> requiredType)

-----------------------------------------

above method similar to previous method but one argument (3rd) is SQLTypes.

Examples

--------

To return int types use following syntax.

int i = template.queryForObject("sql query", Integer.class)

To return long types use following syntax.

long l = template.queryForObject("sql query", Long.class)

To return String types use following syntax.

String s = template.queryForObject("sql query", String.class)

--

--

Above examples return single column and single value.

queryForObject doesn't return any value(result or null) it throws exception i.e EmptyResultDataAccessException

<T> T queryForObject(String sql, RowMapper<T> rowMapper)

--------------------------------------------------------

this method is used to execute a query, and mapping a single result row to a Java object via a RowMapper.

RowMapper is an interface and it has only one method

i.e T mapRow(ResultSet rs, int rowNum)

If we want to use RowMapper we have to implement RowMapper interface and override mapRow(--) method.

RowMapper is used to map the resultset to our Object.

If the query doesn't returning exactly one row it throws IncorrectResultSizeDataAccessException.

<T> T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)

above method is used when the query having any place holder.

<T> T queryForObject(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)

above method similar to previous method but last argument is SQLTypes.

Example for queryForObject(--) using RowMapper.

-----------------------------------------

write the RowMapper class.

public class StudentRowMapper implements RowMapper<Student>

{

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

Student student = new Student();

student.setSno(rs.getInt(1));

student.setSname(rs.getString(2));

student.setSemail(rs.getString(3));

student.setSmobile(rs.getString(4));

return student;

}

}

use the rowMapper in DAO

--

--

public Student getStudent(){

String sql = "select \* from student\_details where no=1";

Student st = template.queryForObject(sql, new StudentRowMapper());

return st;

}

NOTE:

Instead of writing seperate class for rowmapper out side of DAO we can write the inner class in DAO class.

public class StudentDAO

{

--

--

--

private static final class StudentRowmaper implements RowMapper<Student>{

public Student mapRow( -- ){

//map the row with object.

}

}

}

Another way to write row mapper

-------------------------------

--

--

Student student = jdbcTemplate.queryForObject(SQL\_QUERY, new RowMapper<Student>(){

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

Student student = new Student();

student.setSno(rs.getInt(1));

student.setSname(rs.getString(2));

student.setSemail(rs.getString(3));

student.setSmobile(rs.getString(4));

//student.setSno(rs.getInt("NO"));

//student.setSname(rs.getString("NAME"));

//student.setSemail(rs.getString("EMAIL"));

//student.setSmobile(rs.getString("MOBILE"));

return student;

}

});

queryForMap()

------------

Map<String, Object> queryForMap(String sql)

this method executes the query and returns the result as Map.

The query is expected to be a single row query.

In the map column name as the key and column value as the value.

public Map<String, Object> queryForMap(String sql, Object... args)

this method similar to above method, but it takes second paramter that is used to give the values for place holders

public Map<String, Object> queryForMap(String sql, Object[] args, int[] argTypes)

this method similar to above method, but third paramter takes SQL types of place holders.

Ex:

String sql = "select \* from s\_d where no=2";

Map<String, Object> map = template.queryForMap(sql);

output:

{NO=2, NAME=Ramesh, EMAIL=ramesh@gmail.com, MOBILE=22222}

in the above output NO, NAME, EMAIL, MOBILE are column names.

queryForList()

--------------

List<Map<String, Object>> queryForList(String sql)

this method returns multiple rows i.e List.

List contains the Map, Map contains key value value pair, In the map column name as the key and column value as the value.

How many rows will be there in the DB that many maps will be creates.

<T> List<T> queryForList(String sql, Class<T> elementType)

This method is similar to above method and also it takes the required type of element.

Ex:

List<Integer> queryForList(sql, Integer.class)

List<String> querForList(sql, String.class)

query()

-------

There are multiple overloaded query() methods

<T> List<T> query(String sql, RowMapper<T> rowMapper)

<T> T query(String sql, ResultSetExtractor<T> rse)

--

--

<T> List<T> query(String sql, RowMapper<T> rowMapper)

------------------------------------------------------

It returns list of objects.

RowMapper

---------

If we want to use RowMapper we have to implement RowMapper interface and override mapRow(--) method.

RowMapper used to map the resultset data with our Object.

Ex:

---

public List<Student> getStudentList() {

String sql = "select \* from student\_details";

List<Student> studentList = jdbcTemplate.query(sql, new StudentRowMapper());

//display the list data using for loop

for(Student student : studentList){

sop(student.getSno() + "\t");

sop(student.getSname() + "\t");

sop(student.getSemail() + "\t");

sop(student.getSmobile() + "\t");

}

return studentList;

}

private static final class StudentRowMapper implements RowMapper<Student>

{

Student mapRow(--){

//get the data from rs, set into student.

