PART- 2

Spring-ORM

Spring ORM

- The Spring Framework provides integration with *Hibernate, JDO, Oracle TopLink, iBATIS SQL Maps* and *JPA*: in terms of resource management, DAO implementation support, and transaction strategies.
- For example for Hibernate, there is first-class support with lots of IoC convenience features, addressing many typical Hibernate integration issues.
- All of these support packages for O/R (Object Relational) mappers comply with Spring's generic transaction and DAO exception hierarchies.
- There are usually two integration styles: either using Spring's DAO 'templates' or coding DAOs against plain Hibernate/JDO/TopLink/etc APIs. In both cases, DAOs can be configured through Dependency Injection and participate in Spring's resource and transaction management.

Q) What is the traditional style of achieving persistence in java based enterprise application?

- Sending SQL statements to the Database using JDBC API

Q.) What are the limitations of the traditional approach?

- Application portability to the DB is lost. (Vendor lock: diff SQL statement for the db's)
- Mismatches between Object oriented data representation and relational data representation are not properly addressed.
- Requires the extensive knowledge of DB
- Too many steps to maintain transactions
- Manual operations on Result sets
- Need to tune your queries
- Need to implement caching manually.

Q.) what is an alternative for traditional approach?

- ORM (Object Relational mapping)
- Object Relational mapping is technique of mapping objected oriented data representation to that of relational data representation
- Through ORM technique persistence services (database) are provided to business layer in pure object oriented manner by overcoming all limitations of the traditional approach.

Q.) What are the ORM frameworks are there?

- Hibernate
- Toplink
- Ibatis
- JDO
- JPA(Java Persistence API)

Q.) Is Spring ORM an implementation of ORM (Object Relational Model) like Hibernate?

⇨ No, it is not ORM implementation

⇨ It is the one of the modules of spring framework, which is used to integrate any ORM into spring framework

Q.) Why to integrate ORM modules into spring framework?

There are some of the following common practices are there in all most all ORM implementation frameworks

1. Exception Handling Logic
2. Transaction Management Logic
3. Resource allocation Logic
4. Resource Releasing logic

Exception Handling Loigc

⇨ Spring Frame Work provides the **Fine Grind Exception Handling** mechanism to deal with DataBase i.e. it defines specific Exceptions for each and every problem that occurs while dealing with the DB.

⇨ **DataAccessException:** org.springframework.dao.DataAccessException is the **top most Exception** in spring DAO exception hierarchy.

⇨ In Spring DAO Exception Hierarchy we have a support (or **Spring ExceptionTranslator)** to Transform lo-level Data Access API Exceptions to the Spring DAO Exceptions.

⇨ **DataAccessException** is the child class of **RuntimeException** so all Spring DAO Exceptions are unchecked Exceptions

⇨ Spring provides **Declarative Exception Handling** Mechanism, means we can handle the Exceptions in the **"Spring–Configuration"** file.

Transaction Management Logic

⇨ Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the Java code level.

- ⇨ In either case, transaction semantics are handled for you, and prop transaction handling (rollback, etc) in case of exceptions is taken care of. As discussed below, you so get the benefit of being able to use and swap various transaction managers, without your Hiberna /JDO related code being affected:
- ⇨ For example, between local transactions and JTA, with the same fu services (such as declarative transactions) available in both scenarios.
- ⇨ As an additional benefit, JDBC-related code can fully integrate tran ctionally with the code you use to do O/R mapping.
- ⇨ Spring Framework supports Distributive Transaction Management.

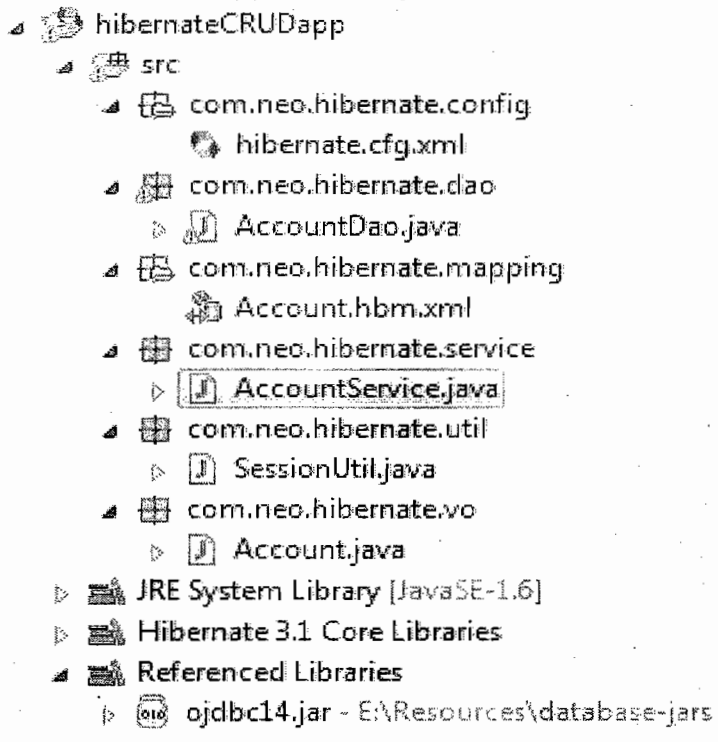
Resource allocation Logic

- ⇨ **XXXTemplate** provided by the spring which, abstract the DB Resou e Allocation Logic.

Resource Releasing logic

- ⇨ Spring provided **XXXTemplate**, which automatically releases the DB esources.

Q.) Develop Hibernate application, in which we can perform account creation, retrieve, update and delete?



ACCOUNT TABLE

ACNO	NAME	BAL
1003	sekharreddy	6500
1001	sekhar	9800
1002	somu	6899

hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.           "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.           "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5. <hibernate-configuration>
6. <session-factory>
7.
8.    <property name="dialect">org.hibernate.dialect.OracleDialect</property>
9.    <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
10.         <property name="connection.username">system</property>
11.         <property name="connection.password">tiger</property>
12.         <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver
13.                                                                    </property>
14.         <property name="connection.pool_size">9</property>
15.         <property name="show_sql">true</property>
16.         <property name="format_sql">true</property>
17.         <property name="use_sql_comments">true</property>
18.         <property name="hbm2ddl.auto">update</property>
```

```
19.         <mapping resource="com/neo/hibernate/mapping/Account.hbm.xml" />
20.
21.    </session-factory>
22.    </hibernate-configuration>
```

SessionUtil.java

```
1. package com.neo.hibernate.util;
2.
3. import org.hibernate.Session;
4. import org.hibernate.SessionFactory;
5. import org.hibernate.cfg.Configuration;
6.
7. public class SessionUtil {
8.     private static final String CONFIGURATION_FILE_LOCATION =
9.                                         "com/neo/hibernate/config/hibernate.cfg.xml";
10.         private static SessionFactory factory;
11.         private static final ThreadLocal<Session> tl = new ThreadLocal<Session>();
12.
13.         private SessionUtil() {
14.         }
15.
16.         static {
17.               try {
18.                     factory = new Configuration()
19.                               .configure(CONFIGURATION_FILE_LOCATION)
20.                               .buildSessionFactory();
21.               } catch (Exception e) {
22.                     e.printStackTrace();
23.               }
24.         }
25.
26.         public static Session getSession() {
27.               return factory.openSession();
28.         }
29.
30.         public static void closeSession(Session session) {
31.               if (session != null) {
32.                     session.close();
33.               }
34.         }
35.
36.         public static Session getCurrentSession() {
37.               Session session = tl.get();
38.               if (session == null) {
39.                     session = factory.openSession();
40.                     tl.set(session);
41.               }
42.               return session;
43.         }
44.
45.         public static void closeCurrentSession() {
46.               Session session = tl.get();
47.               tl.set(null);
48.               if (session != null) {
49.                     session.close();
```

```
50.                 }
51.         }
52.
53.   }
```

Account.java

```
1. package com.neo.hibernate.vo;
2.
3. public class Account {
4.     private int accno;
5.     private String name;
6.     private double balance;
7.     public int getAccno() {
8.           return accno;
9.     }
10.          public void setAccno(int accno) {
11.                this.accno = accno;
12.          }
13.          public String getName() {
14.                return name;
15.          }
16.          public void setName(String name) {
17.                this.name = name;
18.          }
19.          public double getBalance() {
20.                return balance;
21.          }
22.          public void setBalance(double balance) {
23.                this.balance = balance;
24.          }
25.
26.   }
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.
6.     <class name="com.neo.hibernate.vo.Account" table="ACCOUNT">
7.           <id name="accno" column="aCNO"></id>
8.           <property name="name" ></property>
9.           <property name="balance" column="BaL"></property>
10.
11.          </class>
12.
13.   </hibernate-mapping>
```

AccountDao.java

```
1. package com.neo.hibernate.dao;
2.
3. import org.hibernate.HibernateException;
4. import org.hibernate.Session;
5. import org.hibernate.SessionFactory;
```

```
6. import org.hibernate.Transaction;
7. import org.hibernate.cfg.Configuration;
8.
9. import com.neo.hibernate.util.SessionUtil;
10.   import com.neo.hibernate.vo.Account;
11.
12.   public class AccountDao {
13.         public Account get(int accno) {
14.
15.               Session session = null;
16.               Account account = null;
17.               try {
18.                     session = SessionUtil.getSession();
19.                     account = (Account) session.get(Account.class, accno);
20.               } catch (HibernateException e) {
21.                     e.printStackTrace();
22.               } finally {
23.                     SessionUtil.closeSession(session);
24.               }
25.
26.               return account;
27.         }
28.
29.         public void insert(Account account) {
30.
31.               Session session = null;
32.               try {
33.                     session = SessionUtil.getSession();
34.                     session.getTransaction().begin();
35.                     session.save(account);
36.                     session.getTransaction().commit();
37.               } catch (HibernateException e) {
38.                     session.getTransaction().rollback();
39.                     e.printStackTrace();
40.               } finally {
41.                     SessionUtil.closeSession(session);
42.               }
43.         }
44.
45.         public void update(Account account) {
46.               Session session = null;
47.               try {
48.                     session = SessionUtil.getSession();
49.                     session.getTransaction().begin();
50.                     session.update(account);
51.                     session.getTransaction().commit();
52.               } catch (HibernateException e) {
53.                     session.getTransaction().rollback();
54.                     e.printStackTrace();
55.               } finally {
56.                     SessionUtil.closeSession(session);
57.               }
58.         }
59.
60.         public void delete(Account account) {
```

```
61.                 Session session = null;
62.                 Transaction transaction = null;
63.                 try {
64.                       session = SessionUtil.getSession();
65.                       transaction = session.beginTransaction();
66.                       session.delete(account);
67.                       transaction.commit();
68.                 } catch (HibernateException e) {
69.                       transaction.rollback();
70.                       e.printStackTrace();
71.                 } finally {
72.                       SessionUtil.closeSession(session);
73.                 }
74.           }
75.
76.     }
```

AccountService.java

```
1. package com.neo.hibernate.service;
2. import com.neo.hibernate.dao.AccountDao;
3. import com.neo.hibernate.vo.Account;
4.
5. public class AccountService {
6.     public static void main(String[] args) {
7.           AccountDao dao = new AccountDao();
8.
9.           // Retrieve Account
10.                Account rAccount = dao.get(1001);
11.                System.out.println("Account details ....");
12.                System.out.println("Accno : " + rAccount.getAccno());
13.                System.out.println("Name : " + rAccount.getName());
14.                System.out.println("Balance : " + rAccount.getBalance());
15.
16.                // Create Account
17.                Account cAccount = new Account();
18.                cAccount.setAccno(1004);
19.                cAccount.setName("somasekhar");
20.                cAccount.setBalance(6899);
21.                dao.insert(cAccount);
22.                System.out.println("Account created successfully");
23.
24.                // Update Account
25.                Account uAccount = new Account();
26.                uAccount.setAccno(1002);
27.                uAccount.setName("l.n.rao");
28.                uAccount.setBalance(4500);
29.                dao.update(uAccount);
30.                System.out.println("Account updated successfully");
31.
32.                // Delete Account
33.                Account dAccount = new Account();
34.                dAccount.setAccno(1003);
35.                dao.delete(dAccount);
36.                System.out.println("Account is deleted successfully");
```

