|  |
| --- |
| Here is an overview of the **basic steps to perform CRUD (Create, Read, Update, and Delete) operations in Hibernat**e:   1. Create a Hibernate configuration file (hibernate.cfg.xml) and specify the database details, such as the URL, username, and password. 2. Create a Hibernate mapping file (XML or annotations) for the entity you want to perform CRUD operations on. 3. Create a SessionFactory object using the configuration file. 4. Open a session using the SessionFactory object and begin a transaction. 5. Perform the CRUD operation:  * To create a new record, create an instance of the entity class, set its properties, and save it using the session's save() method. * To read a record, use the session's get() or load() method to retrieve the record by its primary key. * To update a record, retrieve the record using the session's get() or load() method, modify its properties, and update it using the session's update() or merge() method. * To delete a record, retrieve the record using the session's get() or load() method and delete it using the session's delete() method.  1. Commit the transaction and close the session. |

|  |
| --- |
| Here is an **overview of the basic steps to perform CRUD (Create, Read, Update, and Delete) operations in Spring Framework using Hibernate Template**:   1. Create a Spring configuration file (applicationContext.xml) and specify the database details, such as the URL, username, and password. Also configure Hibernate session factory and Hibernate template bean in the configuration file. 2. Create a Hibernate mapping file (XML or annotations) for the entity you want to perform CRUD operations on. 3. Inject the Hibernate template bean into the DAO class where you will perform the CRUD operations. 4. Perform the CRUD operation:  * To create a new record, use the Hibernate template's save() method to save the entity instance. * To read a record, use the Hibernate template's get() or load() method to retrieve the record by its primary key. * To update a record, use the Hibernate template's update() method to update the entity instance. * To delete a record, use the Hibernate template's delete() method to delete the entity instance by its primary key.  1. The transactions are handled by Spring framework internally, so no explicit commit or rollback needed in the DAO methods. |

. 1. create a new maven project in STS. Add the following dependencies

* 1. Spring core
  2. Spring context
  3. Spring ORM
  4. Hibernate Core Relocation
  5. Mysql connector

2. create a package for dao--- and in it

a. Entity class

|  |
| --- |
| **package** com.bitlabs.dao;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  @Entity  **public** **class** Student {  @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  **private** **int** id;  **private** String name;  **private** String address;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getAddress() {  **return** address;  }  **public** **void** setAddress(String address) {  **this**.address = address;  }  @Override  **public** String toString() {  **return** "Student [id=" + id + ", name=" + name + ", address=" + address + "]";  }      } |

b. Interface

|  |
| --- |
| **package** com.bitlabs.dao;  **import** java.util.List;  **public** **interface** StudentDaoInterface {  **public** **int** saveStudent(Student st);  **public** Student getStudent(**int** id);  **public** List<Student> getAllStudents();  **public** **void** UpdateStudent(Student st);  **public** **void** deleteStudent(**int** id);    } |

c. Implementation class –

use HibernateTemplate and Transactional in transaction.annotation.Transactional

|  |
| --- |
| **package** com.bitlabs.dao;  **import** java.util.List;  **import** org.springframework.orm.hibernate5.HibernateTemplate;  **import** org.springframework.transaction.annotation.Transactional;  **public** **class** StudentDaoImpl **implements** StudentDaoInterface{  **private** HibernateTemplate hibernatetemplate;      **public** HibernateTemplate getHibernatetemplate() {  **return** hibernatetemplate;  }  **public** **void** setHibernatetemplate(HibernateTemplate hibernatetemplate) {  **this**.hibernatetemplate = hibernatetemplate;  }  **@Transactional //important as it is changing table**  **public** **int** saveStudent(Student st) {    **int** i= (Integer)hibernatetemplate.save(st);  **return** i;  }  **public** Student getStudent(**int** id) {  Student st=hibernatetemplate.get(Student.**class**,id);    **return** st;  }  **public** List<Student> getAllStudents() {  List<Student> list= hibernatetemplate.loadAll(Student.**class**);  **return** list;  }  **@Transactional //important as it is changing table**  **public** **void** UpdateStudent(Student st) {  hibernatetemplate.update(st);  }    **@Transactional //important as it is changing table**  **public** **void** deleteStudent(**int** id) {  Student stm=hibernatetemplate.get(Student.**class**,id);  hibernatetemplate.delete(stm);    }  }  //@Transactional is not necessary for remaining methods as they are fetching the data from table. No change in the database table |

Now create another package and in that add

config.xml and

app.java files

config.xml

Initial configuaration

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:util=*"http://www.springframework.org/schema/util"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-3.0.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"*>  <tx:annotation-driven/> |

Now add the following beans for the following classes

**a) org.springframework.jdbc.datasource.DriverManagerDataSource with database driver, url, username and password**

Simple implementation of the standard JDBC javax.sql.DataSource interface, configuring the plain old JDBC java.sql.DriverManager via bean properties, and returning a new java.sql.Connection from every getConnection call.

|  |
| --- |
| <bean name=*"ds"*  class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"*  value=*"com.mysql.cj.jdbc.Driver"* />  <property name=*"url"*  value=*"jdbc:mysql://localhost:3306/hospital"* />  <property name=*"username"* value=*"root"* />  <property name=*"password"* value=*"root"* />  </bean> |

Hibernate requires a Session object in order to access the database. A Session is created  
from the SessionFactory.

**When using Spring framework, you can use LocalSessionFactoryBean to create a SessionFactory.**

**The LocalSessionFactoryBean requires a data-source to be wired in, along with hibernate properties and mapping resources. The hibernateProperties enables the properties such as database dialect, pool sizes, and other options**

|  |
| --- |
| <bean name=*"factory"* class=*"org.springframework.orm.hibernate5.LocalSessionFactoryBean"*>      <property name=*"dataSource"* ref=*"ds"* />  <property name=*"hibernateProperties"* >    <props>    <prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQL57Dialect</prop>  <prop key=*"hibernate.hbm2ddl.auto"*>update</prop>  <prop key=*"hibernate.show\_sql"*>true</prop>  <prop key=*"hibernate.format\_sql"*>true</prop>  </props>  </property>    <property name=*"annotatedClasses"*>  <list>  <value>com.bitlabs.dao.Student</value>  </list>  </property>    </bean> |

Now that the sessionFactory is defined, the next bit is to define HibernateTemplate.  
The HibernateTemplate requires a SessionFactory instance, so the following declaration  
wires the sessionFactory we defined earlier.

|  |
| --- |
| <bean name=*"template"* class=*"org.springframework.orm.hibernate5.HibernateTemplate"*>    <property name=*"SessionFactory"* ref=*"factory"*></property>    </bean> |

Now that the HibernateTransactionManager is defined

|  |
| --- |
| <bean name=*"transactionManager"* class=*"org.springframework.orm.hibernate5.HibernateTransactionManager"*>    <property name=*"sessionFactory"* ref=*"factory"*></property>    </bean> |

Create a bean for implementationclass

|  |
| --- |
| <bean name=*"stDao"* class=*"com.bitlabs.dao.StudentDaoImpl"*>    <property name=*"hibernatetemplate"* ref=*"template"*></property>    </bean> |