Spring DAO and JDBC

BY

Apparao G

DAO (Data Access Object)

Context:

- We have to implement a system that access multiple persistence storages, such as database, flat text files, XML document etc.
- And we have tried to implement the persistence logic to access the persistence storage along with the business logic in the business process components.
- Since the data access logic varies depending on the data store type we have found that this approach is complicated in implementing, testing, and maintaining.

Problem:

We want to separate the persistence logic or integration logic from the business logic.

Forces: The following are the characteristics of the forces in developing a system that applies the Data Access Object(DAO) pattern:

1. Want to reuse the persistence logic

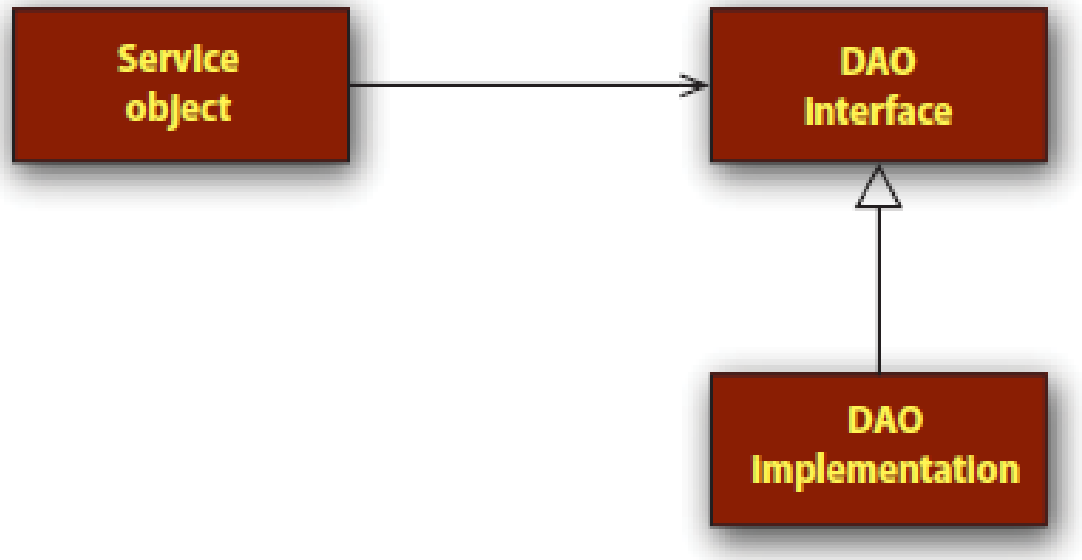
2. Want to abstract the data access logic and backend system exception handling from business logic components.

3. Want to reduce the number of interactions to the data store.

Solution:

Design a DAO that takes the responsibility of communicating with the backend system and present the data to the business object/components in our application understandable format.

Component Diagram:



Example: **Using JDBC to query a row from a database**

```
private static final String SQL_SELECT_SPITTER =
      "select id, username, fullname from spitter where id = ?";
public Spitter getSpitterById(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();

    stmt = conn.prepareStatement(SQL_SELECT_SPITTER);

    stmt.setLong(1, id);

    rs = stmt.executeQuery();

    Spitter spitter = null;
    if (rs.next()) {
      spitter = new Spitter();
      spitter.setId(rs.getLong("id"));
      spitter.setUsername(rs.getString("username"));
```

Get connection

Create statement

Bind parameter

Execute query

Process results

```
      spitter.setPassword(rs.getString("password"));
      spitter.setFullName(rs.getString("fullname"));
    }
    return spitter;
  } catch (SQLException e) {

  } finally {
    if(rs != null) {
      try {
        rs.close();
      } catch(SQLException e) {}
    }

    if(stmt != null) {
      try {
      stmt.close();
      } catch(SQLException e) {}
    }

    if(conn != null) {
      try {
        conn.close();
      } catch(SQLException e) {}
    }
  }

  return null;
}
```

Handle exceptions (somehow)

Clean up

Problems with JDBC:

If we are using JDBC API as a low-level data access API to implements the DAOs there are some shortcoming, which can be found from the sample code of above example as shown in slide 5 and 6.

