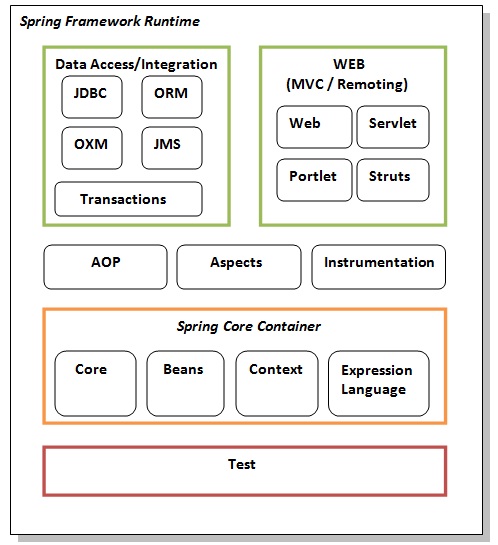
**SPRING**

Spring is a lightweight framework **developed by Rod Johnson in 2003**. 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. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

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.



## **Core Container**

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

* The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
* The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
* The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
* The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

## **Data Access/Integration**

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

* The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
* The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
* The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
* The Java Messaging Service **JMS** module contains features for producing and consuming messages.
* The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

## **Web**

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

* The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
* The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
* The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
* The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

## **Miscellaneous**

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

* The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
* The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
* The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
* The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
* The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

# **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 java project**
* **add spring jar files**
* **create the class**
* **create the xml file to provide the values**
* **create the test class**

## **Steps to create spring application in Eclipse IDE**

Let's see the 5 steps to create the first spring application using eclipse IDE.

### 1) Create the Java Project

**File** ->**New** 🡪 **project** 🡪**Java**🡪 **Java Project**.

Write the project name e.g. firstspring - **Finish.**

### 2) Add spring jar files

There are mainly three jar files required to run this application.

* **org.springframework.core-3.0.1.RELEASE-A**
* **com.springsource.org.apache.commons.logging-1.1.1**
* **org.springframework.beans-3.0.1.RELEASE-A**

For the future use, You can download the required jar files for spring core application.

[download the core jar files for spring](https://static.javatpoint.com/src/sp/spcorejars.zip)

[download the all jar files for spring including aop, mvc, j2ee, remoting, oxm, etc.](https://static.javatpoint.com/src/sp/springjars.zip)

To load the jar files in eclipse IDE, **Right click on your project** - **Build Path** - **Add external archives** - **select all the required jar files** - **finish.**.

### 3) Create Java class

**Right click on src** - **New**- **class** - **Write the class name e.g. Student** - **finish**. Write the following code:

**package** com.firstspring

**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);

}

}

This is simple bean class, containing only one property name with its getters and setters method. This class contains one extra method named displayInfo() that prints the student name by the hello message.

### 4) Create the xml file

To create the xml file click on src - new - file - give the file name such as applicationContext.xml - finish. Open the applicationContext.xml file, and write the following code:

<?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="studentbean" **class**="firstsrping.Student">

<property name="name" value="Vimal Jaiswal"></property>

</bean>

</beans>

The **bean** element is used to define the bean for the given class. The **property** sub element of bean specifies the property of the Student class named name. The value specified in the property element will be set in the Student class object by the IOC container.

### 5) Create the test class

Create the java class e.g. Test. Here we are getting the object of Student class from the IOC container using the getBean() method of BeanFactory. Let's see the code of test class.

**package** firstspring;

**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** Test {

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

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

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

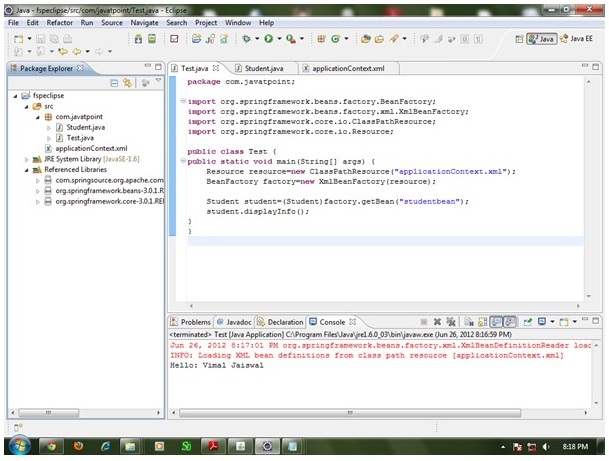
    Student student=(Student)factory.getBean("studentbean");

    student.displayInfo();

}

}

Now run this class. You will get the output Hello: Vimal Jaiswal.



# Spring - IoC Containers

  (Inversion of Control)

# 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**

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:

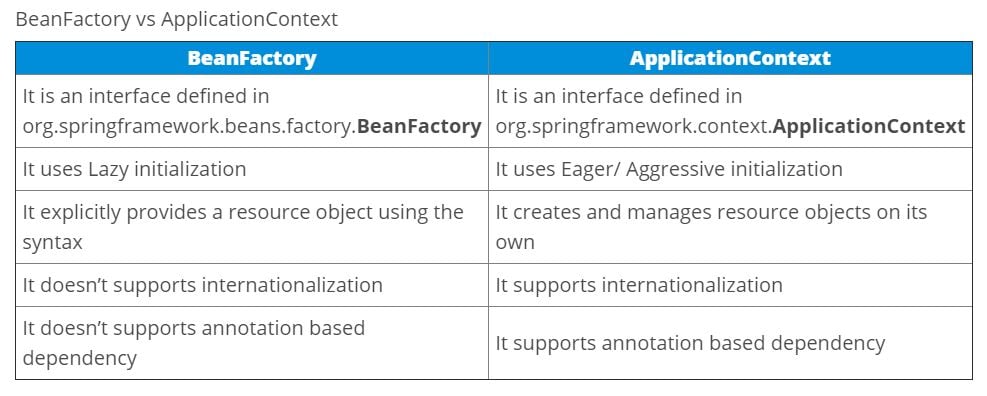
1. Resource resource=**new** ClassPathResource("applicationContext.xml");
2. BeanFactory factory=**new** XmlBeanFactory(resource);

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

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.



# **Dependency Injection in Spring**

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled. To understand the DI better, Let's understand the Dependency Lookup (DL) first:

Dependency Lookup

The Dependency Lookup is an approach where we get the resource after demand. There can be various ways to get the resource for example:

1. A obj = **new** AImpl();

In such way, we get the resource(instance of A class) directly by new keyword. Another way is factory method:

1. A obj = A.getA();

This way, we get the resource (instance of A class) by calling the static factory method getA().

Alternatively, we can get the resource by JNDI (Java Naming Directory Interface) as:

1. Context ctx = **new** InitialContext();
2. Context environmentCtx = (Context) ctx.lookup("java:comp/env");
3. A obj = (A)environmentCtx.lookup("A");

There can be various ways to get the resource to obtain the resource. Let's see the problem in this approach.

Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

* **tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
* **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

Dependency Injection

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

1. **class** Employee{
2. Address address;
4. Employee(Address address){
5. **this**.address=address;
6. }
7. **public** **void** setAddress(Address address){
8. **this**.address=address;
9. }
11. }

In such case, instance of Address class is provided by external souce such as XML file either by constructor or setter method.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

# **Dependency Injection by Constructor**

We can inject the dependency by constructor. The **<constructor-arg>** subelement of **<bean>** is used for constructor injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

Injecting primitive and string-based values

Let's see the simple example to inject primitive and string-based values. We have created three files here:

* Employee.java
* applicationContext.xml
* Test.java

**Employee.java**

It is a simple class containing two fields id and name. There are four constructors and one method in this class.

**package** com.javatpoint;

**public** **class** Employee {

**private** **int** id;

**private** String name;

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

**public** Employee(**int** id) {**this**.id = id;}

**public** Employee(String name) {  **this**.name = name;}

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

**this**.id = id;

**this**.name = name;

}

**void** show(){

    System.out.println(id+" "+name);

}

}

**applicationContext.xml**

We are providing the information into the bean by this file. The constructor-arg element invokes the constructor. In such case, parameterized constructor of int type will be invoked. The value attribute of constructor-arg element will assign the specified value. The type attribute specifies that int parameter constructor will be invoked.

<?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="e" **class**="com.javatpoint.Employee">

<constructor-arg value="10" type="int"></constructor-arg>

</bean>

</beans>

**Test.java**

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

1. **package** com.javatpoint;

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

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

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

**public** **class** Test {

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

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

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

        Employee s=(Employee)factory.getBean("e");

        s.show();

    }

}

**Output:**10 null

Injecting string-based values

If you don't specify the type attribute in the constructor-arg element, by default string type constructor will be invoked.

1. ....
2. <bean id="e" **class**="com.javatpoint.Employee">
3. <constructor-arg value="10"></constructor-arg>
4. </bean>
5. ....

If you change the bean element as given above, string parameter constructor will be invoked and the output will be 0 10.

**Output:**0 10

You may also pass the string literal as following:

1. ....
2. <bean id="e" **class**="com.javatpoint.Employee">
3. <constructor-arg value="Sonoo"></constructor-arg>
4. </bean>
5. ....

**Output:**0 Sonoo

You may pass integer literal and string both as following

1. ....

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

<constructor-arg value="10" type="int" ></constructor-arg>

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

</bean>

1. ....

We can inject the dependency by setter method also. The **<property>** subelement of **<bean>** is used for setter injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

Injecting primitive and string-based values by setter method

Let's see the simple example to inject primitive and string-based values by setter method. We have created three files here:

* Employee.java
* applicationContext.xml
* Test.java

**Employee.java**

It is a simple class containing three fields id, name and city with its setters and getters and a method to display these informations.

**package** com.javatpoint;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** String city;

**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 getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

**void** display(){

    System.out.println(id+" "+name+" "+city);

}

}

**applicationContext.xml**

We are providing the information into the bean by this file. The property element invokes the setter method. The value subelement of property will assign the specified value.

<?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="obj" **class**="com.javatpoint.Employee">

<property name="id">

<value>20</value>

</property>

<property name="name">

<value>Arun</value>

</property>

<property name="city">

<value>ghaziabad</value>

</property>

</bean>

</beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.\*;
7. **public** **class** Test {
8. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Employee e=(Employee)factory.getBean("obj");
14. s.display();
16. }
17. }

# **Constructor Injection with Dependent Object**

1. [Constructor Injection with Dependent Object](https://www.javatpoint.com/spring-tutorial-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.

1. **package** com.javatpoint;
3. **public** **class** Address {
4. **private** String city;
5. **private** String state;
6. **private** String country;
8. **public** Address(String city, String state, String country) {
9. **super**();
10. **this**.city = city;
11. **this**.state = state;
12. **this**.country = country;
13. }
15. **public** String toString(){
16. **return** city+" "+state+" "+country;
17. }
18. }

**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.

1. **package** com.javatpoint;

**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());

}

}

**applicationContext.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.javatpoint.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.javatpoint.Employee">

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

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

<constructor-arg>

<ref bean="a1"/>

</constructor-arg>

</bean>

</beans>

**Test.java**

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

1. **package** com.javatpoint;

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

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

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

**public** **class** Test {

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

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

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

        Employee s=(Employee)factory.getBean("e");

        s.show();

    }

}

# **Constructor Injection with Collection Example**

1. [Constructor Injection with Collection](https://www.javatpoint.com/spring-tutorial-constructor-injection-with-collection)

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.javatpoint;

**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.javatpoint.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>

**Test.java**

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

**package** com.javatpoint;

**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** Test {

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

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

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

    Question q=(Question)factory.getBean("q");

    q.displayInfo();

}

}

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

1. [Constructor Injection with Non-String Collection](https://www.javatpoint.com/spring-tutorial-constructor-injection-with-non-string-collection)

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.

1. **package** com.javatpoint;

**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.javatpoint;

**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.javatpoint.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.javatpoint.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.javatpoint.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>

**Test.java**

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

**package** com.javatpoint;

**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** Test {

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

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

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

    Question q=(Question)factory.getBean("q");

    q.displayInfo();

}

}

# **Constructor Injection with Map Example**

1. [Constructor Injection with Map Example](https://www.javatpoint.com/spring-tutorial-constructor-injection-with-map)

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.

Like previous examples, 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.javatpoint;

**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.javatpoint.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>

**Test.java**

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

**package** com.javatpoint;

**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** Test {

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

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

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

    Question q=(Question)factory.getBean("q");

    q.displayInfo();

}

}

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

1. [Constructor Injection with Non-String Map](https://www.javatpoint.com/spring-tutorial-constructor-injection-with-non-string-map)

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.

Like previous examples, 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.

1. **package** com.javatpoint;
2. **import** java.util.Iterator;
3. **import** java.util.Map;
4. **import** java.util.Set;
5. **import** java.util.Map.Entry;
7. **public** **class** Question {
8. **private** **int** id;
9. **private** String name;
10. **private** Map<Answer,User> answers;
12. **public** Question() {}
13. **public** Question(**int** id, String name, Map<Answer, User> answers) {
14. **super**();
15. **this**.id = id;
16. **this**.name = name;
17. **this**.answers = answers;
18. }

21. **public** **void** displayInfo(){
22. System.out.println("question id:"+id);
23. System.out.println("question name:"+name);
24. System.out.println("Answers....");
25. Set<Entry<Answer, User>> set=answers.entrySet();
26. Iterator<Entry<Answer, User>> itr=set.iterator();
27. **while**(itr.hasNext()){
28. Entry<Answer, User> entry=itr.next();
29. Answer ans=entry.getKey();
30. User user=entry.getValue();
31. System.out.println("Answer Information:");
32. System.out.println(ans);
33. System.out.println("Posted By:");
34. System.out.println(user);
35. }
36. }
37. }

**Answer.java**

1. **package** com.javatpoint;
3. **import** java.util.Date;
5. **public** **class** Answer {
6. **private** **int** id;
7. **private** String answer;
8. **private** Date postedDate;
9. **public** Answer() {}
10. **public** Answer(**int** id, String answer, Date postedDate) {
11. **super**();
12. **this**.id = id;
13. **this**.answer = answer;
14. **this**.postedDate = postedDate;
15. }
17. **public** String toString(){
18. **return** "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;
19. }
20. }

**User.java**

1. **package** com.javatpoint;
3. **public** **class** User {
4. **private** **int** id;
5. **private** String name,email;
6. **public** User() {}
7. **public** User(**int** id, String name, String email) {
8. **super**();
9. **this**.id = id;
10. **this**.name = name;
11. **this**.email = email;
12. }
14. **public** String toString(){
15. **return** "Id:"+id+" Name:"+name+" Email Id:"+email;
16. }
17. }

**applicationContext.xml**

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

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="answer1" **class**="com.javatpoint.Answer">
10. <constructor-arg value="1"></constructor-arg>
11. <constructor-arg value="Java is a Programming Language"></constructor-arg>
12. <constructor-arg value="12/12/2001"></constructor-arg>
13. </bean>
14. <bean id="answer2" **class**="com.javatpoint.Answer">
15. <constructor-arg value="2"></constructor-arg>
16. <constructor-arg value="Java is a Platform"></constructor-arg>
17. <constructor-arg value="12/12/2003"></constructor-arg>
18. </bean>
20. <bean id="user1" **class**="com.javatpoint.User">
21. <constructor-arg value="1"></constructor-arg>
22. <constructor-arg value="Arun Kumar"></constructor-arg>
23. <constructor-arg value="arun@gmail.com"></constructor-arg>
24. </bean>
25. <bean id="user2" **class**="com.javatpoint.User">
26. <constructor-arg value="2"></constructor-arg>
27. <constructor-arg value="Varun Kumar"></constructor-arg>
28. <constructor-arg value="Varun@gmail.com"></constructor-arg>
29. </bean>
31. <bean id="q" **class**="com.javatpoint.Question">
32. <constructor-arg value="1"></constructor-arg>
33. <constructor-arg value="What is Java?"></constructor-arg>
34. <constructor-arg>
35. <map>
36. <entry key-ref="answer1" value-ref="user1"></entry>
37. <entry key-ref="answer2" value-ref="user2"></entry>
38. </map>
39. </constructor-arg>
40. </bean>
42. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method to display the information.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Question q=(Question)factory.getBean("q");
14. q.displayInfo();
16. }
17. }

# **Inheriting Bean in Spring**

1. [Inheriting Bean in Spring](https://www.javatpoint.com/spring-tutorial-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.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** Address address;
7. **public** Employee() {}
9. **public** Employee(**int** id, String name) {
10. **super**();
11. **this**.id = id;
12. **this**.name = name;
13. }
14. **public** Employee(**int** id, String name, Address address) {
15. **super**();
16. **this**.id = id;
17. **this**.name = name;
18. **this**.address = address;
19. }
21. **void** show(){
22. System.out.println(id+" "+name);
23. System.out.println(address);
24. }
26. }

**Address.java**

1. **package** com.javatpoint;
3. **public** **class** Address {
4. **private** String addressLine1,city,state,country;
6. **public** Address(String addressLine1, String city, String state, String country) {
7. **super**();
8. **this**.addressLine1 = addressLine1;
9. **this**.city = city;
10. **this**.state = state;
11. **this**.country = country;
12. }
13. **public** String toString(){
14. **return** addressLine1+" "+city+" "+state+" "+country;
15. }
17. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="e1" **class**="com.javatpoint.Employee">
10. <constructor-arg value="101"></constructor-arg>
11. <constructor-arg  value="Sachin"></constructor-arg>
12. </bean>
14. <bean id="address1" **class**="com.javatpoint.Address">
15. <constructor-arg value="21,Lohianagar"></constructor-arg>
16. <constructor-arg value="Ghaziabad"></constructor-arg>
17. <constructor-arg value="UP"></constructor-arg>
18. <constructor-arg value="USA"></constructor-arg>
19. </bean>
21. <bean id="e2" **class**="com.javatpoint.Employee" parent="e1">
22. <constructor-arg ref="address1"></constructor-arg>
23. </bean>
25. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Employee e1=(Employee)factory.getBean("e2");
14. e1.show();
16. }
17. }

# **Setter Injection with Dependent Object Example**

1. [Setter Injection with Dependent Object](https://www.javatpoint.com/spring-tutorial-setter-injection-with-dependent-object)

Like Constructor Injection, we can inject the dependency of another bean using setters. In such case, we use **property** element. 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 four properties, setters and getters and toString() method.

1. **package** com.javatpoint;
3. **public** **class** Address {
4. **private** String addressLine1,city,state,country;
6. //getters and setters
8. **public** String toString(){
9. **return** addressLine1+" "+city+" "+state+" "+country;
10. }

**Employee.java**

It contains three properties id, name and address(dependent object) , setters and getters with displayInfo() method.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** Address address;
8. //setters and getters
10. **void** displayInfo(){
11. System.out.println(id+" "+name);
12. System.out.println(address);
13. }
14. }

**applicationContext.xml**

The **ref** attribute of **property** elements is used to define the reference of another bean.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="address1" **class**="com.javatpoint.Address">
10. <property name="addressLine1" value="51,Lohianagar"></property>
11. <property name="city" value="Ghaziabad"></property>
12. <property name="state" value="UP"></property>
13. <property name="country" value="India"></property>
14. </bean>
16. <bean id="obj" **class**="com.javatpoint.Employee">
17. <property name="id" value="1"></property>
18. <property name="name" value="Sachin Yadav"></property>
19. <property name="address" ref="address1"></property>
20. </bean>
22. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.context.ApplicationContext;
6. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
7. **import** org.springframework.core.io.ClassPathResource;
8. **import** org.springframework.core.io.Resource;
10. **public** **class** Test {
11. **public** **static** **void** main(String[] args) {
12. Resource r=**new** ClassPathResource("applicationContext.xml");
13. BeanFactory factory=**new** XmlBeanFactory(r);
15. Employee e=(Employee)factory.getBean("obj");
16. e.displayInfo();
18. }
19. }

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

1. [Setter Injection with Non-String Collection](https://www.javatpoint.com/spring-tutorial-setter-injection-with-non-string-collection)

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

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.

