## **Spring Framework**

Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as [Struts](https://www.javatpoint.com/struts-2-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), Tapestry, [EJB](https://www.javatpoint.com/ejb-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

### Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

#### 1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

#### 2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

#### 3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

#### 4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

#### 5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

#### 6) Powerful abstraction

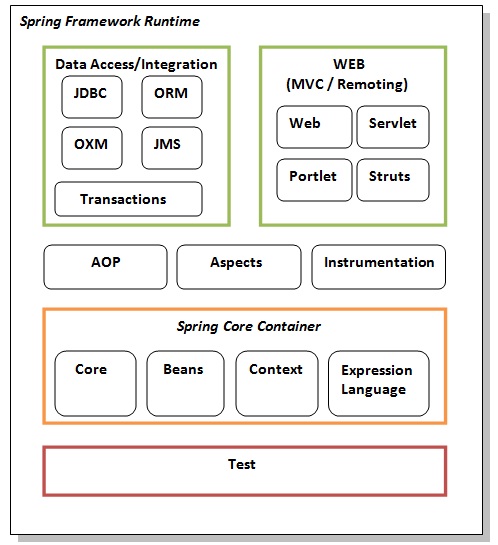
It provides powerful abstraction to JavaEE specifications such as [JMS](https://www.javatpoint.com/jms-tutorial), [JDBC](https://www.javatpoint.com/java-jdbc), JPA and JTA.

#### 7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Spring Modules

The Spring framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc. These modules are grouped into Test, Core Container, AOP, Aspects, Instrumentation, Data Access / Integration, Web (MVC / Remoting) as displayed in the following diagram.



Test

This layer provides support of testing with JUnit and TestNG.

Spring Core Container

The Spring Core container contains core, beans, context and expression language (EL) modules.

Core and Beans

These modules provide IOC and Dependency Injection features.

Context

This module supports internationalization (I18N), EJB, JMS, Basic Remoting.

Expression Language

It is an extension to the EL defined in JSP. It provides support to setting and getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrieval of objects by name etc.

AOP, Aspects and Instrumentation

These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.

The aspects module provides support to integration with AspectJ.

The instrumentation module provides support to class instrumentation and classloader implementations.

Data Access / Integration

This group comprises of JDBC, ORM, OXM, JMS and Transaction modules. These modules basically provide support to interact with the database.

Web

This group comprises of Web, Web-Servlet, Web-Struts and Web-Portlet. These modules provide support to create web application.

### Inversion Of Control (IOC) and Dependency Injection

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain. Let's understand this with the following code:

**class** Employee{

Address address;

Employee(){

address=**new** Address();

}

}

In such case, there is dependency between the Employee and Address (tight coupling). In the Inversion of Control scenario, we do this something like this:

**class** Employee{

Address address;

Employee(Address address){

**this**.address=address;

}

}

Thus, IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

#### Advantage of Dependency Injection

* makes the code loosely coupled so easy to maintain
* makes the code easy to test

IoC Container

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets information from the XML file and works accordingly. The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.**BeanFactory** and the org.springframework.context.**ApplicationContext** interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

1. Resource resource=**new** ClassPathResource("applicationContext.xml");
2. BeanFactory factory=**new** XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Using ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context =

**new** ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

# Creating spring application in Eclipse IDE

Here, we are going to create a simple application of spring framework using eclipse IDE. Let's see the simple steps to create the spring application in Eclipse IDE.

* **create the maven java project**
* **add dependencies in pom.xml**
* **create the class**
* **create the xml file to provide the values**
* **create the test class**

**Step1**

**package** sp;

**public** **class** Student {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** displayInfo(){

    System.out.println("Hello: "+name);

}

}

Add below dependency in Pom.xml

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.2</version>

</dependency>

Bean.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-4.0.xsd"*>

<bean id=*"studentbean"* class=*"sp.Student"*>

<property name=*"sname"* value=*"Vijay"*></property>

</bean>

</beans>

Step4

**package** mproject.com;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** sp.Student;

**public** **class** App

{

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

{

ApplicationContext ac=**new** ClassPathXmlApplicationContext("bean.xml");

Student s=(Student) ac.getBean("studentbean");

s.disp();

}

}

# Constructor Injection with Dependent Object

If there is HAS-A relationship between the classes, we create the instance of dependent object (contained object) first then pass it as an argument of the main class constructor. Here, our scenario is Employee HAS-A Address. The Address class object will be termed as the dependent object. Let's see the Address class first:

**Address.java**

This class contains three properties, one constructor and toString() method to return the values of these object.

**package** com.java;

**public** **class** Address {

**private** String city;

**private** String state;

**private** String country;

**public** Address(String city, String state, String country) {

**super**();

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

**public** String toString(){

**return** city+" "+state+" "+country;

}

}

**Employee.java**

It contains three properties id, name and address(dependent object) ,two constructors and show() method to show the records of the current object including the depedent object.

**package** com.java;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address address;//Aggregation

**public** Employee() {System.out.println("def cons");}

**public** Employee(**int** id, String name, Address address) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.address = address;

}

**void** show(){

    System.out.println(id+" "+name);

    System.out.println(address.toString());

}

}

**bean.xml**

The **ref** attribute is used to define the reference of another object, such way we are passing the dependent object as an constructor argument.

