Facilitybe’s .xhtml file has entityActionBean.save(contextBean)- > -> EntityActionsBean.class-> click on save(entity, DefaultConfirm.class); -> EntityService.class-> click on dao.save(entity); -> HibernateDAO.class

1. Facilitybe’s .xhtml file

<p:commandButton value=*"#{label.save}"* image=*"ui-icon-disk"*

action=*"#{entityActionsBean.save(ctxBean)}"* rendered=*"#{ctxPage.isInStdPage() and pageAuthorization.allowed('facilityBE\_crud.saveEntityBtn','facilityBE:',webContext,'edit','R')}"* id=*"saveEntityBtn"* title=*"#{label.save}"*immediate=*"false"*

ajax=*"true"* global=*"false"* onstart=*"pageSubmitAjaxStart();"* onerror=*"pageSubmitAjaxError();"* update=*":pageHeaderForm:sessionDirtyFlagId @form :page\_messages :ajax-complete :growl"*> </p:commandButton>

1. *entityActionsBean.class*

**private** EntityService entityService;

**public** **void** save(EntityBean entityBean) {

// Validating the Entity before saving

EntityContextEntry entry = entityContext.peek();

saveEntity(entityBean, **false**, **true**);

}

**public** **boolean** saveEntity(EntityBean entityBean, **final** **boolean** close, **boolean** confirm) {

BaseDataObject entity = (BaseDataObject) entityBean.getEntity();

conversationSessionDirtyHolder.setSessionDirty(**false**);

**try** {

// Embeddable entities do not extend BaseDataObject

/\*if (entity instanceof BaseDataObject) {

entityService.validate((BaseDataObject) entity);

}\*/

// **TODO** Triggering validations for Embeddable type

**if** (confirm) {

entityService.save(entity, DefaultConfirm.**class**);

} **else** {

entityService.save(entity);

}

entityService.flush();

cleanUpCache();

**if** (close) {

**if** (Conversation.*instance*().isNested()) {

ConversationCallbackValue callbackValue = (ConversationCallbackValue) Component.*getInstance*(

"conversationCallbackValue");

callbackValue.setValue(Conversation.*instance*().getParentId(), entity);

}

doneEntity(**true**);

}

**if** (SeamUtils.*getCodifyMessageBean*().containsErrorMsg()) {

addFailureMessage("save", entityBean.getEname());

**return** **false**;

} **else** {

addSuccessMessage("save", entityBean.getEname());

**return** **true**;

}

} **catch** (ConfirmationException ce) {

Iterator iter = ce.getConstraintValidations().iterator();

StringBuilder messageBuilder = **new** StringBuilder();

**while** (iter.hasNext()) {

ConstraintViolation violation = (ConstraintViolation) iter.next();

messageBuilder = messageBuilder.append(violation.getMessage()).append(" ");

}

UIComponent component = FacesContext.*getCurrentInstance*().getViewRoot().findComponent(

"entityForm:confirmDialogButton");

**if** ((component != **null**) && (component **instanceof** CommandButton)) {

CommandButton cb = (CommandButton) component;

List<ClientBehavior> behaviors = cb.getClientBehaviors().get("click");

**if** ((behaviors != **null**) && (behaviors.size() > 0)) {

**for** (ClientBehavior behavior : behaviors) {

**if** (behavior **instanceof** CodifyConfirmBehavior) {

((CodifyConfirmBehavior) behavior).setMessage(messageBuilder.toString());

}

}

}

}

entityBean.setActionCallback(**new** DynamicActionCallback() {

@Override **public** **void** doAction(EntityActionsBean entityActionsBean, Object obj) {

entityActionsBean.saveConfirmed((EntityBean) obj, close);

}

});

RequestContext context = RequestContext.*getCurrentInstance*();

context.update("entityForm:confirmButton\_panel");

context.execute("PF('confirmButton').jq.click();");

} **catch** (ValidationException ve) {

processConstraintViolations(ve.getConstraintValidations());

*LOG*.info("Validation failure for entity:" + entity.getClass());