1. **package** com.javatpoint;
3. **import** java.util.Iterator;
4. **import** java.util.List;
6. **public** **class** Question {
7. **private** **int** id;
8. **private** String name;
9. **private** List<Answer> answers;
11. //setters and getters
13. **public** **void** displayInfo(){
14. System.out.println(id+" "+name);
15. System.out.println("answers are:");
16. Iterator<Answer> itr=answers.iterator();
17. **while**(itr.hasNext()){
18. System.out.println(itr.next());
19. }
20. }
22. }

**Answer.java**

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

1. **package** com.javatpoint;
3. **public** **class** Answer {
4. **private** **int** id;
5. **private** String name;
6. **private** String by;
8. //setters and getters
10. **public** String toString(){
11. **return** id+" "+name+" "+by;
12. }
13. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="answer1" **class**="com.javatpoint.Answer">
10. <property name="id" value="1"></property>
11. <property name="name" value="Java is a programming language"></property>
12. <property name="by" value="Ravi Malik"></property>
13. </bean>
14. <bean id="answer2" **class**="com.javatpoint.Answer">
15. <property name="id" value="2"></property>
16. <property name="name" value="Java is a platform"></property>
17. <property name="by" value="Sachin"></property>
18. </bean>
20. <bean id="q" **class**="com.javatpoint.Question">
21. <property name="id" value="1"></property>
22. <property name="name" value="What is Java?"></property>
23. <property name="answers">
24. <list>
25. <ref bean="answer1"/>
26. <ref bean="answer2"/>
27. </list>
28. </property>
29. </bean>
31. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Question q=(Question)factory.getBean("q");
14. q.displayInfo();
16. }
17. }

# **Setter Injection with Map Example**

1. [Setter Injection with Map Example](https://www.javatpoint.com/spring-tutorial-setter-injection-with-map)

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

Like previous examples, it is the example of forum where **one question can have multiple answers**.

**Question.java**

This class contains three properties, getters & setters and displayInfo() method to display the information.

1. **package** com.javatpoint;
2. **import** java.util.Iterator;
3. **import** java.util.Map;
4. **import** java.util.Set;
5. **import** java.util.Map.Entry;
7. **public** **class** Question {
8. **private** **int** id;
9. **private** String name;
10. **private** Map<String,String> answers;
12. //getters and setters
14. **public** **void** displayInfo(){
15. System.out.println("question id:"+id);
16. System.out.println("question name:"+name);
17. System.out.println("Answers....");
18. Set<Entry<String, String>> set=answers.entrySet();
19. Iterator<Entry<String, String>> itr=set.iterator();
20. **while**(itr.hasNext()){
21. Entry<String,String> entry=itr.next();
22. System.out.println("Answer:"+entry.getKey()+" Posted By:"+entry.getValue());
23. }
24. }
25. }

**applicationContext.xml**

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

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="q" **class**="com.javatpoint.Question">
10. <property name="id" value="1"></property>
11. <property name="name" value="What is Java?"></property>
12. <property name="answers">
13. <map>
14. <entry key="Java is a programming language"  value="Sonoo Jaiswal"></entry>
15. <entry key="Java is a Platform" value="Sachin Yadav"></entry>
16. </map>
17. </property>
18. </bean>
20. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Question q=(Question)factory.getBean("q");
14. q.displayInfo();
16. }
17. }

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

1. [Setter Injection with Non-String Map](https://www.javatpoint.com/spring-tutorial-setter-injection-with-non-string-map)

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.

Like previous examples, it is the example of forum where **one question can have multiple answers**.

**Question.java**

This class contains three properties, getters & setters and displayInfo() method to display the information.

1. **package** com.javatpoint;
2. **import** java.util.Iterator;
3. **import** java.util.Map;
4. **import** java.util.Set;
5. **import** java.util.Map.Entry;
7. **public** **class** Question {
8. **private** **int** id;
9. **private** String name;
10. **private** Map<Answer,User> answers;
12. //getters and setters

15. **public** **void** displayInfo(){
16. System.out.println("question id:"+id);
17. System.out.println("question name:"+name);
18. System.out.println("Answers....");
19. Set<Entry<Answer, User>> set=answers.entrySet();
20. Iterator<Entry<Answer, User>> itr=set.iterator();
21. **while**(itr.hasNext()){
22. Entry<Answer, User> entry=itr.next();
23. Answer ans=entry.getKey();
24. User user=entry.getValue();
25. System.out.println("Answer Information:");
26. System.out.println(ans);
27. System.out.println("Posted By:");
28. System.out.println(user);
29. }
30. }
31. }

**Answer.java**

1. **package** com.javatpoint;
3. **import** java.util.Date;
5. **public** **class** Answer {
6. **private** **int** id;
7. **private** String answer;
8. **private** Date postedDate;
9. **public** Answer() {}
10. **public** Answer(**int** id, String answer, Date postedDate) {
11. **super**();
12. **this**.id = id;
13. **this**.answer = answer;
14. **this**.postedDate = postedDate;
15. }
17. **public** String toString(){
18. **return** "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;
19. }
20. }

**User.java**

1. **package** com.javatpoint;
3. **public** **class** User {
4. **private** **int** id;
5. **private** String name,email;
6. **public** User() {}
7. **public** User(**int** id, String name, String email) {
8. **super**();
9. **this**.id = id;
10. **this**.name = name;
11. **this**.email = email;
12. }
14. **public** String toString(){
15. **return** "Id:"+id+" Name:"+name+" Email Id:"+email;
16. }
17. }

**applicationContext.xml**

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

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="answer1" **class**="com.javatpoint.Answer">
10. <property name="id" value="1"></property>
11. <property name="answer" value="Java is a Programming Language"></property>
12. <property name="postedDate" value="12/12/2001"></property>
13. </bean>
14. <bean id="answer2" **class**="com.javatpoint.Answer">
15. <property name="id" value="2"></property>
16. <property name="answer" value="Java is a Platform"></property>
17. <property name="postedDate" value="12/12/2003"></property>
18. </bean>
20. <bean id="user1" **class**="com.javatpoint.User">
21. <property name="id" value="1"></property>
22. <property name="name" value="Arun Kumar"></property>
23. <property name="email" value="arun@gmail.com"></property>
24. </bean>
25. <bean id="user2" **class**="com.javatpoint.User">
26. <property name="id" value="2"></property>
27. <property name="name" value="Varun Kumar"></property>
28. <property name="email" value="Varun@gmail.com"></property>
29. </bean>
31. <bean id="q" **class**="com.javatpoint.Question">
32. <property name="id" value="1"></property>
33. <property name="name" value="What is Java?"></property>
34. <property name="answers">
35. <map>
36. <entry key-ref="answer1" value-ref="user1"></entry>
37. <entry key-ref="answer2" value-ref="user2"></entry>
38. </map>
39. </property>
40. </bean>
42. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method to display the information.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Question q=(Question)factory.getBean("q");
14. q.displayInfo();
16. }
17. }

# **Difference between constructor and setter injection**

1. [Difference between constructor and setter injection](https://www.javatpoint.com/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:

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 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.

1. <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>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the display 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=context.getBean("a",A.**class**);

    a.display();

}

}

Output:

b is created

a is created

hello a

hello b

## **1) byName autowiring mode**

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

It internally uses setter injection.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <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.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <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.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <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.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="b2" **class**="org.sssit.B"></bean>
3. <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.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <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.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <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**.

1. **public** **class** A {
2. **public** **static** A getA(){//factory method
3. **return** **new** A();
4. }
5. }

## **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.

1. <bean id="a" **class**="com.javatpoint.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.

1. <bean id="b" **class**="com.javatpoint.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.

1. <bean id="a" **class**="com.javatpoint.A"></bean>
2. <bean id="b" **class**="com.javatpoint.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.javatpoint.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.javatpoint;

**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.javatpoint.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**

1. **package** com.javatpoint;
2. **public** **interface** Printable {
3. **void** print();
4. }

**A.java**

1. **package** com.javatpoint;
2. **public** **class** A **implements** Printable{
3. @Override
4. **public** **void** print() {
5. System.out.println("hello a");
6. }
8. }

**B.java**

1. **package** com.javatpoint;
2. **public** **class** B **implements** Printable{
3. @Override
4. **public** **void** print() {
5. System.out.println("hello b");
6. }
7. }

**PrintableFactory.java**

1. **package** com.javatpoint;
2. **public** **class** PrintableFactory {
3. **public** **static** Printable getPrintable(){
4. //return new B();
5. **return** **new** A();//return any one instance, either A or B
6. }
7. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="p" **class**="com.javatpoint.PrintableFactory" factory-method="getPrintable"></bean>
11. </beans>

**Test.java**

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

1. **package** org.sssit;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test {
5. **public** **static** **void** main(String[] args) {
6. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
7. Printable p=(Printable)context.getBean("p");
8. p.print();
9. }
10. }

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**

1. **package** com.javatpoint;
2. **public** **class** PrintableFactory {
3. //non-static factory method
4. **public** Printable getPrintable(){
5. **return** **new** A();//return any one instance, either A or B
6. }
7. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="pfactory" **class**="com.javatpoint.PrintableFactory"></bean>
10. <bean id="p" **class**="com.javatpoint.PrintableFactory" factory-method="getPrintable"
11. factory-bean="pfactory"></bean>
13. </beans>

Output:

hello a

|  |
| --- |
| **Spring AOP Example**  1. [Before Advice Example](https://www.javatpoint.com/spring-aop-example) 2. [After Returning Advice Example](https://www.javatpoint.com/spring-aop-example) 3. [Around Advice Example](https://www.javatpoint.com/spring-aop-example) 4. [After Throwing Advice Example](https://www.javatpoint.com/spring-aop-example)   There are given examples of **Spring1.2 old style AOP** (dtd based) implementation.  Though it is supported in spring 3, but it is recommended to use spring aop with aspectJ that we are going to learn in next page.  There are 4 types of advices supported in spring1.2 old style aop implementation.   1. **Before Advice** it is executed before the actual method call. 2. **After Advice** it is executed after the actual method call. If method returns a value, it is executed after returning value. 3. **Around Advice** it is executed before and after the actual method call. 4. **Throws Advice** it is executed if actual method throws exception.  To understand the basic concepts of Spring AOP, visit the previous page.Understanding the hierarchy of advice interfaces Let's understand the advice hierarchy by the diagram given below:  spring aop advice interfaces  All are interfaces in aop.  **MethodBeforeAdvice** interface extends the **BeforeAdvice** interface.  **AfterReturningAdvice** interface extends the **AfterAdvice** interface.  **ThrowsAdvice** interface extends the **AfterAdvice** interface.  **MethodInterceptor** interface extends the **Interceptor** interface. It is used in around advice. 1) MethodBeforeAdvice Example Create a class that contains actual business logic.  *File: A.java*   1. **package** com.javatpoint; 2. **public** **class** A { 3. **public** **void** m(){System.out.println("actual business logic");} 4. }   Now, create the advisor class that implements MethodBeforeAdvice interface.  *File: BeforeAdvisor.java*   1. **package** com.javatpoint; 2. **import** java.lang.reflect.Method; 3. **import** org.springframework.aop.MethodBeforeAdvice; 4. **public** **class** BeforeAdvisor **implements** MethodBeforeAdvice{ 5. @Override 6. **public** **void** before(Method method, Object[] args, Object target)**throws** Throwable { 7. System.out.println("additional concern before actual logic"); 8. } 9. }   In xml file, create 3 beans, one for A class, second for Advisor class and third for **ProxyFactoryBean** class.  *File: applicationContext.xml*   1. <?xml version="1.0" encoding="UTF-8"?> 2. <beans 3. xmlns="http://www.springframework.org/schema/beans" 4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 5. xmlns:p="http://www.springframework.org/schema/p" 6. xsi:schemaLocation="http://www.springframework.org/schema/beans 7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"> 9. <bean id="obj" **class**="com.javatpoint.A"></bean> 10. <bean id="ba" **class**="com.javatpoint.BeforeAdvisor"></bean> 12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean"> 13. <property name="target" ref="obj"></property> 14. <property name="interceptorNames"> 15. <list> 16. <value>ba</value> 17. </list> 18. </property> 19. </bean> 21. </beans>   **Understanding ProxyFactoryBean class:**  The **ProxyFactoryBean** class is provided by Spring Famework. It contains 2 properties target and interceptorNames. The instance of A class will be considered as target object and the instance of advisor class as interceptor. You need to pass the advisor object as the list object as in the xml file given above.  The ProxyFactoryBean class is written something like this:   1. **public** **class** ProxyFactoryBean{ 2. **private** Object target; 3. **private** List interceptorNames; 4. //getters and setters 5. }   Now, let's call the actual method.  *File: Test.java*   1. **package** com.javatpoint; 2. **import** org.springframework.beans.factory.BeanFactory; 3. **import** org.springframework.beans.factory.xml.XmlBeanFactory; 4. **import** org.springframework.core.io.ClassPathResource; 5. **import** org.springframework.core.io.Resource; 6. **public** **class** Test { 7. **public** **static** **void** main(String[] args) { 8. Resource r=**new** ClassPathResource("applicationContext.xml"); 9. BeanFactory factory=**new** XmlBeanFactory(r); 11. A a=factory.getBean("proxy",A.**class**); 12. a.m(); 13. } 14. }  Output  1. additional concern before actual logic 2. actual business logic  Printing additional information in MethodBeforeAdvice We can print additional information like method name, method argument, target object, target object class name, proxy class etc.  You need to change only two classes BeforeAdvisor.java and Test.java.  *File: BeforeAdvisor.java*   1. **package** com.javatpoint; 2. **import** java.lang.reflect.Method; 3. **import** org.springframework.aop.MethodBeforeAdvice; 5. **public** **class** BeforeAdvisor **implements** MethodBeforeAdvice{ 6. @Override 7. **public** **void** before(Method method, Object[] args, Object target)**throws** Throwable { 8. System.out.println("additional concern before actual logic"); 9. System.out.println("method info:"+method.getName()+" "+method.getModifiers()); 10. System.out.println("argument info:"); 11. **for**(Object arg:args) 12. System.out.println(arg); 13. System.out.println("target Object:"+target); 14. System.out.println("target object class name: "+target.getClass().getName()); 15. } 16. }   *File: Test.java*   1. **package** com.javatpoint; 2. **import** org.springframework.beans.factory.BeanFactory; 3. **import** org.springframework.beans.factory.xml.XmlBeanFactory; 4. **import** org.springframework.core.io.ClassPathResource; 5. **import** org.springframework.core.io.Resource; 6. **public** **class** Test { 7. **public** **static** **void** main(String[] args) { 8. Resource r=**new** ClassPathResource("applicationContext.xml"); 9. BeanFactory factory=**new** XmlBeanFactory(r); 11. A a=factory.getBean("proxy",A.**class**); 12. System.out.println("proxy class name: "+a.getClass().getName()); 13. a.m(); 14. } 15. }  Output  1. proxy **class** name: com.javatpoint.A$$EnhancerByCGLIB$$409872b1 2. additional concern before actual logic 3. method info:m 1 4. argument info: 5. target Object:com.javatpoint.A@11dba45 6. target object **class** name: com.javatpoint.A 7. actual business logic  2) AfterReturningAdvice Example Create a class that contains actual business logic.  *File: A.java*  Same as in the previous example.  Now, create the advisor class that implements AfterReturningAdvice interface.  *File: AfterAdvisor.java*   1. **package** com.javatpoint; 2. **import** java.lang.reflect.Method; 3. **import** org.springframework.aop.AfterReturningAdvice; 4. **public** **class** AfterAdvisor **implements** AfterReturningAdvice{ 5. @Override 6. **public** **void** afterReturning(Object returnValue, Method method, 7. Object[] args, Object target) **throws** Throwable { 9. System.out.println("additional concern after returning advice"); 10. } 12. }   Create the xml file as in the previous example, you need to change only the advisor class here.  *File: applicationContext.xml*   1. <?xml version="1.0" encoding="UTF-8"?> 2. <beans 3. xmlns="http://www.springframework.org/schema/beans" 4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 5. xmlns:p="http://www.springframework.org/schema/p" 6. xsi:schemaLocation="http://www.springframework.org/schema/beans 7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"> 9. <bean id="obj" **class**="com.javatpoint.A"></bean> 10. <bean id="ba" **class**="com.javatpoint.AfterAdvisor"></bean> 12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean"> 13. <property name="target" ref="obj"></property> 14. <property name="interceptorNames"> 15. <list> 16. <value>ba</value> 17. </list> 18. </property> 19. </bean> 21. </beans>   *File: Test.java*  Same as in the previous example. Output  1. actual business logic 2. additional concern after returning advice  3) MethodInterceptor (AroundAdvice) Example Create a class that contains actual business logic.  *File: A.java*  Same as in the previous example.  Now, create the advisor class that implements MethodInterceptor interface.  *File: AroundAdvisor.java*   1. **package** com.javatpoint; 2. **import** org.aopalliance.intercept.MethodInterceptor; 3. **import** org.aopalliance.intercept.MethodInvocation; 4. **public** **class** AroundAdvisor **implements** MethodInterceptor{ 6. @Override 7. **public** Object invoke(MethodInvocation mi) **throws** Throwable { 8. Object obj; 9. System.out.println("additional concern before actual logic"); 10. obj=mi.proceed(); 11. System.out.println("additional concern after actual logic"); 12. **return** obj; 13. } 15. }   Create the xml file as in the previous example, you need to change only the advisor class here.  *File: applicationContext.xml*   1. <?xml version="1.0" encoding="UTF-8"?> 2. <beans 3. xmlns="http://www.springframework.org/schema/beans" 4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 5. xmlns:p="http://www.springframework.org/schema/p" 6. xsi:schemaLocation="http://www.springframework.org/schema/beans 7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"> 9. <bean id="obj" **class**="com.javatpoint.A"></bean> 10. <bean id="ba" **class**="com.javatpoint.AroundAdvisor"></bean> 12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean"> 13. <property name="target" ref="obj"></property> 14. <property name="interceptorNames"> 15. <list> 16. <value>ba</value> 17. </list> 18. </property> 19. </bean> 21. </beans>   *File: Test.java*  Same as in the previous example. Output  1. additional concern before actual logic 2. actual business logic 3. additional concern after actual logic  4) ThrowsAdvice Example Create a class that contains actual business logic.  *File: Validator.java*   1. **package** com.javatpoint; 2. **public** **class** Validator { 3. **public** **void** validate(**int** age)**throws** Exception{ 4. **if**(age<18){ 5. **throw** **new** ArithmeticException("Not Valid Age"); 6. } 7. **else**{ 8. System.out.println("vote confirmed"); 9. } 10. } 11. }   Now, create the advisor class that implements ThrowsAdvice interface.  *File: ThrowsAdvisor.java*   1. **package** com.javatpoint; 2. **import** org.springframework.aop.ThrowsAdvice; 3. **public** **class** ThrowsAdvisor **implements** ThrowsAdvice{ 4. **public** **void** afterThrowing(Exception ex){ 5. System.out.println("additional concern if exception occurs"); 6. } 7. }   Create the xml file as in the previous example, you need to change only the Validator class and advisor class.  *File: applicationContext.xml*   1. <?xml version="1.0" encoding="UTF-8"?> 2. <beans 3. xmlns="http://www.springframework.org/schema/beans" 4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 5. xmlns:p="http://www.springframework.org/schema/p" 6. xsi:schemaLocation="http://www.springframework.org/schema/beans 7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"> 9. <bean id="obj" **class**="com.javatpoint.Validator"></bean> 10. <bean id="ba" **class**="com.javatpoint.ThrowsAdvisor"></bean> 12. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean"> 13. <property name="target" ref="obj"></property> 14. <property name="interceptorNames"> 15. <list> 16. <value>ba</value> 17. </list> 18. </property> 19. </bean> 21. </beans>   *File: Test.java*   1. **package** com.javatpoint; 3. **import** org.springframework.beans.factory.BeanFactory; 4. **import** org.springframework.beans.factory.xml.XmlBeanFactory; 5. **import** org.springframework.core.io.ClassPathResource; 6. **import** org.springframework.core.io.Resource; 8. **public** **class** Test { 9. **public** **static** **void** main(String[] args) { 10. Resource r=**new** ClassPathResource("applicationContext.xml"); 11. BeanFactory factory=**new** XmlBeanFactory(r); 13. Validator v=factory.getBean("proxy",Validator.**class**); 14. **try**{ 15. v.validate(12); 16. }**catch**(Exception e){e.printStackTrace();} 17. } 18. }  Output  1. java.lang.ArithmeticException: Not Valid Age 3. additional concern **if** exception occurs 5. at com.javatpoint.Validator.validate(Validator.java:7) 6. at com.javatpoint.Validator$$FastClassByCGLIB$$562915cf.invoke(<generated>) 7. at net.sf.cglib.proxy.MethodProxy.invoke(MethodProxy.java:191) 8. at org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation.invoke 9. Joinpoint(Cglib2AopProxy.java:692) 10. at org.springframework.aop.framework.ReflectiveMethodInvocation. 11. proceed(ReflectiveMethodInvocation.java:150) 12. at org.springframework.aop.framework.adapter.ThrowsAdviceInterceptor. 13. invoke(ThrowsAdviceInterceptor.java:124) 14. at org.springframework.aop.framework.ReflectiveMethodInvocation. 15. proceed(ReflectiveMethodInvocation.java:172) 16. at org.springframework.aop.framework.Cglib2AopProxy$DynamicAdvisedInterceptor. 17. intercept(Cglib2AopProxy.java:625) 18. at com.javatpoint.Validator$$EnhancerByCGLIB$$4230ed28.validate(<generated>) 19. at com.javatpoint.Test.main(Test.java:15) |

# **Spring AOP AspectJ Annotation Example**

1. [@Before Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
2. [@After Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
3. [@AfterReturning Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
4. [@Around Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
5. [@AfterThrowing Example](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)

The **Spring Framework** recommends you to use **Spring AspectJ AOP implementation** over the Spring 1.2 old style dtd based AOP implementation because it provides you more control and it is easy to use.