<?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="a1" **class**="com.java.Address">

<constructor-arg value="ghaziabad"></constructor-arg>

<constructor-arg value="UP"></constructor-arg>

<constructor-arg value="India"></constructor-arg>

</bean>

<bean id="e" **class**="com.java.Employee">

<constructor-arg value="12" type="int"></constructor-arg>

<constructor-arg value="Sonoo"></constructor-arg>

<constructor-arg>

<ref bean="a1"/>

</constructor-arg>

</bean>

</beans>

# Constructor Injection with Collection Example

We can inject collection values by constructor in spring framework. There can be used three elements inside the **constructor-arg** element.

It can be:

1. **list**
2. **set**
3. **map**

Each collection can have string based and non-string based values.

In this example, we are taking the example of Forum where **One question can have multiple answers**. There are three pages:

1. **Question.java**
2. **applicationContext.xml**
3. **Test.java**

In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

**Question.java**

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

**package** com.java;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** Question {

**private** **int** id;

**private** String name;

**private** List<String> answers;

**public** Question() {}

**public** Question(**int** id, String name, List<String> answers) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.answers = answers;

}

**public** **void** displayInfo(){

    System.out.println(id+" "+name);

    System.out.println("answers are:");

    Iterator<String> itr=answers.iterator();

**while**(itr.hasNext()){

        System.out.println(itr.next());

    }

}

}

**applicationContext.xml**

The list element of constructor-arg is used here to define the list.

<?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="q" **class**="com.java.Question">

<constructor-arg value="111"></constructor-arg>

<constructor-arg value="What is java?"></constructor-arg>

<constructor-arg>

<list>

<value>Java is a programming language</value>

<value>Java is a Platform</value>

<value>Java is an Island of Indonasia</value>

</list>

</constructor-arg>

</bean>

</beans>

# Constructor Injection with Non-String Collection (having Dependent Object) Example

If we have dependent object in the collection, we can inject these information by using the **ref** element inside the **list**, **set** or **map**.

In this example, we are taking the example of Forum where **One question can have multiple answers**. But Answer has its own information such as answerId, answer and postedBy. There are four pages used in this example:

1. **Question.java**
2. **Answer.java**
3. **applicationContext.xml**
4. **Test.java**

In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

**Question.java**

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

**package** com.java;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** Question {

**private** **int** id;

**private** String name;

**private** List<Answer> answers;

**public** Question() {}

**public** Question(**int** id, String name, List<Answer> answers) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.answers = answers;

}

**public** **void** displayInfo(){

    System.out.println(id+" "+name);

    System.out.println("answers are:");

    Iterator<Answer> itr=answers.iterator();

**while**(itr.hasNext()){

        System.out.println(itr.next());

    }

}

}

**Answer.java**

This class has three properties id, name and by with constructor and toString() method.

**package** com.java;

**public** **class** Answer {

**private** **int** id;

**private** String name;

**private** String by;

**public** Answer() {}

**public** Answer(**int** id, String name, String by) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.by = by;

}

**public** String toString(){

**return** id+" "+name+" "+by;

}

}

**applicationContext.xml**

The **ref** element is used to define the reference of another bean. Here, we are using **bean** attribute of **ref** element to specify the reference of another bean.

<?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="ans1" **class**="com.java.Answer">

<constructor-arg value="1"></constructor-arg>

<constructor-arg value="Java is a programming language"></constructor-arg>

<constructor-arg value="John"></constructor-arg>

</bean>

<bean id="ans2" **class**="com.java.Answer">

<constructor-arg value="2"></constructor-arg>

<constructor-arg value="Java is a Platform"></constructor-arg>

<constructor-arg value="Ravi"></constructor-arg>

</bean>

<bean id="q" **class**="com.java.Question">

<constructor-arg value="111"></constructor-arg>

<constructor-arg value="What is java?"></constructor-arg>

<constructor-arg>

<list>

<ref bean="ans1"/>

<ref bean="ans2"/>

</list>

</constructor-arg>

</bean>

</beans>

# Constructor Injection with Map Example

In this example, we are using **map** as the answer that have answer with posted username. Here, we are using key and value pair both as a string.

It is the example of forum where **one question can have multiple answers**.

**Question.java**

This class contains three properties, two constructors and displayInfo() method to display the information.

**package** com.java;

**import** java.util.Iterator;

**import** java.util.Map;

**import** java.util.Set;

**import** java.util.Map.Entry;

**public** **class** Question {

**private** **int** id;

**private** String name;

**private** Map<String,String> answers;

**public** Question() {}

**public** Question(**int** id, String name, Map<String, String> answers) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.answers = answers;

}

**public** **void** displayInfo(){

    System.out.println("question id:"+id);

    System.out.println("question name:"+name);

    System.out.println("Answers....");

    Set<Entry<String, String>> set=answers.entrySet();

    Iterator<Entry<String, String>> itr=set.iterator();

**while**(itr.hasNext()){

        Entry<String,String> entry=itr.next();

        System.out.println("Answer:"+entry.getKey()+" Posted By:"+entry.getValue());

    }

}

}

**applicationContext.xml**

The **entry** attribute of **map** is used to define the key and value information.