```
37.         }
38.  }
```

After Execution ACCOUNT TABLE :

ACNO	NAME	BAL
1004	somasekhar	6899
1001	sekhar	9800
1002	[illegible]	4500

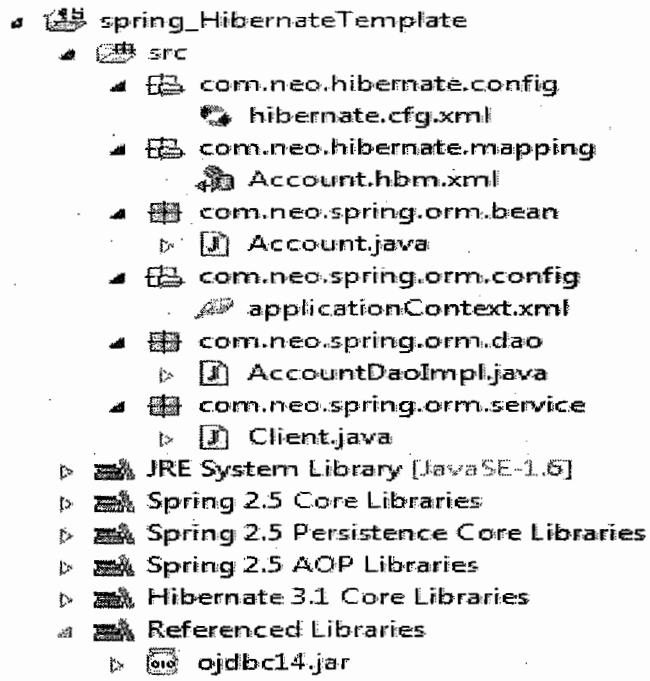
Spring-HibernateIntegration

- In spring-hibernate integration, we have to add both spring and hibernate capabilities.
- In hibernate, we have configuration file(hibernate.cfg.xml) and mapping file (Account.hbm.xml).
- In hibernate.cfg.xml we provide the Database details and Dialect class and mapping file information.
- So, there will be two configuration files i.e., spring configuration file(applicationContext.xml) and hibernate configuration file(hibernate.cfg.xml).
- But in general while integrating spring and hibernate we won't write hibernate.cfg.xml, we give this file(hibernate.cfg.xml) information also in the spring configuration file. it is recommend. So Database details(ur username, password, driver class), Dialect class, other hibernate properties(show_sql, hbm2.ddl.auto ...etc.), hibernate mapping file information and some in the spring configuration file.
- We will configure all these information to **LocalSessionFacotyBean** of spring framework.
- Note that switching from a local Jakarta Commons DBCP `BasicDataSource` to a JNDI-locate `DataSource` (usually managed by an application server) is just a matter of configuration:

```
<beans>

  <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/myds"/>
  </bean>

</beans>
```



hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.           "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.           "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.
6. <hibernate-configuration>
7.
8. <session-factory>
9.       <property name="hibernate.dialect">
10.                          org.hibernate.dialect.OracleDialect</property>
11.    <property name="connection.url">
12.                jdbc:oracle:thin:@localhost:1521:XE</property>
13.    <property name="connection.username">system</property>
14.    <property name="connection.password">tiger</property>
15.    <property name="connection.driver_class">
16.                oracle.jdbc.driver.OracleDriver</property>
17.    <property name="hibernate.show_sql">true</property>
18.    <mapping resource="com/neo/hibernate/mapping/Account.hbm.xml" />
19.    </session-factory>
20.
21.    </hibernate-configuration>
```

NOTE : this file is not required if we use, DataSource in spring configuration file itself.

Account.hbm.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3.  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.  <hibernate-mapping>
5.      <class name="com.neo.spring.orm.bean.Account" table="ACCOUNT">
6.              <id name="accno" column="ACCNO"></id>
7.              <property name="name" column="NAME"></property>
8.              <property name="balance" column="BAL"></property>
9.  </class>
10.
11. </hibernate-mapping>
```

applicationContext.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.  xmlns:p="http://www.springframework.org/schema/p"
5.      xsi:schemaLocation="http://www.springframework.org/schema/beans
6.  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.      <bean id="ds"
9.              class="org.springframework.jdbc.datasource.DriverManagerDataSource">
10.             <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
11.             <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"></property>
12.             <property name="username" value="system"></property>
13.             <property name="password" value="tiger"></property>
```

```
14.     </bean>
15.
16.     <bean id="lsfb"
17.             class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
18.
19.     <!--
20.     <property name="configLocations" value="com/neo/hibernate/config/hibernate.cfg.xml" />
21.     -->
22.     <property name="dataSource" ref="ds"></property>
23.     <property name="mappingResources">
24.             <list>
25.                     <value>com/neo/hibernate/mapping/Account.hbm.xml</value>
26.             </list>
27.     </property>
28.
29.     <property name="hibernateProperties">
30.             <props>
31.                     <prop key="hibernate.dialect">org.hibernate.dialect.OracleDialect</prop>
32.                     <prop key="hibernate.show_sql">true</prop>
33.             </props>
34.     </property>
35.     </bean>
36.
37.     <bean id="ht" class="org.springframework.orm.hibernate3.HibernateTemplate">
38.             <constructor-arg ref="lsfb"></constructor-arg>
39.     </bean>
40.
41.     <bean id="daoImpl" class="com.neo.spring.orm.dao.AccountDaoImpl">
42.             <property name="hibernateTemplate" ref="ht"></property>
43.     </bean>
44.
45. </beans>
```

Account.java

```
1.  package com.neo.spring.orm.bean;
2.  public class Account {
3.      private int accno;
4.      private String name;
5.      private double balance;
6.
7.      // setters & getters
8.  }
```

AccountDaoImpl.java

```
1.  package com.neo.spring.orm.dao;
2.  import org.springframework.orm.hibernate3.HibernateTemplate;
3.
4.  import com.neo.spring.orm.bean.Account;
5.
```

```
6.  public class AccountDaoImpl {
7.
8.      public void insert(Account account){
9.              hibernateTemplate.save(account);
10.             System.out.println("Account is inserted successfully");
11.
12.     }
13.
14.     public Account get(int accno){
15.             return (Account) hibernateTemplate.get(Account.class, accno);
16.     }
17.
18.     public void update(Account account){
19.             hibernateTemplate.update(account);
20.             System.out.println("Account is updated successfully");
21.     }
22.
23.
24.     public void delete(int accno){
25.             Account account = new Account();
26.             account.setAccno(accno);
27.             hibernateTemplate.delete(account);
28.             System.out.println("Account is deleted successfully");
29.     }
30.
31.     private HibernateTemplate hibernateTemplate;
32.     public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
33.             this.hibernateTemplate = hibernateTemplate;
34.     }
35.
36. }
```

Client.java

```
1.  package com.neo.spring.orm.service;
2.
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6.  import com.neo.spring.orm.bean.Account;
7.  import com.neo.spring.orm.dao.AccountDaoImpl;
8.
9.  public class Client {
10.     private static ApplicationContext context = new ClassPathXmlApplicationContext(
11.                     "com/neo/spring/orm/config/applicationContext.xml");
12.
13.     public static void main(String[] args) {
14.             AccountDaoImpl daoImpl = (AccountDaoImpl) context.getBean("daoImpl");
15.
16.             // insert
17.             Account accountA = new Account();
```

```
18.             accountA.setAccno(8001);
19.             accountA.setName("somasekhar");
20.             accountA.setBalance(5978.0);
21.             daoImpl.insert(accountA);
22.
23.             // select
24.             Account accountB = daoImpl.get(8004);
25.             System.out.println(accountB.getAccno());
26.             System.out.println(accountB.getName());
27.             System.out.println(accountB.getBalance());
28.
29.             // update
30.             Account accountC = daoImpl.get(8005);
31.             accountC.setName("yellareddy-2");
32.             accountC.setBalance(8600.0);
33.             daoImpl.update(accountC);
34.
35.             // delete
36.             daoImpl.delete(8002);
37.
38.     }
39. }
```

Before application execution ACCOUNT table :

ACCNO	NAME	BAL
8002	sekhar	6790
8004	kesavareddy	9999
8005	yellareddy	8690

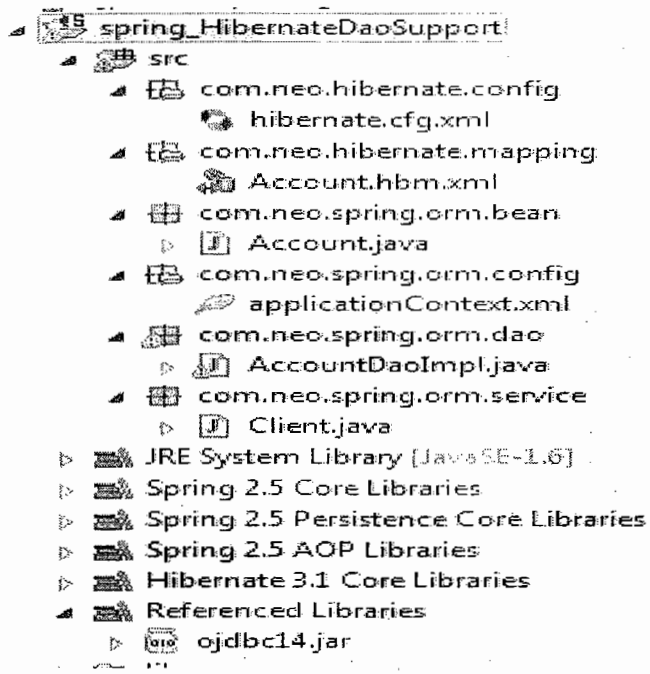
Before application execution ACCOUNT table :

ACCNO	NAME	BAL
8001	somasekhar	5978
8004	kesavareddy	9999
8005	yellareddy-2	8600

Q.) What is *HibernateDaoSupport* ?

- ⇨ While working with hibernate, into all dao classes we need to inject **HibernateTemplate** object. So injecting **HibernateTemplate** object, logic is repeating in all the dao classes. So spring people has given one abstract class called **HibernateDaoSupport**, which contains this common **HibernateTemplate** object injection logic.
- ⇨ So while writing Dao class it is better to extend **HibernateDaoSupport**, so that we no need to write injection logic of **HibernateTemplate.**
- ⇨ When we need **HibernateTemplate** object in dao class just we call **getHibernateTemplate().** So, by extending this class we can get **HibernateTemplate**, on that we can perform our operations.
- ⇨ One more advantage of **HibernateDaoSupport** is we can directly inject **LocalSessionFactoryBean** to dao class instead of **HibernateTemplate**

Q.) Develop an application where we can use *HibernateDaoSupport* instead of *HibernateTemplate*?



hibernate.cfg.xml

```
<-- SAME AS ABOVE -->
```

NOTE : this file is not required if we use, DataSource in spring configuration file itself.