There are two ways to use Spring AOP AspectJ implementation:

1. By annotation: We are going to learn it here.
2. By xml configuration (schema based): We will learn it in next page.

#### To understand the aop concepts, its advantage etc. visit here [AOP Concepts Tutorial](http://www.javatpoint.com/spring-aop-tutorial)

[download all examples (developed using MyEclipse IDE)](https://static.javatpoint.com/src/sp/aopaspectjannotation.zip)

Spring AspectJ AOP implementation provides many annotations:

1. **@Aspect** declares the class as aspect.
2. **@Pointcut** declares the pointcut expression.

The annotations used to create advices are given below:

1. **@Before** declares the before advice. It is applied before calling the actual method.
2. **@After** declares the after advice. It is applied after calling the actual method and before returning result.
3. **@AfterReturning** declares the after returning advice. It is applied after calling the actual method and before returning result. But you can get the result value in the advice.
4. **@Around** declares the around advice. It is applied before and after calling the actual method.
5. **@AfterThrowing** declares the throws advice. It is applied if actual method throws exception.

## **Understanding Pointcut**

Pointcut is an expression language of Spring AOP.

The **@Pointcut** annotation is used to define the pointcut. We can refer the pointcut expression by name also. Let's see the simple example of pointcut expression.

1. @Pointcut("execution(\* Operation.\*(..))")
2. **private** **void** doSomething() {}

The name of the pointcut expression is doSomething(). It will be applied on all the methods of Operation class regardless of return type.

#### Understanding Pointcut Expressions

Let's try the understand the pointcut expressions by the examples given below:

1. @Pointcut("execution(public \* \*(..))")

It will be applied on all the public methods.

1. @Pointcut("execution(public Operation.\*(..))")

It will be applied on all the public methods of Operation class.

1. @Pointcut("execution(\* Operation.\*(..))")

It will be applied on all the methods of Operation class.

1. @Pointcut("execution(public Employee.set\*(..))")

It will be applied on all the public setter methods of Employee class.

1. @Pointcut("execution(int Operation.\*(..))")

It will be applied on all the methods of Operation class that returns int value.

#### 1) @Before Example

The AspectJ Before Advice is applied before the actual business logic method. You can perform any operation here such as conversion, authentication etc.

Create a class that contains actual business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** msg(){System.out.println("msg method invoked");}
4. **public** **int** m(){System.out.println("m method invoked");**return** 2;}
5. **public** **int** k(){System.out.println("k method invoked");**return** 3;}
6. }

Now, create the aspect class that contains before advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.aspectj.lang.annotation.Before;
6. **import** org.aspectj.lang.annotation.Pointcut;
8. @Aspect
9. **public** **class** TrackOperation{
10. @Pointcut("execution(\* Operation.\*(..))")
11. **public** **void** k(){}//pointcut name
13. @Before("k()")//applying pointcut on before advice
14. **public** **void** myadvice(JoinPoint jp)//it is advice (before advice)
15. {
16. System.out.println("additional concern");
17. //System.out.println("Method Signature: "  + jp.getSignature());
18. }
19. }

Now create the applicationContext.xml file that defines beans.

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop.xsd">

11. <bean id="opBean" **class**="com.javatpoint.Operation">   </bean>
12. <bean id="trackMyBean" **class**="com.javatpoint.TrackOperation"></bean>
14. <bean **class**="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"></bean>
16. </beans>

Now, let's call the actual method.

*File: Test.java*

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation e = (Operation) context.getBean("opBean");
9. System.out.println("calling msg...");
10. e.msg();
11. System.out.println("calling m...");
12. e.m();
13. System.out.println("calling k...");
14. e.k();
15. }
16. }

#### Output

1. calling msg...
2. additional concern
3. msg() method invoked
4. calling m...
5. additional concern
6. m() method invoked
7. calling k...
8. additional concern
9. k() method invoked

As you can see, additional concern is printed before msg(), m() and k() method is invoked.

Now if you change the pointcut expression as given below:

1. @Pointcut("execution(\* Operation.m\*(..))")

Now additional concern will be applied for the methods starting with m in Operation class. Output will be as this:

1. calling msg...
2. additional concern
3. msg() method invoked
4. calling m...
5. additional concern
6. m() method invoked
7. calling k...
8. k() method invoked

Now you can see additional concern is not printed before k() method invoked.

#### 2) @After Example

The AspectJ after advice is applied after calling the actual business logic methods. It can be used to maintain log, security, notification etc.

Here, We are assuming that **Operation.java**, **applicationContext.xml** and **Test.java** files are same as given in @Before example.

Create the aspect class that contains after advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.aspectj.lang.annotation.After;
6. **import** org.aspectj.lang.annotation.Pointcut;
8. @Aspect
9. **public** **class** TrackOperation{
10. @Pointcut("execution(\* Operation.\*(..))")
11. **public** **void** k(){}//pointcut name
13. @After("k()")//applying pointcut on after advice
14. **public** **void** myadvice(JoinPoint jp)//it is advice (after advice)
15. {
16. System.out.println("additional concern");
17. //System.out.println("Method Signature: "  + jp.getSignature());
18. }
19. }

#### Output

1. calling msg...
2. msg() method invoked
3. additional concern
4. calling m...
5. m() method invoked
6. additional concern
7. calling k...
8. k() method invoked
9. additional concern

You can see that additional concern is printed after calling msg(), m() and k() methods.

#### 3) @AfterReturning Example

By using after returning advice, we can get the result in the advice.

Create the class that contains business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **int** m(){System.out.println("m() method invoked");**return** 2;}
4. **public** **int** k(){System.out.println("k() method invoked");**return** 3;}
5. }

Create the aspect class that contains after returning advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.annotation.AfterReturning;
5. **import** org.aspectj.lang.annotation.Aspect;
7. @Aspect
8. **public** **class** TrackOperation{
9. @AfterReturning(
10. pointcut = "execution(\* Operation.\*(..))",
11. returning= "result")
13. **public** **void** myadvice(JoinPoint jp,Object result)//it is advice (after returning advice)
14. {
15. System.out.println("additional concern");
16. System.out.println("Method Signature: "  + jp.getSignature());
17. System.out.println("Result in advice: "+result);
18. System.out.println("end of after returning advice...");
19. }
20. }

*File: applicationContext.xml*

It is same as given in @Before advice example

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation e = (Operation) context.getBean("opBean");
9. System.out.println("calling m...");
10. System.out.println(e.m());
11. System.out.println("calling k...");
12. System.out.println(e.k());
13. }
14. }

#### Output

1. calling m...
2. m() method invoked
3. additional concern
4. Method Signature: **int** com.javatpoint.Operation.m()
5. Result in advice: 2
6. end of after returning advice...
7. 2
8. calling k...
9. k() method invoked
10. additional concern
11. Method Signature: **int** com.javatpoint.Operation.k()
12. Result in advice: 3
13. end of after returning advice...
14. 3

You can see that return value is printed two times, one is printed by TrackOperation class and second by Test class.

#### 4) @Around Example

The AspectJ around advice is applied before and after calling the actual business logic methods.

Here, we are assuming that **applicationContext.xml** file is same as given in @Before example.

Create a class that contains actual business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** msg(){System.out.println("msg() is invoked");}
4. **public** **void** display(){System.out.println("display() is invoked");}
5. }

Create the aspect class that contains around advice.

You need to pass the **PreceedingJoinPoint** reference in the advice method, so that we can proceed the request by calling the proceed() method.

*File: TrackOperation.java*

1. **package** com.javatpoint;
2. **import** org.aspectj.lang.ProceedingJoinPoint;
3. **import** org.aspectj.lang.annotation.Around;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.aspectj.lang.annotation.Pointcut;
7. @Aspect
8. **public** **class** TrackOperation
9. {
10. @Pointcut("execution(\* Operation.\*(..))")
11. **public** **void** abcPointcut(){}
13. @Around("abcPointcut()")
14. **public** Object myadvice(ProceedingJoinPoint pjp) **throws** Throwable
15. {
16. System.out.println("Additional Concern Before calling actual method");
17. Object obj=pjp.proceed();
18. System.out.println("Additional Concern After calling actual method");
19. **return** obj;
20. }
21. }

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test{
5. **public** **static** **void** main(String[] args){
6. ApplicationContext context = **new** classPathXmlApplicationContext("applicationContext.xml");
8. Operation op = (Operation) context.getBean("opBean");
9. op.msg();
10. op.display();
11. }
12. }

#### Output

1. Additional Concern Before calling actual method
2. msg() is invoked
3. Additional Concern After calling actual method
4. Additional Concern Before calling actual method
5. display() is invoked
6. Additional Concern After calling actual method

You can see that additional concern is printed before and after calling msg() and display methods.

#### 5) @AfterThrowing Example

By using after throwing advice, we can print the exception in the TrackOperation class. Let's see the example of AspectJ AfterThrowing advice.

Create the class that contains business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** validate(**int** age)**throws** Exception{
4. **if**(age<18){
5. **throw** **new** ArithmeticException("Not valid age");
6. }
7. **else**{
8. System.out.println("Thanks for vote");
9. }
10. }
12. }

Create the aspect class that contains after throwing advice.

Here, we need to pass the Throwable reference also, so that we can intercept the exception here.

*File: TrackOperation.java*

1. **package** com.javatpoint;
2. **import** org.aspectj.lang.JoinPoint;
3. **import** org.aspectj.lang.annotation.AfterThrowing;
4. **import** org.aspectj.lang.annotation.Aspect;
5. @Aspect
6. **public** **class** TrackOperation{
7. @AfterThrowing(
8. pointcut = "execution(\* Operation.\*(..))",
9. throwing= "error")
11. **public** **void** myadvice(JoinPoint jp,Throwable error)//it is advice
12. {
13. System.out.println("additional concern");
14. System.out.println("Method Signature: "  + jp.getSignature());
15. System.out.println("Exception is: "+error);
16. System.out.println("end of after throwing advice...");
17. }
18. }

*File: applicationContext.xml*

It is same as given in @Before advice example

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation op = (Operation) context.getBean("opBean");
9. System.out.println("calling validate...");
10. **try**{
11. op.validate(19);

        }**catch**(Exception e){System.out.println(e);}

1. System.out.println("calling validate again...");
3. **try**{
4. op.validate(11);
5. }**catch**(Exception e){System.out.println(e);}
6. }
7. }

#### Output

1. calling validate...
2. Thanks **for** vote
3. calling validate again...
4. additional concern
5. Method Signature: **void** com.javatpoint.Operation.validate(**int**)
6. Exception is: java.lang.ArithmeticException: Not valid age
7. end of after throwing advice...

# **Spring AOP AspectJ Xml Configuration Example**

1. [aop:before example](https://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)
2. [aop:after example](https://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)
3. [aop:after-returning example](https://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)
4. [aop:around example](https://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)
5. [aop:after-throwing example](https://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)

Spring enables you to define the aspects, advices and pointcuts in xml file.

In the previous page, we have seen the aop examples using annotations. Now we are going to see same examples by the xml configuration file.

Let's see the xml elements that are used to define advice.

1. **aop:before** It is applied before calling the actual business logic method.
2. **aop:after** It is applied after calling the actual business logic method.
3. **aop:after-returning** it is applied after calling the actual business logic method. It can be used to intercept the return value in advice.
4. **aop:around** It is applied before and after calling the actual business logic method.
5. **aop:after-throwing** It is applied if actual business logic method throws exception.

#### To understand the aop concepts, its advantage etc. visit here [AOP Concepts Tutorial](http://www.javatpoint.com/spring-aop-tutorial)

[download all examples (developed using MyEclipse IDE)](https://static.javatpoint.com/src/sp/aopaspectjxml.zip)

#### 1) aop:before Example

The AspectJ Before Advice is applied before the actual business logic method. You can perform any operation here such as conversion, authentication etc.

Create a class that contains actual business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** msg(){System.out.println("msg method invoked");}
4. **public** **int** m(){System.out.println("m method invoked");**return** 2;}
5. **public** **int** k(){System.out.println("k method invoked");**return** 3;}
6. }

Now, create the aspect class that contains before advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
2. **import** org.aspectj.lang.JoinPoint;
3. **public** **class** TrackOperation{
4. **public** **void** myadvice(JoinPoint jp)//it is advice
5. {
6. System.out.println("additional concern");
7. //System.out.println("Method Signature: "  + jp.getSignature());
8. }
9. }

Now create the applicationContext.xml file that defines beans.

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.javatpoint.Operation">   </bean>
13. <bean id="trackAspect" **class**="com.javatpoint.TrackOperation"></bean>
15. <aop:config>
16. <aop:aspect id="myaspect" ref="trackAspect" >
17. <!-- @Before -->
18. <aop:pointcut id="pointCutBefore"   expression="execution(\* com.javatpoint.Operation.\*(..))" />
19. <aop:before method="myadvice" pointcut-ref="pointCutBefore" />
20. </aop:aspect>
21. </aop:config>
23. </beans>

Now, let's call the actual method.

*File: Test.java*

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation e = (Operation) context.getBean("opBean");
9. System.out.println("calling msg...");
10. e.msg();
11. System.out.println("calling m...");
12. e.m();
13. System.out.println("calling k...");
14. e.k();
15. }
16. }

#### Output

1. calling msg...
2. additional concern
3. msg() method invoked
4. calling m...
5. additional concern
6. m() method invoked
7. calling k...
8. additional concern
9. k() method invoked

As you can see, additional concern is printed before msg(), m() and k() method is invoked.

#### 2) aop:after example

The AspectJ after advice is applied after calling the actual business logic methods. It can be used to maintain log, security, notification etc.

Here, We are assuming that **Operation.java**, **TrackOperation.java** and **Test.java** files are same as given in aop:before example.

Now create the applicationContext.xml file that defines beans.

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.javatpoint.Operation">   </bean>
13. <bean id="trackAspect" **class**="com.javatpoint.TrackOperation"></bean>
15. <aop:config>
16. <aop:aspect id="myaspect" ref="trackAspect" >
17. <!-- @After -->
18. <aop:pointcut id="pointCutAfter"    expression="execution(\* com.javatpoint.Operation.\*(..))" />
19. <aop:after method="myadvice" pointcut-ref="pointCutAfter" />
20. </aop:aspect>
21. </aop:config>
23. </beans>

#### Output

1. calling msg...
2. msg() method invoked
3. additional concern
4. calling m...
5. m() method invoked
6. additional concern
7. calling k...
8. k() method invoked
9. additional concern

You can see that additional concern is printed after calling msg(), m() and k() methods.

#### 3) aop:after-returning example

By using after returning advice, we can get the result in the advice.

Create the class that contains business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **int** m(){System.out.println("m() method invoked");**return** 2;}
4. **public** **int** k(){System.out.println("k() method invoked");**return** 3;}
5. }

Create the aspect class that contains after returning advice.

*File: TrackOperation.java*

1. **package** com.javatpoint;
3. **import** org.aspectj.lang.JoinPoint;
5. **public** **class** TrackOperation{
6. **public** **void** myadvice(JoinPoint jp,Object result)//it is advice (after advice)
7. {
8. System.out.println("additional concern");
9. System.out.println("Method Signature: "  + jp.getSignature());
10. System.out.println("Result in advice: "+result);
11. System.out.println("end of after returning advice...");
12. }
13. }

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.javatpoint.Operation">   </bean>
14. <bean id="trackAspect" **class**="com.javatpoint.TrackOperation"></bean>
16. <aop:config>
17. <aop:aspect id="myaspect" ref="trackAspect" >
18. <!-- @AfterReturning -->
19. <aop:pointcut id="pointCutAfterReturning"   expression="execution(\* com.javatpoint.Operation.\*(..))" />
20. <aop:after-returning method="myadvice" returning="result" pointcut-ref="pointCutAfterReturning" />
21. </aop:aspect>
22. </aop:config>
24. </beans>

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation e = (Operation) context.getBean("opBean");
9. System.out.println("calling m...");
10. System.out.println(e.m());
11. System.out.println("calling k...");
12. System.out.println(e.k());
13. }
14. }

#### Output

1. calling m...
2. m() method invoked
3. additional concern
4. Method Signature: **int** com.javatpoint.Operation.m()
5. Result in advice: 2
6. end of after returning advice...
7. 2
8. calling k...
9. k() method invoked
10. additional concern
11. Method Signature: **int** com.javatpoint.Operation.k()
12. Result in advice: 3
13. end of after returning advice...
14. 3

You can see that return value is printed two times, one is printed by TrackOperation class and second by Test class.

#### 4) aop:around example

The AspectJ around advice is applied before and after calling the actual business logic methods.

Create a class that contains actual business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** msg(){System.out.println("msg() is invoked");}
4. **public** **void** display(){System.out.println("display() is invoked");}
5. }

Create the aspect class that contains around advice.

You need to pass the **PreceedingJoinPoint** reference in the advice method, so that we can proceed the request by calling the proceed() method.