<?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="q" **class**="com.java.Question">

<constructor-arg value="11"></constructor-arg>

<constructor-arg value="What is Java?"></constructor-arg>

<constructor-arg>

<map>

<entry key="Java is a Programming Language"  value="Ajay Kumar"></entry>

<entry key="Java is a Platform" value="John Smith"></entry>

<entry key="Java is an Island" value="Raj Kumar"></entry>

</map>

</constructor-arg>

</bean>

</beans>

# Constructor Injection with Non-String Map (having dependent Object) Example

In this example, we are using **map** as the answer that have Answer and User. Here, we are using key and value pair both as an object. Answer has its own information such as answerId, answer and postedDate, User has its own information such as userId, username, emailId.

It is the example of forum where **one question can have multiple answers**.

**Question.java**

This class contains three properties, two constructors and displayInfo() method to display the information.

**package** com.java;

**import** java.util.Iterator;

**import** java.util.Map;

**import** java.util.Set;

**import** java.util.Map.Entry;

**public** **class** Question {

**private** **int** id;

**private** String name;

**private** Map<Answer,User> answers;

**public** Question() {}

**public** Question(**int** id, String name, Map<Answer, User> answers) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.answers = answers;

}

**public** **void** displayInfo(){

    System.out.println("question id:"+id);

    System.out.println("question name:"+name);

    System.out.println("Answers....");

    Set<Entry<Answer, User>> set=answers.entrySet();

    Iterator<Entry<Answer, User>> itr=set.iterator();

**while**(itr.hasNext()){

        Entry<Answer, User> entry=itr.next();

        Answer ans=entry.getKey();

        User user=entry.getValue();

        System.out.println("Answer Information:");

        System.out.println(ans);

        System.out.println("Posted By:");

        System.out.println(user);

    }

}

}

**Answer.java**

**package** com.java;

**import** java.util.Date;

**public** **class** Answer {

**private** **int** id;

**private** String answer;

**private** Date postedDate;

**public** Answer() {}

**public** Answer(**int** id, String answer, Date postedDate) {

**super**();

**this**.id = id;

**this**.answer = answer;

**this**.postedDate = postedDate;

}

**public** String toString(){

**return** "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;

}

}

**User.java**

**package** com.java;

**public** **class** User {

**private** **int** id;

**private** String name,email;

**public** User() {}

**public** User(**int** id, String name, String email) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.email = email;

}

**public** String toString(){

**return** "Id:"+id+" Name:"+name+" Email Id:"+email;

}

}

**applicationContext.xml**

The **key-ref** and **value-ref** attributes of entry **element** is used to define the reference of bean in the map.

<?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="answer1" **class**="com.java.Answer">

<constructor-arg value="1"></constructor-arg>

<constructor-arg value="Java is a Programming Language"></constructor-arg>

<constructor-arg value="12/12/2001"></constructor-arg>

</bean>

<bean id="answer2" **class**="com.java.Answer">

<constructor-arg value="2"></constructor-arg>

<constructor-arg value="Java is a Platform"></constructor-arg>

<constructor-arg value="12/12/2003"></constructor-arg>

</bean>

<bean id="user1" **class**="com.java.User">

<constructor-arg value="1"></constructor-arg>

<constructor-arg value="Arun Kumar"></constructor-arg>

<constructor-arg value="arun@gmail.com"></constructor-arg>

</bean>

<bean id="user2" **class**="com.java.User">

<constructor-arg value="2"></constructor-arg>

<constructor-arg value="Varun Kumar"></constructor-arg>

<constructor-arg value="Varun@gmail.com"></constructor-arg>

</bean>

<bean id="q" **class**="com.java.Question">

<constructor-arg value="1"></constructor-arg>

<constructor-arg value="What is Java?"></constructor-arg>

<constructor-arg>

<map>

<entry key-ref="answer1" value-ref="user1"></entry>

<entry key-ref="answer2" value-ref="user2"></entry>

</map>

</constructor-arg>

</bean>

</beans>

# Inheriting Bean in Spring

By using the **parent** attribute of **bean**, we can specify the inheritance relation between the beans. In such case, parent bean values will be inherited to the current bean.

Let's see the simple example to inherit the bean.

**Employee.java**

This class contains three properties, three constructor and show() method to display the values.

**package** com.java;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address address;

**public** Employee() {}

**public** Employee(**int** id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

**public** Employee(**int** id, String name, Address address) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.address = address;

}

**void** show(){

    System.out.println(id+" "+name);

    System.out.println(address);

}

}

**Address.java**

**package** com.java;

**public** **class** Address {

**private** String addressLine1,city,state,country;

**public** Address(String addressLine1, String city, String state, String country){

**super**();

**this**.addressLine1 = addressLine1;

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

**public** String toString(){

**return** addressLine1+" "+city+" "+state+" "+country;

}

}

**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="e1" **class**="com.java.Employee">

<constructor-arg value="101"></constructor-arg>

<constructor-arg  value="Sachin"></constructor-arg>

</bean>

<bean id="address1" **class**="com.java.Address">

<constructor-arg value="21,Lohianagar"></constructor-arg>

<constructor-arg value="Ghaziabad"></constructor-arg>

<constructor-arg value="UP"></constructor-arg>

<constructor-arg value="USA"></constructor-arg>

</bean>

<bean id="e2" **class**="com.java.Employee" parent="e1">

<constructor-arg ref="address1"></constructor-arg>

</bean>

</beans>

# Difference between constructor and setter injection

There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

# Autowiring in Spring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

## **Advantage of Autowiring**

It requires the **less code** because we don't need to write the code to inject the dependency explicitly.

