# Data Access in Spring Framework:

If you have to describe how data access is handled by Spring Framework using 2 keywords then those 2 words will be-

* Abstraction
* Agnostic

Now JDBC or any ORM framework do provide abstraction of its own and you use JDBC or ORM as an abstract layer to interact with DB then what abstraction does spring provide?

Again two keywords are-

* Template
* Exception Handling

Before delving any further into those keywords let’s talk about the “**agnostic**” feature too (which will cover the Exception handling point too!). It is said that data access exception handling in Spring framework is platform agnostic. What does that mean?

### Spring’s data access exception handling is platform agnostic

Spring framework it provides data access mechanism for getting data through **JDBC**, various ORM frameworks like **Hibernate**, **Ibatis**. Now think of a situation where Spring provides the layer over the JDBC or any ORM framework but doesn’t provide any mechanism to handle exceptions thrown by them. In that case your code will become a hotchpotch of Spring templates and then exception handling by JDBC, Hibernate, Ibatis or any other way of data access. That’s where Spring’s platform agnostic exception handling comes to the rescue.

For JDBC **org.springframework.jdbc.support** package provides *SQLException translation functionality* and some utility classes. Exceptions thrown during JDBC processing are translated to exceptions defined in the **org.springframework.dao** package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling.

All translated exceptions are unchecked exceptions, which gives you the option of catching the exceptions from which you can recover while allowing other exceptions to be propagated to the caller.

Same way implementations of **PersistenceExceptionTranslator** interface in Spring provides exception translation for data access technologies that throw runtime exceptions, such as JPA, TopLink, JDO and Hibernate.

Using these **translator classes** *Spring translates the platform specific exception to the Spring specific exception under the hood*, so you are abstracted from how it is done.

In order to take advantage of Spring’s data-access exceptions, you need to use one of the **data access templates** provided by Spring framework. That brings us to the point about templates.

### Using Templates for Data Access in Spring

Spring provides many templates for data access (For JDBC, for Hibernate, for MongoDB and many more). If you have idea about template design pattern you can get an idea what these templates are doing.

You generally use **Template design pattern** if you have to design the functionality where some implementation is common among the classes and some implementation differs. So in template design pattern -

* There is a base class with common implementations.
* Methods that require individual implementation are mere place holder in base class.
* Sub classes can provide implementation for those place holder methods.

So, base class provides the template of the implementation and fill what it can (common implementation) and delegates the other parts to the implementing classes.

Same process is used by the Spring data access. Whatever persistence mechanism is used there are some **common steps** like getting the DB connection, handling exception if thrown and cleaning up the resources (closing the connection) once done. These can be termed as **fixed part**.

But how and what data is accessed, what data is updated is different for different application. That can be termed as **variable part**.

Template classes in Spring framework provide implementation for the **fixed part** and uses a **callback approach to handle variable part** which is your custom data access code, benefit of template class is that it frees application code from having to do the boilerplate tasks like getting DB connection, handling exceptions, closing connection and results in code that is intention driven. That way the code that is written focuses solely on what the developer wants to do.

When you use the JdbcTemplate for your code, you only need to implement **callback interfaces**, giving them a clearly defined contract. The **PreparedStatementCreator** callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The same is true for the **CallableStatementCreator** interface, which creates callable statements. The **RowCallbackHandler** interface extracts values from each row of a ResultSet.

Some of the templates provided by Spring framework-

* **jdbc.core.JdbcTemplate**– For JDBC connections.
* **jdbc.core.namedparam.NamedParameterJdbcTemplate**- For JDBC connections with named parameters.
* **orm.hibernate3.HibernateTemplate**– For Hibernate 3.x sessions.
* **orm.hibernate4.HibernateTemplate**- For Hibernate 4.x sessions.
* **org.springframework.orm.hibernate5.HibernateTemplate**- For Hibernate 5.x sessions.

In order to manage the fixed part like getting connection, releasing resources Spring template needs a reference to a DataSource. Refer Configuring DataSource in Spring Framework to see different ways to configure data source in Spring.