Account.hbm.xml

```
<-- SAME AS ABOVE -->
```

applicationContext.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.  xmlns:p="http://www.springframework.org/schema/p"
5.      xsi:schemaLocation="http://www.springframework.org/schema/beans
6.  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.      <bean id="ds"
9.              class="org.springframework.jdbc.datasource.DriverManagerDataSource">
10.             <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
11.             <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"></property>
12.             <property name="username" value="system"></property>
13.             <property name="password" value="tiger"></property>
14.     </bean>
15.
16.     <bean id="lsfb"
17.             class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
18.
19.     <!--
```

```
20.     <property name="configLocations" value="com/neo/hibernate/config/hibernate.cfg.xml" />
21.     -->
22.     <property name="dataSource" ref="ds"></property>
23.     <property name="mappingResources">
24.             <list>
25.                     <value>com/neo/hibernate/mapping/Account.hbm.xml</value>
26.             </list>
27.     </property>
28.
29.     <property name="hibernateProperties">
30.             <props>
31.                     <prop key="hibernate.dialect">org.hibernate.dialect.OracleDialect</prop>
32.                     <prop key="hibernate.show_sql">true</prop>
33.             </props>
34.     </property>
35.     </bean>
36.
37.     <bean id="daoImpl" class="com.neo.spring.orm.dao.AccountDaoImpl">
38.             <property name="sessionFactory" ref="lsfb"></property>
39.     </bean>
40.
41. </beans>
```

Account.java

```
<-- SAME AS ABOVE -->
```

AccountDaoImpl.java

```
1.  package com.neo.spring.orm.dao;
2.  import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
3.
4.  import com.neo.spring.orm.bean.Account;
5.
6.  public class AccountDaoImpl extends HibernateDaoSupport{
7.
8.      public void insert(Account account){
9.              getHibernateTemplate().save(account);
10.             System.out.println("Account is inserted successfully");
11.
12.     }
13.
14.     public Account get(int accno){
15.             return (Account) getHibernateTemplate().get(Account.class, accno);
16.     }
17.
18.     public void update(Account account){
19.             getHibernateTemplate().update(account);
20.             System.out.println("Account is updated successfully");
21.     }
22.
23.
```

```
24.     public void delete(int accno){
25.             Account account = new Account();
26.             account.setAccno(accno);
27.             getHibernateTemplate().delete(account);
28.             System.out.println("Account is deleted successfully");
29.     }
30.
31. }
```

Client.java

```
<-- SAME AS ABOVE -->
```

Before application execution ACCOUNT table :

ACCNO	NAME	BAL
8002	sekhar	6790
8004	kesavareddy	9999
8005	yellareddy	8600

Before application execution ACCOUNT table :

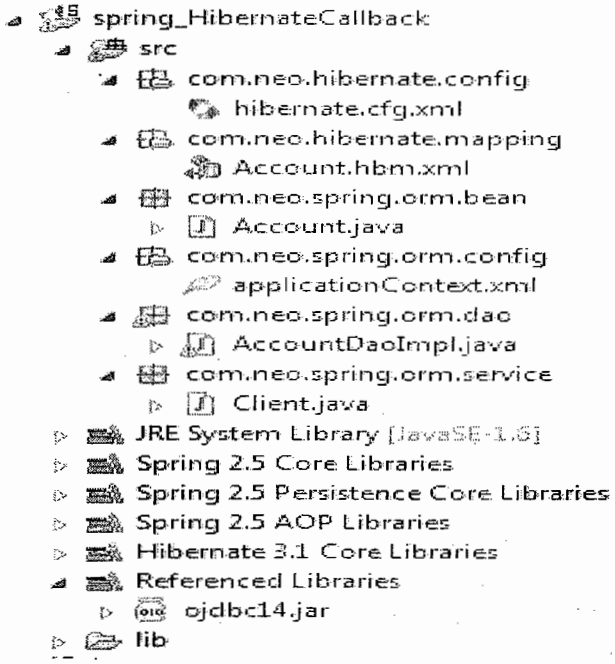
ACCNO	NAME	BAL
8001	somasekhar	5978
8004	kesavareddy	9999
8005	yellareddy-2	8600

Q) Why we are going for Callback mechanism?
Ans: When we are unable to implement underlying framework or technology functionality using xxxTemplate then we need to go for Callback mechanism.

Q) What we can achieve with xxxCallback?
Ans: Any specific functionality of underlying framework or technology can be implemented. Because in the Callback mechanism they will pass underlying framework resource (Connection, Session, SqlMap ... etc).

HibernateCallback:
When we are not getting the specific functionality of hibernate by using HibernateTemplate those functionalities we can achieve by using HibernateCallback.

Q.) Develop an application where I can use where we can use *HibernateCallback*?



hibernate.cfg.xml

```
<-- SAME AS ABOVE -->
```

NOTE : this file is not required if we use, DataSource in spring configuration file itself.

Account.hbm.xml

```
<-- SAME AS ABOVE -->
```

applicationContext.xml

```
<-- SAME AS ABOVE -->
```

Account.java

```
<-- SAME AS ABOVE -->
```

AccountDaoImpl.java

```
1.  package com.neo.spring.orm.dao;
2.  import java.sql.SQLException;
3.  import java.util.List;
4.
5.  import org.hibernate.HibernateException;
6.  import org.hibernate.Session;
7.  import org.springframework.orm.hibernate3.HibernateCallback;
8.  import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
9.
10. import com.neo.spring.orm.bean.Account;
11.
12. public class AccountDaoImpl  extends HibernateDaoSupport {
13.      public List<Account> get() {
14.
15.              return (List<Account>) getHibernateTemplate()
16.                              .execute(new AccountHibernateCallback());
17.      }
```

```
18.
19.     private class AccountHibernateCallback implements HibernateCallback {
20.         @Override
21.         public Object doInHibernate(Session session) throws HibernateException, SQLException {
22.             List<Account> accounts = session.createCriteria(Account.class).list();
23.             return accounts;
24.         }
25.     }
26.
27. }
```

Client.java

```
1.  package com.neo.spring.orm.service;
2.
3.  import java.util.List;
4.
5.  import org.springframework.context.ApplicationContext;
6.  import org.springframework.context.support.ClassPathXmlApplicationContext;
7.
8.  import com.neo.spring.orm.bean.Account;
9.  import com.neo.spring.orm.dao.AccountDaoImpl;
10.
11. public class Client {
12.     private static ApplicationContext context = new ClassPathXmlApplicationContext(
13.                     "com/neo/spring/orm/config/applicationContext.xml");
14.
15.     public static void main(String[] args) {
16.         AccountDaoImpl daoImpl = (AccountDaoImpl) context.getBean("daoImpl");
17.
18.
19.         // select
20.         List<Account> accounts  = daoImpl.get();
21.         for(Account account : accounts){
22.             System.out.println(account.getAccno());
23.             System.out.println(account.getName());
24.             System.out.println(account.getBalance());
25.         }
26.
27.     }
28. }
```

iBATIS

What is iBatis ?

- A JDBC Framework
  - Developers write SQL, iBATIS executes it using JDBC.
  - No more try/catch/finally/try/catch.
- An SQL Mapper
  - Automatically maps object properties to prepared statement parameters.
  - Automatically maps result sets to objects.
  - Support for getting rid of N+1 queries.
- A Transaction Manager
  - iBATIS will provide transaction management for database operations if no other transaction manager is available.
  - iBATIS will use external transaction management (Spring, EJB CMT, etc.) if available.
- Great integration with Spring, but can also be used without Spring (the Spring folks were early supporters of iBATIS).

What isn't iBATIS ?

- An ORM
  - Does not generate SQL
  - Does not have a proprietary query language
  - Does not know about object identity
  - Does not transparently persist objects
  - Does not build an object cache

iBATIS Design Philosophies:

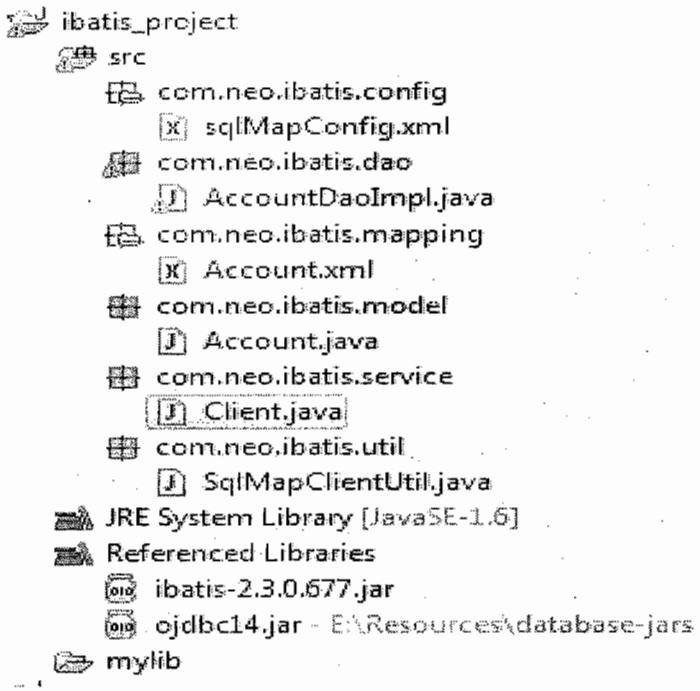
iBATIS comes with the following design philosophies:

- **Simplicity:** iBATIS is widely regarded as being one of the simplest persistence frameworks available today.
- **Fast Development:** iBATIS's philosophy is to do all it can to facilitate hyper-fast development.
- **Portability:** iBATIS can be implemented for nearly any language or platform like Java, Ruby, and C# for Microsoft .NET.
- **Independent Interfaces:** iBATIS provides database-independent interfaces and APIs that help the rest of the application remain independent of any persistence-related resources,
- **Open source:** iBATIS is free and an open source software.

Advantages of IBATIS

Here are few advantages of using IBATIS:

- Suppports Stored procedures: iBATIS encapsulates SQL in the form of stored procedures so that business logic is kept out of the database, and the application is easier to deploy and test, and is more portable.
- Supports Inline SQL: No precompiler is needed, and you have full access to all of the features of SQL.
- Supports Dynamic SQL: iBATIS provides features for dynamically building SQL queries based on parameters.
- Supports O/RM: iBATIS supports many of the same features as an O/RM tool, such as lazy loading, join fetching, caching, runtime code generation, and inheritance



sqlMapConfig.xml

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE sqlMapConfig PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
3. "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
4. <sqlMapConfig>
5.     <settings useStatementNamespaces="true" />
6.     <transactionManager type="JDBC">
7.   <dataSource type="SIMPLE">
8.           <property name="JDBC.Driver" value="oracle.jdbc.driver.OracleDriver" />
9.          <property name="JDBC.ConnectionURL"
10.                          value="jdbc:oracle:thin:@localhost:1521:xe" />
11.   <property name="JDBC.Username" value="system" />
12.   <property name="JDBC.Password" value="tiger" />
13.
14.   <!-- these properties are optional -->
15.   <property name="Pool.MaximumActiveConnections" value="10" />
16.   <property name="Pool.MaximumIdleConnections" value="5" />
17.   <property name="Pool.MaximumCheckoutTime" value="120000" />
18.   <property name="Pool.TimeToWait" value="10000" />
19.   <property name="Pool.PingConnectionsOlderThan" value="0" />
20.   <property name="Pool.PingConnectionsNotUsedFor" value="0" />
21.   <!-- these properties are optional -->
```

```
22.   </dataSource>
23.   </transactionManager>
24.   <sqlMap resource="com/neo/ibatis/mapping/Account.xml" />
25.   </sqlMapConfig>
```

Account.xml

```
26.   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
27.   <!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
28.   "http://www.ibatis.com/dtd/sql-map-2.dtd">
29.   <sqlMap namespace="accountNamespace">
30.               <insert id="insert" parameterClass="com.neo.ibatis.model.Account">
31.                     INSERT INTO
32.                     ACCOUNT(ACCNO,NAME,BAL)
33.                     VALUES(#accno#, #name#, #balance#)
34.               </insert>
35.
36.   <delete id="delete" parameterClass="integer">
37.         DELETE FROM ACCOUNT WHERE
38.         ACCNO=#accno#
39.   </delete>
40.
41.   <update id="update" parameterClass="com.neo.ibatis.model.Account">
42.         UPDATE ACCOUNT SET
43.         NAME=#name#,
44.         BAL=#balance# WHERE ACCNO=#accno#
45.   </update>
46.
47.   <select id="myselect" parameterClass="integer"
48.               resultClass="com.neo.ibatis.model.Account">
49.         SELECT ACCNO AS accno, NAME AS name,
50.         BAL AS balance FROM
51.         ACCOUNT
52.         WHERE ACCNO=#accno#
53.   </select>
54.
55.   <!--
56.   <select id="myselects" resultClass="com.neo.ibatis.model.Account">
57.         SELECT ACCNO AS accno, NAME AS name, BAL AS balance FROM ACCOUNT
58.   </select>
59.   -->
60.
61.   <resultMap id="resultId" class="com.neo.ibatis.model.Account">
62.         <result property="accno" column="ACCNO" columnIndex="1" />
63.         <result property="name" column="NAME" columnIndex="2" />
64.         <result property="balance" column="BAL" columnIndex="3" />
65.   </resultMap>
66.
67.   <select id="myselects" resultMap="resultId">
68.         SELECT ACCNO , NAME, BAL FROM ACCOUNT
69.   </select>
70.
71.   <parameterMap id="accountProcParams" class="map" >
72.         <parameter property="P_ACCNO"     jdbcType="INTEGER"
73.                           javaType="java.lang.Integer" mode="IN"/>
74.         <parameter property="P_NAME"      jdbcType="VARCHAR"
75.                           javaType="java.lang.String" mode="IN" />
```

```
76.         <parameter property="P_BAL"      jdbcType="DOUBLE"
77.                         javaType="java.lang.Double" mode="IN"/>
78.         <parameter property="P_RESULT"   jdbcType="VARCHAR"
79.                         javaType="java.lang.String" mode="OUT" />
80.
81.   </parameterMap>
82.
83.   <procedure id="myprocedure" parameterMap="accountProcParams" >
84.         {call ACCOUNT_INSERT_PROC(?,?,?,?)}
85.   </procedure>
86.
87.   </sqlMap>
```