## **Disadvantage of Autowiring**

No control of programmer.

It can't be used for primitive and string values.

## **Autowiring Modes**

There are many autowiring modes:

1)no It is the default autowiring mode. It means no autowiring bydefault.

2)byName The byName mode injects the object dependency according to name of the bean.

In such case, property name and bean name must be same.

It internally calls setter method.

3)byType The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.

4)constructor The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

5)autodetect It is deprecated since Spring 3.

## **Example of Autowiring**

Let's see the simple code to use autowiring in spring. You need to use autowire attribute of bean element to apply the autowire modes.

<bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

Let's see the full example of autowiring in spring. To create this example, we have created 4 files.

1. **B.java**
2. **A.java**
3. **applicationContext.xml**
4. **Test.java**

**B.java**

This class contains a constructor and method only.

**package** org.sssit;

**public** **class** B {

B(){System.out.println("b is created");}

**void** print(){System.out.println("hello b");}

}

**A.java**

This class contains reference of B class and constructor and method.

**package** org.sssit;

**public** **class** A {

B b;

A(){System.out.println("a is created");}

**public** B getB() {

**return** b;

}

**public** **void** setB(B b) {

**this**.b = b;

}

**void** print(){System.out.println("hello a");}

**void** display(){

    print();

    b.print();

}

}

**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="b" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

</beans>

## **1) byName autowiring mode**

In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

<bean id="b" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are changing the name of the bean from b to b1.

<bean id="b1" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

## **2) byType autowiring mode**

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.

It internally uses setter injection.

<bean id="b1" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="byType"></bean>

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

Let's see the code where are many bean of type B.

<bean id="b1" **class**="org.sssit.B"></bean>

<bean id="b2" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

In such case, it will throw exception.

## **3) constructor autowiring mode**

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

<bean id="b" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="constructor"></bean>

## **4) no autowiring mode**

In case of no autowiring mode, spring container doesn't inject the dependency by autowiring.

<bean id="b" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="no"></bean>

# Dependency Injection with Factory Method in Spring

Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of bean element.

1. **factory-method:** represents the factory method that will be invoked to inject the bean.
2. **factory-bean:** represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

A method that returns instance of a class is called **factory method**.

**public** **class** A {

**public** **static** A getA(){//factory method

**return** **new** A();

}

}

## **Factory Method Types**

There can be three types of factory method:

1) A **static factory method** that returns instance of **its own** class. It is used in singleton design pattern.

<bean id="a" **class**="com.java.A" factory-method="getA"></bean>

2) A **static factory method** that returns instance of **another** class. It is used instance is not known and decided at runtime.

<bean id="b" **class**="com.java.A" factory-method="getB"></bean>

3) A **non-static factory** method that returns instance of **another** class. It is used instance is not known and decided at runtime.

<bean id="a" **class**="com.java.A"></bean>

1. <bean id="b" **class**="com.java.A" factory-method="getB" factory-bean="a"></bean>

## **Type 1**

Let's see the simple code to inject the dependency by static factory method.

1. <bean id="a" **class**="com.java.A" factory-method="getA"></bean>

Let's see the full example to inject dependency using factory method in spring. To create this example, we have created 3 files.

1. **A.java**
2. **applicationContext.xml**
3. **Test.java**

**A.java**

This class is a singleton class.

**package** com.java;

**public** **class** A {

**private** **static** **final** A obj=**new** A();

**private** A(){System.out.println("private constructor");}

**public** **static** A getA(){

    System.out.println("factory method ");

**return** obj;

}

**public** **void** msg(){

    System.out.println("hello user");

}

}

**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="a" **class**="com.java.A" factory-method="getA"></bean>

</beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the msg method.

**package** org.sssit;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

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

    ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");

   A a=(A)context.getBean("a");

    a.msg();

}

}

Output:

private constructor

factory method

hello user

## **Type 2**

Let's see the simple code to inject the dependency by static factory method that returns the instance of another class.

To create this example, we have created 6 files.

1. **Printable.java**
2. **A.java**
3. **B.java**
4. **PrintableFactory.java**
5. **applicationContext.xml**
6. **Test.java**

**Printable.java**

**package** com.java;

**public** **interface** Printable {

**void** print();

}

**A.java**

**package** com.java;

**public** **class** A **implements** Printable{

    @Override

**public** **void** print() {

        System.out.println("hello a");

    }

}

**B.java**

**package** com.java;

**public** **class** B **implements** Printable{

    @Override

**public** **void** print() {

        System.out.println("hello b");

    }

}

**PrintableFactory.java**

**package** com.java;

**public** **class** PrintableFactory {

**public** **static** Printable getPrintable(){

    //return new B();

**return** **new** A();//return any one instance, either A or B

}

}

**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="p" **class**="com.java.PrintableFactory" factory-method="getPrintable"></bean>

</beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the print() method.