} **catch** (ConstraintViolationException cve) {

processConstraintViolations(cve.getConstraintViolations());

*LOG*.info("Validation failure for entity:" + entity.getClass());

}

addFailureMessage("save", entityBean.getEname());

**return** **false**;

}

1. **EntityService.class**

@Transactional(

readOnly = **false**,

noRollbackFor = { ValidationException.**class** }

)

**public** Long save(ENTITY entity, Class confirmationGroup) **throws** DataAccessException {

**if** (*LOG*.isInfoEnabled()) {

*LOG*.info(String.*format*("Executing EntityService.save: %s", entity.toShortString()));

}

sequenceProcessor.parseEntity(entity);

// this.validate(entity);

**return** dao.save(entity);

}

1. HibernateDAO.class

/\*\*

\* **@author** Suren Krishnamurthy

\* **@author** Anil Babu

\* **@see** HibernateTemplate

\*/

//@Repository

//@Configurable

**public** **class** HibernateDAO **extends** HibernateDaoSupport {

**protected** **final** Log log = LogFactory.*getLog*(getClass());

// A hack to bypass seamManagedSessionFactory, which creates a proxy (typically, in case of webapps).

// This is to be used in cases where sessionFactory injected is seamManaged, but we need to

// the underlying session factory directly.

**protected** SessionFactory springSessionFactory;

@Autowired

**protected** TagService tagService;

**public** Long save(Object entity) **throws** DataAccessException {

**return** (Long) getHibernateTemplate().save(entity);

}

**public** List find(String queryString, Object value) throws DataAccessException {

**return** getHibernateTemplate().find(queryString, value);

}

**public** List find(String queryString, Object[] values) **throws** DataAccessException {

**return** getHibernateTemplate().find(queryString, values);

}

**public** List find(String queryString) **throws** DataAccessException {

**return** getHibernateTemplate().find(queryString);

}

**By extending DaoSupport or directly instantiating HibernateTemplate ( with Autowiring Spring DI) we can perform DB related task**

// Compiled from HibernateDaoSupport.java (version 1.5 : 49.0, super bit)

public abstract class org.springframework.orm.hibernate3.support.HibernateDaoSupport extends org.springframework.dao.support.DaoSupport {

// Field descriptor #6 Lorg/springframework/orm/hibernate3/HibernateTemplate;

private org.springframework.orm.hibernate3.HibernateTemplate hibernateTemplate;

http://www.javatpoint.com/hibernate-and-spring-integration

Hibernate and Spring Integration

We can simply integrate **hibernate application with spring application**.

In hibernate framework, we provide all the database information in hibernate.cfg.xml file.

But if we are going to integrate the hibernate application with spring, we don't need to create the hibernate.cfg.xml file. We can provide all the information in the applicationContext.xml file.

Advantage of Spring framework with hibernate

The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc. So **it saves a lot of code**. **Understanding problem without using spring:** Let's understand it by the code of hibernate given below:

//creating configuration

Configuration cfg=**new** Configuration();

cfg.configure("hibernate.cfg.xml");

//creating seession factory object

SessionFactory factory=cfg.buildSessionFactory();

//creating session object

Session session=factory.openSession();

//creating transaction object

Transaction t=session.beginTransaction();

Employee e1=**new** Employee(111,"arun",40000);

session.persist(e1);//persisting the object

t.commit();//transaction is commited

session.close();

As you can see in the code of sole hibernate, you have to follow so many steps.

