DAO Support and JDBC in Spring

Introduction

- **DAO support in Spring is primarily aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way**
- **Spring JDBC framework is built on top of the core JDBC API.**
  - Its main aim is to leverage the features of JDBC that work well and to abstract away all problem areas.
- **Spring provides a consistent exception hierarchy that is used across all of its DAO frameworks**

Problem with JDBC code

```
public void GettingRows() {
Connection conn=null;
Statement stmt=null;
Resultset rset=null;
try{
      conn = dataSource.getConnection();
      stmt = conn.createStatement ();
      rset = stmt.executeQuery ("select empno, ename,job from emp");
              while (rset.next()) {
                System.out.print (rset.getString (1));
              }
  } catch (SQLException e) {
              LOGGER.error(e);
           throw e;
   }
finally{
         //code to clean up resources
```

Declare connection parameters

Open connection

Create statement

Execute statement

Iterate over resultset

Handle exceptions

clean up resources

Using JDBC with Spring

- Spring separates the fixed and variant parts of the data access process into two distinct classes:
  - Templates: manage the fixed part of the process like controlling transactions, managing resources, handling exceptions etc
  - Callbacks: define implementation details, specific to application such as creating statements, binding parameters etc.

Using JDBC with Spring

Action	Developer	Spring
Define connection parameters	Yes	
Open the connection		Yes
Specify the SQL statement	Yes	
Declare parameters and provide values	Yes	
Prepare and execute the statement		Yes
Set up the loop to iterate through the results		Yes
Do the work for each iteration(Logic)	Yes	
Process any exception		Yes
Handle transactions		Yes
Close connection, statement and Resultset		Yes

Data Access Templates

- Spring offers several data access templates, each suitable for a different persistent mechanism. Some of them are listed below:

Template class:	**Suitable for:**
jdbc.core.JdbcTemplate	JDBC connections
Jdbc.core.namedparam.NamedParameterJdbcTemplate	JDBC connections with support for named parameters
orm.hibernate.HibernateTemplate	Hibernate 2.x sessions
orm.jdo.JdoTemplate	Java data objects implementations
orm.jpa.JpaTemplate	Java persistent API entity managers

The JdbcTemplate class

- Central class in JDBC framework
  - Manages all database communication and exception handling
  - Based on template style of programming; some calls are handled entirely by JdbcTemplate while others require the calling class to provide callback methods that contain implementation for parts of the JDBC workflow

The DataSource

- **A DataSource obtains a connection to the database for working with data.**
- **Some of the implementations of DataSource are:**
  - BasicDataSource
  - PoolingDataSource
  - SingleConnectionDataSource
  - DriverManagerDataSource
- **The datasource can be programmatically configured, for example:**

```
DriverManagerDataSource ds = new   DriverManagerDataSource();
ds.setDriverClassName(driver);
ds.setUrl(url);
ds.setUsername(username);
ds.setPassword(password);
```

Example

```
<bean id="dataSource"   class=
  “org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
                value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url“
                value="jdbc:oracle:thin:@oraserver:1521:oradb"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
</bean>
```

```
<bean id="dataSource"
      class=“org.springframework.jndi.JndiObjectFactoryBean">
    <property name=“jndiName" value=“/jdbc/TrgDatasource"/>
    <property name=“resourceRef “ value=“true"/>
</bean>
```

Wiring beans in the Spring Context file

```
<bean id=”jdbcTemplate”
        class=” org.springframework.jdbc.core. JdbcTemplate ”>
    <property name=”dataSource”><ref local=” myDataSource” /></property>
</bean>
<bean id=“myDataSource "    ….
  <!– the datasource configuration comes here -- >
</bean>
<bean id=“jdbcTemplateDemo” class=“JdbcTemplateDemo”>
      <property name= “jdbcTemplate” ref= “jdbcTemplate” />
<bean>
```

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

Query and update methods: Examples

- JdbcTemplate jt = new JdbcTemplate(dataSource);

```
int count = jt.queryForObject("select count(*) from emp",Integer.class);
```

query method examples

```
String name = (String) jt.queryForObject("select name from mytable where
empno=1022", String.class);
```

```
List rows = jt.queryForList("select * from mytable");
```

```
Object params[] = new Object[]{new Double(1000.0)};
List rows1 = jt.queryForList("Select * from emp where sal > ?",params);
```

```
int x=jt.update("insert into Book(id,name) values(1,'Core Java')");
```

update method examples

```
int x=jt.update("Update Book set name='Advanced Java' where id=?", new
Object[]{new Integer(3)});
```

```
int x=jt.update("Delete from Book where id=2");
```