**package** org.sssit;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

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

    ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");

    Printable p=(Printable)context.getBean("p");

    p.print();

}

}

Output:

hello a

## **Type 3**

Let's see the example to inject the dependency by non-static factory method that returns the instance of another class.

To create this example, we have created 6 files.

1. **Printable.java**
2. **A.java**
3. **B.java**
4. **PrintableFactory.java**
5. **applicationContext.xml**
6. **Test.java**

All files are same as previous, you need to change only 2 files: PrintableFactory and applicationContext.xml.

**PrintableFactory.java**

**package** com.java;

**public** **class** PrintableFactory {

//non-static factory method

**public** Printable getPrintable(){

**return** **new** A();//return any one instance, either A or B

}

}

**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="pfactory" **class**="com.java.PrintableFactory"></bean>

<bean id="p" **class**="com.java.PrintableFactory" factory-method="getPrintable"

factory-bean="pfactory"></bean>

</beans>

Output:

hello a

**AnnotationBased configuration**

The **@Required** annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time. Otherwise, the container throws a BeanInitializationException exception. Following is an example to show the use of @Required annotation.

package com.java;

import org.springframework.beans.factory.annotation.Required;

public class Student {

private Integer age;

private String name;

@Required

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Required

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

Bean.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:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for student bean -->

<bean id = "student" class = "com.java.Student">

<property name = "name" value = "Zara" />

<property name = "age" value = "11"/>

</bean>

</beans>

## **@Autowired on Properties**

You can use **@Autowired** annotation on properties to get rid of the setter methods. When you will pass values of autowired properties using <property> Spring will automatically assign those properties with the passed values or references

**package** sp;

**public** **class** SpellChecker {

**public** SpellChecker(){

System.***out***.println("Inside SpellChecker constructor." );

}

**public** **void** checkSpelling(){

System.***out***.println("Inside checkSpelling." );

}

}

**package** sp;

**import** org.springframework.beans.factory.annotation.Autowired;

**public** **class** TextEditor {

@Autowired

**private** SpellChecker spellChecker;

**public** TextEditor() {

System.***out***.println("Inside TextEditor constructor." );

}

**public** SpellChecker getSpellChecker( ){

**return** spellChecker;

}

**public** **void** spellCheck(){

spellChecker.checkSpelling();

}

}

<?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:context = *"http://www.springframework.org/schema/context"*

xsi:schemaLocation = *"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id = *"textEditor"* class = *"sp.TextEditor"*>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = *"spellChecker"* class = *"sp.SpellChecker"*>

</bean>

</beans>

package sp;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

## **@Autowired on Setter Methods**

You can use **@Autowired** annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform **byType** autowiring on the method.

package com.java;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

package com.java;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker( ) {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

<?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:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id = "textEditor" class = "com.java.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.java.SpellChecker">

</bean>

</beans>

package com.java;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

## **@Autowired on Constructors**

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file

package com.java;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

package com.java;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

<?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:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id = "textEditor" class = "com.java.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.java.SpellChecker">

</bean>

</beans>

**@Qualifier**

|  |
| --- |
| The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.  There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired. Following is an example to show the use of @Qualifier annotation.  package com.java;  public class Student {  private Integer age;  private String name;  public void setAge(Integer age) {  this.age = age;  }  public Integer getAge() {  return age;  }  public void setName(String name) {  this.name = name;  }  public String getName() {  return name;  }  } |
|  |

package com.java;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

public class Profile {

@Autowired

@Qualifier("student1")

private Student student;

public Profile(){

System.out.println("Inside Profile constructor." );

}

public void printAge() {

System.out.println("Age : " + student.getAge() );

}

public void printName() {

System.out.println("Name : " + student.getName() );

}

}

Bean.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:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for profile bean -->

<bean id = "profile" class = "com.java.Profile"></bean>

<!-- Definition for student1 bean -->

<bean id = "student1" class = "com.java.Student">

<property name = "name" value = "Zara" />

<property name = "age" value = "11"/>

</bean>

<!-- Definition for student2 bean -->

<bean id = "student2" class = "com.java.Student">

<property name = "name" value = "Nuha" />

<property name = "age" value = "2"/>

</bean>

</beans>

package com.java;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

Profile profile = (Profile) context.getBean("profile");

profile.printAge();

profile.printName();

}

}

**Java Based Configuration**

Java-based configuration option enables you to write most of your Spring configuration without XML

## **@Configuration & @Bean Annotations**

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

**package** com.java;

**public** **class** HelloWorld {

**private** String message;

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

package com.java;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

The above code will be equivalent to the following XML configuration −

<beans>

<bean id = "helloWorld" class = "com.java.HelloWorld" />

</beans>

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

System.out.println( helloWorld.getMessage());

}

DAO Module

**Spring JDBC Template**

Spring JdbcTemplate is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

**Problems of JDBC API**

We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.

We need to perform exception handling code on the database logic.

We need to handle transaction.

Repetition of all these codes from one to another database logic is a time consuming task.

**Advantage of Spring JdbcTemplate**

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

**Spring Jdbc Approaches**

Spring framework provides following approaches for JDBC database access:

JdbcTemplate

NamedParameterJdbcTemplate

SimpleJdbcTemplate

SimpleJdbcInsert and SimpleJdbcCall

**JdbcTemplate class**

It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

It handles the exception and provides the informative exception messages by the help of exception classes defined in the org.springframework.dao package.