Account.java

```
1. package com.neo.ibatis.model;
2.
3. public class Account {
4.    private int accno;
5.    private String name;
6.    private double balance;
7.     // setters & getters
8. }
```

SqlMapClientUtil.java

```
1. package com.neo.ibatis.util;
2.
3. import java.io.Reader;
4.
5. import com.ibatis.common.resources.Resources;
6. import com.ibatis.sqlmap.client.SqlMapClient;
7. import com.ibatis.sqlmap.client.SqlMapClientBuilder;
8.
9. public class SqlMapClientUtil {
10.         private static SqlMapClient mapClient;
11.         static{
12.               try {
13.                     Reader reader = Resources.getResourceAsReader(
14.                                 "com/neo/ibatis/config/sqlMapConfig.xml");
15.                     mapClient =
16.                           SqlMapClientBuilder.buildSqlMapClient(reader);
17.               } catch (Exception e) {
18.                     e.printStackTrace();
19.               }
20.         }
21.
22.         public static SqlMapClient getSqlMapClient(){
23.               return mapClient;
24.         }
25.
26.   }
```

AccountDaoImpl.java

```
1. package com.neo.ibatis.dao;
2.
3. import java.io.IOException;
```

```
4. import java.sql.SQLException;
5. import java.util.HashMap;
6. import java.util.List;
7. import java.util.Map;
8.
9. import com.ibatis.sqlmap.client.SqlMapClient;
10.    import com.neo.ibatis.model.Account;
11.    import com.neo.ibatis.util.SqlMapClientUtil;
12.
13.    public class AccountDaoImpl {
14.    public void insert(Account account) throws IOException{
15.           try {
16.                 SqlMapClient mapClient = SqlMapClientUtil.getSqlMapClient();
17.                 mapClient.insert("accountNamespace.insert", account);
18.           } catch (SQLException e) {
19.                 e.printStackTrace();
20.           }
21.    }
22.
23.
24.    public void delete(int accno){
25.           try {
26.                 SqlMapClient mapClient = SqlMapClientUtil.getSqlMapClient();
27.                 mapClient.delete("accountNamespace.delete", accno);
28.           } catch (Exception e) {
29.                 e.printStackTrace();
30.           }
31.    }
32.
33.    public void update(Account account){
34.           try {
35.                 SqlMapClientUtil.getSqlMapClient().update(
36.                       "accountNamespace.update", account);
37.           } catch (Exception e) {
38.                 e.printStackTrace();
39.           }
40.    }
41.
42.    public Account get(int accno){
43.           Account account = null;
44.           try {
45.                 account = (Account)SqlMapClientUtil.getSqlMapClient().
46.                       queryForObject("accountNamespace.myselect",accno);
47.           } catch (Exception e) {
48.                 e.printStackTrace();
49.           }
50.           return account;
51.    }
52.
53.    public List<Account> get(){
54.           List<Account> accounts = null;
55.           try {
56.                 accounts =
57.                       (List<Account>)SqlMapClientUtil.getSqlMapClient().
58.                             queryForList("accountNamespace.myselects");
59.           } catch (Exception e) {
```

```
60.                 e.printStackTrace();
61.           }
62.           return accounts;
63.     }
64.
65.     public void callProcedure(Account account){
66.           Map<String, Object> map = new HashMap<String, Object>();
67.           map.put("P_ACCNO", account.getAccno());
68.           map.put("P_NAME", account.getName());
69.           map.put("P_BAL", account.getBalance());
70.           try {
71.                 SqlMapClientUtil.getSqlMapClient().
72.                       queryForObject("accountNamespace.myprocedure",map);
73.                 System.out.println("Procedure execution status : "
74.                                                   +map.get("P_RESULT"));
75.           } catch (Exception e) {
76.                 e.printStackTrace();
77.           }
78.     }
79.     }
```

Client.java

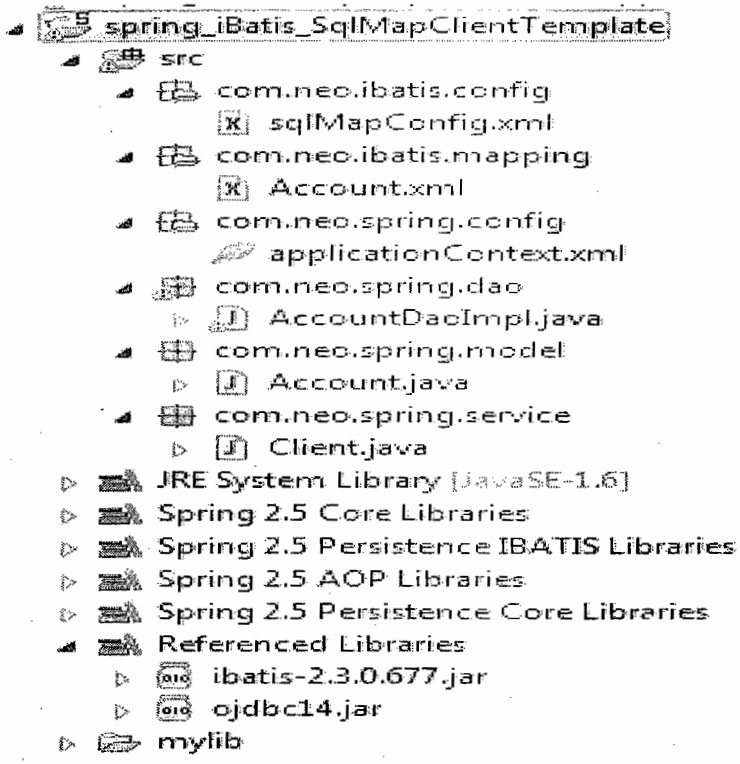
```
1. package com.neo.ibatis.service;
2.
3. import java.io.IOException;
4. import java.util.List;
5.
6. import com.neo.ibatis.dao.AccountDaoImpl;
7. import com.neo.ibatis.model.Account;
8.
9. public class Client {
10.         public static void main(String[] args) throws IOException {
11.               AccountDaoImpl impl = new AccountDaoImpl();
12.               // Insert
13.               Account accountA = new Account();
14.               accountA.setAccno(8002);
15.               accountA.setName("sekhar");
16.               accountA.setBalance(9569.0);
17.               impl.insert(accountA);
18.               sop("Account inserted");
19.
20.               // Delete
21.               impl.delete(8001);
22.               sop("Account deleted");
23.
24.               // Retrieve
25.               Account account = impl.get(8004);
26.               sop("Account Detials are...");
27.               sop(account.getAccno());
28.               sop(account.getName());
29.               sop(account.getBalance());
30.
31.               // Update
32.               Account accountB = new Account();
33.               accountB.setAccno(8005);
```

```
34.                accountB.setName("sekhar-updated");
35.                accountB.setBalance(678.9);
36.                impl.update(accountB);
37.                sop("Account updated");
38.
39.                // Retrieve multiple
40.                List<Account> accounts = impl.get();
41.                sop("Account Detials are...");
42.                for (Account accountC : accounts) {
43.                      sop(accountC.getAccno());
44.                      sop(accountC.getName());
45.                      sop(accountC.getBalance());
46.                }
47.
48.                Account accountD = new Account();
49.                accountD.setAccno(6002);
50.                accountD.setName("sekhar");
51.                accountD.setBalance(9569.0);
52.                impl.callProcedure(accountD);
53.
54.          }
55.
56.          public static void sop(Object object) {
57.                System.out.println(object);
58.          }
59.    }
```

Spring Framework with iBatis – Integration

Place the ibatis-2.3.0.677.jar in the application's classpath. At minimum, you must define and place the following three configuration files in the classpath

- Spring config - This file defines the database connection parameters, the location of the SQL Map config file, and one or more Spring beans for use within the application. (applicationContext.xml).
- SQL Map config - This file defines any iBATIS-specific configuration settings that you may need and declares the location for any SQL Map files that should be accessible through this config file. (SqlMapConfig.xml)
- SQL Map(s) ï¿½ One or more SQL Map files are declared in the SQL Map config and typically mapped to a single business entity within the application ,often represented by a single Java class (domainObject.xml).



Account.java

```
1.  package com.neo.spring.model;
2.
3.  public class Account {
4.      private int accno;
5.      private String name;
6.      private double balance;
7.
8.  //getters & setters
9.
10. }
```

sqlMapConfig.xml

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.  <!DOCTYPE sqlMapConfig PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
3.                                  "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
4.  <sqlMapConfig>
5.      <settings useStatementNamespaces="true" />
6.      <sqlMap resource="com/neo/ibatis/mapping/Account.xml" />
7.  </sqlMapConfig>
```

Account.xml

```
1.  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2.  <!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
3.  "http://www.ibatis.com/dtd/sql-map-2.dtd">
4.  <sqlMap namespace="accountNamespace">
5.
```

```
6.    <typeAlias alias="accountAlias" type="com.neo.spring.model.Account"/>
7.
8.    <insert id="insert" parameterClass="accountAlias">
9.          INSERT INTO
10.         ACCOUNT(ACCNO,NAME,BAL)
11.         VALUES(#accno#, #name#, #balance#)
12.   </insert>
13.
14.   <delete id="delete" parameterClass="integer">
15.         DELETE FROM ACCOUNT WHERE
16.         ACCNO=#accno#
17.   </delete>
18.
19.   <update id="update" parameterClass="accountAlias">
20.         UPDATE ACCOUNT SET
21.         NAME=#name#,
22.         BAL=#balance# WHERE ACCNO=#accno#
23.   </update>
24.
25.   <select id="myselect" parameterClass="integer"
26.         resultClass="accountAlias">
27.         SELECT ACCNO AS accno, NAME AS name,
28.         BAL AS balance FROM
29.         ACCOUNT
30.         WHERE ACCNO=#accno#
31.   </select>
32.
33.   <!--
34.         <select id="myselects" resultClass="accountAlias">
35.         SELECT ACCNO AS accno, NAME AS name, BAL AS balance FROM ACCOUNT
36.         </select>
37.   -->
38.
39.   <resultMap id="resultId" class="accountAlias">
40.         <result property="accno" column="ACCNO" columnIndex="1" />
41.         <result property="name" column="NAME" columnIndex="2" />
42.         <result property="balance" column="BAL" columnIndex="3" />
43.   </resultMap>
44.
45.   <select id="myselects" resultMap="resultId">
46.         SELECT ACCNO , NAME, BAL FROM
47.         ACCOUNT
48.   </select>
49.
50.   <parameterMap id="accountProcParams" class="map" >
51.         <parameter property="P_ACCNO"       jdbcType="INTEGER"
52.                                                javaType="java.lang.Integer" mode="IN"/>
53.         <parameter property="P_NAME"        jdbcType="VARCHAR"
54.                                                javaType="java.lang.String" mode="IN" />
55.         <parameter property="P_BAL"  jdbcType="DOUBLE"
```

```
                                                    javaType="java.lang.Double"  mode="IN"/>
        <parameter property="P_RESULT"      jdbcType="VARCHAR"
                                                    javaType="java.lang.String" mode="OUT" />