*File: TrackOperation.java*

1. **package** com.javatpoint;
2. **import** org.aspectj.lang.ProceedingJoinPoint;
3. **public** **class** TrackOperation
4. {
5. **public** Object myadvice(ProceedingJoinPoint pjp) **throws** Throwable
6. {
7. System.out.println("Additional Concern Before calling actual method");
8. Object obj=pjp.proceed();
9. System.out.println("Additional Concern After calling actual method");
10. **return** obj;
11. }
12. }

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
10. <aop:aspectj-autoproxy />
12. <bean id="opBean" **class**="com.javatpoint.Operation">   </bean>
14. <bean id="trackAspect" **class**="com.javatpoint.TrackOperation"></bean>
16. <aop:config>
17. <aop:aspect id="myaspect" ref="trackAspect" >
18. <!-- @Around -->
19. <aop:pointcut id="pointCutAround"   expression="execution(\* com.javatpoint.Operation.\*(..))" />
20. <aop:around method="myadvice" pointcut-ref="pointCutAround" />
21. </aop:aspect>
22. </aop:config>
24. </beans>

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test{
5. **public** **static** **void** main(String[] args){
6. ApplicationContext context = **new** classPathXmlApplicationContext("applicationContext.xml");
8. Operation op = (Operation) context.getBean("opBean");
9. op.msg();
10. op.display();
11. }
12. }

#### Output

1. Additional Concern Before calling actual method
2. msg() is invoked
3. Additional Concern After calling actual method
4. Additional Concern Before calling actual method
5. display() is invoked
6. Additional Concern After calling actual method

You can see that additional concern is printed before and after calling msg() and display methods.

#### 5) aop:after-throwing example

By using after throwing advice, we can print the exception in the TrackOperation class. Let's see the example of AspectJ AfterThrowing advice.

Create the class that contains business logic.

*File: Operation.java*

1. **package** com.javatpoint;
2. **public**  **class** Operation{
3. **public** **void** validate(**int** age)**throws** Exception{
4. **if**(age<18){
5. **throw** **new** ArithmeticException("Not valid age");
6. }
7. **else**{
8. System.out.println("Thanks for vote");
9. }
10. }
12. }

Create the aspect class that contains after throwing advice.

Here, we need to pass the Throwable reference also, so that we can intercept the exception here.

*File: TrackOperation.java*

1. **package** com.javatpoint;
2. **import** org.aspectj.lang.JoinPoint;
3. **public** **class** TrackOperation{
5. **public** **void** myadvice(JoinPoint jp,Throwable error)//it is advice
6. {
7. System.out.println("additional concern");
8. System.out.println("Method Signature: "  + jp.getSignature());
9. System.out.println("Exception is: "+error);
10. System.out.println("end of after throwing advice...");
11. }
12. }

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
9. <aop:aspectj-autoproxy />
10. <bean id="opBean" **class**="com.javatpoint.Operation">   </bean>
11. <bean id="trackAspect" **class**="com.javatpoint.TrackOperation"></bean>
13. <aop:config>
14. <aop:aspect id="myaspect" ref="trackAspect" >
15. <!-- @AfterThrowing -->
16. <aop:pointcut id="pointCutAfterThrowing"    expression="execution(\* com.javatpoint.Operation.\*(..))" />
17. <aop:after-throwing method="myadvice" throwing="error" pointcut-ref="pointCutAfterThrowing" />
18. </aop:aspect>
19. </aop:config>
21. </beans>

*File: Test.java*

Now create the Test class that calls the actual methods.

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. Operation op = (Operation) context.getBean("opBean");
9. System.out.println("calling validate...");
10. **try**{
11. op.validate(19);
12. }**catch**(Exception e){System.out.println(e);}
13. System.out.println("calling validate again...");
15. **try**{
16. op.validate(11);
17. }**catch**(Exception e){System.out.println(e);}
18. }
19. }

#### Output

1. calling validate...
2. Thanks **for** vote
3. calling validate again...
4. additional concern
5. Method Signature: **void** com.javatpoint.Operation.validate(**int**)
6. Exception is: java.lang.ArithmeticException: Not valid age
7. end of after throwing advice...
8. java.lang.ArithmeticException: Not valid age

# **Spring JdbcTemplate Tutorial**

1. [Spring JDBC Template](https://www.javatpoint.com/spring-JdbcTemplate-tutorial)
2. [Understanding the need for Spring JDBC Template](https://www.javatpoint.com/spring-JdbcTemplate-tutorial)
3. [Advantage of Spring JDBC Template](https://www.javatpoint.com/spring-JdbcTemplate-tutorial)
4. [JDBC Template classes](https://www.javatpoint.com/spring-JdbcTemplate-tutorial)
5. [Example of JdbcTemplate class](https://www.javatpoint.com/spring-JdbcTemplate-tutorial)

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**

The problems of JDBC API are as follows:

* 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 excepion 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.

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**EmployeeDao.java**

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

1. **package** com.javatpoint;
2. **import** org.springframework.jdbc.core.JdbcTemplate;
4. **public** **class** EmployeeDao {
5. **private** JdbcTemplate jdbcTemplate;
7. **public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {
8. **this**.jdbcTemplate = jdbcTemplate;
9. }
11. **public** **int** saveEmployee(Employee e){
12. String query="insert into employee values(
13. '"+e.getId()+"','"+e.getName()+"','"+e.getSalary()+"')";
14. **return** jdbcTemplate.update(query);
15. }
16. **public** **int** updateEmployee(Employee e){
17. String query="update employee set
18. name='"+e.getName()+"',salary='"+e.getSalary()+"' where id='"+e.getId()+"' ";
19. **return** jdbcTemplate.update(query);
20. }
21. **public** **int** deleteEmployee(Employee e){
22. String query="delete from employee where id='"+e.getId()+"' ";
23. **return** jdbcTemplate.update(query);
24. }
26. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>
24. </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.

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test {
7. **public** **static** **void** main(String[] args) {
8. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
10. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
11. **int** status=dao.saveEmployee(**new** Employee(102,"Amit",35000));
12. System.out.println(status);
14. /\*int status=dao.updateEmployee(new Employee(102,"Sonoo",15000));
15. System.out.println(status);
16. \*/
18. /\*Employee e=new Employee();
19. e.setId(102);
20. int status=dao.deleteEmployee(e);
21. System.out.println(status);\*/
23. }
25. }

# **Example of PreparedStatement in Spring JdbcTemplate**

1. [PreparedStatement in Spring JDBC Template](https://www.javatpoint.com/example-of-PreparedStatement-in-Spring-JdbcTemplate)
2. [PreparedStatementCallback interface](https://www.javatpoint.com/example-of-PreparedStatement-in-Spring-JdbcTemplate)
3. [Example of using PreparedStatement in Spring](https://www.javatpoint.com/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

1. **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:

1. **public** T doInPreparedStatement(PreparedStatement ps)**throws** SQLException, DataAccessException

### Example of using PreparedStatement in Spring

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

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**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.

1. **package** com.javatpoint;
2. **import** java.sql.PreparedStatement;
3. **import** java.sql.SQLException;
5. **import** org.springframework.dao.DataAccessException;
6. **import** org.springframework.jdbc.core.JdbcTemplate;
7. **import** org.springframework.jdbc.core.PreparedStatementCallback;
9. **public** **class** EmployeeDao {
10. **private** JdbcTemplate jdbcTemplate;
12. **public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {
13. **this**.jdbcTemplate = jdbcTemplate;
14. }
16. **public** Boolean saveEmployeeByPreparedStatement(**final** Employee e){
17. String query="insert into employee values(?,?,?)";
18. **return** jdbcTemplate.execute(query,**new** PreparedStatementCallback<Boolean>(){
19. @Override
20. **public** Boolean doInPreparedStatement(PreparedStatement ps)
21. **throws** SQLException, DataAccessException {
23. ps.setInt(1,e.getId());
24. ps.setString(2,e.getName());
25. ps.setFloat(3,e.getSalary());
27. **return** ps.execute();
29. }
30. });
31. }

34. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>

25. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test {
7. **public** **static** **void** main(String[] args) {
8. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
10. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
11. dao.saveEmployeeByPreparedStatement(**new** Employee(108,"Amit",35000));
12. }
13. }

# **ResultSetExtractor Example | Fetching Records by Spring JdbcTemplate**

1. [ResultSetExtractor Example](https://www.javatpoint.com/ResultSetExtractor-example)
2. [ResultSetExtractor Interface](https://www.javatpoint.com/ResultSetExtractor-example)
3. [Method of ResultSetExtractor Interface](https://www.javatpoint.com/ResultSetExtractor-example)
4. [Example of ResultSetExtractor](https://www.javatpoint.com/ResultSetExtractor-example)

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

1. **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:

1. **public** T extractData(ResultSet rs)**throws** SQLException,DataAccessException

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

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

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
10. **public** String toString(){
11. **return** id+" "+name+" "+salary;
12. }
13. }

**EmployeeDao.java**

It contains on property jdbcTemplate and one method getAllEmployees.

1. **package** com.javatpoint;
2. **import** java.sql.ResultSet;
3. **import** java.sql.SQLException;
4. **import** java.util.ArrayList;
5. **import** java.util.List;
6. **import** org.springframework.dao.DataAccessException;
7. **import** org.springframework.jdbc.core.JdbcTemplate;
8. **import** org.springframework.jdbc.core.ResultSetExtractor;
10. **public** **class** EmployeeDao {
11. **private** JdbcTemplate template;
13. **public** **void** setTemplate(JdbcTemplate template) {
14. **this**.template = template;
15. }
17. **public** List<Employee> getAllEmployees(){
18. **return** template.query("select \* from employee",**new** ResultSetExtractor<List<Employee>>(){
19. @Override
20. **public** List<Employee> extractData(ResultSet rs) **throws** SQLException,
21. DataAccessException {
23. List<Employee> list=**new** ArrayList<Employee>();
24. **while**(rs.next()){
25. Employee e=**new** Employee();
26. e.setId(rs.getInt(1));
27. e.setName(rs.getString(2));
28. e.setSalary(rs.getInt(3));
29. list.add(e);
30. }
31. **return** list;
32. }
33. });
34. }
35. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>
24. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** java.util.List;
5. **import** org.springframework.context.ApplicationContext;
6. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
7. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
11. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
12. List<Employee> list=dao.getAllEmployees();
14. **for**(Employee e:list)
15. System.out.println(e);
17. }
19. }

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

1. [RowMapper](https://www.javatpoint.com/RowMapper-example)
2. [RowMapper Interface](https://www.javatpoint.com/RowMapper-example)
3. [Method of RowMapper Interface](https://www.javatpoint.com/RowMapper-example)
4. [Example of RowMapper Interface](https://www.javatpoint.com/RowMapper-example)

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

1. **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:

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

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

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

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. **public** String toString(){
10. **return** id+" "+name+" "+salary;
11. }
12. }

**EmployeeDao.java**

It contains on property jdbcTemplate and one method getAllEmployeesRowMapper.

1. **package** com.javatpoint;
2. **import** java.sql.ResultSet;
3. **import** java.sql.SQLException;
4. **import** java.util.ArrayList;
5. **import** java.util.List;
6. **import** org.springframework.dao.DataAccessException;
7. **import** org.springframework.jdbc.core.JdbcTemplate;
8. **import** org.springframework.jdbc.core.ResultSetExtractor;
9. **import** org.springframework.jdbc.core.RowMapper;
11. **public** **class** EmployeeDao {
12. **private** JdbcTemplate template;
14. **public** **void** setTemplate(JdbcTemplate template) {
15. **this**.template = template;
16. }
18. **public** List<Employee> getAllEmployeesRowMapper(){
19. **return** template.query("select \* from employee",**new** RowMapper<Employee>(){
20. @Override
21. **public** Employee mapRow(ResultSet rs, **int** rownumber) **throws** SQLException {
22. Employee e=**new** Employee();
23. e.setId(rs.getInt(1));
24. e.setName(rs.getString(2));
25. e.setSalary(rs.getInt(3));
26. **return** e;
27. }
28. });
29. }
30. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>
24. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** java.util.List;
5. **import** org.springframework.context.ApplicationContext;
6. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
7. **public** **class** Test {
8. **public** **static** **void** main(String[] args) {
9. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
10. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
11. List<Employee> list=dao.getAllEmployeesRowMapper();
13. **for**(Employee e:list)
14. System.out.println(e);
15. }
16. }

# **Spring NamedParameterJdbcTemplate Example**

1. [NamedParameterJdbcTemplate](https://www.javatpoint.com/spring-NamedParameterJdbcTemplate-example)
2. [Method of NamedParameterJdbcTemplate](https://www.javatpoint.com/spring-NamedParameterJdbcTemplate-example)
3. [Spring NamedParameterJdbcTemplate Example](https://www.javatpoint.com/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

1. 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:

1. 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.

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**EmployeeDao.java**

It contains on property jdbcTemplate and one method save.

1. **package** com.javatpoint;
3. **import** java.sql.PreparedStatement;
4. **import** java.sql.SQLException;
5. **import** org.springframework.dao.DataAccessException;
6. **import** org.springframework.jdbc.core.PreparedStatementCallback;
7. **import** org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
8. **import** java.util.\*;
10. **public** **class** EmpDao {
11. NamedParameterJdbcTemplate template;
13. **public** EmpDao(NamedParameterJdbcTemplate template) {
14. **this**.template = template;
15. }
16. **public**  **void** save (Emp e){
17. String query="insert into employee values (:id,:name,:salary)";
19. Map<String,Object> map=**new** HashMap<String,Object>();
20. map.put("id",e.getId());
21. map.put("name",e.getName());
22. map.put("salary",e.getSalary());
24. template.execute(query,map,**new** PreparedStatementCallback() {
25. @Override
26. **public** Object doInPreparedStatement(PreparedStatement ps)
27. **throws** SQLException, DataAccessException {
28. **return** ps.executeUpdate();
29. }
30. });
31. }
32. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jtemplate"
17. **class**="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
18. <constructor-arg ref="ds"></constructor-arg>
19. </bean>
21. <bean id="edao" **class**="com.javatpoint.EmpDao">
22. <constructor-arg>
23. <ref bean="jtemplate"/>
24. </constructor-arg>
25. </bean>
27. </beans>

**SimpleTest.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** SimpleTest {
9. **public** **static** **void** main(String[] args) {
11. Resource r=**new** ClassPathResource("applicationContext.xml");
12. BeanFactory factory=**new** XmlBeanFactory(r);
14. EmpDao dao=(EmpDao)factory.getBean("edao");
15. dao.save(**new** Emp(23,"sonoo",50000));
17. }
18. }

# **Spring SimpleJdbcTemplate Example**

1. [Spring SimpleJdbcTemplate](https://www.javatpoint.com/spring-SimpleJdbcTemplate-example)
2. [Example of SimpleJdbcTemplate](https://www.javatpoint.com/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

1. **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.

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**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.

1. **package** com.javatpoint;
3. **import** org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
4. **public** **class** EmpDao {
5. SimpleJdbcTemplate template;
7. **public** EmpDao(SimpleJdbcTemplate template) {
8. **this**.template = template;
9. }
10. **public** **int** update (Emp e){
11. String query="update employee set name=? where id=?";
12. **return** template.update(query,e.getName(),e.getId());
14. //String query="update employee set name=?,salary=? where id=?";
15. //return template.update(query,e.getName(),e.getSalary(),e.getId());
16. }
18. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jtemplate" **class**="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
17. <constructor-arg ref="ds"></constructor-arg>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmpDao">
21. <constructor-arg>
22. <ref bean="jtemplate"/>
23. </constructor-arg>
24. </bean>
26. </beans>

**SimpleTest.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** SimpleTest {
9. **public** **static** **void** main(String[] args) {
11. Resource r=**new** ClassPathResource("applicationContext.xml");
12. BeanFactory factory=**new** XmlBeanFactory(r);
14. EmpDao dao=(EmpDao)factory.getBean("edao");
15. **int** status=dao.update(**new** Emp(23,"Tarun",35000));
16. System.out.println(status);
17. }
18. }

To understand the concepts related to Spring JDBC framework with JdbcTemplate class, let us write a simple example, which will implement all the CRUD operations on the following Student table.

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Before proceeding, let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Add Spring JDBC specific latest libraries **mysql-connector-java.jar**, **org.springframework.jdbc.jar** and **org.springframework.transaction.jar** in the project. You can download required libraries if you do not have them already. |
| 4 | Create DAO interface *StudentDAO* and list down all the required methods. Though it is not required and you can directly write *StudentJDBCTemplate* class, but as a good practice, let's do it. |
| 5 | Create other required Java classes *Student*, *StudentMapper*, *StudentJDBCTemplate* and *MainApp* under the *com.tutorialspoint* package. |
| 6 | Make sure you already created **Student** table in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the give username and password. |
| 7 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 8 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Following is the content of the Data Access Object interface file **StudentDAO.java** −

package com.tutorialspoint;

import java.util.List;

import javax.sql.DataSource;

public interface StudentDAO {

/\*\*

\* This is the method to be used to initialize

\* database resources ie. connection.

\*/

public void setDataSource(DataSource ds);

/\*\*

\* This is the method to be used to create

\* a record in the Student table.

\*/

public void create(String name, Integer age);

/\*\*

\* This is the method to be used to list down

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public Student getStudent(Integer id);

/\*\*

\* This is the method to be used to list down

\* all the records from the Student table.

\*/

public List<Student> listStudents();

/\*\*

\* This is the method to be used to delete

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public void delete(Integer id);

/\*\*

\* This is the method to be used to update

\* a record into the Student table.

\*/

public void update(Integer id, Integer age);

}

Following is the content of the **Student.java** file

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

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;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of the **StudentMapper.java** file

package com.tutorialspoint;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setId(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO.

package com.tutorialspoint;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplateObject;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplateObject = new JdbcTemplate(dataSource);

}

public void create(String name, Integer age) {

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, name, age);

System.out.println("Created Record Name = " + name + " Age = " + age);

return;

}

public Student getStudent(Integer id) {

String SQL = "select \* from Student where id = ?";

Student student = jdbcTemplateObject.queryForObject(SQL,

new Object[]{id}, new StudentMapper());

return student;

}

public List<Student> listStudents() {

String SQL = "select \* from Student";

List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());

return students;

}

public void delete(Integer id) {

String SQL = "delete from Student where id = ?";

jdbcTemplateObject.update(SQL, id);

System.out.println("Deleted Record with ID = " + id );

return;

}

public void update(Integer id, Integer age){

String SQL = "update Student set age = ? where id = ?";

jdbcTemplateObject.update(SQL, age, id);

System.out.println("Updated Record with ID = " + id );

return;

}

}

Following is the content of the **MainApp.java** file

package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {

public static void main(String[] args) {

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

StudentJDBCTemplate studentJDBCTemplate =

(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

System.out.println("------Records Creation--------" );

studentJDBCTemplate.create("Zara", 11);

studentJDBCTemplate.create("Nuha", 2);

studentJDBCTemplate.create("Ayan", 15);

System.out.println("------Listing Multiple Records--------" );

List<Student> students = studentJDBCTemplate.listStudents();

for (Student record : students) {

System.out.print("ID : " + record.getId() );

System.out.print(", Name : " + record.getName() );

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

}

System.out.println("----Updating Record with ID = 2 -----" );

studentJDBCTemplate.update(2, 20);

System.out.println("----Listing Record with ID = 2 -----" );

Student student = studentJDBCTemplate.getStudent(2);

System.out.print("ID : " + student.getId() );

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

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

}

}

Following is the configuration file **Beans.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"

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

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

<!-- Initialization for data source -->

<bean id="dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

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

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

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

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

</bean>

<!-- Definition for studentJDBCTemplate bean -->

<bean id = "studentJDBCTemplate"

class = "com.tutorialspoint.StudentJDBCTemplate">

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

</bean>

</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

------Records Creation--------

Created Record Name = Zara Age = 11

Created Record Name = Nuha Age = 2

Created Record Name = Ayan Age = 15

------Listing Multiple Records--------

ID : 1, Name : Zara, Age : 11

ID : 2, Name : Nuha, Age : 2

ID : 3, Name : Ayan, Age : 15

----Updating Record with ID = 2 -----

Updated Record with ID = 2

----Listing Record with ID = 2 -----

ID : 2, Name : Nuha, Age : 20

You can try and delete the operation yourself, which we have not used in the example, but now you have one working application based on Spring JDBC framework, which you can extend to add sophisticated functionality based on your project requirements. There are other approaches to access the database where you will use **NamedParameterJdbcTemplate** and **SimpleJdbcTemplate** classes, so if you are interested in learning these classes then kindly check the reference manual for Spring Framework.

