Spring JdbcTemplate is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

Problems of JDBC API

The problems of JDBC API are as follows:

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

Spring provides a simplification in handling database access with the Spring JDBC Template. The Spring JDBC Template has the following advantages compared with standard JDBC.

1. The Spring JDBC template allows to clean-up the resources automatically, e.g. release the database connections.
2. The Spring JDBC template converts the standard JDBC SQLExceptions into RuntimeExceptions. This allows the programmer to react more flexible to the errors. The Spring JDBC template converts also the vendor specific error messages into better understandable error messages.

JdbcTemplate class:

- 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.

Methods:

1. public int update(String query):
   This is used to insert, update and delete records.
2. public int update(String query,Object... args):
   This is used to insert, update and delete records using PreparedStatement by using given number of arguments.
3. public void execute(String query):
   It is used to execute DDL query
4. public T execute(String sql, PreparedStatementCallback action):
   It executes the query by using PreparedStatement callback.

5. public T query(String sql, ResultSetExtractor resultSetExecutor):
   It is used to fetch records using ResultSetExtractor.
6. public List query(String sql, RowMapper rowMapper):
   It is used to fetch records using RowMapper.

applicationContext.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"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jdbc="http://www.springframework.org/schema/jdbc"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
            http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-4.3.xsd">

      <bean id="ds"

      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <property name="driverClassName"
                  value="com.mysql.jdbc.Driver" />
            <property name="url"
                  value="jdbc:mysql://localhost:3306/B200109" />
            <property name="username" value="root" />
            <property name="password" value="pankaj" />
      </bean>

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

      <bean id="employeeDAOImpl"
class="com.spring.templates.dao.impl.EmployeeDAOImpl">
            <property name="jdbcTemplate" ref="jdbcTemplate"></property>
      </bean>
</beans>
```

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

There are 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.

Here, we are using the JdbcTemplate object in the EmployeeDAO class, so we are passing it by the setter method but you can use constructor also.

EmployeeDAOImpl.java

Create JdbcTemplate object and setter method

```
private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
	this.jdbcTemplate = jdbcTemplate;
}
```

Method to Save Employee

```
	public boolean saveEmployee(Employee employee) {
		/*
		  String query = "insert into employee values('" +
		  employee.getId() + "','" +
		  employee.getName() + "','" + employee.getAddress() + "','" +
		  employee.getEmail() + "','" + employee.getMobile() + "')";
	try {
		  jdbcTemplate.update(query); return true; } catch
	(DataAccessException e) {
		  e.printStackTrace();
		return false;
		}
	  */

		String query = "insert into employee values(?,?,?,?,?)";
		try {
			jdbcTemplate.update(query, employee.getId(),
			employee.getName(), employee.getAddress(),
			employee.getEmail(),  employee.getMobile());
			return true;
		} catch (DataAccessException e) {
			e.printStackTrace();
			return false;
		}
	}
```

Method to Update Employee

```
	public boolean updateEmployee(Employee employee) {
	String query = "update employee set name=?, address=?,email=?,mobile=?
	 where id =?";
	try {
	jdbcTemplate.update(query, employee.getName(), employee.getAddress(),
	employee.getEmail(),employee.getMobile(), employee.getId());
		return true;
	}
	catch (DataAccessException e) {
			e.printStackTrace();
```

```
                return false;
            }
        }
```

Delete Employee by Id

```
    public boolean deleteEmployee(Employee employee) {
        String query = "delete from employee where id =?";
        try {
            jdbcTemplate.update(query, employee.getId());
            return true;
        } catch (DataAccessException e) {
            e.printStackTrace();
            return false;
        }
    }
```

Get Employee By Id

```
    public Employee getEmployeeById(int id) {
        String query = "select * from employee where id = ?";
            RowMapper<Employee> rowMapper = new
    BeanPropertyRowMapper<Employee>(Employee.class);
        return jdbcTemplate.queryForObject(query, rowMapper, id);
    }
```

Get All Employees

```
public List<Employee> getAllEmployees() {

// 1. ResultSetExecutor Example
return jdbcTemplate.query("select * from employee", new ResultSetExtractor<List<Employee>>()
{
public List<Employee> extractData(ResultSet rs) throws SQLException, DataAccessException {

    List<Employee> list = new ArrayList<Employee>();
        while (rs.next()) {
            Employee employee = new Employee();
            employee.setId(rs.getInt(1));
            employee.setName(rs.getString(2));
            employee.setAddress(rs.getString(3));
            employee.setEmail(rs.getString(4));
            employee.setMobile(rs.getLong(5));
            list.add(employee);
        }
        return list;
    }
});