We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

Let's see the methods of spring JdbcTemplate class.

No. Method Description

1) public int update(String query) is used to insert, update and delete records.

2) public int update(String query,Object... args) is used to insert, update and delete records using PreparedStatement using given arguments.

3) public void execute(String query) is used to execute DDL query.

4) public T execute(String sql, PreparedStatementCallback action) executes the query by using PreparedStatement callback.

5) public T query(String sql, ResultSetExtractor rse) is used to fetch records using ResultSetExtractor.

6) public List query(String sql, RowMapper rse) is used to fetch records using RowMapper.

Example of Spring JdbcTemplate

We are assuming that you have created the following table inside the Oracle10g database.

create table employee(

id int,

name varchar(20),

salary double

);

Pom.xml

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com</groupId>

<artifactId>springjdbc</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>springjdbc</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.4</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.3.4</version>

</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.19</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

Employee.java

This class contains 3 properties with constructors and setter and getters.

package com.java;

public class Employee {

private int id;

private String name;

private float salary;

//no-arg and parameterized constructors

//getters and setters

}

EmployeeDao.java

It contains one property jdbcTemplate and three methods saveEmployee(), updateEmployee and deleteEmployee().

package com.java;

import org.springframework.jdbc.core.JdbcTemplate;

public class EmployeeDao {

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public int saveEmployee(Employee e){

String query="insert into employee values(

'"+e.getId()+"','"+e.getName()+"','"+e.getSalary()+"')";

return jdbcTemplate.update(query);

}

public int updateEmployee(Employee e){

String query="update employee set

name='"+e.getName()+"',salary='"+e.getSalary()+"' where id='"+e.getId()+"' ";

return jdbcTemplate.update(query);

}

public int deleteEmployee(Employee e){

String query="delete from employee where id='"+e.getId()+"' ";

return jdbcTemplate.update(query);

}

}

applicationContext.xml

The DriverManagerDataSource is used to contain the information about the database such as driver class name, connnection URL, username and password.

There are a property named datasource in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.

Here, we are using the JdbcTemplate object in the EmployeeDao class, so we are passing it by the setter method but you can use constructor also.

<?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="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name=*"driverClassName"* value=*"com.mysql.cj.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/dumdi"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"Password"* />

</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

<bean id="edao" class="com.java.EmployeeDao">

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

</bean>

</beans>

Test.java

This class gets the bean from the applicationContext.xml file and calls the saveEmployee() method. You can also call updateEmployee() and deleteEmployee() method by uncommenting the code as well.

package com.java;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");

int status=dao.saveEmployee(new Employee(102,"Amit",35000));

System.out.println(status);

/\*int status=dao.updateEmployee(new Employee(102,"Sonoo",15000));

System.out.println(status);

\*/

/\*Employee e=new Employee();

e.setId(102);

int status=dao.deleteEmployee(e);

System.out.println(status);\*/

}

}

**Example of PreparedStatement in Spring JdbcTemplate**

We can execute parameterized query using Spring JdbcTemplate by the help of execute() method of JdbcTemplate class. To use parameterized query, we pass the instance of PreparedStatementCallback in the execute method.

Syntax of execute method to use parameterized query

public T execute(String sql,PreparedStatementCallback<T>);

PreparedStatementCallback interface

It processes the input parameters and output results. In such case, you don't need to care about single and double quotes.

Method of PreparedStatementCallback interface

It has only one method doInPreparedStatement. Syntax of the method is given below:

public T doInPreparedStatement(PreparedStatement ps)throws SQLException, DataAccessException

create table employee(

id int,

name varchar(20),

salary int

);

Employee.java

This class contains 3 properties with constructors and setter and getters.

package com.java;

public class Employee {

private int id;

private String name;

private float salary;

//no-arg and parameterized constructors

//getters and setters

}

EmployeeDao.java

It contains one property jdbcTemplate and one method saveEmployeeByPreparedStatement. You must understand the concept of annonymous class to understand the code of the method.

package com.java;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import org.springframework.dao.DataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.PreparedStatementCallback;

public class EmployeeDao {

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public Boolean saveEmployeeByPreparedStatement(final Employee e){

String query="insert into employee values(?,?,?)";

return jdbcTemplate.execute(query,new PreparedStatementCallback<Boolean>(){

@Override

public Boolean doInPreparedStatement(PreparedStatement ps)

throws SQLException, DataAccessException {

ps.setInt(1,e.getId());

ps.setString(2,e.getName());

ps.setFloat(3,e.getSalary());

return ps.execute();

}

});

}

}

applicationContext.xml

The DriverManagerDataSource is used to contain the information about the database such as driver class name, connnection URL, username and password.

There are a property named datasource in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.

Here, we are using the JdbcTemplate object in the EmployeeDao class, so we are passing it by the setter method but you can use constructor also.