# **Example of PreparedStatement in Spring JdbcTemplate**

1. [PreparedStatement in Spring JDBC Template](https://www.javatpoint.com/example-of-PreparedStatement-in-Spring-JdbcTemplate)
2. [PreparedStatementCallback interface](https://www.javatpoint.com/example-of-PreparedStatement-in-Spring-JdbcTemplate)
3. [Example of using PreparedStatement in Spring](https://www.javatpoint.com/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

1. **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:

1. **public** T doInPreparedStatement(PreparedStatement ps)**throws** SQLException, DataAccessException

### Example of using PreparedStatement in Spring

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

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**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.

1. **package** com.javatpoint;
2. **import** java.sql.PreparedStatement;
3. **import** java.sql.SQLException;
5. **import** org.springframework.dao.DataAccessException;
6. **import** org.springframework.jdbc.core.JdbcTemplate;
7. **import** org.springframework.jdbc.core.PreparedStatementCallback;
9. **public** **class** EmployeeDao {
10. **private** JdbcTemplate jdbcTemplate;
12. **public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {
13. **this**.jdbcTemplate = jdbcTemplate;
14. }
16. **public** Boolean saveEmployeeByPreparedStatement(**final** Employee e){
17. String query="insert into employee values(?,?,?)";
18. **return** jdbcTemplate.execute(query,**new** PreparedStatementCallback<Boolean>(){
19. @Override
20. **public** Boolean doInPreparedStatement(PreparedStatement ps)
21. **throws** SQLException, DataAccessException {
23. ps.setInt(1,e.getId());
24. ps.setString(2,e.getName());
25. ps.setFloat(3,e.getSalary());
27. **return** ps.execute();
29. }
30. });
31. }

34. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>

25. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test {
7. **public** **static** **void** main(String[] args) {
8. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
10. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
11. dao.saveEmployeeByPreparedStatement(**new** Employee(108,"Amit",35000));
12. }
13. }

# **ResultSetExtractor Example | Fetching Records by Spring JdbcTemplate**

1. [ResultSetExtractor Example](https://www.javatpoint.com/ResultSetExtractor-example)
2. [ResultSetExtractor Interface](https://www.javatpoint.com/ResultSetExtractor-example)
3. [Method of ResultSetExtractor Interface](https://www.javatpoint.com/ResultSetExtractor-example)
4. [Example of ResultSetExtractor](https://www.javatpoint.com/ResultSetExtractor-example)

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

1. **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:

1. **public** T extractData(ResultSet rs)**throws** SQLException,DataAccessException

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

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

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
10. **public** String toString(){
11. **return** id+" "+name+" "+salary;
12. }
13. }

**EmployeeDao.java**

It contains on property jdbcTemplate and one method getAllEmployees.

1. **package** com.javatpoint;
2. **import** java.sql.ResultSet;
3. **import** java.sql.SQLException;
4. **import** java.util.ArrayList;
5. **import** java.util.List;
6. **import** org.springframework.dao.DataAccessException;
7. **import** org.springframework.jdbc.core.JdbcTemplate;
8. **import** org.springframework.jdbc.core.ResultSetExtractor;
10. **public** **class** EmployeeDao {
11. **private** JdbcTemplate template;
13. **public** **void** setTemplate(JdbcTemplate template) {
14. **this**.template = template;
15. }
17. **public** List<Employee> getAllEmployees(){
18. **return** template.query("select \* from employee",**new** ResultSetExtractor<List<Employee>>(){
19. @Override
20. **public** List<Employee> extractData(ResultSet rs) **throws** SQLException,
21. DataAccessException {
23. List<Employee> list=**new** ArrayList<Employee>();
24. **while**(rs.next()){
25. Employee e=**new** Employee();
26. e.setId(rs.getInt(1));
27. e.setName(rs.getString(2));
28. e.setSalary(rs.getInt(3));
29. list.add(e);
30. }
31. **return** list;
32. }
33. });
34. }
35. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>
24. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** java.util.List;
5. **import** org.springframework.context.ApplicationContext;
6. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
7. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
11. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
12. List<Employee> list=dao.getAllEmployees();
14. **for**(Employee e:list)
15. System.out.println(e);
17. }
19. }

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

1. [RowMapper](https://www.javatpoint.com/RowMapper-example)
2. [RowMapper Interface](https://www.javatpoint.com/RowMapper-example)
3. [Method of RowMapper Interface](https://www.javatpoint.com/RowMapper-example)
4. [Example of RowMapper Interface](https://www.javatpoint.com/RowMapper-example)

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

1. **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:

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

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

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

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. **public** String toString(){
10. **return** id+" "+name+" "+salary;
11. }
12. }

**EmployeeDao.java**

It contains on property jdbcTemplate and one method getAllEmployeesRowMapper.

1. **package** com.javatpoint;
2. **import** java.sql.ResultSet;
3. **import** java.sql.SQLException;
4. **import** java.util.ArrayList;
5. **import** java.util.List;
6. **import** org.springframework.dao.DataAccessException;
7. **import** org.springframework.jdbc.core.JdbcTemplate;
8. **import** org.springframework.jdbc.core.ResultSetExtractor;
9. **import** org.springframework.jdbc.core.RowMapper;
11. **public** **class** EmployeeDao {
12. **private** JdbcTemplate template;
14. **public** **void** setTemplate(JdbcTemplate template) {
15. **this**.template = template;
16. }
18. **public** List<Employee> getAllEmployeesRowMapper(){
19. **return** template.query("select \* from employee",**new** RowMapper<Employee>(){
20. @Override
21. **public** Employee mapRow(ResultSet rs, **int** rownumber) **throws** SQLException {
22. Employee e=**new** Employee();
23. e.setId(rs.getInt(1));
24. e.setName(rs.getString(2));
25. e.setSalary(rs.getInt(3));
26. **return** e;
27. }
28. });
29. }
30. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>
24. </beans>

**Test.java**

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

1. **package** com.javatpoint;
3. **import** java.util.List;
5. **import** org.springframework.context.ApplicationContext;
6. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
7. **public** **class** Test {
8. **public** **static** **void** main(String[] args) {
9. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
10. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
11. List<Employee> list=dao.getAllEmployeesRowMapper();
13. **for**(Employee e:list)
14. System.out.println(e);
15. }
16. }

# **Spring NamedParameterJdbcTemplate Example**

1. [NamedParameterJdbcTemplate](https://www.javatpoint.com/spring-NamedParameterJdbcTemplate-example)
2. [Method of NamedParameterJdbcTemplate](https://www.javatpoint.com/spring-NamedParameterJdbcTemplate-example)
3. [Spring NamedParameterJdbcTemplate Example](https://www.javatpoint.com/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

1. 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:

1. 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.

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**EmployeeDao.java**

It contains on property jdbcTemplate and one method save.

1. **package** com.javatpoint;
3. **import** java.sql.PreparedStatement;
4. **import** java.sql.SQLException;
5. **import** org.springframework.dao.DataAccessException;
6. **import** org.springframework.jdbc.core.PreparedStatementCallback;
7. **import** org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
8. **import** java.util.\*;
10. **public** **class** EmpDao {
11. NamedParameterJdbcTemplate template;
13. **public** EmpDao(NamedParameterJdbcTemplate template) {
14. **this**.template = template;
15. }
16. **public**  **void** save (Emp e){
17. String query="insert into employee values (:id,:name,:salary)";
19. Map<String,Object> map=**new** HashMap<String,Object>();
20. map.put("id",e.getId());
21. map.put("name",e.getName());
22. map.put("salary",e.getSalary());
24. template.execute(query,map,**new** PreparedStatementCallback() {
25. @Override
26. **public** Object doInPreparedStatement(PreparedStatement ps)
27. **throws** SQLException, DataAccessException {
28. **return** ps.executeUpdate();
29. }
30. });
31. }
32. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jtemplate"
17. **class**="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
18. <constructor-arg ref="ds"></constructor-arg>
19. </bean>
21. <bean id="edao" **class**="com.javatpoint.EmpDao">
22. <constructor-arg>
23. <ref bean="jtemplate"/>
24. </constructor-arg>
25. </bean>
27. </beans>

**SimpleTest.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** SimpleTest {
9. **public** **static** **void** main(String[] args) {
11. Resource r=**new** ClassPathResource("applicationContext.xml");
12. BeanFactory factory=**new** XmlBeanFactory(r);
14. EmpDao dao=(EmpDao)factory.getBean("edao");
15. dao.save(**new** Emp(23,"sonoo",50000));
17. }
18. }

# **Spring SimpleJdbcTemplate Example**

1. [Spring SimpleJdbcTemplate](https://www.javatpoint.com/spring-SimpleJdbcTemplate-example)
2. [Example of SimpleJdbcTemplate](https://www.javatpoint.com/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

1. **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.

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

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

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**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.

1. **package** com.javatpoint;
3. **import** org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
4. **public** **class** EmpDao {
5. SimpleJdbcTemplate template;
7. **public** EmpDao(SimpleJdbcTemplate template) {
8. **this**.template = template;
9. }
10. **public** **int** update (Emp e){
11. String query="update employee set name=? where id=?";
12. **return** template.update(query,e.getName(),e.getId());
14. //String query="update employee set name=?,salary=? where id=?";
15. //return template.update(query,e.getName(),e.getSalary(),e.getId());
16. }
18. }

**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.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jtemplate" **class**="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
17. <constructor-arg ref="ds"></constructor-arg>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmpDao">
21. <constructor-arg>
22. <ref bean="jtemplate"/>
23. </constructor-arg>
24. </bean>
26. </beans>

**SimpleTest.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** SimpleTest {
9. **public** **static** **void** main(String[] args) {
11. Resource r=**new** ClassPathResource("applicationContext.xml");
12. BeanFactory factory=**new** XmlBeanFactory(r);
14. EmpDao dao=(EmpDao)factory.getBean("edao");
15. **int** status=dao.update(**new** Emp(23,"Tarun",35000));
16. System.out.println(status);
17. }
18. }

# **Spring with ORM Frameworks**

1. [Spring with ORM Frameworks](https://www.javatpoint.com/spring-with-orm-frameworks)
2. [Advantage of Spring with ORM](https://www.javatpoint.com/spring-with-orm-frameworks)

Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA(Java Persistence API), JDO(Java Data Objects), Oracle Toplink and iBATIS.

### Advantage of ORM Frameworks with Spring

There are a lot of advantage of Spring framework in respect to ORM frameworks. There are as follows:

* **Less coding is required**: By the help of Spring framework, you don't need to write extra codes before and after the actual database logic such as getting the connection, starting transaction, commiting transaction, closing connection etc.
* **Easy to test**: Spring's IoC approach makes it easy to test the application.
* **Better exception handling**: Spring framework provides its own API for exception handling with ORM framework.
* **Integrated transaction management**: By the help of Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP style method interceptor.

### Upcoming Topics in Spring with ORM

[Spring and JPA Integration](https://www.javatpoint.com/spring-and-jpa-integration)

Example of spring and JPA Integration by implementation of Hibernate.

[Hibernate and Spring Integration](https://www.javatpoint.com/hibernate-and-spring-integration)

Example of hibernate and spring integration by the help of HibernateTemplate class.

# **Hibernate and Spring Integration**

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

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

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

### Advantage of Spring framework with hibernate

The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

So **it saves a lot of code**.

**Understanding problem without using spring:**

Let's understand it by the code of hibernate given below:

1. //creating configuration
2. Configuration cfg=**new** Configuration();
3. cfg.configure("hibernate.cfg.xml");
5. //creating seession factory object
6. SessionFactory factory=cfg.buildSessionFactory();
8. //creating session object
9. Session session=factory.openSession();
11. //creating transaction object
12. Transaction t=session.beginTransaction();
14. Employee e1=**new** Employee(111,"arun",40000);
15. session.persist(e1);//persisting the object
17. t.commit();//transaction is commited
18. session.close();

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

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

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

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

## **Methods of HibernateTemplate class**

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

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

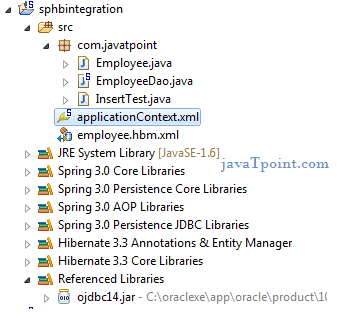
## **Steps**

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

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

### Example of Hibernate and spring integration

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



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

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

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

**2) Employee.java**

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

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

**3) employee.hbm.xml**

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

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

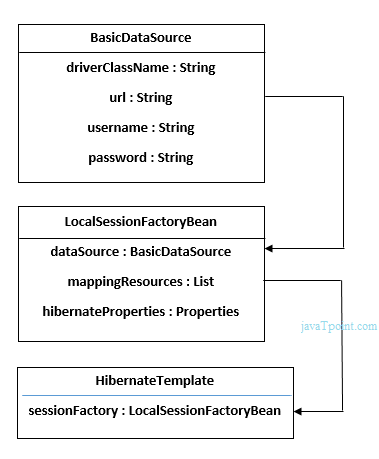
**4) EmployeeDao.java**

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

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

**5) applicationContext.xml**

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



*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

10. <bean id="dataSource" **class**="org.apache.commons.dbcp.BasicDataSource">
11. <property name="driverClassName"  value="oracle.jdbc.driver.OracleDriver"></property>
12. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>
13. <property name="username" value="system"></property>
14. <property name="password" value="oracle"></property>
15. </bean>
17. <bean id="mysessionFactory"  **class**="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
18. <property name="dataSource" ref="dataSource"></property>
20. <property name="mappingResources">
21. <list>
22. <value>employee.hbm.xml</value>
23. </list>
24. </property>
26. <property name="hibernateProperties">
27. <props>
28. <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
29. <prop key="hibernate.hbm2ddl.auto">update</prop>
30. <prop key="hibernate.show\_sql">**true**</prop>
32. </props>
33. </property>
34. </bean>
36. <bean id="template" **class**="org.springframework.orm.hibernate3.HibernateTemplate">
37. <property name="sessionFactory" ref="mysessionFactory"></property>
38. </bean>
40. <bean id="d" **class**="com.javatpoint.EmployeeDao">
41. <property name="template" ref="template"></property>
42. </bean>

45. </beans>

**6) InsertTest.java**

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

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** InsertTest {
9. **public** **static** **void** main(String[] args) {
11. Resource r=**new** ClassPathResource("applicationContext.xml");
12. BeanFactory factory=**new** XmlBeanFactory(r);
14. EmployeeDao dao=(EmployeeDao)factory.getBean("d");
16. Employee e=**new** Employee();
17. e.setId(114);
18. e.setName("varun");
19. e.setSalary(50000);
21. dao.saveEmployee(e);
23. }
24. }

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

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

### Enabling automatic table creation, showing sql queries etc.

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

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

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

# **Spring Data JPA Tutorial**

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

JPA (Java Persistent API) is the sun specification for persisting objects in the enterprise application. It is currently used as the replacement for complex entity beans.

The implementation of JPA specification are provided by many vendors such as:

* Hibernate
* Toplink
* iBatis
* OpenJPA etc.

## **Advantage of Spring JpaTemplate**

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

So, it **save a lot of code**.

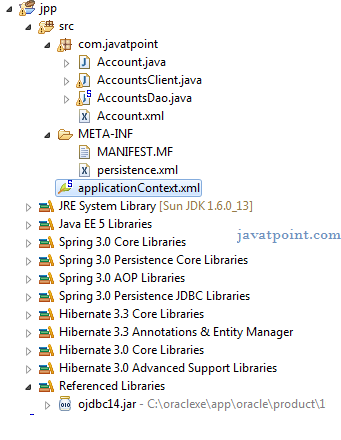
In this example, we are going to use hibernate for the implementation of JPA.

### Example of Spring and JPA Integration

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

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

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



**1) Account.java**

It is a simple POJO class.

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

**2) Account.xml**

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

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

**3) AccountDao.java**

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

**4) persistence.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<persistence** xmlns="http://java.sun.com/xml/ns/persistence"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5. http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd" version="1.0"**>**
7. **<persistence-unit** name="ForAccountsDB"**>**
8. **<mapping-file>**com/javatpoint/Account.xml**</mapping-file>**
9. **<class>**com.javatpoint.Account**</class>**
10. **</persistence-unit>**
11. **</persistence>**

**5) applicationContext.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:tx="http://www.springframework.org/schema/tx"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/tx
8. http://www.springframework.org/schema/tx/spring-tx-3.0.xsd"**>**
10. **<tx:annotation-driven** transaction-manager="jpaTxnManagerBean" proxy-target-class="true"**/>**
12. **<bean** id="dataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>**
13. **<property** name="driverClassName" value="oracle.jdbc.driver.OracleDriver"**></property>**
14. **<property** name="url" value="jdbc:oracle:thin:@localhost:1521:xe"**></property>**
15. **<property** name="username" value="system"**></property>**
16. **<property** name="password" value="oracle"**></property>**
17. **</bean>**
19. **<bean** id="hbAdapterBean" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"**>**
20. **<property** name="showSql" value="true"**></property>**
21. **<property** name="generateDdl" value="true"**></property>**
22. **<property** name="databasePlatform" value="org.hibernate.dialect.OracleDialect"**></property>**
23. **</bean>**
25. **<bean** id="emfBean" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"**>**
26. **<property** name="dataSource" ref="dataSourceBean"**></property>**
27. **<property** name="jpaVendorAdapter" ref="hbAdapterBean"**></property>**
28. **</bean>**
30. **<bean** id="jpaTemplateBean" class="org.springframework.orm.jpa.JpaTemplate"**>**
31. **<property** name="entityManagerFactory" ref="emfBean"**></property>**
32. **</bean>**
34. **<bean** id="accountsDaoBean" class="com.javatpoint.AccountsDao"**>**
35. **<property** name="template" ref="jpaTemplateBean"**></property>**
36. **</bean>**
37. **<bean** id="jpaTxnManagerBean" class="org.springframework.orm.jpa.JpaTransactionManager"**>**
38. **<property** name="entityManagerFactory" ref="emfBean"**></property>**
39. **</bean>**
41. **</beans>**

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

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

**6) Accountsclient.java**

1. **package** com.javatpoint;
3. **import** java.util.List;
4. **import** org.springframework.context.ApplicationContext;
5. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
6. **import** org.springframework.context.support.FileSystemXmlApplicationContext;
8. **public** **class** AccountsClient{
9. **public** **static** **void** main(String[] args){
10. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
11. AccountsDao accountsDao = context.getBean("accountsDaoBean",AccountsDao.**class**);
13. accountsDao.createAccount(15, "Jai Kumar", 41000);
14. accountsDao.createAccount(20, "Rishi ", 35000);
15. System.out.println("Accounts created");
17. //accountsDao.updateBalance(20, 50000);
18. //System.out.println("Account balance updated");

21. /\*List<Account> accounts = accountsDao.getAllAccounts();
22. for (int i = 0; i < accounts.size(); i++) {
23. Account acc = accounts.get(i);
24. System.out.println(acc.getAccountNumber() + " : " + acc.getOwner() + " (" + acc.getBalance() + ")");
25. }\*/
27. //accountsDao.deleteAccount(111);
28. //System.out.println("Account deleted");
30. }
31. }

#### Output

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

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

Accounts created

# **Spring Expression Language (SPEL) Tutorial**

1. [SPEL](https://www.javatpoint.com/spring-expression-language-tutorial)
2. [SPEL API](https://www.javatpoint.com/spring-expression-language-tutorial)
3. [Hello SPEL Example](https://www.javatpoint.com/spring-expression-language-tutorial)
4. [Other Examples of SPEL](https://www.javatpoint.com/spring-expression-language-tutorial)

**SpEL** is an exression language supporting the features of querying and manipulating an object graph at runtime.

There are many expression languages available such as JSP EL, OGNL, MVEL and JBoss EL. SpEL provides some additional features such as method invocation and string templating functionality.