**Solution by using HibernateTemplate class of Spring Framework:**

Now, you don't need to follow so many steps. You can simply write this:

1. Employee e1=**new** Employee(111,"arun",40000);
2. hibernateTemplate.save(e1);

Methods of HibernateTemplate class

Let's see a list of commonly used methods of HibernateTemplate class.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | void persist(Object entity) | persists the given object. |
| 2) | Serializable save(Object entity) | persists the given object and returns id. |
| 3) | void saveOrUpdate(Object entity) | persists or updates the given object. If id is found, it updates the record otherwise saves the record. |
| 4) | void update(Object entity) | updates the given object. |
| 5) | void delete(Object entity) | deletes the given object on the basis of id. |
| 6) | Object get(Class entityClass, Serializable id) | returns the persistent object on the basis of given id. |
| 7) | Object load(Class entityClass, Serializable id) | returns the persistent object on the basis of given id. |
| 8) | List loadAll(Class entityClass) | returns the all the persistent objects. |

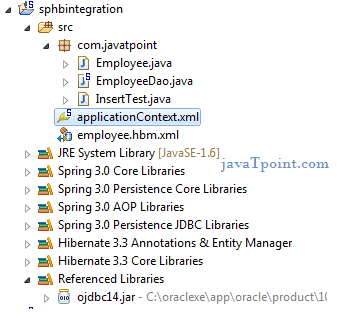
Steps

Let's see what are the simple steps for hibernate and spring integration:

1. **create table in the database** It is optional.
2. **create applicationContext.xml file** It contains information of DataSource, SessionFactory etc.
3. **create Employee.java file** It is the persistent class
4. **create employee.hbm.xml file** It is the mapping file.
5. **create EmployeeDao.java file** It is the dao class that uses HibernateTemplate.
6. **create InsertTest.java file** It calls methods of EmployeeDao class.

Example of Hibernate and spring integration

In this example, we are going to integrate the hibernate application with spring. Let's see the **directory structure** of spring and hibernate example.



**1) create the table in the database**

In this example, we are using the Oracle as the database, but you may use any database. Let's create the table in the oracle database

1. CREATE TABLE  "EMP558"
2. (    "ID" NUMBER(10,0) NOT NULL ENABLE,
3. "NAME" VARCHAR2(255 CHAR),
4. "SALARY" FLOAT(126),
5. PRIMARY KEY ("ID") ENABLE
6. )
7. /

**2) Employee.java**

It is a simple POJO class. Here it works as the persistent class for hibernate.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
8. //getters and setters
10. }

**3) employee.hbm.xml**

This mapping file contains all the information of the persistent class.

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.Employee" table="emp558">
8. <id name="id">
9. <generator **class**="assigned"></generator>
10. </id>
12. <property name="name"></property>
13. <property name="salary"></property>
14. </**class**>
16. </hibernate-mapping>

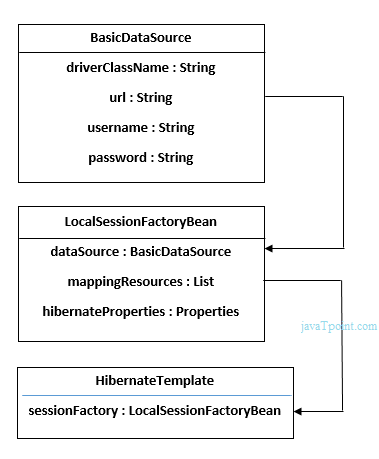
**4) EmployeeDao.java**

It is a java class that uses the **HibernateTemplate** class method to persist the object of Employee class.

1. **package** com.javatpoint;
2. **import** org.springframework.orm.hibernate3.HibernateTemplate;
3. **import** java.util.\*;
4. **public** **class** EmployeeDao {
5. HibernateTemplate template;
6. **public** **void** setTemplate(HibernateTemplate template) {
7. **this**.template = template;
8. }
9. //method to save employee
10. **public** **void** saveEmployee(Employee e){
11. template.save(e);
12. }
13. //method to update employee
14. **public** **void** updateEmployee(Employee e){
15. template.update(e);
16. }
17. //method to delete employee
18. **public** **void** deleteEmployee(Employee e){
19. template.delete(e);
20. }
21. //method to return one employee of given id
22. **public** Employee getById(**int** id){
23. Employee e=(Employee)template.get(Employee.**class**,id);
24. **return** e;
25. }
26. //method to return all employees
27. **public** List<Employee> getEmployees(){
28. List<Employee> list=**new** ArrayList<Employee>();
29. list=template.loadAll(Employee.**class**);
30. **return** list;
31. }
32. }

**5) applicationContext.xml**

In this file, we are providing all the informations of the database in the **BasicDataSource** object. This object is used in the**LocalSessionFactoryBean** class object, containing some other informations such as mappingResources and hibernateProperties. The object of **LocalSessionFactoryBean** class is used in the HibernateTemplate class. Let's see the code of applicationContext.xml file.