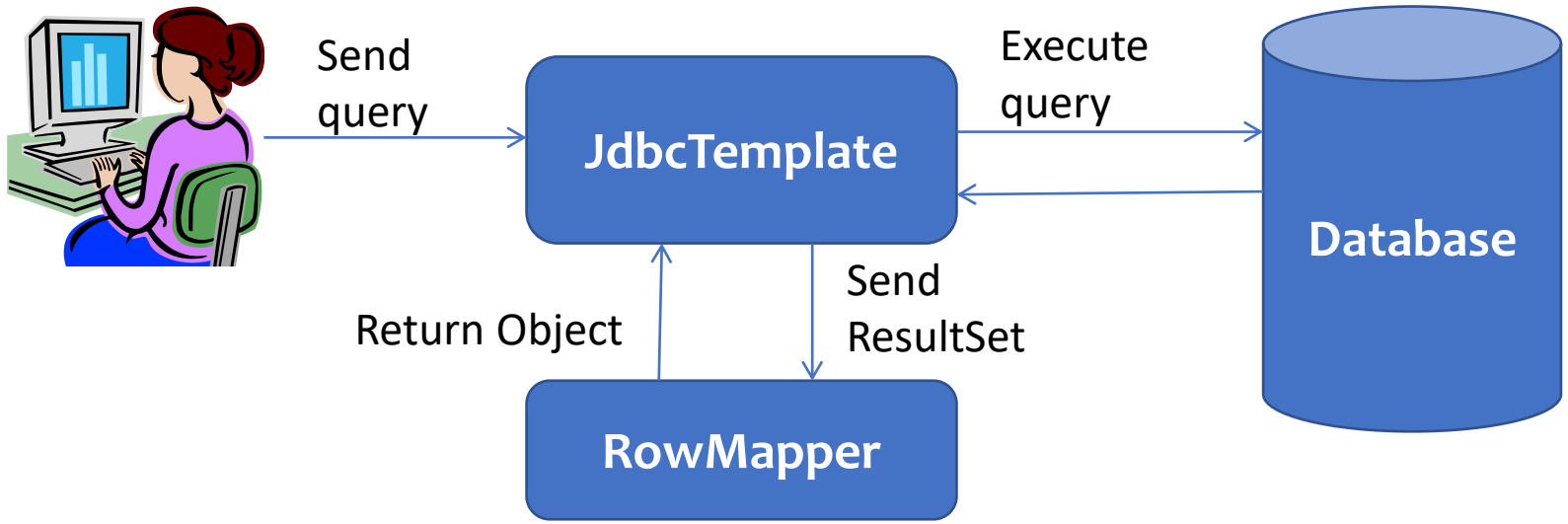
```
String sql = "insert into person(id,fname,lname) values (?,?,?)";
  Object[] params = new Object[]{ p.getId(), p.getFname(), p.getLname()};
   int count = jdbcTemplate.update(sql,params);
```

JdbcTemplate : execute methods

```
public class ExecuteAStatement {
   private JdbcTemplate jt, DataSource dataSource;
   public void doExecute() {
        jt = new JdbcTemplate(dataSource);
        jt.execute("create table mytable (id integer, name  varchar(100))");
   }
   public void setDataSource(DataSource dataSource) {
            this.dataSource = dataSource;
    }
}
```

RowMapper

- **RowMapper is an interface used to bind each row of data to an object**



- **Steps to implement RowMapper:**
  - Create a class implements with RowMapper
  - Override mapRow method which has 2 arguments
    - ResultSet – ResultSet created after executing query.
    - rownum – Number of current row.
  - MapRow method map's each row of data in the ResultSet to an Object

Implementation of RowMapper

```
String sql="SELECT * FROM employee";
 List<Employee> empList =getJdbcTemplate().query(sql,new EmployeeRowMapper());
```