}



    public List<Employee> getAllEmployees() {
        // 2. RowMapper Example

        return jdbcTemplate.query("select * from employee", new RowMapper<Employee>() {

            @Override
            public Employee mapRow(ResultSet rs, int rownumber) throws SQLException {
```

```
                        Employee employee = new Employee();
                        employee.setId(rs.getInt(1));
                        employee.setName(rs.getString(2));
                        employee.setAddress(rs.getString(3));
                        employee.setEmail(rs.getString(4));
                        employee.setMobile(rs.getLong(5));
                        return employee;
                    }
                });
    }
```

```
public List<Employee> getAllEmployees() {
            // 3. By using BeanPropertyRowMapper
            String sql = "SELECT * FROM employee";
            RowMapper<Employee> rowMapper = new BeanPropertyRowMapper<>(Employee.class);
            return this.jdbcTemplate.query(sql, rowMapper);
        }
}
```

ResultSetExtractor Interface(To Fetch Records):

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

public T query(String sql,ResultSetExtractor<T> resultSetExecutor)

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

Method of ResultSetExtractor

It defines only one method extractData that accepts ResultSet instance as a parameter.

public T extractData(ResultSet rs)throws SQLException,DataAccessException

RowMapper:(To Fetch Records):

Like ResultSetExtractor, 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.

query() method Signature:

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

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 because 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:

public T mapRow(ResultSet resultSet, int rowNumber)throws SQLException

BeanPropertyRowMapper:

Spring JDBC provides **BeanPropertyRowMapper** that implements **RowMapper**. We can directly use it in place of custom RowMapper. We use BeanPropertyRowMapper in the scenario when database table column names and our class fields name are of same.

```
public List<Employee> getAllEmployees() {
      String sql = "SELECT articleId, title, category FROM articles";
RowMapper<Employee> rowMapper = new
BeanPropertyRowMapper<>(Employee.class);
return this.jdbcTemplate.query(sql, rowMapper);
}
```

JdbcTemplate : Run SQL Queries

JdbcTemplate provides methods to run DML and DDL SQL queries. Find the example of some of them.

JdbcTemplate.queryForObject :

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

This method fetches data for a given SQL query as an object using RowMapper. SQL query can have bind parameters. Find the description of parameters.

sql: SQL containing bind parameter.

rowMapper: Object of RowMapper implemented class. RowMapper will map one object per row.

args: Arguments that bind to the query.

By using this method we can get the Employee By using the Id.

```
 public Employee getEmployeeById(int id) {
           String query = "select * from employee where id = ?";
           RowMapper<Employee> rowMapper = new
BeanPropertyRowMapper<Employee>(Employee.class);
           return jdbcTemplate.queryForObject(query, rowMapper, id);
      }
```

Spring and ORM

Hibernate and Spring Integration:

To integrate Hibernate with Spring MVC application, you can use the LocalSessionFactoryBean class, which set up a shared SessionFactory object within a Spring application context. This SessionFactory object can be passed to DAO classes via dependencies injection.

Basically, in order to support Hibernate integration, Spring provides two key beans available in the org.springframework.orm.hibernate5 package:

LocalSessionFactoryBean: creates a Hibernate's SessionFactory which is injected into Hibernate-based DAO classes.

HibernateTransactionManager: provides transaction support code for a SessionFactory. Programmers can use @Transactional annotation in DAO methods to avoid writing boiler-plate transaction code explicitly.

The spring-orm module provides the Spring integration with Hibernate:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.1.8.RELEASE</version>
</dependency>
```

Hibernate Integrating with Java File:

```
@Configuration
@EnableTransactionManagement
@ComponentScan({ "com.shopping" })
public class HibernateConfiguration {

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new
        LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(new String[] { "com.shopping.model"
});
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
     }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/s190040");
        dataSource.setUsername("root");
        dataSource.setPassword("pankaj");
```

```
        return dataSource;
    }

    private Properties hibernateProperties() {
        Properties properties = new Properties();

properties.put("hibernate.dialect","org.hibernate.dialect.MySQL5Dialect");
        properties.put("hibernate.show_sql",true);
        properties.put("hibernate.format_sql",true);
        properties.put("hibernate.hbm2ddl.auto", "update");
        return properties;
    }

    @Bean
    @Autowired
    public HibernateTransactionManager transactionManager(SessionFactory s) {
       HibernateTransactionManager txManager = new
HibernateTransactionManager();
       txManager.setSessionFactory(s);
       return txManager;
    }
}
```