*File: 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-3.0.xsd">

10. <bean id="dataSource" **class**="org.apache.commons.dbcp.BasicDataSource">
11. <property name="driverClassName"  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="oracle"></property>
15. </bean>
17. <bean id="mysessionFactory"  **class**="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
18. <property name="dataSource" ref="dataSource"></property>
20. <property name="mappingResources">
21. <list>
22. <value>employee.hbm.xml</value>
23. </list>
24. </property>
26. <property name="hibernateProperties">
27. <props>
28. <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
29. <prop key="hibernate.hbm2ddl.auto">update</prop>
30. <prop key="hibernate.show\_sql">**true**</prop>
32. </props>
33. </property>
34. </bean>
36. <bean id="template" **class**="org.springframework.orm.hibernate3.HibernateTemplate">
37. <property name="sessionFactory" ref="mysessionFactory"></property>
38. </bean>
40. <bean id="d" **class**="com.javatpoint.EmployeeDao">
41. <property name="template" ref="template"></property>
42. </bean>

45. </beans>

**6) InsertTest.java**

This class uses the EmployeeDao class object and calls its saveEmployee method by passing the object of Employee class.

1. **package** com.javatpoint;
2. **import** org.springframework.beans.factory.BeanFactory;
3. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
4. **import** org.springframework.core.io.ClassPathResource;
5. **import** org.springframework.core.io.Resource;
6. **public** **class** InsertTest {
7. **public** **static** **void** main(String[] args) {
8. Resource r=**new** ClassPathResource("applicationContext.xml");
9. BeanFactory factory=**new** XmlBeanFactory(r);
11. EmployeeDao dao=(EmployeeDao)factory.getBean("d");
13. Employee e=**new** Employee();
14. e.setId(114);
15. e.setName("varun");
16. e.setSalary(50000);
18. dao.saveEmployee(e);  }}

Now, if you see the table in the oracle database, record is inserted successfully.

[download this example (developed using MyEclipse IDE)](http://www.javatpoint.com/src/hb/sphbinteg.zip)

Enabling automatic table creation, showing sql queries etc.

You can enable many hibernate properties like automatic table creation by hbm2ddl.auto etc. in applicationContext.xml file. Let's see the code:

1. <property name="hibernateProperties">
2. <props>
3. <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
4. <prop key="hibernate.hbm2ddl.auto">update</prop>
5. <prop key="hibernate.show\_sql">**true**</prop>
7. </props>

If you write this code, you don't need to create table because table will be created automatically.

|  |  |
| --- | --- |
|  | **Why HibernateTemplate isn't recommended?**  Because its main goal was to get a Hibernate session tied to the current Spring transaction, when SessionFactory.getCurrentSession() didn't exist. Since it now exists (and for a long time: HibenateTemplate usage is discouraged even in the hibernate3 package), there is no reason to use this Spring-specific class instead of using SessionFactory.getCurrentSession() to get a session tied to the current Spring transaction.  If you use Spring, then you should use its declarative transaction management, which allows you to avoid opening, committing, closing and flushing. It's all done by Spring automatically:  @Autowired  private SessionFactory sessionFactory;  @Transactional  public void someMethod() {  // get the session for the current transaction:  Session session = sessionFactory.getCurrentSession();  // do things with the session (queries, merges, persists, etc.)  }  In the above example, a transaction will be started (if not already started) before the method invocation; A session will be created by Spring for the transaction, and the session will be automatically flushed before the commit of the transaction, that will be done by Spring automatically when the method returns. |