}

}

ResultSetExtractor

-------------------

<T> T query(String sql, ResultSetExtractor<T> rse)

it returns object. it takes ResultSetExtractor as argument

ResultSetExtractor is an interface and it has T extractData(ResultSet rs) method.

it is similar to RowMapper

Ex:

String sql = "select \* from s\_d where no=2";

Student st = jdbcTemplate.query(sql, new StudentResultSetExtractor);

------------------------

public class StudentResultSetExtractor implements ResultSetExtractor<Student>

{

public Student extractData(ResultSet rs) throws SQLException, DataAccessException {

Student student = new Student();

while (rs.next()) {

student.setSno(rs.getInt(1));

student.setSname(rs.getString(2));

student.setSemail(rs.getString(3));

student.setSmobile(rs.getString(4));

}

return student;

}

}

----------------------------------

In above all applications we are injecting jdbcTemplate object into StudentDAO.

If your application has mutliple DAOs, in each DAO we have to inject jdbctemplte to connect the DB. it is burden to the developer.

solution.

---------

write the BaseDAO class, which contains JdbcTemplte object.

there are two ways to write the BaseDAO.

1st approach

----------

public class BaseDAO

{

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public JdbcTemplate getJdbcTemplate() {

return jdbcTemplate;

}

//write any common logic for all the DAO

}

use above DAO class in all the DAO's

Ex:

public class StudentDAO extends BaseDAO{

//get the jdbctemplate from baseDAO and

//call the methods defined in jdbctemplate.

--

--

getJdbcTemplate().query(---)

}

public class EmployeeDAO extends BaseDAO{

//get the jdbctemplate from baseDAO and call the method defined in jdbctemplate.

}

xml

---

similar to previous applications.

2 Approach

----------

BaseDAO

-------

public class BaseDAO {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

jdbcTemplate = new JdbcTemplate(dataSource);

}

//setters and getters for jdbctemplate

}

StudentDAO

-----------

similar to above approach.

xml

---

<beans>

<bean id="studentDAO" class="StudentDAO">

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

</bean>

</bean>

<bean id="dataSourceBean"

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

--

--

--

</bean>

</beans>

JdbcDaoSupport

--------------

It is similar to BaseDAO class given by the spring framework.

So instead of developing the BaseDAO class, directly we can use JdbcDaoSupport.

public class StudentDAO extends JdbcDaoSupport{

--

--

---

---

public void insertStudent(){

getJdbcTemplate().update(sql);

}

}

xml

---

similar to BaseDAO 2nd approach application.

Here we are using getJdbcTemplate() to get the JdbcTemplate object, this method is defined in JdbcDaoSupport.

NOTE: Spring has multiple support class to get the template objects.

for exmaple to get the JdbcTemplate object

spring provdies JdbcDaoSupport.

for NamedParameterJdbcTemplate --- NamedParameterJdbcDaoSupport

for SimpleJdbcTemplate --- SimpleJdbcDaoSupport

NamedParameterJdbcTemplate

--------------------------

In JdbcTemplate, SQL parameters are represented by a placeholder "?" symbol and bind it by position. The problem is whenever the order of parameter is changed, you have to change the parameters bindings as well. other wise we will get the errors.

To fix it, you can use "Named Parameter", whereas SQL parameters are defined by a starting colon follow by a name, rather than by position.

The named parameters are support in SimpleJdbcTemplate and NamedParameterJdbcTemplate.

NOTE: JdbcTemplate doesn't support named paramters.

INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (:no, :name, :email, :mobile)

in the above example query :no, :name, :email, :mobile are named paramters.

There are two ways to set the values for named parameters.

Map and SqlParameterSource

Map

---

String sql = "INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (:no, :name, :email, :mobile)";

Map<String, Object> parameters = new HashMap<String, Object>();

parameters.put("mobile", "123455");

parameters.put("no", 2222);

parameters.put("name", "Ramesh");

parameters.put("email", "ramesh@gmail.com");

int i = namedParameterJdbcTemplate.update(sql, parameters);

NOTE:

The names declared in the query must match the keys of our Map(case sensitive).

here order of named parameter names is not required.

SqlParameterSource

------------------

it is an interface defined in spring jdbc.

MapSqlParameterSource implements SqlParameterSource.

MapSqlParameterSource internally uses Map only.

String sql = "INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (:no, :name, :email, :mobile)";

SqlParameterSource parameters = new MapSqlParameterSource().addValue("no", 1111).addValue("name", "ramesh") .addValue("email", "ramesh@gmail.com") .addValue("mobile", "666666");

int i = namedParameterJdbcTemplate.update(sql, parameters);

----------------------------------------

NOTE:

While working with NamedParameterJdbcTemplate, we have to inject dataSource object into NamedParameterJdbcTemplate object.