1. **Code Duplication**:

- As evident from slide 5 ,Code duplication with the code related to exception handling, getting the connection, preparing a JDBC statement, etc., is a major problem while using JDBC API directly to access database.
- As we know that writing boilerplate code over and over again is a clear violation of the basic object-oriented(OO) principal of code reuse.
- This has some side effects in terms of projects cost, timelines, and effort.

2. Resource Leakage:

As shown in slide 6, all DAO methods must hand over control of obtained database resources like connection, statement, and resultset.

This is a risky plan because a beginner programmer can very easily skip those code fragments. As result, resources would run out and bring the system to stop.

3. Error Handling:

- When using JDBC directly we need to handle SQLException since the JDBC drivers report all error situations by raising the SQLException.

- Most of these exception are not possible to be recovered. Moreover, the error code and message obtained from the SQLException object are database vendor-specific, so it is difficult to write portable DAO error messaging code.

To solve the above described problems spring introduced spring JDBC module.

Spring JDBC

- The Spring frame work provides a solution fro these problems by giving a thin, robust, and highly extensible JDBC abstraction framework.
- The JDBC abstraction framework provides under spring framework as a value –added service takes cares of all the low-level details like retrieving connection, preparing the statement, executing the query, and releasing the database resources.
- While using the JDBC abstraction framework of spring framework for data access the application developer needs to specify the SQL statements to execute and read the results.
- Since all the low-level logic have been written once and correctly into the abstraction layer provided by spring JDBC frame work this helps to eliminate the shortcomings found in using JDBC API to implement DAO's, which are described above.

JdbcTemplate:

The most basic of Spring's JDBC templates, this class provides simple access to a database through JDBC and simple indexed-parameter queries.

NamedParameterJdbcTemplate:

This JDBC template class enables you to per- form queries where values are bound to named parameters in SQL, rather than indexed parameters.

SimpleJdbcTemplate:

This version of the JDBC template takes advantage of Java 5 features such as autoboxing, generics, and variable parameter lists to sim- plify how a JDBC template is used.

Configuring Data Source :

Spring offers several options for configuring data source beans in your Spring application, including

1. Data sources that are defined by a JDBC driver
2. Data sources that are looked up by JNDI
3. Data sources that pool connections

1. Data sources that are looked up by JNDI

```
<jee:jndi-lookupid="dataSource" jndi-name="/jdbc/SpitterDS"
                    resource-ref="true"/>
```

The jndi-name attribute is used to specify the name of the resource in JNDI.

If only the jndi-name property is set, then the data source will be looked up using the name given as is. But if the application is running within a Java application server, then you'll want to set the resource-ref property to true so that the value given in jndi- name will be prepended with java:comp/env/.

2. JDBC driver-based* data *source

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <propertyname="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <propertyname="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
    <propertyname="username"value="system"/>
    <propertyname="password"value="oracle"/>
</bean>
```

3. Using a pooled data source

```
<bean id="dataSource" 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 name="username" value="system"></property>
    <property name="password" value="oracle"></property>
    <propertyname="initialSize“  value="5"/>
    <propertyname="maxActive“  value="10"/>
</bean>
```

Configuring the JdbcTemplate:

```
<?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-2.5.xsd">
	<bean id="dataSource" 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="oracle"></property>
	</bean>
	<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
		<property name="dataSource">
			<ref bean="dataSource" />
		</property>
	</bean>
	<bean id="empDao" class="com.apparao.dao.EmpDaoImpl">
		<property name="jdbcTemplate">
			<ref bean="jdbcTemplate" />
		</property>
	</bean>
</beans>
```

Examples of JdbcTemplate class usage:

Insert:

```
public int createEmp(Emp emp) {
        Object[] values = new Object[] {
                emp.getEno(),
                emp.getEname(),
                emp.getSal(),
                emp.getComm(),
                emp.getJob() };
        return jdbcTemplate.update(
                        "insert into emp values(?,?,?,?,?)", values);
    }
```

Delete:

```
public int deleteEmp(Emp emp) {
        return jdbcTemplate.update(
                        "delete from emp where eno = ?",
                         new Object[]{emp.getEno()});
    }
```

Update:

```
public int updateEmp(Emp emp) {
        return jdbcTemplate.update(
                    "update emp set sal=? where eno=?",
                     new Object[]{emp.getSal(),emp.getEno()});
    }
```

Executing SQL Select Queries:

Using JdbcTemplate to query the database includes two simple steps described below:

1. Prepare an SQL query and execute
2. Read the results

   JdbcTemplate class includes query() methods to prepare and execute the SQL queries.

   Spring JDBC abstraction framework provides three different types of callbacks to read the results after executing the SQL query() methods.

   The three callbacks of spring JDBC abstraction framework to retrive the data are:

   1. ResultSetExtractor

   2. RowCallbackHandler

   3.RowMapper

1. ResultSetExtractor:

The ResultSetExtractor is a callback interface used by JdbcTemplate's query methods. This interface contains only one method and signature of method shown below.

Public Object extractData(ResultSet rs) throws SQLException,DataAccessException

Example:

```
public Emp getEmp(int empno) throws DataAccessException{
        System.out.println("emp no in daoimpl:"+empno);
        return(Emp)jdbcTemplate.query("select eno,ename,sal,comm,job from emp where
              eno=?", new Object[]{empno}, new ResultSetExtractor<Emp>(){
         public Emp extractData(ResultSet resultSet)
               throws SQLException, DataAccessException {
           Emp emp = new Emp();
           if(resultSet.next()) {
              emp.setEno(resultSet.getInt(1));
              emp.setEname(resultSet.getString(2));
              emp.setSal(resultSet.getDouble(3));
              emp.setComm(resultSet.getDouble(4));
              emp.setJob(resultSet.getString(5));
           }
          return emp;
        }});
    }
```

2. RowCallbackHandler

The RowCallbackHandler is a callback interface used by JdbcTemplate's query methods. This interface contains only one method and signature of method shown below.

Public void processRow(ResultSet rs) throws SQLException

```
public Emp getEmployeeDetail(int empno) throws DataAccessException{
        EmpResults empResults = new EmpResults();
        jdbcTemplate.query(
                "select eno,ename,sal,comm,job from emp where eno=?",
                 new Object[]{empno}, empResults);
        return empResults.emp;
 }
```

```
Public class EmpResults implements RowCallbackHandler{
    public Emp emp;
    public void processRow(ResultSet resultSet) throws SQLException {
        emp = new Emp();
        emp.setEno(resultSet.getInt(1));
        emp.setEname(resultSet.getString(2));
        emp.setSal(resultSet.getDouble(3));
        emp.setComm(resultSet.getDouble(4));
        emp.setJob(resultSet.getString(5));
    }
}
```

3. RowMapper:

The RowMapper is a callback interface used by JdbcTemplate's query methods. This interface contains only one method and signature of method shown below.

Public Object mapRow(ResultSet rs ,int rowNum) throws SQLException

```
public List<Emp> getEmployeeDetails() throws DataAccessException {
      String query = "select eno,ename,sal,comm,job from emp";
      List<Emp> list = jdbcTemplate.query(query, new RowMapper<Emp>() {
         public Emp mapRow(ResultSet resultSet, int i) throws SQLException {
            Emp emp = new Emp();
            emp.setEno(resultSet.getInt(1));
            emp.setEname(resultSet.getString(2));
            emp.setSal(resultSet.getDouble(3));
            emp.setComm(resultSet.getDouble(4));
            emp.setJob(resultSet.getString(5));
            return emp;
         }
      });
      return list;
   }
```

BeanPropertyRowMapper:

```
public Emp getEmp(int empno) throws DataAccessException {
    System.out.println("emp no in daoimpl:" + empno);
    return (Emp) getJdbcTemplate().query("select eno,ename,sal,comm,job
        from emp where eno=?",
        new Object[] {empno },new
        BeanPropertyRowMapper(Emp.class)).get(0);
 }
```

Example:

```
int rowCount = this.jdbcTemplate.queryForInt("select count(*) from emp");
int countOfActorsNamedJoe = this.jdbcTemplate.queryForInt(
        "select count(*) from emp where first_name = ?", "Rao");
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from emp where empno = ?",
        new Object[]{19}, String.class);
```

Batch Updates using JdbcTemplate:

JdbcTemplate includes two overloaded batchUpdate() methods in support of this feature.

one for eqecuting a batch of SQL statements using JDBC Statement and the other for executing the SQL statement for multiple times with different parameters using PreparedStatement.

The method signature of these two methods are shown below.

Public int[] batchUpdate(String[] sql) throws DataAcessException

Example:

```
jdbcTemplate.batchUpdate(new String[]{
        "update emp set sal=sal*1.2 where empno=19",
         "update emp set sal=sal*1.1 where empno=19",
        "update dept set loc='Bangalore' where deptno=10"
});
```

Public int[] batchUpdate(String sql,BatchPreparedStatementSetter bpss) throws DataAcessException

Example:

```
jdbcTemplate.batchUpdate("update emp set sal*=? where empno=?",
    new BatchPreparedStatementSetter(){
        public int getBatchSize(){return 2;}
        public void setValues(PreparedStatement ps, int i){
            if(i==1){
                ps.setDouble(1,1.3);
                ps.setInt(2,19);
            }else{
                ps.setDouble(1,1.1);
                ps.setInt(2,9);
            }
        }
    }
)
```

JDBC's exception hierarchy versus Spring's data access exceptions

JDBC's exceptions	Spring's data access exceptions
BatchUpdateException DataTruncation SQLException SQLWarning	CannotAcquireLockException CannotSerializeTransactionException CleanupFailureDataAccessException ConcurrencyFailureException DataAccessException DataAccessResourceFailureException DataIntegrityViolationException DataRetrievalFailureException DeadlockLoserDataAccessException EmptyResultDataAccessException IncorrectResultSizeDataAccessException IncorrectUpdateSemanticsDataAccessException InvalidDataAccessApiUsageException InvalidDataAccessResourceUsageException OptimisticLockingFailureException PermissionDeniedDataAccessException PessimisticLockingFailureException TypeMismatchDataAccessException UncategorizedDataAccessException

Spring JdbcTemplate example using Annotation:

```
public interface EmpService {
   public void createEmp(final Emp emp)throws DataAccessException ;

    public int updateEmp(final Emp emp)throws DataAccessException ;

    public int deleteEmp(final Emp emp)throws DataAccessException ;

   public Emp getEmp(final int empno)throws DataAccessException ;

    public List<Emp> getEmployeeDetails()throws DataAccessException ;

}
```

```
@Service("empService")
public class EmpServiceImpl implements EmpService {
   @Autowired
   public EmpDao empDao;

   @Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.DEFAULT,readOnly = false )
   public void createEmp(Emp emp)throws DataAccessException {
      empDao.createEmp(emp);
   }
   @Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.DEFAULT,readOnly = false )
   public int deleteEmp(Emp emp) throws DataAccessException {
      return empDao.deleteEmp(emp);
   }
   public Emp getEmp(int empno) throws DataAccessException {
      return empDao.getEmp(empno);
   }
   public List<Emp> getEmployeeDetails() throws DataAccessException {
      return empDao.getEmployeeDetails();
   }
   @Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.DEFAULT,readOnly = false )
   public int updateEmp(Emp emp) throws DataAccessException {
      return empDao.updateEmp(emp);
   }
}
```