### SpEL API

The SpEL API provides many interfaces and classes. They are as follows:

* Expression interface
* SpelExpression class
* ExpressionParser interface
* SpelExpressionParser class
* EvaluationContext interface
* StandardEvaluationContext class

### Hello SPEL Example

1. **import** org.springframework.expression.Expression;
2. **import** org.springframework.expression.ExpressionParser;
3. **import** org.springframework.expression.spel.standard.SpelExpressionParser;
5. **public** **class** Test {
6. **public** **static** **void** main(String[] args) {
7. ExpressionParser parser = **new** SpelExpressionParser();
9. Expression exp = parser.parseExpression("'Hello SPEL'");
10. String message = (String) exp.getValue();
11. System.out.println(message);
12. //OR
13. //System.out.println(parser.parseExpression("'Hello SPEL'").getValue());
14. }
15. }

[download this example (developed using MyEclipse IDE)](https://static.javatpoint.com/src/sp/spel1.zip)

### Other SPEL Example

Let's see a lot of useful examples of SPEL. Here, we are assuming all the examples have been written inside the main() method.

#### Using concat() method with String

1. ExpressionParser parser = **new** SpelExpressionParser();
2. Expression exp = parser.parseExpression("'Welcome SPEL'.concat('!')");
3. String message = (String) exp.getValue();
4. System.out.println(message);

#### Converting String into byte array

1. Expression exp = parser.parseExpression("'Hello World'.bytes");
2. **byte**[] bytes = (**byte**[]) exp.getValue();
3. **for**(**int** i=0;i<bytes.length;i++){
4. System.out.print(bytes[i]+" ");
5. }

#### Getting length after converting string into bytes

1. Expression exp = parser.parseExpression("'Hello World'.bytes.length");
2. **int** length = (Integer) exp.getValue();
3. System.out.println(length);

#### Converting String contents into uppercase letter

1. Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
2. String message = exp.getValue(String.**class**);
3. System.out.println(message);
4. //OR
5. System.out.println(parser.parseExpression("'hello world'.toUpperCase()").getValue());

# **Operators in SPEL**

1. [Examples of Operators in SPEL](https://www.javatpoint.com/operators-in-spel)

We can use many operators in SpEL such as arithmetic, relational, logical etc. There are given a lot of examples of using different operators in SpEL.

### Examples of using operators in SPEL

1. **import** org.springframework.expression.ExpressionParser;
2. **import** org.springframework.expression.spel.standard.SpelExpressionParser;

5. **public** **class** Test {
6. **public** **static** **void** main(String[] args) {
7. ExpressionParser parser = **new** SpelExpressionParser();
9. //arithmetic operator
10. System.out.println(parser.parseExpression("'Welcome SPEL'+'!'").getValue());
11. System.out.println(parser.parseExpression("10 \* 10/2").getValue());
12. System.out.println(parser.parseExpression("'Today is: '+ new java.util.Date()").getValue());
14. //logical operator
15. System.out.println(parser.parseExpression("true and true").getValue());
17. //Relational operator
18. System.out.println(parser.parseExpression("'sonoo'.length()==5").getValue());
19. }
20. }

# **Variable in SPEL | StandardEvaluationContext**

1. [Using Variable in SPEL](https://www.javatpoint.com/using-variable-in-spel)
2. [StandardEvaluationContext class](https://www.javatpoint.com/using-variable-in-spel)
3. [Example of Using variable in SPEL](https://www.javatpoint.com/using-variable-in-spel)

In SpEL, we can store a value in the variable and use the variable in the method and call the method. To work on variable, we need to use **StandardEvaluationContext** class.

### Example of Using variable in SPEL

**Calculation.java**

1. **public** **class** Calculation {
2. **private** **int** number;
3. **public** **int** getNumber() {
4. **return** number;
5. }
6. **public** **void** setNumber(**int** number) {
7. **this**.number = number;
8. }
9. **public** **int** cube(){
10. **return** number\*number\*number;
11. }
12. }

**Test.java**

1. **import** org.springframework.expression.ExpressionParser;
2. **import** org.springframework.expression.spel.standard.SpelExpressionParser;
3. **import** org.springframework.expression.spel.support.StandardEvaluationContext;
5. **public** **class** Test {
6. **public** **static** **void** main(String[] args) {
7. Calculation calculation=**new** Calculation();
8. StandardEvaluationContext context=**new** StandardEvaluationContext(calculation);
10. ExpressionParser parser = **new** SpelExpressionParser();
11. parser.parseExpression("number").setValue(context,"5");
13. System.out.println(calculation.cube());
14. }
15. }

# **Spring MVC**

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

## **Spring Web Model-View-Controller**



* **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
* **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
* **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
* **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

## **Understanding the flow of Spring Web MVC**



* As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
* The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
* The controller returns an object of ModelAndView.
* The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

## **Advantages of Spring MVC Framework**

Let's see some of the advantages of Spring MVC Framework:-

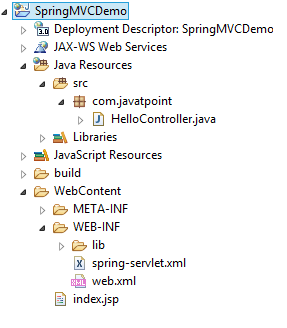
* **Separate roles** - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
* **Light-weight** - It uses light-weight servlet container to develop and deploy your application.
* **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
* **Rapid development** - The Spring MVC facilitates fast and parallel development.
* **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
* **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
* **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

## **Spring Web MVC Framework Example**

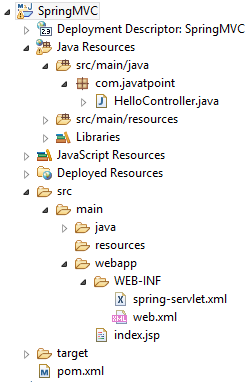
Let's see the simple example of a Spring Web MVC framework. The steps are as follows:

* Load the spring jar files or add dependencies in the case of Maven
* Create the controller class
* Provide the entry of controller in the web.xml file
* Define the bean in the separate XML file
* Display the message in the JSP page
* Start the server and deploy the project

## **Directory Structure of Spring MVC**



## **Directory Structure of Spring MVC using Maven**



## **Required Jar files or Maven Dependency**

To run this example, you need to load:

* Spring Core jar files
* Spring Web jar files
* JSP + JSTL jar files (If you are using any another view technology then load the corresponding jar files).

**Download Link:** [Download all the jar files for spring including JSP and JSTL](https://static.javatpoint.com/src/sp/springjars.zip).

If you are using Maven, you don't need to add jar files. Now, you need to add maven dependency to the pom.xml file.

### 1. Provide project information and configuration in the pom.xml file.

**pom.xml**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.javatpoint**</groupId>**
5. **<artifactId>**SpringMVC**</artifactId>**
6. **<packaging>**war**</packaging>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **<name>**SpringMVC Maven Webapp**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<dependencies>**
11. **<dependency>**
12. **<groupId>**junit**</groupId>**
13. **<artifactId>**junit**</artifactId>**
14. **<version>**3.8.1**</version>**
15. **<scope>**test**</scope>**
16. **</dependency>**
18. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
19. **<dependency>**
20. **<groupId>**org.springframework**</groupId>**
21. **<artifactId>**spring-webmvc**</artifactId>**
22. **<version>**5.1.1.RELEASE**</version>**
23. **</dependency>**
25. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
26. **<dependency>**
27. **<groupId>**javax.servlet**</groupId>**
28. **<artifactId>**servlet-api**</artifactId>**
29. **<version>**3.0-alpha-1**</version>**
30. **</dependency>**
32. **</dependencies>**
33. **<build>**
34. **<finalName>**SpringMVC**</finalName>**
35. **</build>**
36. **</project>**

### 2. Create the controller class

To create the controller class, we are using two annotations @Controller and @RequestMapping.

The @Controller annotation marks this class as Controller.

The @Requestmapping annotation is used to map the class with the specified URL name.

**HelloController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. **public** **class** HelloController {
6. @RequestMapping("/")
7. **public** String display()
8. {
9. **return** "index";
10. }
11. }

### 3. Provide the entry of controller in the web.xml file

In this xml file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the html file will be forwarded to the DispatcherServlet.

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 4. Define the bean in the xml file

This is the important configuration file where we need to specify the View components.

The context:component-scan element defines the base-package where DispatcherServlet will search the controller class.

This xml file should be located inside the WEB-INF directory.

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
14. <!-- Provide support for component scanning -->
15. **<context:component-scan** base-package="com.javatpoint" **/>**
17. <!--Provide support for conversion, formatting and validation -->
18. **<mvc:annotation-driven/>**
20. **</beans>**

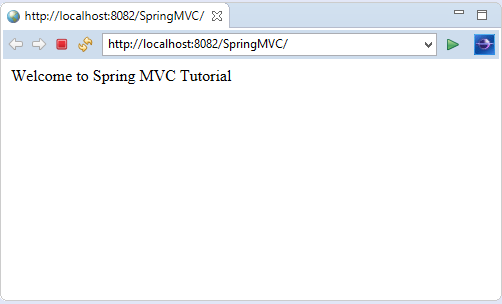
### 5. Display the message in the JSP page

This is the simple JSP page, displaying the message returned by the Controller.

**index.jsp**

1. **<html>**
2. **<body>**
3. **<p>**Welcome to Spring MVC Tutorial**</p>**
4. **</body>**
5. **</html>**

**Output:**



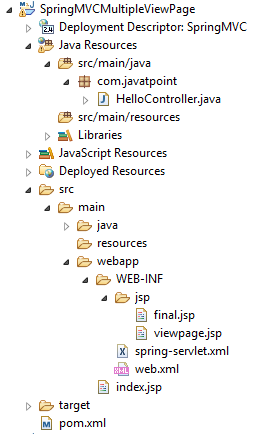
# **Spring MVC Multiple View page Example**

Here, we redirect a view page to another view page.

Let's see the simple example of a Spring Web MVC framework. The steps are as follows:

* Load the spring jar files or add dependencies in the case of Maven
* Create the controller class
* Provide the entry of controller in the web.xml file
* Define the bean in the separate XML file
* Create the other view components
* Start the server and deploy the project

## **Directory Structure of Spring MVC**



### 1. Add dependencies to pom.xml

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**

### 2. Create the request page

Let's create a simple jsp page containing a link.

**index.jsp**

1. **<html>**
2. **<body>**
3. **<a** href="hello"**>**Click here...**</a>**
4. **</body>**
5. **</html>**

### 3. Create the controller class

Let's create a controller class that returns the JSP pages. Here, we pass the specific name with a @Requestmapping annotation to map the class.

**HelloController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. **public** **class** HelloController {
6. @RequestMapping("/hello")
7. **public** String redirect()
8. {
9. **return** "viewpage";
10. }
11. @RequestMapping("/helloagain")
12. **public** String display()
13. {
14. **return** "final";
15. }
16. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

Now, we also provide view resolver with view component.

Here, the InternalResourceViewResolver class is used for the ViewResolver.

The prefix+string returned by controller+suffix page will be invoked for the view component.

This xml file should be located inside the WEB-INF directory.

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
14. <!-- Provide support for component scanning -->
15. **<context:component-scan** base-package="com.javatpoint" **/>**
17. <!--Provide support for conversion, formatting and validation -->
18. **<mvc:annotation-driven/>**
19. <!-- Define Spring MVC view resolver -->
20. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
21. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
22. **<property** name="suffix" value=".jsp"**></property>**
23. **</bean>**
24. **</beans>**

### 6. Create the other view components

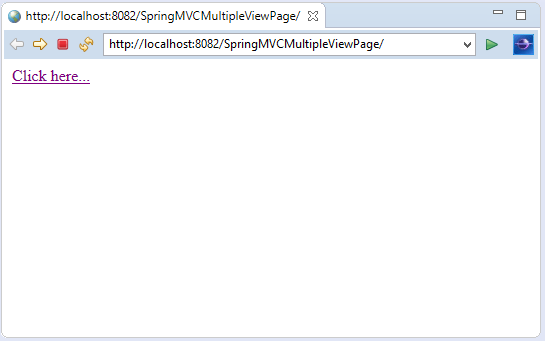
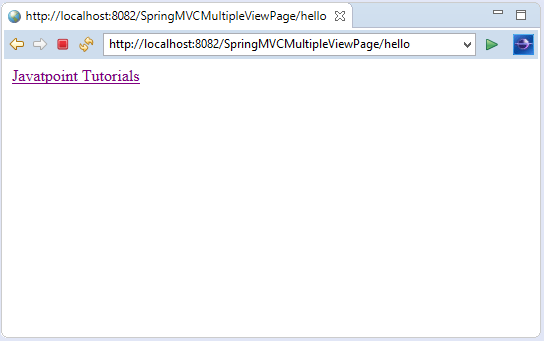
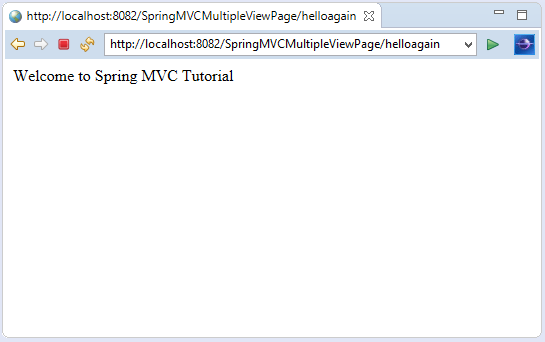
**viewpage.jsp**

1. **<html>**
2. **<body>**
3. **<a** href="helloagain"**>**Javatpoint Tutorials**</a>**
4. **</body>**
5. **</html>**

**final.jsp**

1. **<html>**
2. **<body>**
3. **<p>**Welcome to Spring MVC Tutorial**</p>**
4. **</body>**
5. **</html>**

**Output:**

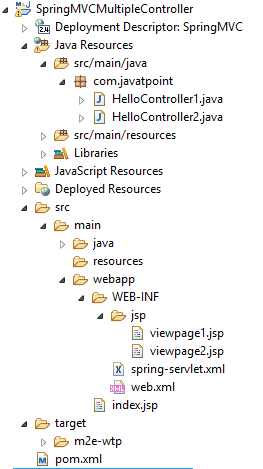
  
  


# **Spring MVC Multiple Controller Example**

In Spring MVC, we can create multiple controllers at a time. It is required to map each controller class with **@Controller** annotation. Here, we see a Spring MVC example of multiple controllers. The steps are as follows:

* Load the spring jar files or add dependencies in the case of Maven
* Create the controller class
* Provide the entry of controller in the web.xml file
* Define the bean in the separate XML file
* Create the other view components
* Start the server and deploy the project

## **Directory Structure of Spring MVC**



### 1. Add dependencies to pom.xml

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**

### 2. Create the request page

Let's create a simple JSP page containing two links.

**index.jsp**

1. **<html>**
2. **<body>**
3. **<a** href="hello1"**>**Spring MVC**</a>** ||
4. **<a** href="hello2"**>**Spring Boot**</a>**
5. **</body>**
6. **</html>**

### 3. Create the controller class

Let's create two controller classes, where each returns the particular view page.

**HelloController1.java**

1. package com.javatpoint;
2. import org.springframework.stereotype.Controller;
3. import org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. public class HelloController1 {
6. @RequestMapping("/hello1")
7. public String display()
8. {
9. return "viewpage1";
10. }
11. }

**HelloController2.java**

1. package com.javatpoint;
2. import org.springframework.stereotype.Controller;
3. import org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. public class HelloController2 {
6. @RequestMapping("/hello2")
7. public String display()
8. {
9. return "viewpage2";
10. }
11. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
14. <!-- Provide support for component scanning -->
15. **<context:component-scan** base-package="com.javatpoint" **/>**
17. <!--Provide support for conversion, formatting and validation -->
18. **<mvc:annotation-driven/>**
19. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
20. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
21. **<property** name="suffix" value=".jsp"**></property>**
22. **</bean>**
23. **</beans>**

### 6. Create the other view components

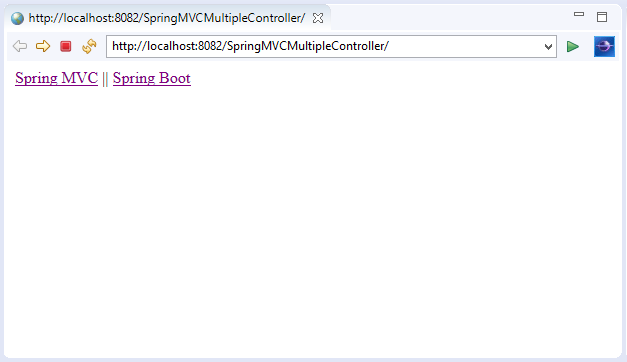
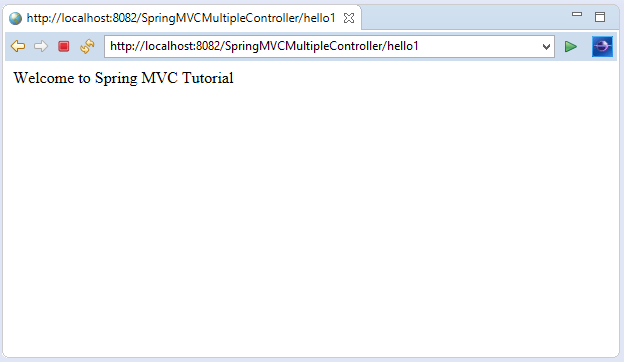
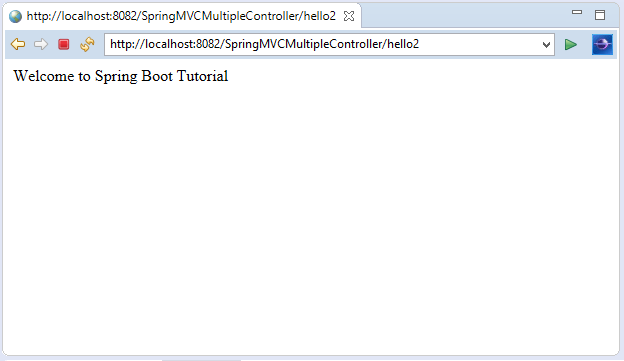
**viewpage1.jsp**

1. **<html>**
2. **<body>**
3. **<p>**Welcome to Spring MVC Tutorial**</p>**
4. **</body>**
5. **</html>**

**viewpage1.jsp**

1. **<html>**
2. **<body>**
3. **<p>**Welcome to Spring Boot Tutorial**</p>**
4. **</body>**
5. **</html>**

**Output:**

# **Spring MVC Model Interface**

In Spring MVC, the model works a container that contains the data of the application. Here, a data can be in any form such as objects, strings, information from the database, etc.

It is required to place the **Model** interface in the controller part of the application. The object of **HttpServletRequest** reads the information provided by the user and pass it to the **Model** interface. Now, a view page easily accesses the data from the model part.

## **Methods of Model Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Model addAllAttributes(Collection<?> arg) | It adds all the attributes in the provided Collection into this Map. |
| Model addAllAttributes(Map<String,?> arg) | It adds all the attributes in the provided Map into this Map. |
| Model addAllAttribute(Object arg) | It adds the provided attribute to this Map using a generated name. |
| Model addAllAttribute(String arg0, Object arg1) | It binds the attribute with the provided name. |
| Map<String, Object> asMap() | It return the current set of model attributes as a Map. |
| Model mergeAttributes(Map< String,?> arg) | It adds all attributes in the provided Map into this Map, with existing objects of the same name taking precedence. |
| boolean containsAttribute(String arg) | It indicates whether this model contains an attribute of the given name |

## **Spring MVC Model Example**

Let's create a login page that contains a username and password. Here, we validate the password with a specific value.