To inject datasource object we have to use constructor injection.

Ex:

<beans>

<bean id="studentDAO" class="StudentDAO">

<property name="namedParameterJdbcTemplate" ref="namedParameterJdbcTemplateBean"></property>

</bean>

<bean id="namedParameterJdbcTemplateBean"

class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">

<constructor-arg ref="dataSourceBean"></constructor-arg>

</bean>

<bean id="dataSourceBean" class="----">

--

--

</bean>

</beans>

working with NamedParameterJdbcDaoSupport

------------------------------------------

Instead of injecting NamedParameterJdbcTempalate in each DAO we can get the template object by using support class.

steps

------

1.write the dao class which extends NamedParameterJdbcDaoSupport

public class StudentDAO extends NamedParameterJdbcDaoSupport

{

public void insertStudent(){

getNamedParameterJdbcTemplate() .update(--)

}

}

2.develop xml

-------------

<beans>

<bean id="studentDAO"

class="StudentDAO">

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

</bean>

<bean id="dataSourceBean" class="">

--

--

</bean>

</beans>

SimpleJdbcTemplate

------------------

It is similar to jdbctemplate object but its uses java5 features. like varargs autoboxing.

String sql = "INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (?, ?, ?, ?)";

int i = simpleJdbcTemplate.update(sql, 1111, "abc", "abc@gmail.com", "666666");

NOTE:

since Spring 3.1 SimpleJdbcTemplate was deprecated. The JdbcTemplate and NamedParameterJdbcTemplate now provide all the functionality of the SimpleJdbcTemplate.

public class StudentDAO

{

private SimpleJdbcTemplate simpleJdbcTemplate;

//setter for simpleJdbcTemplate

//use the template object.

}

}

BatchUpdate

-----------

In some cases, you may required to insert a batch of records into database at a time. If you call a single insert method for every record it decreases the performance.

to resolve this problem we will go for batchUpdate() method.

Ex:

String sql1 = "INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (201, 'ramesh', 'ramesh@gmail.com', '88888')";

String sql2 = "INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (202, 'ramesh', 'ramesh@gmail.com', '88888')";

String sql3 = "INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (203, 'ramesh', 'ramesh@gmail.com', '88888')";

int[] i = jdbcTemplate.batchUpdate(new String[] { sql1, sql2, sql3 });

note: there are multiple methods in jdbctemplate clsss to work with batch updations.

public int[] batchUpdate(final String[] sql)

public int[] batchUpdate(final String sql, final BatchPreparedStatementSetter pss)

Ex:

---

String sql = "INSERT INTO STUDENT\_DETAILS (NO, NAME, EMAIL, MOBILE) VALUES (?, ?, ?, ?)";

jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {

@Override

public void setValues(PreparedStatement ps, int i) throws SQLException {

Student student = studentsList.get(i);

ps.setInt(1, student.getNo());

ps.setString(2, student.getName());

ps.setString(3, student.getEmail);

ps.setString(4, student.getMobile);

}

public int getBatchSize() {

System.out.println("Batch Size: " + studentsList.size());

return studentsList.size();

}

});

Callback interfaces

-------------------

There are multiple callback intefaces in spring DAO.

ConnectionCallback

StatementCallback

PreparedStatementCallback

CallableStatementCallback

In spring jdbctemplate class doesnt provides the connection, statement preparedstatement callablestatement directly.

to get those objects in the applciation we will go for callback interfaces.

To get the connection we have to use ConnectionCallback ineterface.

ConnectionCallback interface having only one metod.

T doInConnection(Connection connection, ConnectionFactory connectionFactory) ----

for examples refer meterail.

Spring supprots to work with diffrent types of datasource implementations.

Ex:

DriverManagerDataSource

BasicDataSource

ComboPooledDataSource

JNDI(Server supplied Datasource).

procedure to work with BasicDataSource.

1.configure BasicDataSource in xml

<beans>

<bean id="studentDAO" class="------">

<propery name="jdbcTemplate" ref="----"/>

</bean>

<bean id="jdbcTemplateBean" class="------">

<propery name="dataSource" ref="----"/>

</bean>

<bean id="dataSourceBean" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>

<property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>

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

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

</bean>

</beans>

2.use the jdbc template in DAO.

NOTE:To execute BasicDatasource add following jars in the classpath.

1.common-dbcp.jar

2.common-pooling.jar

Example for BasicDataSource, ComboPooledDatasource and JNDI refer meterial.