    </parameterMap>

    <procedure id="myprocedure" parameterMap="accountProcParams" >{call
                                                    ACCOUNT_INSERT_PROC(?,?,?,?)}</procedure>
</sqlMap>
```

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:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="ds"
        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="tiger"></property>
    </bean>

    <bean id="smcfb" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="dataSource" ref="ds"></property>
        <property name="configLocation"  value="com/neo/ibatis/config/sqlMapConfig.xml" />
    </bean>

    <bean id="smct" class="org.springframework.orm.ibatis.SqlMapClientTemplate">
        <property name="sqlMapClient" ref="smcfb"></property>
    </bean>

    <bean id="daoImpl" class="com.neo.spring.dao.AccountDaoImpl">
        <property name="sqlMapClientTemplate" ref="smct"></property>
    </bean>

</beans>
```

AccountDaoImpl.java

```
package com.neo.spring.dao;

import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

```

```
8.  import org.springframework.orm.ibatis.SqlMapClientTemplate;
9.
10. import com.neo.spring.model.Account;
11.
12. public class AccountDaoImpl {
13.     public void insert(Account account) throws IOException{
14.                     sqlMapClientTemplate.insert("accountNamespace.insert", account);
15.     }
16.
17.
18.     public void delete(int accno){
19.             sqlMapClientTemplate.delete("accountNamespace.delete", accno);
20.     }
21.
22.     public void update(Account account){
23.             sqlMapClientTemplate.update("accountNamespace.update", account);
24.     }
25.
26.     public Account get(int accno){
27.             return
28.             (Account)sqlMapClientTemplate.queryForObject("accountNamespace.myselect",accno);
29.     }
30.
31.     public List<Account> get(){
32.             return
33.                 (List<Account>)sqlMapClientTemplate.queryForList("accountNamespace.myselects");
34.     }
35.
36.     public void callProcedure(Account account){
37.             Map<String, Object> map = new HashMap<String, Object>();
38.             map.put("P_ACCNO", account.getAccno());
39.             map.put("P_NAME", account.getName());
40.             map.put("P_BAL", account.getBalance());
41.             sqlMapClientTemplate.queryForObject("accountNamespace.myprocedure",map);
42.             System.out.println("Account procedure execution status : "+map.get("P_RESULT"));
43.     }
44.
45.     private SqlMapClientTemplate sqlMapClientTemplate;
46.     public void setSqlMapClientTemplate(
47.                     SqlMapClientTemplate sqlMapClientTemplate) {
48.             this.sqlMapClientTemplate = sqlMapClientTemplate;
49.     }
50. }
```

Client.java

```
1.  package com.neo.spring.service;
2.
3.  import java.io.IOException;
4.  import java.util.List;
5.
```

```
6.  import org.springframework.context.ApplicationContext;
7.  import org.springframework.context.support.ClassPathXmlApplicationContext;
8.
9.  import com.neo.spring.dao.AccountDaoImpl;
10. import com.neo.spring.model.Account;
11.
12. public class Client {
13.     private static ApplicationContext context = new ClassPathXmlApplicationContext(
14.     "com/neo/spring/config/applicationContext.xml");
15.
16.     public static void main(String[] args) throws IOException {
17.             AccountDaoImpl impl = (AccountDaoImpl)context.getBean("daoImpl");
18.             // Insert
19.             Account accountA = new Account();
20.             accountA.setAccno(8005);
21.             accountA.setName("sekhar");
22.             accountA.setBalance(9569.0);
23.             impl.insert(accountA);
24.             sop("Account inserted");
25.
26.             // Delete
27.             impl.delete(8001);
28.             sop("Account deleted");
29.
30.             // Retrieve
31.             Account account = impl.get(8002);
32.             sop("Account Detials are...");
33.             sop(account.getAccno());
34.             sop(account.getName());
35.             sop(account.getBalance());
36.
37.             // Update
38.             Account accountB = new Account();
39.             accountB.setAccno(1001);
40.             accountB.setName("yellareddy-2");
41.             accountB.setBalance(2678.9);
42.             impl.update(accountB);
43.             sop("Account updated");
44.
45.             // Retrieve multiple
46.             List<Account> accounts = impl.get();
47.             sop("Account Detials are...");
48.             for (Account accountC : accounts) {
49.                     sop(accountC.getAccno());
50.                     sop(accountC.getName());
51.                     sop(accountC.getBalance());
52.             }
53.
54.             Account accountD = new Account();
55.             accountD.setAccno(6002);
```

```
56.             accountD.setName("sekhar");
57.             accountD.setBalance(9569.0);
58.             impl.callProcedure(accountD);
59.
60.     }
61.
62.     public static void sop(Object object) {
63.             System.out.println(object);
64.     }
65. }
```

Before Execution ACCOUNT table

ACCNO	NAME	BAL
1001	yellareddy	5689
8001	somu	9569
8002	kesavareddy	3456

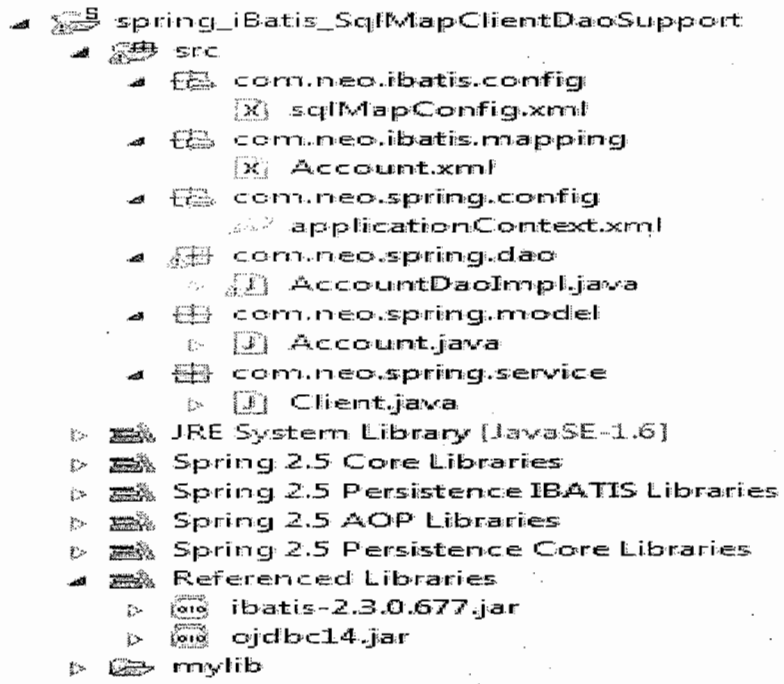
After Execution ACCOUNT table

ACCNO	NAME	BAL
1001	yellareddy-2	2678.9
8005	sekhar	9569
6002	sekhar	9569
8002	kesavareddy	3456

Q.) What is **SqlMapClientDaoSupport ?**

- ⇨ While working with IBatis, into all dao classes we need to inject **SqlMapClientTemplate** object. So injecting **SqlMapClientTemplate** object, logic is repeating in all the dao classes. So spring people has given one abstract class called **SqlMapClientDaoSupport**, which contains this common **SqlMapClientTemplate** object injection logic.
- ⇨ So while writing Dao class it is better to extend **SqlMapClientDaoSupport**, so that we no need to write injection logic of **SqlMapClientTemplate**.
- ⇨ When we need **SqlMapClientTemplate**object in dao class just we call **getSqlMapClientTemplate()** . So, by extending this class we can get **SqlMapClientTemplate**, on that we can perform our operations.
- ⇨ One more advantage of **SqlMapClientDaoSupport** is we can directly inject **SqlMapClientFactoryBean** to dao class instead of **SqlMapClientTemplate**

Q.) Develop an application where we can use **SqlMapClientDaoSupport** instead of **SqlMapClientTemplate**?



Account.java

```
<-- SAME AS ABOVE -->
```

sqlMapConfig.xml

```
<-- SAME AS ABOVE -->
```

Account.xml

```
<-- SAME AS ABOVE -->
```

Client.java

```
<-- SAME AS ABOVE -->
```

AccountDaoImpl.java

```
1.  package com.neo.spring.dao;
2.
3.  import java.io.IOException;
4.  import java.util.HashMap;
5.  import java.util.List;
6.  import java.util.Map;
7.
8.  import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;
9.
10. import com.neo.spring.model.Account;
11.
12. public class AccountDaoImpl extends SqlMapClientDaoSupport{
13.     public void insert(Account account) throws IOException{
14.                         getSqlMapClientTemplate().insert("accountNamespace.insert", account);
15.     }
16.
17.
```

```
18.     public void delete(int accno){
19.             getSqlMapClientTemplate().delete("accountNamespace.delete", accno);
20.     }
21.
22.     public void update(Account account){
23.             getSqlMapClientTemplate().update("accountNamespace.update", account);
24.     }
25.
26.     public Account get(int accno){
27.             return
28.        (Account)getSqlMapClientTemplate().queryForObject("accountNamespace.myselect",accno);
29.     }
30.
31.     public List<Account> get(){
32.             return
33.           (List<Account>)getSqlMapClientTemplate().queryForList("accountNamespace.myselects");
34.     }
35.
36.     public void callProcedure(Account account){
37.             Map<String, Object> map = new HashMap<String, Object>();
38.             map.put("P_ACCNO", account.getAccno());
39.             map.put("P_NAME", account.getName());
40.             map.put("P_BAL", account.getBalance());
41.             getSqlMapClientTemplate().queryForObject("accountNamespace.myprocedure",map);
42.             System.out.println("Account procedure execution status : "+map.get("P_RESULT"));
43.     }
44.
45. }
```

applicationContext.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.  xmlns:p="http://www.springframework.org/schema/p"
5.  xsi:schemaLocation="http://www.springframework.org/schema/beans
6.  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.      <bean id="ds"
9.              class="org.springframework.jdbc.datasource.DriverManagerDataSource">
10.             <property name="driverClassName"
11.                                             value="oracle.jdbc.driver.OracleDriver"></property>
12.             <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"></property>
13.             <property name="username" value="system"></property>
14.             <property name="password" value="tiger"></property>
15.     </bean>
16.
17.     <bean id="smcfb" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
18.             <property name="dataSource" ref="ds"></property>
19.             <property name="configLocation"  value="com/neo/ibatis/config/sqlMapConfig.xml" />
20.     </bean>
```

```
21.
22.     <bean id="daoImpl" class="com.neo.spring.dao.AccountDaoImpl">
23.             <property name="sqlMapClient" ref="smcfb"></property>
24.     </bean>
25. </beans>
```

Before Execution ACCOUNT table

ACCNO	NAME	BAL
1001	yellareddy	5689
8001	somu	9569
8002	kesavareddy	3456

After Execution ACCOUNT table

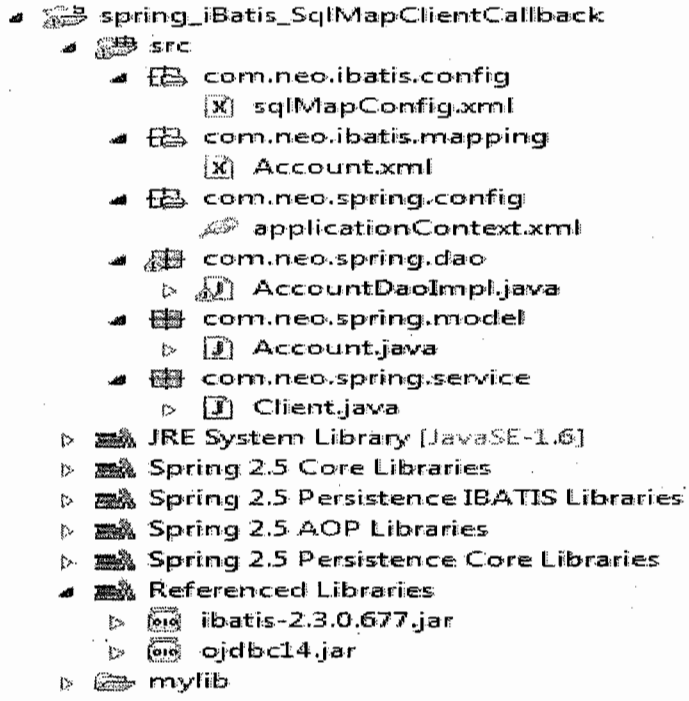
ACCNO	NAME	BAL
1001	yellareddy-2	2678.9
8005	sekhar	9569
6002	sekhar	9569
8002	kesavareddy	3456

Q) Why we are going for Callback mechanism?
Ans: When we are unable to implement underlying framework or technology functionality using xxxTemplate then we need to go for XXXCallback mechanism.

Q) What we can achieve with xxxCallback?
Ans: Any specific functionality of underlying framework or technology can be implemented, Because in the Callback mechanism they will pass underlying framework resource (Connection, Session, SqlMap, EntityManager ... etc).

SqlMapClientback:
When we are not getting the specific functionality of iBatis by using **SqlMapClientTemplate** those functionalities we can achieve by using **SqlMapClientCallback**.

Q.) Develop an application where I can use where we can use **SqlMapClientCallback**?



Account.java