<?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="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name=*"driverClassName"* value=*"com.mysql.cj.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/dumdi"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"Password"* />

</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

<bean id="edao" class="com.java.EmployeeDao">

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

</bean>

</beans>

Test.java

This class gets the bean from the applicationContext.xml file and calls the saveEmployeeByPreparedStatement() method.

package com.java;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");

dao.saveEmployeeByPreparedStatement(new Employee(108,"Amit",35000));

}

}

Fetching Records by Spring JdbcTemplate

We can easily fetch the records from the database using query() method of JdbcTemplate class where we need to pass the instance of ResultSetExtractor.

Syntax of query method using ResultSetExtractor

public T query(String sql,ResultSetExtractor<T> rse)

ResultSetExtractor Interface

ResultSetExtractor interface can be used to fetch records from the database. It accepts a ResultSet and returns the list.

Method of ResultSetExtractor interface

It defines only one method extractData that accepts ResultSet instance as a parameter. Syntax of the method is given below:

public T extractData(ResultSet rs)throws SQLException,DataAccessException

Example of ResultSetExtractor Interface to show all the records of the table

create table employee(

id int,

name varchar(20),

salary double

);

Employee.java

This class contains 3 properties with constructors and setter and getters. It defines one extra method toString().

package com.java;

public class Employee {

private int id;

private String name;

private float salary;

//no-arg and parameterized constructors

//getters and setters

public String toString(){

return id+" "+name+" "+salary;

}

}

EmployeeDao.java

It contains on property jdbcTemplate and one method getAllEmployees.

package com.java;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.List;

import org.springframework.dao.DataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.ResultSetExtractor;

public class EmployeeDao {

private JdbcTemplate template;

public void setTemplate(JdbcTemplate template) {

this.template = template;

}

public List<Employee> getAllEmployees(){

return template.query("select \* from employee",new ResultSetExtractor<List<Employee>>(){

@Override

public List<Employee> extractData(ResultSet rs) throws SQLException,

DataAccessException {

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

while(rs.next()){

Employee e=new Employee();

e.setId(rs.getInt(1));

e.setName(rs.getString(2));

e.setSalary(rs.getInt(3));

list.add(e);

}

return list;

}

});

}

}

applicationContext.xml

The DriverManagerDataSource is used to contain the information about the database such as driver class name, connnection URL, username and password.

There are a property named datasource in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.

Here, we are using the JdbcTemplate object in the EmployeeDao class, so we are passing it by the setter method but you can use constructor also.

<?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="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name=*"driverClassName"* value=*"com.mysql.cj.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/dumdi"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"Password"* />

</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

<bean id="edao" class="com.java.EmployeeDao">

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

</bean>

</beans>

Test.java

This class gets the bean from the applicationContext.xml file and calls the getAllEmployees() method of EmployeeDao class.

package com.java;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");

List<Employee> list=dao.getAllEmployees();

for(Employee e:list)

System.out.println(e);

}

}

**RowMapper Example | Fetching records by Spring JdbcTemplate**

Like ResultSetExtractor, we can use RowMapper interface to fetch the records from the database using query() method of JdbcTemplate class. In the execute of we need to pass the instance of RowMapper now.

Syntax of query method using RowMapper

public T query(String sql,RowMapper<T> rm)

RowMapper Interface

RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

Advantage of RowMapper over ResultSetExtractor

RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.

Method of RowMapper interface

It defines only one method mapRow that accepts ResultSet instance and int as the parameter list. Syntax of the method is given below:

public T mapRow(ResultSet rs, int rowNumber)throws SQLException

Example of RowMapper Interface to show all the records of the table

create table employee(

id int,

name varchar(20),

salary double

);

Employee.java

This class contains 3 properties with constructors and setter and getters and one extra method toString().

package com.java;

public class Employee {

private int id;

private String name;

private float salary;

//no-arg and parameterized constructors

//getters and setters

public String toString(){

return id+" "+name+" "+salary;

}

}

EmployeeDao.java

It contains on property jdbcTemplate and one method getAllEmployeesRowMapper.

package com.java;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.List;

import org.springframework.dao.DataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.ResultSetExtractor;

import org.springframework.jdbc.core.RowMapper;

public class EmployeeDao {

private JdbcTemplate template;

public void setTemplate(JdbcTemplate template) {

this.template = template;

}

public List<Employee> getAllEmployeesRowMapper(){

return template.query("select \* from employee",new RowMapper<Employee>(){

@Override

public Employee mapRow(ResultSet rs, int rownumber) throws SQLException {

Employee e=new Employee();

e.setId(rs.getInt(1));

e.setName(rs.getString(2));

e.setSalary(rs.getInt(3));

return e;

}

});

}

}

applicationContext.xml

The DriverManagerDataSource is used to contain the information about the database such as driver class name, connnection URL, username and password.

There are a property named datasource in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.

Here, we are using the JdbcTemplate object in the EmployeeDao class, so we are passing it by the setter method but you can use constructor also.

<?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="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name=*"driverClassName"* value=*"com.mysql.cj.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/dumdi"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"Password"* />

</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

<bean id="edao" class="com.java.EmployeeDao">

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

</bean>

</beans>

Test.java

This class gets the bean from the applicationContext.xml file and calls the getAllEmployeesRowMapper() method of EmployeeDao class.

package com.java;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");

List<Employee> list=dao.getAllEmployeesRowMapper();

for(Employee e:list)

System.out.println(e);

}

}

Spring NamedParameterJdbcTemplate Example

Spring provides another way to insert data by named parameter. In such way, we use names instead of ?(question mark). So it is better to remember the data for the column.