```
public class BaseDao extends JdbcDaoSupport{
   @Autowired
   public DataSource dataSource;

   @PostConstruct
   public void init(){
      System.out.println("In base dao init method...");
      setDataSource(dataSource);
   }
 }

public interface EmpDao {
     public void createEmp(final Emp emp)throws DataAccessException ;
     public int updateEmp(final Emp emp)throws DataAccessException ;
     public int deleteEmp(final Emp emp)throws DataAccessException ;
     public Emp getEmp(final int empno)throws DataAccessException ;
     public List<Emp> getEmployeeDetails()throws DataAccessException ;
}
```

```
@Repository
public class EmpDaoImpl extends BaseDao implements EmpDao {
   public void createEmp(Emp emp) throws DataAccessException {
      Object[] values = new Object[] {
            emp.getEno(), emp.getEname(), emp.getSal(), emp.getComm(), emp.getJob() };
      getJdbcTemplate().update("insert into emp values(?,?,?,?,?)", values);
    }
   public int deleteEmp(Emp emp) throws DataAccessException {
      return getJdbcTemplate().update("delete from emp where eno = ?", new Object[] {emp.getEno() });
   }
   public int updateEmp(Emp emp) throws DataAccessException {
      return getJdbcTemplate().update("update emp set sal=? where eno=?", new Object[] {emp.getSal(), emp.getEno() });
   }
   public Emp getEmp(int empno) throws DataAccessException {
      return getJdbcTemplate().query("select eno,ename,sal,comm,job from emp where eno=?",
            new Object[] {empno }, new EmpMapper()).get(0);
   }
   public List<Emp> getEmployeeDetails() throws DataAccessException {
      String query = "select eno,ename,sal,comm,job from emp";
      List<Emp> list = getJdbcTemplate().query(query, new EmpMapper());
      return list;
   }
   private static final class EmpMapper implements RowMapper<Emp> {
      public Emp mapRow(ResultSet resultSet, int rowNum) throws SQLException {
         Emp emp = new Emp();
         emp.setEno(resultSet.getInt(1));
         emp.setEname(resultSet.getString(2));
         emp.setSal(resultSet.getDouble(3));
         emp.setComm(resultSet.getDouble(4));
         emp.setJob(resultSet.getString(5));
         return emp;
      }
   }
}
```

```
public class Emp {
    private int eno;
    private String ename;
    private double sal;
    private double comm;
    private String job;

    public Emp() {
        System.out.println("Employee entity constructor");
    }
     // Generate setters and getters methods fro above properties.
}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context"
	xmlns:tx="http://www.springframework.org/schema/tx" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:task="http://www.springframework.org/schema/task" xmlns:security="http://www.springframework.org/schema/security"
	xmlns:p="http://www.springframework.org/schema/p" xmlns:jee="http://www.springframework.org/schema/jee"
	xmlns:jms="http://www.springframework.org/schema/jms"
	xsi:schemaLocation="
	http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-3.0.xsd
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
  http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security-3.0.xsd
  http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-3.0.xsd
	http://www.springframework.org/schema/jee
	http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

	<context:component-scan
		base-package="com.apparao.commons,com.apparao.dao,com.apparao.service" />

	<bean id="dataSource" 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="oracle"></property>
	</bean>
</beans>
```