```
<-- SAME AS ABOVE -->
```

sqlMapConfig.xml

```
<-- SAME AS ABOVE -->
```

Account.xml

```
<-- SAME AS ABOVE -->
```

applicationContext.xml

```
<-- SAME AS ABOVE -->
```

AccountDaoImpl.java

```
1.  package com.neo.spring.dao;
2.
3.  import java.sql.SQLException;
4.  import java.util.List;
5.
6.  import org.springframework.orm.ibatis.SqlMapClientCallback;
7.  import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;
8.
9.  import com.ibatis.sqlmap.client.SqlMapExecutor;
10. import com.neo.spring.model.Account;
11.
12. public class AccountDaoImpl extends SqlMapClientDaoSupport{
13.     public List<Account> get() {
14.         return (List<Account>) getSqlMapClientTemplate().execute(
15.                 new AccountSqlMapClientCallback());
16.     }
17.
18.     private class AccountSqlMapClientCallback implements SqlMapClientCallback {
19.         @Override
20.         public Object doInSqlMapClient(SqlMapExecutor executor)
21.                 throws SQLException {
22.             return executor.queryForList("accountNamespace.myselects");
23.         }
24.     }
25.
26. }
```

Client.java

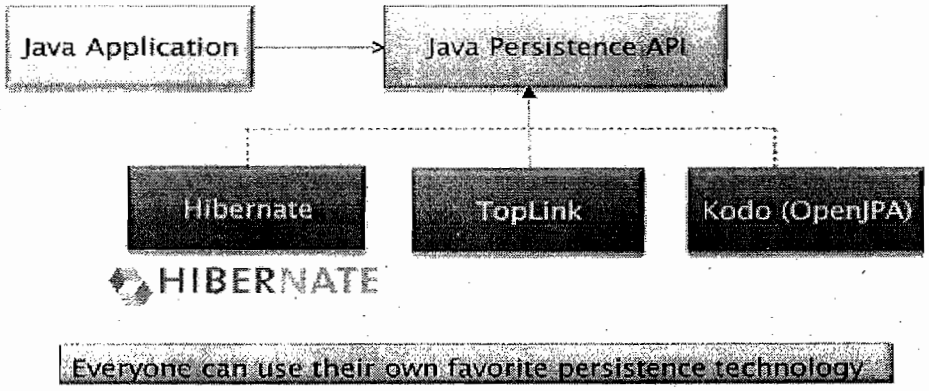
```
1.  package com.neo.spring.service;
2.
3.  import java.io.IOException;
4.  import java.util.List;
5.
6.  import org.springframework.context.ApplicationContext;
7.  import org.springframework.context.support.ClassPathXmlApplicationContext;
8.
9.  import com.neo.spring.dao.AccountDaoImpl;
```

```
10. import com.neo.spring.model.Account;
11.
12. public class Client {
13.     private static ApplicationContext context = new ClassPathXmlApplicationContext(
14.     "com/neo/spring/config/applicationContext.xml");
15.
16.     public static void main(String[] args) throws IOException {
17.             AccountDaoImpl impl = (AccountDaoImpl)context.getBean("daoImpl");
18.
19.             // Retrieve multiple
20.             List<Account> accounts = impl.get();
21.             sop("Account Detials are...");
22.             for (Account accountC : accounts) {
23.                     sop(accountC.getAccno());
24.                     sop(accountC.getName());
25.                     sop(accountC.getBalance());
26.             }
27.
28.     }
29.
30.     public static void sop(Object object) {
31.             System.out.println(object);
32.     }
33. }
```

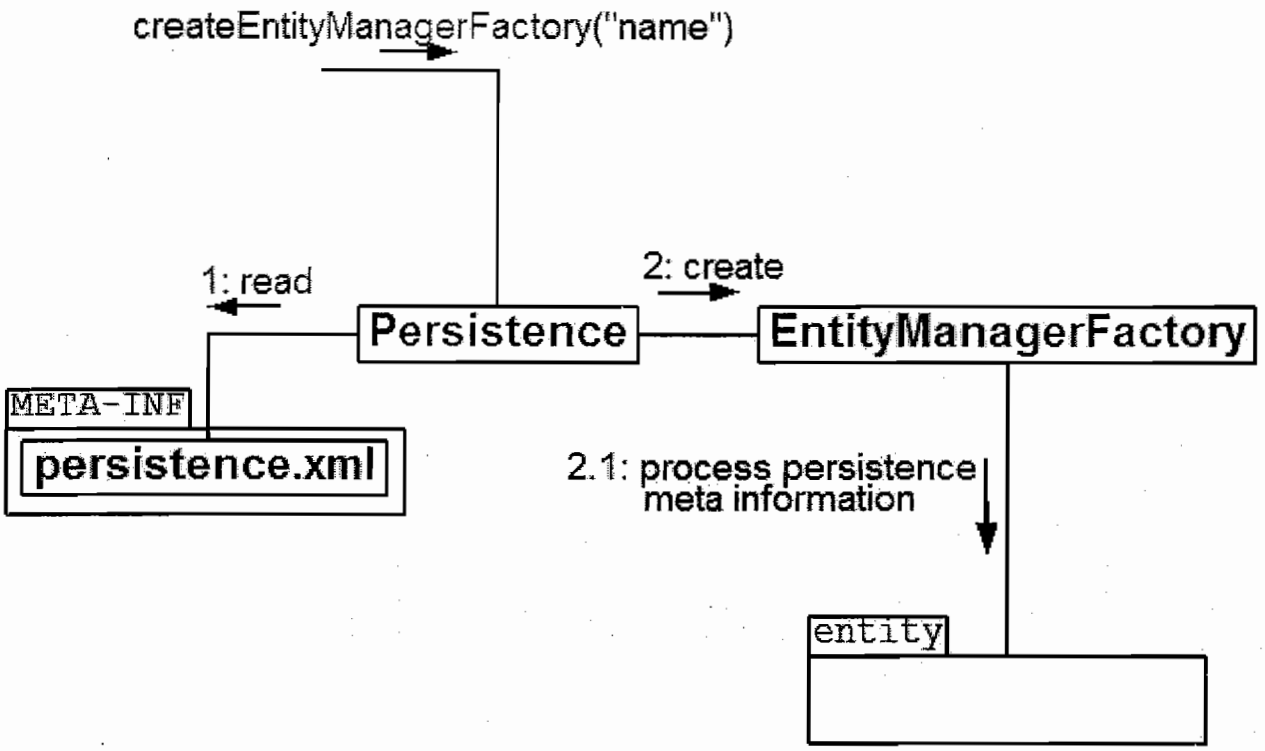
JPA(Java Persistence API)

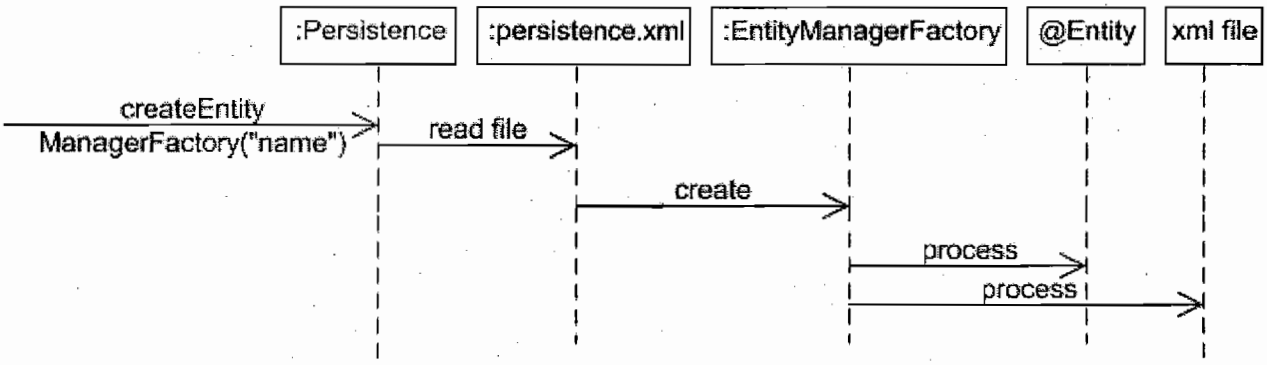
- Java Persistence API (JPA) provides POJO (Plain Old Java Object) standard and object relational mapping (OR mapping) for data persistence among applications.
- Persistence, which deals with storing and retrieving of application data, can now be programmed with Java Persistence API starting from EJB 3.0.
- This API has borrowed many of the concepts and standards from leading persistence frameworks like Toplink (from Oracle) and Hibernate (from JBoss). One of the great benefits of JPA is that it is an independent API.

JPA Architecture



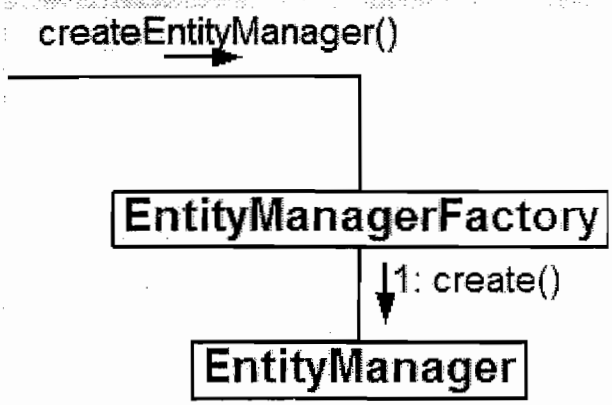
- Here we use annotation to map object with table. Mapping information given in the persistence class itself with annotations.
  - **@Entity:** It is attached to a class and it signifies that a class is persistent.
  - **@Id:** Each entity must have an identity. An Identity of an entity could simply be a class variable annotated with @Id.
  - **@Column:** We have to put on a class variable annotated with @Column.
- JPA allows us to work with entity classes, which are denoted as such using the annotation @Entity or configured in an XML file (we'll call this **persistence meta information**). When we acquire the **Entity Manager Factory** using the**Persistence** class, the **Entity Manager Factory** finds and processes the **persistence meta information**.
- To work with a database using JPA, we need an **Entity Manager**. Before we can do that, we need to create an **Entity Manager Factory.**





- To acquire an **Entity Manager Factory**, we use the class **javax.persistence.Persistence**. It reads a file called persistence.xml in the META-INF directory. It then creates the named **Entity Manager Factory**, which processes persistence meta information stored in XML files or annotations (we only use annotations).

- Creating an **Entity Manager** once we have the **Entity Manager Factory** is simple:

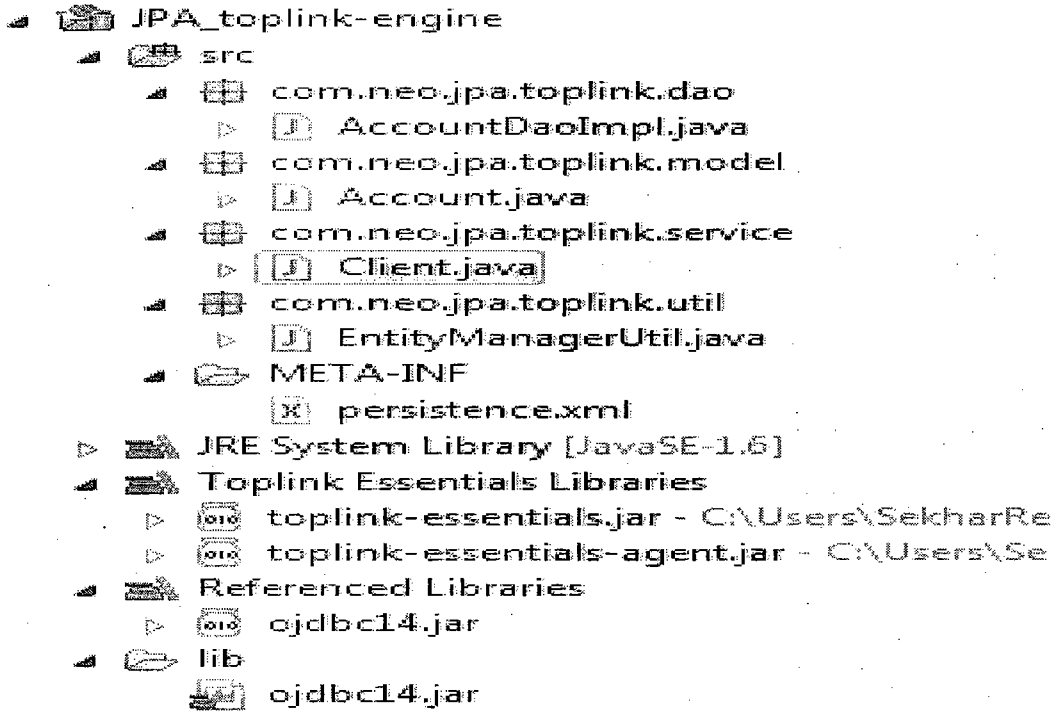


Once we have an **Entity Manager**, we can ask it to perform several operations such as persisting or removing an entity from the database or creating a query.

Term	Description
javax.persistence.Persistence	This is a class used as an entry point for using JPA. The primary method you'll use on this class is createEntityManagerFactory("someName") to retrieve an entity manager factory with the name "someName". This class ***requires*** a file called **persistence.xml** to be in the class path under a directory called **META-INF**.
EntityManagerFactory	An instance of this class provides a way to create entity managers. Entity Managers are not multi-thread safe so we need a way to create one per thread. This class provides that functionality. The Entity Manager Factory is the in-memory representation of a Persistence Unit.
EntityManager	An Entity Manager is **the** interface in your underlying storage mechanism. It provides methods for persisting, merging, removing, retrieving and querying objects. It is **not** multi-thread safe so we need one per thread. The Entity Manager also serves as a first level cache. It maintains changes and then attempts to optimize changes to the database by batching them up when the transaction completes.
persistence.xml	A **required** file that describes one or more persistence units. When you use the javax.persistence.Persistence class to look up an named Entity Manager Factory, the Persistence class looks for this file under the META-INF directory.
Persistence Unit	A Persistence Unit has a name and it describes database connection information either directly (if working in a JSE environment) or indirectly by referencing a JNDI-defined data source (if working in a managed/JEE environment). A Persistence Unit can also specify the classes(entities) it

	should or should not manage .
Persistence Meta Information	Information describing the configuration of entities and the database and the association between entity classes and the persistence units to which they relate. This is either through annotations added to classes or though XML files. Note that XML files take precedence over annotations.

Q.) Develop application where JPA uses toplink, hibernate, openJpa, eclipselink engines?



Account.java