**ApplicationContext.xml**

<bean id=*"hibernateDAO"* class=*"com.thirdpillar.foundation.model.dao.HibernateDAO"*>

<property name=*"springSessionFactory"* ref=*"sessionFactory"* />

<property name=*"sessionFactory"* ref=*"seamSessionFactory"* />

<property name=*"flushModeName"* value=*"FLUSH\_NEVER"* />

<property name=*"allowCreate"* value=*"false"* />

<property name=*"checkWriteOperations"* value=*"false"* />

</bean>

**Spring Data JPA Tutorial** [www.javatpoint.com/spring-and-jpa-integration](http://www.javatpoint.com/spring-and-jpa-integration)

Spring Data JPA API provides JpaTemplate class to integrate spring application with JPA.

JPA (Java Persistent API) is the sun specification for persisting objects in the enterprise application. It is currently used as the replacement for complex entity beans.The implementation of JPA specification are provided by many vendors such as:

* Hibernate
* Toplink
* iBatis
* OpenJPA etc.

**Advantage of Spring JpaTemplate**

You don't need to write the before and after code for persisting, updating, deleting or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, commiting EntityTransaction instance and closing EntityManager.

So, it **save a lot of code**.

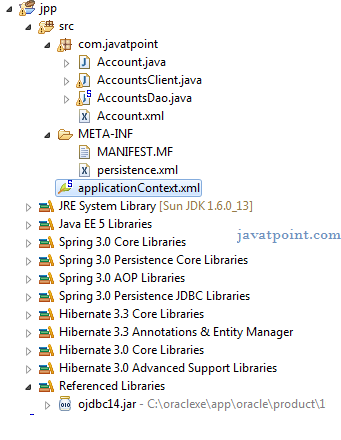
In this example, we are going to use hibernate for the implementation of JPA.

### Example of Spring and JPA Integration

Let's see the simple steps to integration spring application with JPA:

1. **create Account.java file**
2. **create Account.xml file**
3. **create AccountDao.java file**
4. **create persistence.xml file**
5. **create applicationContext.xml file**
6. **create AccountsClient.java file**

In this example, we are going to integrate the hibernate application with spring. Let's see the **directory structure** of jpa example with spring.



**1) Account.java**

It is a simple POJO class.

1. **package** com.javatpoint;
3. **public** **class** Account {
4. **private** **int** accountNumber;
5. **private** String owner;
6. **private** **double** balance;
7. //no-arg and parameterized constructor
8. //getters and setters
9. }

**2) Account.xml**

This mapping file contains all the information of the persistent class.

1. <entity-mappings version="1.0"
2. xmlns="http://java.sun.com/xml/ns/persistence/orm"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
5. http://java.sun.com/xml/ns/persistence/orm\_1\_0.xsd ">
7. <entity **class**="com.javatpoint.Account">
8. <table name="account100"></table>
9. <attributes>
10. <id name="accountNumber">
11. <column name="accountnumber"/>
12. </id>
13. <basic name="owner">
14. <column name="owner"/>
15. </basic>
16. <basic name="balance">
17. <column name="balance"/>
18. </basic>
19. </attributes>
20. </entity>
21. </entity-mappings>

**3) AccountDao.java**

1. **package** com.javatpoint;
2. **import** java.util.List;
3. **import** org.springframework.orm.jpa.JpaTemplate;
4. **import** org.springframework.transaction.annotation.Transactional;
5. @Transactional
6. **public** **class** AccountsDao{
7. JpaTemplate template;
9. **public** **void** setTemplate(JpaTemplate template) {
10. **this**.template = template;
11. }
12. **public** **void** createAccount(**int** accountNumber,String owner,**double** balance){
13. Account account = **new** Account(accountNumber,owner,balance);
14. template.persist(account);
15. }
16. **public** **void** updateBalance(**int** accountNumber,**double** newBalance){
17. Account account = template.find(Account.**class**, accountNumber);
18. **if**(account != **null**){
19. account.setBalance(newBalance);
20. }
21. template.merge(account);
22. }
23. **public** **void** deleteAccount(**int** accountNumber){
24. Account account = template.find(Account.**class**, accountNumber);
25. **if**(account != **null**)
26. template.remove(account);
27. }
28. **public** List<Account> getAllAccounts(){
29. List<Account> accounts =template.find("select acc from Account acc");
30. **return** accounts;
31. }
32. }