### 1. Add dependencies to pom.xml

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**

### 2. Create the request page

Here, we create the login page to receive name and password from the user.

**index.jsp**

1. **<html>**
2. **<body>**
3. **<form** action="hello"**>**
4. UserName : **<input** type="text" name="name"**/>** **<br><br>**
5. Password : **<input** type="text" name="pass"**/>** **<br><br>**
6. **<input** type="submit" name="submit"**>**
7. **</form>**
8. **</body>**
9. **</html>**

### 3. Create the controller class

In controller class:

* The **HttpServletRequest** is used to read the HTML form data provided by the user.
* The **Model** contains the request data and provides it to view page.

**HelloController.java**

1. **package** com.javatpoint;
2. **import** javax.servlet.http.HttpServletRequest;
3. **import** org.springframework.stereotype.Controller;
4. **import** org.springframework.ui.Model;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @Controller
8. **public** **class** HelloController {
10. @RequestMapping("/hello")
11. **public** String display(HttpServletRequest req,Model m)
12. {
13. //read the provided form data
14. String name=req.getParameter("name");
15. String pass=req.getParameter("pass");
16. **if**(pass.equals("admin"))
17. {
18. String msg="Hello "+ name;
19. //add a message to the model
20. m.addAttribute("message", msg);
21. **return** "viewpage";
22. }
23. **else**
24. {
25. String msg="Sorry "+ name+". You entered an incorrect password";
26. m.addAttribute("message", msg);
27. **return** "errorpage";
28. }
29. }
30. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
14. <!-- Provide support for component scanning -->
15. **<context:component-scan** base-package="com.javatpoint" **/>**
17. <!--Provide support for conversion, formatting and validation -->
18. **<mvc:annotation-driven/>**
19. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
20. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
21. **<property** name="suffix" value=".jsp"**></property>**
22. **</bean>**
23. **</beans>**

### 6. Create the other view components

To run this example, the following view components must be located inside the WEB-INF/jsp directory.

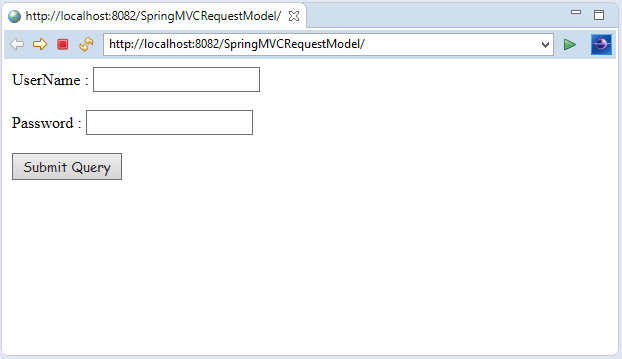
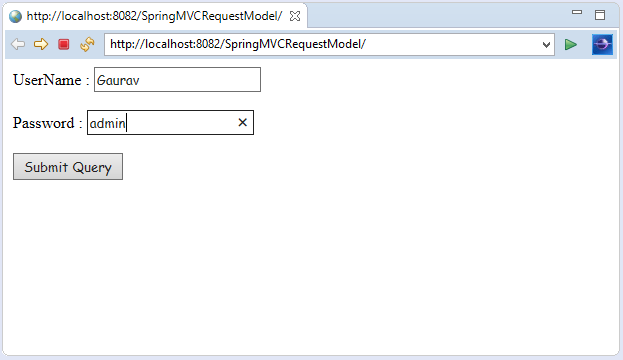
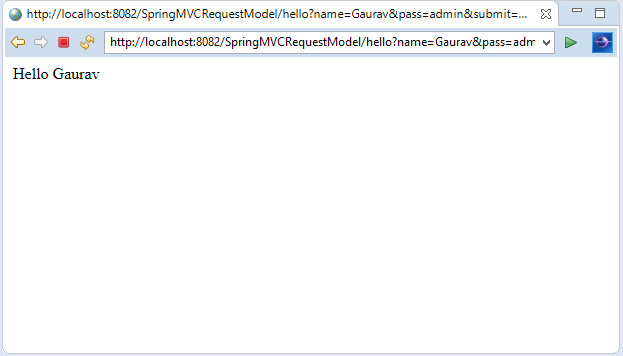
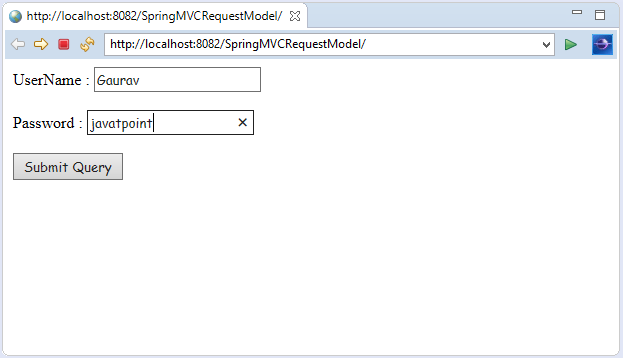
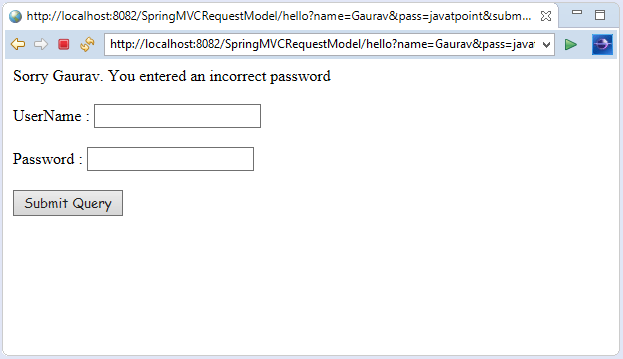
**viewpage.jsp**

1. **<html>**
2. **<body>**
3. ${message}
4. **</body>**
5. **</html>**

**errorpage.jsp**

1. **<html>**
2. **<body>**
3. ${message}
4. **<br><br>**
5. **<jsp:include** page="/index.jsp"**></jsp:include>**
6. **</body>**
7. **</html>**

**Output:**

# **Spring MVC RequestParam Annotation**

In Spring MVC, the **@RequestParam** annotation is used to read the form data and bind it automatically to the parameter present in the provided method. So, it ignores the requirement of **HttpServletRequest** object to read the provided data.

Including form data, it also maps the request parameter to query parameter and parts in multipart requests. If the method parameter type is Map and a request parameter name is specified, then the request parameter value is converted to a Map else the map parameter is populated with all request parameter names and values.

## **Spring MVC RequestParam Example**

Let's create a login page that contains a username and password. Here, we validate the password with a specific value.

### 1. Add dependencies to pom.xml

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**

### 2. Create the request page

It is the login page that receive name and password from the user.

**index.jsp**

1. **<html>**
2. **<body>**
3. **<form** action="hello"**>**
4. UserName : **<input** type="text" name="name"**/>** **<br><br>**
5. Password : **<input** type="text" name="pass"**/>** **<br><br>**
6. **<input** type="submit" name="submit"**>**
7. **</form>**
8. **</body>**
9. **</html>**

### 3. Create the Controller Class

In controller class:

* The @RequestParam is used to read the HTML form data provided by a user and bind it to the request parameter.
* The Model contains the request data and provides it to view page.

**HelloController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.RequestMapping;
5. **import** org.springframework.web.bind.annotation.RequestParam;
7. @Controller
8. **public** **class** HelloController {
10. @RequestMapping("/hello")
11. //read the provided form data
12. **public** String display(@RequestParam("name") String name,@RequestParam("pass") String pass,Model m)
13. {
14. **if**(pass.equals("admin"))
15. {
16. String msg="Hello "+ name;
17. //add a message to the model
18. m.addAttribute("message", msg);
19. **return** "viewpage";
20. }
21. **else**
22. {
23. String msg="Sorry "+ name+". You entered an incorrect password";
24. m.addAttribute("message", msg);
25. **return** "errorpage";
26. }
27. }
28. }

### 4. Create the other view components

To run this example, the following view components must be located inside the WEB-INF/jsp directory.

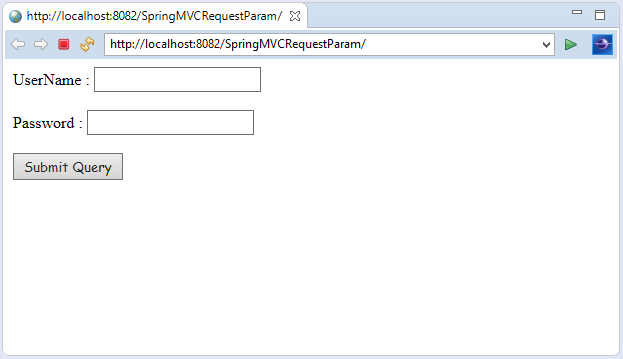
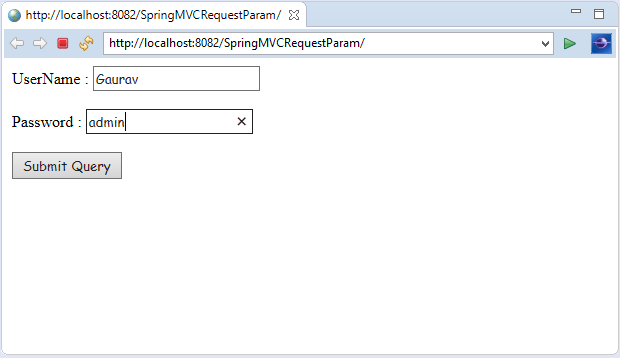
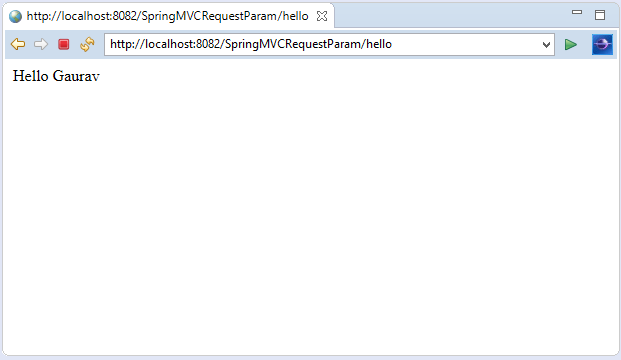
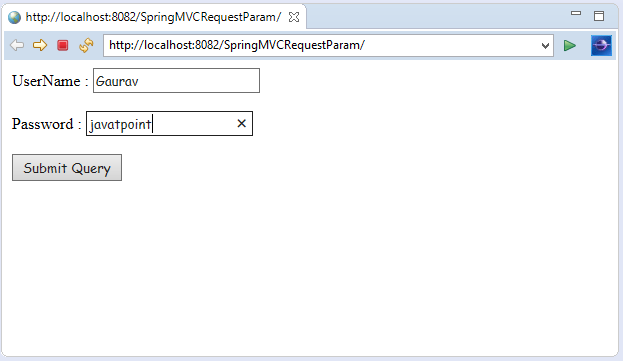
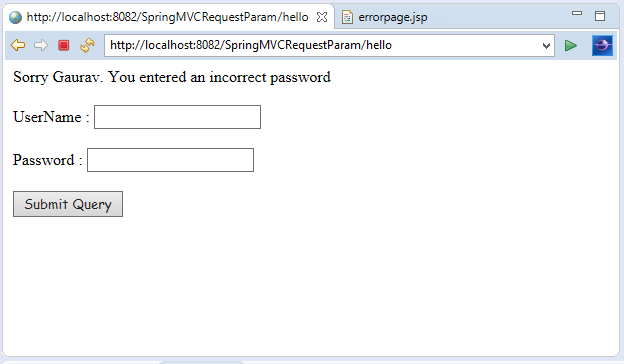
**viewpage.jsp**

1. **<html>**
2. **<body>**
3. ${message}
4. **</body>**
5. **</html>**

**errorpage.jsp**

1. **<html>**
2. **<body>**
3. ${message}
4. **<br><br>**
5. **<jsp:include** page="/index.jsp"**></jsp:include>**
6. **</body>**
7. **</html>**
8. **</html>**

**Output:**

# **Spring MVC Form Text Field**

The Spring MVC form text field tag generates an HTML input tag using the bound value. By default, the type of the input tag is text.

### Syntax

1. **<form:input** path="name" **/>**

Here, **path** attribute binds the form field to the bean property.

The Spring MVC form tag library also provides other input types such as email, date, tel, etc.

### For email:

1. **<form:input** type=?email? path="email" **/>**

### For date:

1. **<form:input** type=?date? path="date" **/>**

## **Example of Spring MVC Form Text Field**

Let's see an example to create a railway reservation form using form tag library.

### 1. Add dependencies to pom.xml file.

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

Here, the bean class contains the variables (along setter and getter methods) corresponding to the input field exist in the form.

**Reservation.java**

1. **package** com.javatpoint;
3. **public** **class** Reservation {
5. **private** String firstName;
6. **private** String lastName;
8. **public** Reservation()
9. {
10. }
11. **public** String getFirstName() {
12. **return** firstName;
13. }
14. **public** **void** setFirstName(String firstName) {
15. **this**.firstName = firstName;
16. }
17. **public** String getLastName() {
18. **return** lastName;
19. }
20. **public** **void** setLastName(String lastName) {
21. **this**.lastName = lastName;
22. }
23. }

### 3. Create the controller class

**ReservationController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.ModelAttribute;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @RequestMapping("/reservation")
8. @Controller
9. **public** **class** ReservationController {
10. @RequestMapping("/bookingForm")
11. **public** String bookingForm(Model model)
12. {
13. //create a reservation object
14. Reservation res=**new** Reservation();
15. //provide reservation object to the model
16. model.addAttribute("reservation", res);
17. **return** "reservation-page";
18. }
19. @RequestMapping("/submitForm")
20. // @ModelAttribute binds form data to the object
21. **public** String submitForm(@ModelAttribute("reservation") Reservation res)
22. {
23. **return** "confirmation-form";
24. }
25. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create other view components

**reservation-page.jsp**

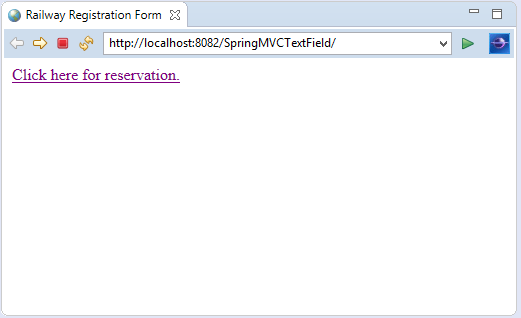
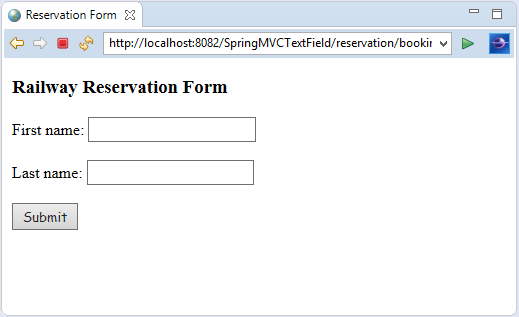
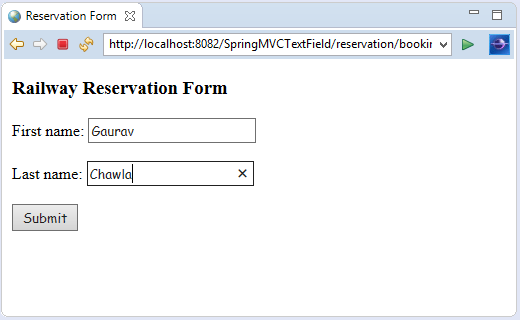
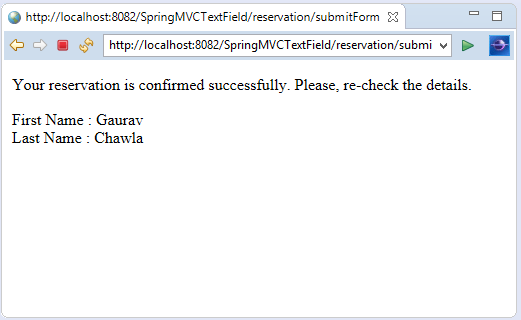
1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. **<input** type="submit" value="Submit" **/>**
15. **</form:form>**
16. **</body>**
17. **</html>**

### Note - The value passed with the @ModelAttribute annotation should be the same to the modelAttribute value present in the view page.

**confirmation-page.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<body>**
4. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
5. First Name : ${reservation.firstName} **<br>**
6. Last Name : ${reservation.lastName}
7. **</body>**
8. **</html>**

**Output:**

# **Spring MVC Form Radio Button**

The Spring MVC form radio button allows us to choose only one option at a time. This tag renders an HTML input tag of type radio.

### Syntax

**<form:radiobutton** path="abc" value="xyz"**/>**

Apart from radio button tag, Spring MVC form tag library also contains **radiobuttons** tag. This tag renders multiple HTML input tags with type radio.

1. **<form:radiobuttons** path="abc" items="${xyz}"**/>**

## **Example of Spring MVC Form Radio Button**

### 1. Add dependencies to pom.xml

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

**Reservation.java**

package com.javatpoint;

public class Reservation {

    private String firstName;

    private String lastName;

    private String Gender;

    public Reservation()

    {

    }

    public String getFirstName() {

        return firstName;

    }

    public void setFirstName(String firstName) {

        this.firstName = firstName;

    }

    public String getLastName() {

        return lastName;

    }

    public void setLastName(String lastName) {

        this.lastName = lastName;

    }

    public String getGender() {

        return Gender;

    }

    public void setGender(String gender) {

        Gender = gender;

    }

}

### 3. Create the controller class

**ReservationController.java**

**package** com.javatpoint;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.ModelAttribute;

**import** org.springframework.web.bind.annotation.RequestMapping;

@RequestMapping("/reservation")

@Controller

**public** **class** ReservationController {

    @RequestMapping("/bookingForm")

**public** String bookingForm(Model model)

{

      //create a reservation object

    Reservation res=**new** Reservation();

      //provide reservation object to the model

    model.addAttribute("reservation", res);

**return** "reservation-page";

}

@RequestMapping("/submitForm")

**public** String submitForm(@ModelAttribute("reservation") Reservation res)

{

**return** "confirmation-form";

}

}

Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create the view components

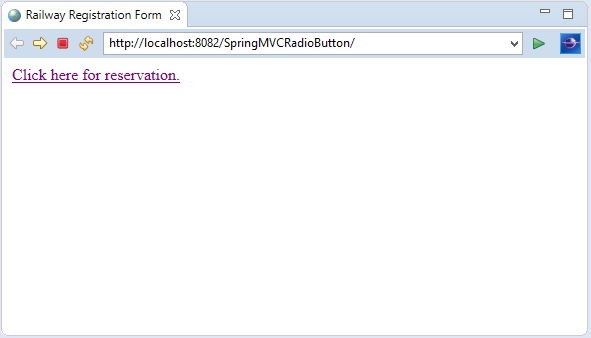
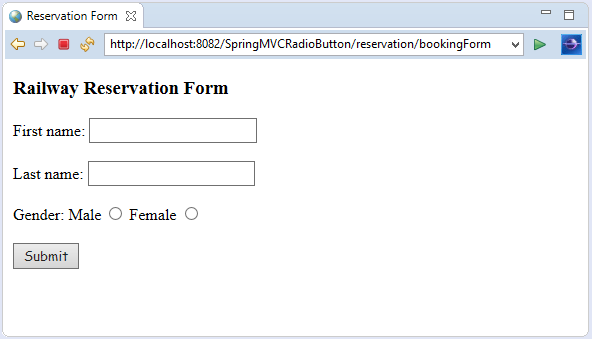
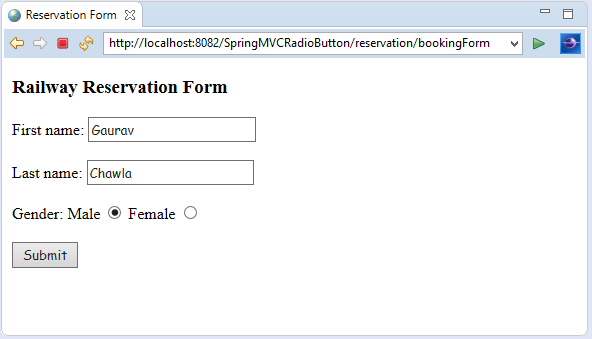
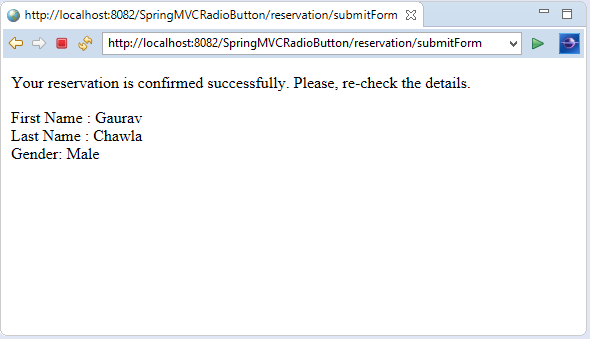
**reservation-page.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. Gender:
15. Male **<form:radiobutton** path="Gender" value="Male"**/>**
16. Female **<form:radiobutton** path="Gender" value="Female"**/>**
17. **<br><br>**
18. **<input** type="submit" value="Submit" **/>**
19. **</form:form>**
20. **</body>**
21. **</html>**

**confirmation-page.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<body>**
4. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
5. First Name : ${reservation.firstName} **<br>**
6. Last Name : ${reservation.lastName} **<br>**
7. Gender: ${reservation.gender}
8. **</body>**
9. **</html>**

**Output:**

# **Spring MVC Form Checkbox**

The Spring MVC form checkbox facilitates to choose multiple options at the same time. This tag renders an HTML input tag of type checkbox.