```
1.  package com.neo.jpa.toplink.model;
2.
3.  import javax.persistence.Column;
4.  import javax.persistence.Entity;
5.  import javax.persistence.Id;
6.
7.  @Entity(name="ACCOUNT")
8.  public class Account {
9.      @Id
10.     @Column(name="ACCNO")
11.     private int accno;
12.     @Column(name="NAME")
13.     private String name;
14.     @Column(name="BAL")
15.     private double balance;
16.
17.     public int getAccno() {
```

```
18.             return accno;
19.     }
20.
21.     public void setAccno(int accno) {
22.             this.accno = accno;
23.     }
24.
25.     public String getName() {
26.             return name;
27.     }
28.
29.     public void setName(String name) {
30.             this.name = name;
31.     }
32.
33.     public double getBalance() {
34.             return balance;
35.     }
36.
37.     public void setBalance(double balance) {
38.             this.balance = balance;
39.     }
40.
41. }
```

persistence.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5.     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
6.
7.       <persistence-unit name="JPA_toplinkEnginePU" transaction-type="RESOURCE_LOCAL">
8.               <provider>oracle.toplink.essentials.PersistenceProvider</provider>
9.               <class>com.neo.jpa.toplink.model.Account</class>
10.              <properties>
11.                <property name = "toplink.jdbc.driver" value = "oracle.jdbc.driver.OracleDriver"/>
12.                <property name = "toplink.jdbc.url" value = "jdbc:oracle:thin:@localhost:1521:XE"/>
13.                <property name = "toplink.jdbc.user" value = "system"/>
14.                <property name = "toplink.jdbc.password" value = "tiger"/>
15.              </properties>
16.     </persistence-unit>
17.
18.  </persistence>
```

EntityManagerUtil.java

```
1.  package com.neo.jpa.toplink.util;
2.
3.  import javax.persistence.EntityManager;
4.  import javax.persistence.EntityManagerFactory;
```

```
5.  import javax.persistence.Persistence;
6.
7.  public class EntityManagerUtil {
8.      private static EntityManagerFactory factory;
9.      static {
10.             try {
11.                 factory = Persistence
12.                         .createEntityManagerFactory("JPA_toplinkEnginePU");
13.             } catch (Exception e) {
14.                 e.printStackTrace();
15.             }
16.     }
17.
18.     public static EntityManager getEntityManager() {
19.             return factory.createEntityManager();
20.     }
21.
22.     public static void closeEntityManager(EntityManager manager) {
23.             if (manager != null) {
24.                 manager.close();
25.             }
26.     }
27. }
```

AccountDaoImpl.java

```
1.  package com.neo.jpa.toplink.dao;
2.
3.  import java.io.IOException;
4.
5.  import javax.persistence.EntityManager;
6.
7.  import com.neo.jpa.toplink.model.Account;
8.  import com.neo.jpa.toplink.util.EntityManagerUtil;
9.
10. public class AccountDaoImpl {
11.     public void insert(Account account) throws IOException {
12.             EntityManager manager = null;
13.             try {
14.                 manager = EntityManagerUtil.getEntityManager();
15.                 manager.getTransaction().begin();
16.                 manager.persist(account);
17.                 manager.getTransaction().commit();
18.             } catch (Exception e) {
19.                 e.printStackTrace();
20.                 manager.getTransaction().rollback();
21.             } finally {
22.                 EntityManagerUtil.closeEntityManager(manager);
23.             }
24.     }
25.
```

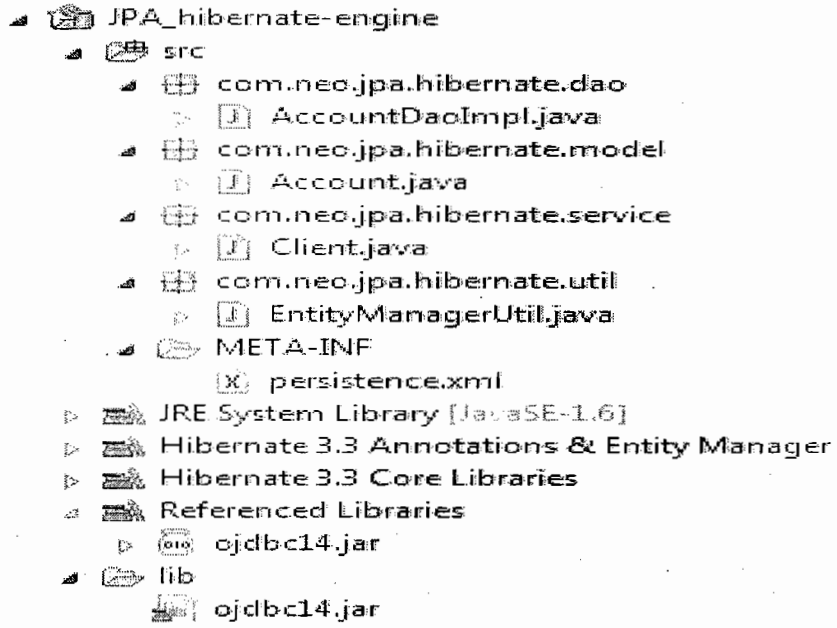
```
26.     public void delete(int accno) {
27.             EntityManager manager = null;
28.             try {
29.                     manager = EntityManagerUtil.getEntityManager();
30.                     manager.getTransaction().begin();
31.                     Account account = manager.find(Account.class, accno);
32.                     manager.remove(account);
33.                     manager.getTransaction().commit();
34.             } catch (Exception e) {
35.                     e.printStackTrace();
36.                     manager.getTransaction().rollback();
37.             } finally {
38.                     EntityManagerUtil.closeEntityManager(manager);
39.             }
40.     }
41.
42.     public void update(Account account) {
43.             EntityManager manager = null;
44.             try {
45.                     manager = EntityManagerUtil.getEntityManager();
46.                     manager.getTransaction().begin();
47.                     Account account2 = manager.find(Account.class, account.getAccno());
48.                     account2.setName(account.getName());
49.                     account2.setBalance(account.getBalance());
50.                     manager.getTransaction().commit();
51.             } catch (Exception e) {
52.                     e.printStackTrace();
53.                     manager.getTransaction().rollback();
54.             } finally {
55.                     EntityManagerUtil.closeEntityManager(manager);
56.             }
57.     }
58.
59.     public Account get(int accno) {
60.             Account account = null;
61.             EntityManager manager = null;
62.             try {
63.                     manager = EntityManagerUtil.getEntityManager();
64.                     account = manager.find(Account.class, accno);
65.             } catch (Exception e) {
66.                     e.printStackTrace();
67.             } finally {
68.                     EntityManagerUtil.closeEntityManager(manager);
69.             }
70.             return account;
71.     }
72. }
```

Client.java

```
1.  package com.neo.jpa.toplink.service;
```

```
2.  import java.io.IOException;
3.  import com.neo.jpa.toplink.dao.AccountDaoImpl;
4.  import com.neo.jpa.toplink.model.Account;
5.  public class Client {
6.      public static void main(String[] args) throws IOException {
7.              AccountDaoImpl impl = new AccountDaoImpl();
8.              // Insert
9.              Account accountA = new Account();
10.             accountA.setAccno(8003);
11.             accountA.setName("sekhar");
12.             accountA.setBalance(9569.0);
13.             impl.insert(accountA);
14.             sop("Account inserted");
15.
16.             // Delete
17.             impl.delete(8002);
18.             sop("Account deleted");
19.
20.             // Retrieve
21.             Account account = impl.get(8004);
22.             sop("Account Detials are...");
23.             sop(account.getAccno());
24.             sop(account.getName());
25.             sop(account.getBalance());
26.
27.             // Update
28.             Account accountB = new Account();
29.             accountB.setAccno(8008);
30.             accountB.setName("sekhar-updated");
31.             accountB.setBalance(678.9);
32.             impl.update(accountB);
33.             sop("Account updated");
34.     }
35.
36.     public static void sop(Object object) {
37.             System.out.println(object);
38.     }
39. }
```