**4) 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"**>**
7. **<persistence-unit** name="ForAccountsDB"**>**
8. **<mapping-file>**com/javatpoint/Account.xml**</mapping-file>**
9. **<class>**com.javatpoint.Account**</class>**
10. **</persistence-unit>**
11. **</persistence>**

**5) 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:tx="http://www.springframework.org/schema/tx"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/tx
8. http://www.springframework.org/schema/tx/spring-tx-3.0.xsd"**>**
10. **<tx:annotation-driven** transaction-manager="jpaTxnManagerBean" proxy-target-class="true"**/>**
12. **<bean** id="dataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>**
13. **<property** name="driverClassName" value="oracle.jdbc.driver.OracleDriver"**></property>**
14. **<property** name="url" value="jdbc:oracle:thin:@localhost:1521:xe"**></property>**
15. **<property** name="username" value="system"**></property>**
16. **<property** name="password" value="oracle"**></property>**
17. **</bean>**
19. **<bean** id="hbAdapterBean" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"**>**
20. **<property** name="showSql" value="true"**></property>**
21. **<property** name="generateDdl" value="true"**></property>**
22. **<property** name="databasePlatform" value="org.hibernate.dialect.OracleDialect"**></property>**
23. **</bean>**
25. **<bean** id="emfBean" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"**>**
26. **<property** name="dataSource" ref="dataSourceBean"**></property>**
27. **<property** name="jpaVendorAdapter" ref="hbAdapterBean"**></property>**
28. **</bean>**
30. **<bean** id="jpaTemplateBean" class="org.springframework.orm.jpa.JpaTemplate"**>**
31. **<property** name="entityManagerFactory" ref="emfBean"**></property>**
32. **</bean>**
34. **<bean** id="accountsDaoBean" class="com.javatpoint.AccountsDao"**>**
35. **<property** name="template" ref="jpaTemplateBean"**></property>**
36. **</bean>**
37. **<bean** id="jpaTxnManagerBean" class="org.springframework.orm.jpa.JpaTransactionManager"**>**
38. **<property** name="entityManagerFactory" ref="emfBean"**></property>**
39. **</bean>**
41. **</beans>**

The **generateDdl** property will create the table automatically.

The **showSql** property will show the sql query on console.

**6) Accountsclient.java**

1. **package** com.javatpoint;
3. **import** java.util.List;
4. **import** org.springframework.context.ApplicationContext;
5. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
6. **import** org.springframework.context.support.FileSystemXmlApplicationContext;
8. **public** **class** AccountsClient{
9. **public** **static** **void** main(String[] args){
10. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
11. AccountsDao accountsDao = context.getBean("accountsDaoBean",AccountsDao.**class**);
13. accountsDao.createAccount(15, "Jai Kumar", 41000);
14. accountsDao.createAccount(20, "Rishi ", 35000);
15. System.out.println("Accounts created");
17. //accountsDao.updateBalance(20, 50000);
18. //System.out.println("Account balance updated");
19. /\*List<Account> accounts = accountsDao.getAllAccounts();
20. for (int i = 0; i < accounts.size(); i++) {
21. Account acc = accounts.get(i);
22. System.out.println(acc.getAccountNumber() + " : " + acc.getOwner() + " (" + acc.getBalance() + ")");
23. }\*/
24. //accountsDao.deleteAccount(111);
25. //System.out.println("Account deleted"); } }

#### Output

Hibernate: insert into account100 (balance, owner, accountnumber) values (?, ?, ?)

Hibernate: insert into account100 (balance, owner, accountnumber) values (?, ?, ?)

Accounts created

[download this example (developed using Myeclipse IDE)](http://www.javatpoint.com/src/sp/jpawithhb.zip)