```
public class TestClient {
	public static void main(String[] args)throws Exception {
			ApplicationContext container=new ClassPathXmlApplicationContext("applicationContext.xml");
			Object o=container.getBean(com.apparao.service.EmpService.class);
			EmpService empService=(EmpService)o;
			Emp emp=new Emp();
			emp.setEname("Ramu");
			emp.setSal(1700.2);
			emp.setComm(100.1);
			emp.setEno(13);
			emp.setJob("Quality Analysist");
			//empService.createEmp(emp);
			//empService.deleteEmp(emp);
			//empService.updateEmp(emp);
			Emp e=empService.getEmp(19);
			System.out.println("Eno:"+e.getEno());
			 System.out.println("Ename:"+e.getEname());
			 System.out.println("Sal:"+e.getSal());
			 System.out.println("Comm:"+e.getComm());
			 System.out.println("Job:"+e.getJob());
			/*List<Emp> empList=empService.getEmployeeDetails();
			for(Emp e:empList){
			System.out.println("eno:"+e.getEno());
			System.out.println("ename:"+e.getEname());
			System.out.println("sal:"+e.getSal());
			System.out.println("comm:"+e.getComm());
			System.out.println("job:"+e.getJob());
			System.out.println("========================================");
			}*/
	}
}
```

HibernateDaoSupport and HibernateTemplate :

While using hibernate as a low-level data access API for implementing DAOs we can identify the following shortcomings:

1. Code Duplication
2. Resource Leakage
3. Error Handling

To solve the above problem spring introduces HibernateTemplate.

HibernateTemplate can be instantiate following three ways.

1. HibernateTemplate() : Constructs a new HibernateTemplate object. This constructor is provided to allow Java Bean style of instantiation. Note that when constructing an object using this constructor we need to use setSessionFactory() method to set the Hibernate SessionFactory before using any of the method of this object.
2. HibernateTemplate(SessionFactory): Constructs a new HibernateTemplate object initializing it with the given Hibernate SessionFactory used to create Session for executing the requested statements.

3. HibernateTemplate(SessionFactory, boolean):

Constructs a new HibernateTemplate object initializing it with the given Hibernate SessionFactory used to create Session for executing the requested statements, and boolean value describes the allowCreate flag.

Setting the allowCreate flag to “true” specifies HibernateTemplate to perform its own Session management instead of participating in a custom Hibernate current session context.

HibernateTemplate example:

Configuring the SessionFactory in applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	<context:component-scan
		base-package="com.apparao.commons,com.apparao.dao,com.apparao.service" />
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
		<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
		<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
		<property name="username" value="system"></property>
		<property name="password" value="oracle"></property>
	</bean>
	<bean id="sessionFactory"
		class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
		<property name="dataSource"><ref bean="dataSource" /></property>
		<property name="hibernateProperties">
			<props>
				<prop key="hibernate.dialect">
					org.hibernate.dialect.Oracle9Dialect
				</prop>
			</props>
		</property>
		<property name="annotatedClasses">
			<list><value>com.apparao.model.Emp</value></list>
		</property>
	</bean>
	<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
		<property name="sessionFactory" ref="sessionFactory" />
	</bean>
</beans>
```

```
@Repository
public class BaseDao extends HibernateDaoSupport{
   @Autowired
   public SessionFactory sessionFactory;
   @PostConstruct
    public void init(){
      System.out.println("in side init in base dao:"+sessionFactory);
      setSessionFactory(sessionFactory);
   }
}
public interface EmpDao {
     public void createEmp(final Emp emp);
     public int updateEmp(final Emp emp);
     public int deleteEmp(final Emp emp);
     public Emp getEmp(final int empno);
}
```

```
@Repository
public class EmpDaoImpl extends BaseDao implements EmpDao {
	public void createEmp(Emp emp) {
		getHibernateTemplate().save(emp);
	}
	public void deleteEmp(Emp emp) {
	   getHibernateTemplate().delete(emp);
	}
	public Emp getEmp(int empno) {
		Emp emp =(Emp)getHibernateTemplate().get(Emp.class, empno);
		return emp;
	}
	public void updateEmp(Emp emp) {
	   getHibernateTemplate().saveOrUpdate(emp);
	}
	public List<Object[]> getMinEmployeeDetails(){
		String query="select e.ename,e.sal from Employee e";
		return getHibernateTemplate().find(query);
	}
}
```