JPA with Hibernate engine



Account.java

```
<-- SAME AS ABOVE -->
```

EntityManagerUtil.java

```
<-- SAME AS ABOVE -->
```

AccountDaoImpl.java

```
<-- SAME AS ABOVE -->
```

Client.java

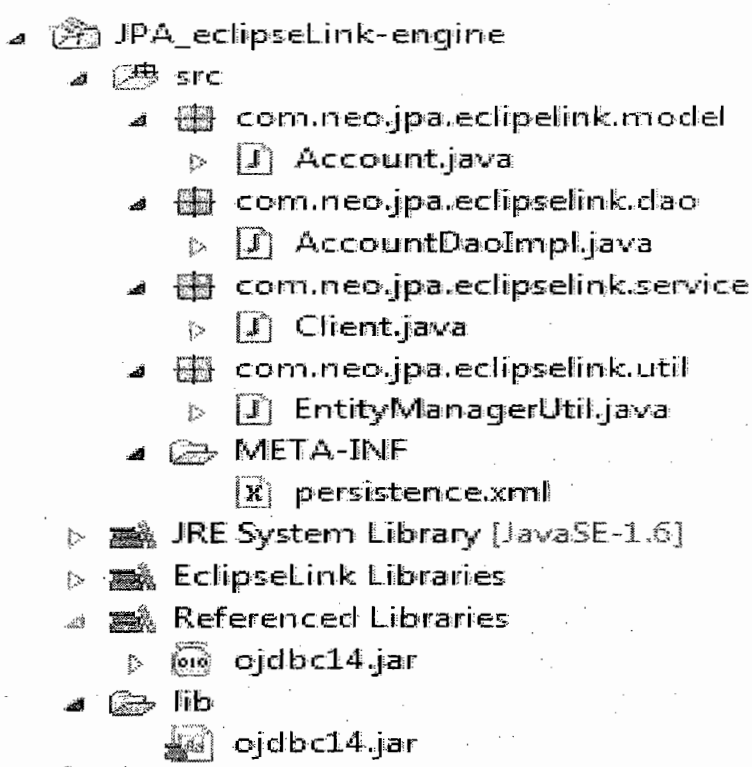
```
<-- SAME AS ABOVE -->
```

persistence.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5.     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
6.
7.       <persistence-unit name="JPA_hibernateEnginePU" transaction-type="RESOURCE_LOCAL">
8.               <provider>org.hibernate.ejb.HibernatePersistence</provider>
9.               <class>com.neo.jpa.hibernate.model.Account</class>
10.              <properties>
11.                     <property name = "hibernate.connection.driver_class" value =
12.                                                     "oracle.jdbc.driver.OracleDriver"/>
13.                     <property name = "hibernate.connection.url" value =
14.                                                     "jdbc:oracle:thin:@localhost:1521:XE"/>
15.                     <property name = "hibernate.connection.username" value = "system"/>
16.                     <property name = "hibernate.connection.password" value = "tiger"/>
17.             </properties>
18.     </persistence-unit>
```

```
19.
20.    </persistence>
```

JPA with eclipseLink engine



Account.java

```
<-- SAME AS ABOVE -->
```

EntityManagerUtil.java

```
<-- SAME AS ABOVE -->
```

AccountDaoImpl.java

```
<-- SAME AS ABOVE -->
```

Client.java

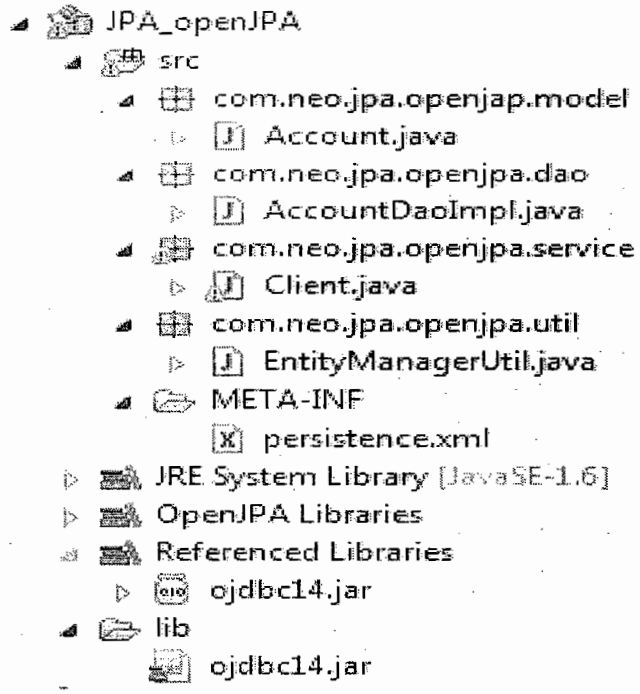
```
<-- SAME AS ABOVE -->
```

persistence.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5.     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
6.
7.       <persistence-unit name="aaPU" transaction-type="RESOURCE_LOCAL">
8.              <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
9.              <class>com.neo.jpa.eclipselink.model.Account</class>
10.             <properties>
```

```
11.                 <property name = "eclipselink.jdbc.driver" value =
12.                                                     "oracle.jdbc.driver.OracleDriver"/>
13.                 <property name = "eclipselink.jdbc.url" value =
14.                                                     "jdbc:oracle:thin:@localhost:1521:XE"/>
15.                 <property name = "eclipselink.jdbc.user" value = "system"/>
16.                 <property name = "eclipselink.jdbc.password" value = "tiger"/>
17.             </properties>
18.     </persistence-unit>
19. </persistence>
```

JPA with openJPA engine



Account.java

```
<-- SAME AS ABOVE -->
```

EntityManagerUtil.java

```
<-- SAME AS ABOVE -->
```

AccountDaoImpl.java

```
<-- SAME AS ABOVE -->
```

Client.java

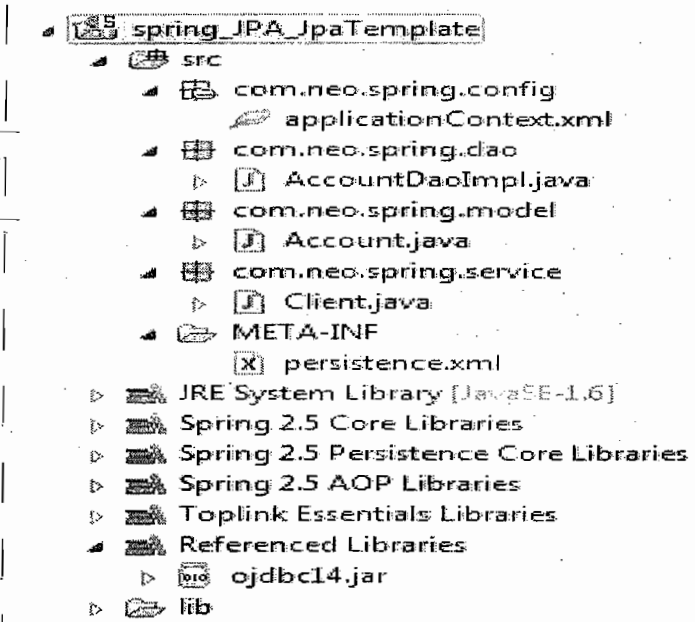
```
<-- SAME AS ABOVE -->
```

persistence.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5.     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
```

```
6.
7.      <persistence-unit name="JPA_openJPAPU" transaction-type="RESOURCE_LOCAL">
8.              <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider>
9.              <class>com.neo.jpa.openjpa.model.Account</class>
10.             <properties>
11.                     <property name = "openjpa.ConnectionDriverName" value =
12.                                                     "oracle.jdbc.driver.OracleDriver"/>
13.                     <property name = "openjpa.ConnectionURL" value =
14.                                                     "jdbc:oracle:thin:@localhost:1521:XE"/>
15.                     <property name = "openjpa.ConnectionUserName" value = "system"/>
16.                     <property name = "openjpa.ConnectionPassword" value = "tiger"/>
17.             </properties>
18.     </persistence-unit>
19. </persistence>
```

JPA-Spring Integration



persistence.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5.    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
6.
7.      <persistence-unit name="spring_JPA_JpaTemplatePU" transaction-type="RESOURCE_LOCAL">
8.              <provider>oracle.toplink.essentials.PersistenceProvider</provider>
9.              <class>com.neo.spring.model.Account</class>
10.             <properties>
11.                     <property name = "toplink.jdbc.driver"
12.                                                     value = "oracle.jdbc.driver.OracleDriver"/>
13.                     <property name = "toplink.jdbc.url"
14.                                                     value = "jdbc:oracle:thin:@localhost:1521:XE"/>
15.                     <property name = "toplink.jdbc.user" value = "system"/>
```

```
16.                     <property name = "toplink.jdbc.password" value = "tiger"/>
17.                 </properties>
18.         </persistence-unit>
19. </persistence>
```

applicationContext.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans
3.      xmlns="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.      xmlns:p="http://www.springframework.org/schema/p"
6.      xsi:schemaLocation="http://www.springframework.org/schema/beans
7.  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
8.  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-
9.  tx-2.5.xsd" xmlns:tx="http://www.springframework.org/schema/tx">
10.
11. <bean id="emf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
12.             <property name="persistenceUnitName" value="spring_JPA_JpaTemplatePU" />
13. </bean>
14.
15.     <bean id="jt" class="org.springframework.orm.jpa.JpaTemplate">
16.             <property name="entityManagerFactory" ref="emf"></property>
17.     </bean>
18.
19.     <bean id="daoImpl" class="com.neo.spring.dao.AccountDaoImpl">
20.             <property name="jpaTemplate" ref="jt"></property>
21.     </bean>
22.
23.     <bean id="tm" class="org.springframework.orm.jpa.JpaTransactionManager">
24.             <property name="entityManagerFactory" ref="emf" />
25.     </bean>
26.     <tx:annotation-driven transaction-manager="tm" />
27.
28. </beans>
```

Account.java

```
1.  package com.neo.spring.model;
2.  import javax.persistence.Column;
3.  import javax.persistence.Entity;
4.  import javax.persistence.Id;
5.
6.  @Entity(name="ACCOUNT")
7.  public class Account {
8.      @Id
9.      @Column(name="ACCNO")
10.     private int accno;
11.     @Column(name="NAME")
12.     private String name;
13.     @Column(name="BAL")
```

```
14.     private double balance;
15.
16.     public int getAccno() {
17.             return accno;
18.     }
19.
20.     public void setAccno(int accno) {
21.             this.accno = accno;
22.     }
23.
24.     public String getName() {
25.             return name;
26.     }
27.     public void setName(String name) {
28.             this.name = name;
29.     }
30.     public double getBalance() {
31.             return balance;
32.     }
33.     public void setBalance(double balance) {
34.             this.balance = balance;
35.     }
36. }
```

AccountDaoImpl.java

```
1.  package com.neo.spring.dao;
2.  import org.springframework.orm.jpa.JpaTemplate;
3.  import org.springframework.transaction.annotation.Transactional;
4.  import com.neo.spring.model.Account;
5.  @Transactional
6.  public class AccountDaoImpl {
7.      private JpaTemplate jpaTemplate;
8.
9.      public void setJpaTemplate(JpaTemplate jpaTemplate) {
10.             this.jpaTemplate = jpaTemplate;
11.     }
12.
13.     public Account get(int accno) {
14.             return jpaTemplate.find(Account.class, accno);
15.     }
16.
17.     public void create(Account account) {
18.             jpaTemplate.persist(account);
19.     }
20.
21.     public void update(Account account) {
22.             Account existingAccount = jpaTemplate.find(Account.class, account
23.                             .getAccno());
24.             existingAccount.setName(account.getName());
25.             existingAccount.setBalance(account.getBalance());
```

```
26.     }
27.
28.     public void delete(int accno) {
29.             Account account = jpaTemplate.find(Account.class, accno);
30.             jpaTemplate.remove(account);
31.     }
32. }
```

Client.java

```
1.  package com.neo.spring.service;
2.  import org.springframework.context.ApplicationContext;
3.  import org.springframework.context.support.ClassPathXmlApplicationContext;
4.  import com.neo.spring.dao.AccountDaoImpl;
5.  import com.neo.spring.model.Account;
6.  public class Client {
7.      private static ApplicationContext context = new ClassPathXmlApplicationContext(
8.                      "com/neo/spring/config/applicationContext.xml");
9.
10.     public static void main(String[] args) {
11.             AccountDaoImpl daoImpl = (AccountDaoImpl) context.getBean("daoImpl");
12.
13.             // insert
14.             Account accountA = new Account();
15.             accountA.setAccno(9001);
16.             accountA.setName("somasekhar");
17.             accountA.setBalance(9999.0);
18.             daoImpl.create(accountA);
19.             System.out.println("Account Inserted");
20.
21.             // select
22.             Account accountB = daoImpl.get(8001);
23.             System.out.println("Account details are...");
24.             System.out.println(accountB.getAccno());
25.             System.out.println(accountB.getName());
26.             System.out.println(accountB.getBalance());
27.
28.             // update
29.             Account accountC = new Account();
30.             accountC.setAccno(8002);
31.             accountC.setName("somu");
32.             accountC.setBalance(8600.0);
33.             daoImpl.update(accountC);
34.             System.out.println("Account updated");
35.
36.             // delete
37.             daoImpl.delete(8003);
38.             System.out.println("Account Deleted...");
39.     }
40. }
```