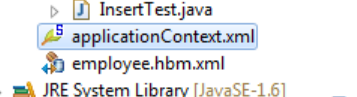
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Integrating Hibernate with Spring:

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.

 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.

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **float** salary;

//getters and setters

}

**3) employee.hbm.xml**

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

<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<**class** name="com.demo.Employee" table="emp558">

          <id name="id">

          <generator **class**="assigned"></generator>

          </id>

          <property name="name"></property>

          <property name="salary"></property>

</**class**>

</hibernate-mapping>

**4) EmployeeDao.java**

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

**import** org.springframework.orm.hibernate3.HibernateTemplate;

**import** java.util.\*;

**public** **class** EmployeeDao {

HibernateTemplate template;

**public** **void** setTemplate(HibernateTemplate template) {

**this**.template = template;

}

//method to save employee

**public** **void** saveEmployee(Employee e){

    template.save(e);

}

//method to update employee

**public** **void** updateEmployee(Employee e){

    template.update(e);

}

//method to delete employee

**public** **void** deleteEmployee(Employee e){

    template.delete(e);

}

//method to return one employee of given id

**public** Employee getById(**int** id){

    Employee e=(Employee)template.get(Employee.**class**,id);

**return** e;

}

//method to return all employees

**public** List<Employee> getEmployees(){

    List<Employee> list=**new** ArrayList<Employee>();

    list=template.loadAll(Employee.**class**);

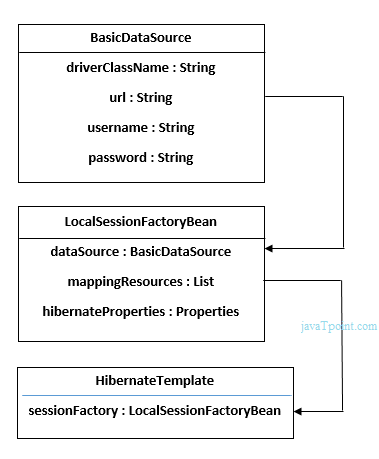
**return** list;

}

}

**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*

<?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-3.0.xsd">

    <bean id="dataSource" **class**="org.apache.commons.dbcp.BasicDataSource">

        <property name="driverClassName"  value="oracle.jdbc.driver.OracleDriver"></property>

        <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>

        <property name="username" value="system"></property>

        <property name="password" value="oracle"></property>

    </bean>

    <bean id="mysessionFactory"  **class**="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

        <property name="dataSource" ref="dataSource"></property>

        <property name="mappingResources">

        <list>

        <value>employee.hbm.xml</value>

        </list>

        </property>

        <property name="hibernateProperties">

            <props>

                <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>

                <prop key="hibernate.hbm2ddl.auto">update</prop>

                <prop key="hibernate.show\_sql">**true**</prop>

            </props>

        </property>

    </bean>

    <bean id="template" **class**="org.springframework.orm.hibernate3.HibernateTemplate">

    <property name="sessionFactory" ref="mysessionFactory"></property>

    </bean>

    <bean id=" empl " **class**="com.demo.EmployeeDao">

    <property name="template" ref="template"></property>

    </bean>

    </beans>

**6) InsertTest.java**

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

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.beans.factory.xml.XmlBeanFactory;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**public** **class** InsertTest {

**public** **static** **void** main(String[] args) {

    Resource r=**new** ClassPathResource("applicationContext.xml");

    BeanFactory factory=**new** XmlBeanFactory(r);

    EmployeeDao dao=(EmployeeDao)factory.getBean("empl");

    Employee e=**new** Employee();

    e.setId(114);

    e.setName("varun");

    e.setSalary(50000);

    dao.saveEmployee(e);

}

}

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

### **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:

<property name="hibernateProperties">

            <props>

                <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>

                <prop key="hibernate.hbm2ddl.auto">update</prop>

                <prop key="hibernate.show\_sql">**true**</prop>

            </props>