Simple example of named parameter query

insert into employee values (:id,:name,:salary)

Method of NamedParameterJdbcTemplate class

In this example,we are going to call only the execute method of NamedParameterJdbcTemplate class. Syntax of the method is as follows:

pubic T execute(String sql,Map map,PreparedStatementCallback psc)

Example of NamedParameterJdbcTemplate class

We are assuming that you have created the following table inside the Oracle10g database.

create table employee(

id int,

name varchar(20),

salary double

);

Employee.java

This class contains 3 properties with constructors and setter and getters.

package com.java;

public class Employee {

private int id;

private String name;

private float salary;

//no-arg and parameterized constructors

//getters and setters

}

EmployeeDao.java

It contains on property jdbcTemplate and one method save.

package com.java;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import org.springframework.dao.DataAccessException;

import org.springframework.jdbc.core.PreparedStatementCallback;

import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

import java.util.\*;

public class EmployeeDao {

NamedParameterJdbcTemplate template;

public EmployeeDao(NamedParameterJdbcTemplate template) {

this.template = template;

}

public void save (Employee e){

String query="insert into employee values (:id,:name,:salary)";

Map<String,Object> map=new HashMap<String,Object>();

map.put("id",e.getId());

map.put("name",e.getName());

map.put("salary",e.getSalary());

template.execute(query,map,new PreparedStatementCallback() {

public Object doInPreparedStatement(PreparedStatement ps)

throws SQLException, DataAccessException {

return ps.executeUpdate();

}

});

}

}

applicationContext.xml

The DriverManagerDataSource is used to contain the information about the database such as driver class name, connnection URL, username and password.

There are a property named datasource in the NamedParameterJdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the NamedParameterJdbcTemplate class for the datasource property.

Here, we are using the NamedParameterJdbcTemplate object in the EmployeeDao class, so we are passing it by the constructor but you can use setter method also.

<?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="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name=*"driverClassName"* value=*"com.mysql.cj.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/dumdi"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"Password"* />

</bean>

<bean id="jtemplate"

class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">

<constructor-arg ref="ds"></constructor-arg>

</bean>

<bean id="edao" class="com.java.EmpDao">

<constructor-arg>

<ref bean="jtemplate"/>

</constructor-arg>

</bean>

</beans>

SimpleTest.java

This class gets the bean from the applicationContext.xml file and calls the save method.

package com.java;

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 SimpleTest {

public static void main(String[] args) {

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

BeanFactory factory=new XmlBeanFactory(r);

EmpDao dao=(EmpDao)factory.getBean("edao");

dao.save(new Emp(23,"sonoo",50000));

}

}

Spring SimpleJdbcTemplate Example

Spring 3 JDBC supports the java 5 feature var-args (variable argument) and autoboxing by the help of SimpleJdbcTemplate class.

SimpleJdbcTemplate class wraps the JdbcTemplate class and provides the update method where we can pass arbitrary number of arguments.

Syntax of update method of SimpleJdbcTemplate class

int update(String sql,Object... parameters)

We should pass the parameter values in the update method in the order they are defined in the parameterized query.

Example of SimpleJdbcTemplate class

We are assuming that you have created the following table inside the Oracle10g database.

create table employee(

id int,

name varchar(20),

salary double

);

Employee.java

This class contains 3 properties with constructors and setter and getters.

package com.java;

public class Employee {

private int id;

private String name;

private float salary;

//no-arg and parameterized constructors

//getters and setters

}

EmployeeDao.java

It contains one property SimpleJdbcTemplate and one method update. In such case, update method will update only name for the corresponding id. If you want to update the name and salary both, comment the above two lines of code of the update method and uncomment the 2 lines of code given below.

package com.java;

import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;

public class EmpDao {

SimpleJdbcTemplate template;

public EmpDao(SimpleJdbcTemplate template) {

this.template = template;

}

public int update (Emp e){

String query="update employee set name=? where id=?";

return template.update(query,e.getName(),e.getId());

//String query="update employee set name=?,salary=? where id=?";

//return template.update(query,e.getName(),e.getSalary(),e.getId());

}

}

applicationContext.xml

The DriverManagerDataSource is used to contain the information about the database such as driver class name, connnection URL, username and password.

There are a property named datasource in the SimpleJdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the SimpleJdbcTemplate class for the datasource property.

Here, we are using the SimpleJdbcTemplate object in the EmployeeDao class, so we are passing it by the constructor but you can use setter method also.

<?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="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

</bean>

<bean id="jtemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">

<constructor-arg ref="ds"></constructor-arg>

</bean>

<bean id="edao" class="com.java.EmpDao">

<constructor-arg>

<ref bean="jtemplate"/>

</constructor-arg>

</bean>

</beans>

SimpleTest.java

This class gets the bean from the applicationContext.xml file and calls the update method of EmpDao class.

package com.java;

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 SimpleTest {

public static void main(String[] args) {

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

BeanFactory factory=new XmlBeanFactory(r);

EmpDao dao=(EmpDao)factory.getBean("edao");

int status=dao.update(new Emp(23,"Tarun",35000));

System.out.println(status);

}

}