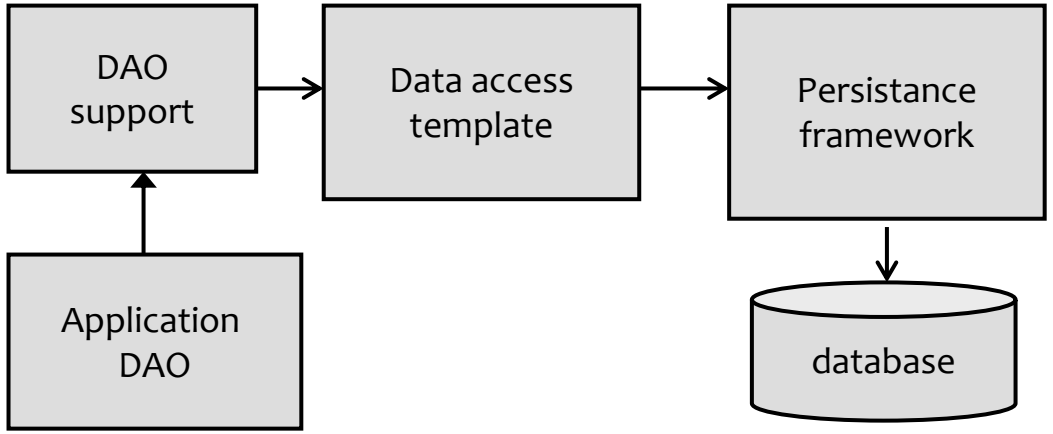
```
public class EmployeeRowMapper  implements RowMapper {
        @Override
        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
                Employee emp = new Employee();
                emp.setEmployeeId(rs.getInt(1));
                emp.setEmployeeName(rs.getString(2));
                emp.setEmployeeSalary(rs.getDouble(3));
                return emp;
        }
}
```

Implementation of RowMapper – Lambda Expressions

```
String sql="SELECT * FROM employee";
 List<Employee> empList =getJdbcTemplate().query(sql, (rs,rowNum) ->{
                        Employee emp = new Employee();
                        emp.setEmployeeId(rs.getInt(1));
                        emp.setEmployeeName(rs.getString(2));
                        emp.setEmployeeSalary(rs.getDouble(3));
                        return emp;
});
```

JdbcDaoSupport class

- **On top of the template-callback design, Spring provides DAO support classes that are meant to be subclassed.**
  - When writing your application DAO implementation, you can call a template retrieval method to have direct access to the underlying data access template.
  - To make it easier to work with a variety of data access technologies like JDBC, JDO and Hibernate in a consistent way, Spring provides a set of abstract DAO classes that you can extend



Handling Exceptions in the Spring JDBC

- Data Access Exception:

The Spring framework offers a consistent data access exception-handling mechanism for its data access module, including the JDBC framework. In general, all exceptions thrown by the Spring JDBC framework are subclasses of DataAccessException, a type of RuntimeException that you are not forced to catch. It's the root exception class for all exceptions in Spring's data access module

Ex 1:

```
try
{
      employeeDao.insert(employee);
}
catch (DuplicateKeyException e)
{
            System.out.println(“Employee Already Exist");
} catch (DataAccessException e)
{
            // Code to handle exception
}
```

4.7: Using Profiling effectively - XML

```
<bean id="properties"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location">
                <value>oracle.properties</value>
        </property>
</bean>
<beans profile="“dev">
        <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="${db.driver}" />
                <property name="url" value="${db.url}" />
                <property name="username" value="${db.user}" />
                <property name="password" value="${db.password}" />
        </bean>
</beans>
```

- To enable a specific profile, use the below code:
  - System.*setProperty("spring.profiles.active", "dev");*

4.7: Using Profiling effectively - @Profile

```
@Component
@Profile("production")
public class DatabaseConfig {

        @Bean
        @Qualifier("myDataSource")
        public DataSource createDataSource()
        {
            DriverManagerDataSource dms=new DriverManagerDataSource();
            dms.setDriverClassName("oracle.jdbc.driver.OracleDriver");
            dms.setUrl("jdbc:oracle:thin:@172.28.40.3:1521:orcl");
            dms.setUsername("user1");
            dms.setPassword("user1");
            return dms;
        }
}
```