### Syntax

1. **<form:checkbox** path="abc" value="element"**/>**

Apart from checkbox tag, Spring MVC form tag library also contains **checkboxes** tag. This tag renders multiple HTML input tags with type checkbox. This tag is used only if you don't want to list all the elements in the view page. In such a case, you can provide the elements at runtime and pass it to the tag. As the user can select multiple options, you need to pass the elements of Array, a List or a Map type.

### Syntax

1. **<form:checkboxes** path="abc" items="${object.elementList}"**/>**

## **Example of Spring MVC Form Check Box**

### 1. Add dependencies to pom.xml file.

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

**Reservation.java**

1. **package** com.javatpoint;
3. **public** **class** Reservation {
5. **private** String firstName;
6. **private** String lastName;
7. **private** String Gender;
8. **private** String[] Food;
9. **public** Reservation()
10. {
11. }
12. **public** String getFirstName() {
13. **return** firstName;
14. }
15. **public** **void** setFirstName(String firstName) {
16. **this**.firstName = firstName;
17. }
18. **public** String getLastName() {
19. **return** lastName;
20. }
21. **public** **void** setLastName(String lastName) {
22. **this**.lastName = lastName;
23. }
24. **public** String getGender() {
25. **return** Gender;
26. }
27. **public** **void** setGender(String gender) {
28. Gender = gender;
29. }
30. **public** String[] getFood() {
31. **return** Food;
32. }
33. **public** **void** setFood(String[] food) {
34. Food = food;
35. }
36. }

### 3. Create the controller class

**ReservationController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.ModelAttribute;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @RequestMapping("/reservation")
8. @Controller
9. **public** **class** ReservationController {
10. @RequestMapping("/bookingForm")
11. **public** String bookingForm(Model model)
12. {
13. //create a reservation object
14. Reservation res=**new** Reservation();
15. //provide reservation object to the model
16. model.addAttribute("reservation", res);
17. **return** "reservation-page";
18. }
19. @RequestMapping("/submitForm")
20. **public** String submitForm(@ModelAttribute("reservation") Reservation res)
21. {
22. **return** "confirmation-form";
23. }
24. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create the view components

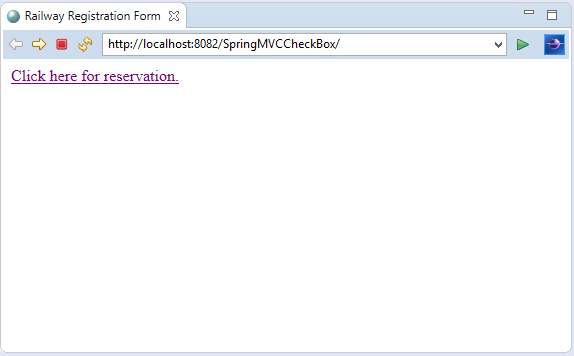
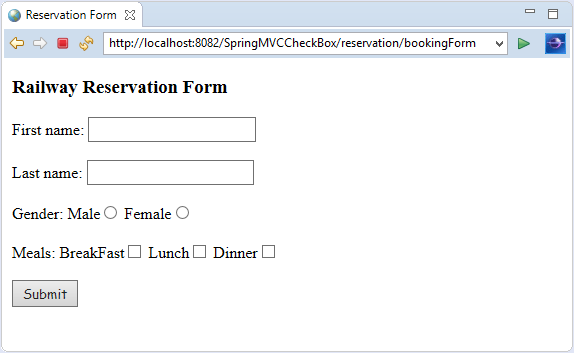
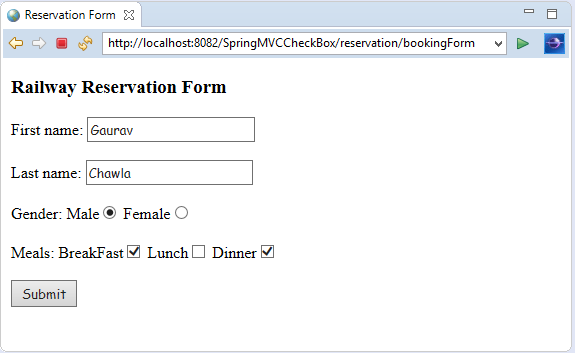
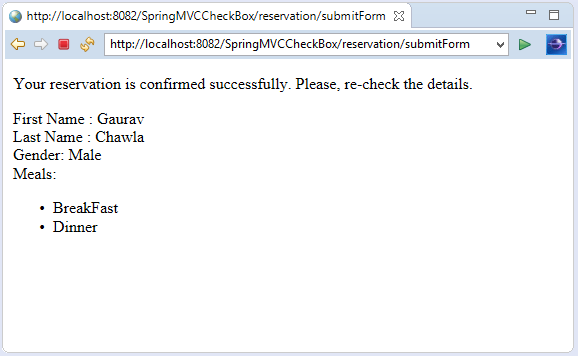
**reservation-page.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. Gender:
15. Male**<form:radiobutton** path="Gender" value="Male"**/>**
16. Female**<form:radiobutton** path="Gender" value="Female"**/>**
17. **<br><br>**
18. Meals:
19. BreakFast**<form:checkbox** path="Food" value="BreakFast"**/>**
20. Lunch**<form:checkbox** path="Food" value="Lunch"**/>**
21. Dinner**<form:checkbox** path="Food" value="Dinner"**/>**
22. **<br><br>**
23. **<input** type="submit" value="Submit" **/>**
24. **</form:form>**
25. **</body>**
26. **</html>**

**confirmation-page.jsp**

1. **<**%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<body>**
5. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
6. First Name : ${reservation.firstName} **<br>**
7. Last Name : ${reservation.lastName} **<br>**
8. Gender: ${reservation.gender}**<br>**
9. Meals:
10. **<ul>**
11. **<c:forEach** var="meal" items="${reservation.food}"**>**
12. **<li>**${meal}**</li>**
13. **</c:forEach>**
14. **</ul>**
15. **</body>**
16. **</html>**

**Output:**

# **Spring MVC Form Drop-Down List**

The Spring MVC form drop-down list contains the list of elements. This tag generates an HTML select element. It allows data binding to the selected element.

### Syntax

1. **<form:select** path="name"**>**

Here are some other tags used to select the options.

### option tag

This tag generates an HTML option tag. Each tag contains a value that can be selected by the user.

### Syntax

1. **<form:option** value="abc" label="xyz"**/>**

### options tag

This tag generates a list of HTML option tags. Each tag contains a list of elements selected by the user.

### Syntax

1. **<form:options** items="${elementList}" itemValue="abc" itemLabel="xyz"**/>**

## **Example of Spring MVC Form Drop-Down List**

### 1. Add dependencies to pom.xml file.

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

**Reservation.java**

1. **package** com.javatpoint;
3. **public** **class** Reservation {
5. **private** String firstName;
6. **private** String lastName;
7. **private** String Gender;
8. **private** String[] Food;
9. **private** String cityFrom;
10. **private** String cityTo;
11. **public** Reservation()
12. {
13. }
14. **public** String getFirstName() {
15. **return** firstName;
16. }
17. **public** **void** setFirstName(String firstName) {
18. **this**.firstName = firstName;
19. }
20. **public** String getLastName() {
21. **return** lastName;
22. }
23. **public** **void** setLastName(String lastName) {
24. **this**.lastName = lastName;
25. }
26. **public** String getGender() {
27. **return** Gender;
28. }
29. **public** **void** setGender(String gender) {
30. Gender = gender;
31. }
32. **public** String[] getFood() {
33. **return** Food;
34. }
35. **public** **void** setFood(String[] food) {
36. Food = food;
37. }
38. **public** String getCityFrom() {
39. **return** cityFrom;
40. }
41. **public** **void** setCityFrom(String cityFrom) {
42. **this**.cityFrom = cityFrom;
43. }
44. **public** String getCityTo() {
45. **return** cityTo;
46. }
47. **public** **void** setCityTo(String cityTo) {
48. **this**.cityTo = cityTo;
49. }
50. }

### 3. Create the controller class

**ReservationController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.ModelAttribute;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @RequestMapping("/reservation")
8. @Controller
9. **public** **class** ReservationController {
10. @RequestMapping("/bookingForm")
11. **public** String bookingForm(Model model)
12. {
13. //create a reservation object
14. Reservation res=**new** Reservation();
15. //provide reservation object to the model
16. model.addAttribute("reservation", res);
17. **return** "reservation-page";
18. }
19. @RequestMapping("/submitForm")
20. **public** String submitForm(@ModelAttribute("reservation") Reservation res)
21. {
22. **return** "confirmation-form";
23. }
24. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create the view components

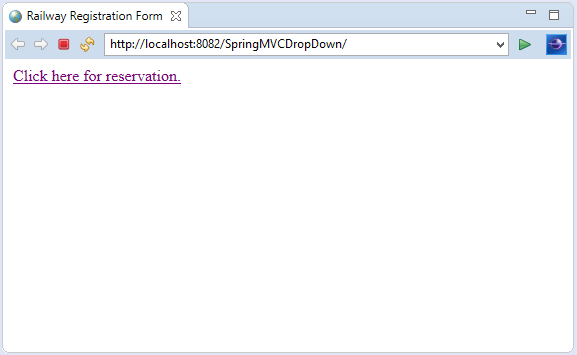
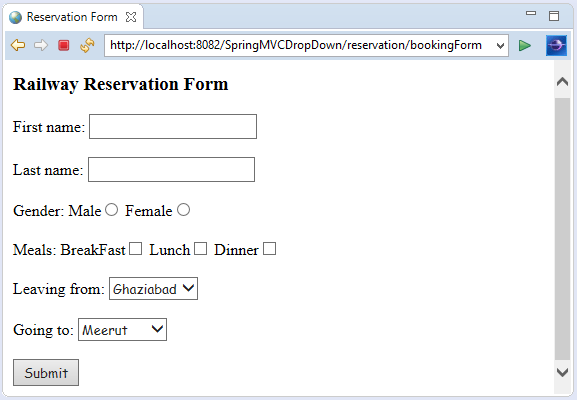
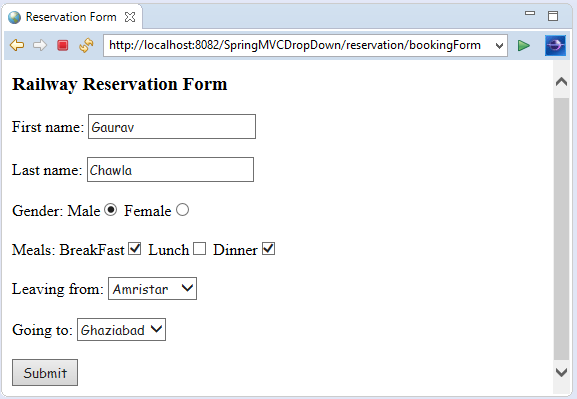
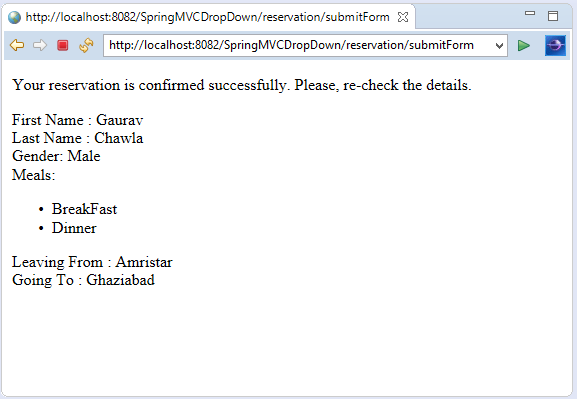
**reservation-page.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. Gender:
15. Male**<form:radiobutton** path="Gender" value="Male"**/>**
16. Female**<form:radiobutton** path="Gender" value="Female"**/>**
17. **<br><br>**
18. Meals:
19. BreakFast**<form:checkbox** path="Food" value="BreakFast"**/>**
20. Lunch**<form:checkbox** path="Food" value="Lunch"**/>**
21. Dinner**<form:checkbox** path="Food" value="Dinner"**/>**
22. **<br><br>**
23. Leaving from: **<form:select** path="cityFrom"**>**
24. **<form:option** value="Ghaziabad" label="Ghaziabad"**/>**
25. **<form:option** value="Modinagar" label="Modinagar"**/>**
26. **<form:option** value="Meerut" label="Meerut"**/>**
27. **<form:option** value="Amristar" label="Amristar"**/>**
28. **</form:select>**
29. **<br><br>**
30. Going to: **<form:select** path="cityTo"**>**
31. **<form:option** value="Ghaziabad" label="Ghaziabad"**/>**
32. **<form:option** value="Modinagar" label="Modinagar"**/>**
33. **<form:option** value="Meerut" label="Meerut"**/>**
34. **<form:option** value="Amristar" label="Amristar"**/>**
35. **</form:select>**
36. **<br><br>**
37. **<input** type="submit" value="Submit" **/>**
38. **</form:form>**
39. **</body>**
40. **</html>**

**confirmation-page.jsp**

1. **<**%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<body>**
5. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
6. First Name : ${reservation.firstName} **<br>**
7. Last Name : ${reservation.lastName} **<br>**
8. Gender: ${reservation.gender}**<br>**
9. Meals:
10. **<ul>**
11. **<c:forEach** var="meal" items="${reservation.food}"**>**
12. **<li>**${meal}**</li>**
13. **</c:forEach>**
14. **</ul>**
15. Leaving From : ${reservation.cityFrom} **<br>**
16. Going To : ${reservation.cityTo}
17. **</body>**
18. **</html>**

**Output:**

# **Spring MVC Form Tag Library**

The Spring MVC form tags are the configurable and reusable building blocks for a web page. These tags provide JSP, an easy way to develop, read and maintain.

The Spring MVC form tags can be seen as data binding-aware tags that can automatically set data to Java object/bean and also retrieve from it. Here, each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and easy to use.

## **Configuration of Spring MVC Form Tag**

The form tag library comes under the spring-webmvc.jar. To enable the support for form tag library, it is required to reference some configuration. So, add the following directive at the beginning of the JSP page:

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**

## **List of Spring MVC Form Tags**

Let's see some of the frequently used Spring MVC form tags.

|  |  |
| --- | --- |
| **Form Tag** | **Description** |
| form:form | It is a container tag that contains all other form tags. |
| form:input | This tag is used to generate the text field. |
| form:radiobutton | This tag is used to generate the radio buttons. |
| form:checkbox | This tag is used to generate the checkboxes. |
| form:password | This tag is used to generate the password input field. |
| form:select | This tag is used to generate the drop-down list. |
| form:textarea | This tag is used to generate the multi-line text field. |
| form:hidden | This tag is used to generate the hidden input field. |

## **The form tag**

The Spring MVC form tag is a container tag. It is a parent tag that contains all the other tags of the tag library. This tag generates an HTML form tag and exposes a binding path to the inner tags for binding.

### Syntax

1. **<form:form** action="nextFormPath" modelAttribute=?abc**?>**

In the next section, we will learn more about each form tag.

# **Spring MVC Form Text Field**

The Spring MVC form text field tag generates an HTML input tag using the bound value. By default, the type of the input tag is text.

### Syntax

1. **<form:input** path="name" **/>**

Here, **path** attribute binds the form field to the bean property.

The Spring MVC form tag library also provides other input types such as email, date, tel, etc.

### For email:

1. **<form:input** type=?email? path="email" **/>**

### For date:

1. **<form:input** type=?date? path="date" **/>**

## **Example of Spring MVC Form Text Field**

Let's see an example to create a railway reservation form using form tag library.

### 1. Add dependencies to pom.xml file.

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

Here, the bean class contains the variables (along setter and getter methods) corresponding to the input field exist in the form.

**Reservation.java**

1. **package** com.javatpoint;
3. **public** **class** Reservation {
5. **private** String firstName;
6. **private** String lastName;
8. **public** Reservation()
9. {
10. }
11. **public** String getFirstName() {
12. **return** firstName;
13. }
14. **public** **void** setFirstName(String firstName) {
15. **this**.firstName = firstName;
16. }
17. **public** String getLastName() {
18. **return** lastName;
19. }
20. **public** **void** setLastName(String lastName) {
21. **this**.lastName = lastName;
22. }
23. }

### 3. Create the controller class

**ReservationController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.ModelAttribute;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @RequestMapping("/reservation")
8. @Controller
9. **public** **class** ReservationController {
10. @RequestMapping("/bookingForm")
11. **public** String bookingForm(Model model)
12. {
13. //create a reservation object
14. Reservation res=**new** Reservation();
15. //provide reservation object to the model
16. model.addAttribute("reservation", res);
17. **return** "reservation-page";
18. }
19. @RequestMapping("/submitForm")
20. // @ModelAttribute binds form data to the object
21. **public** String submitForm(@ModelAttribute("reservation") Reservation res)
22. {
23. **return** "confirmation-form";
24. }
25. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create other view components

**reservation-page.jsp**

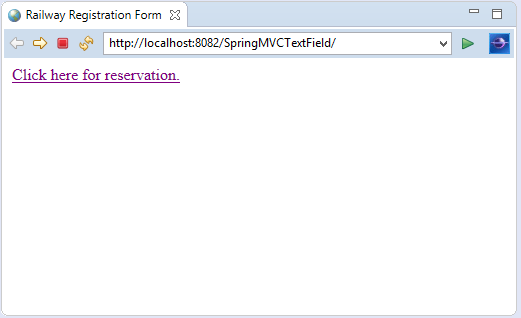
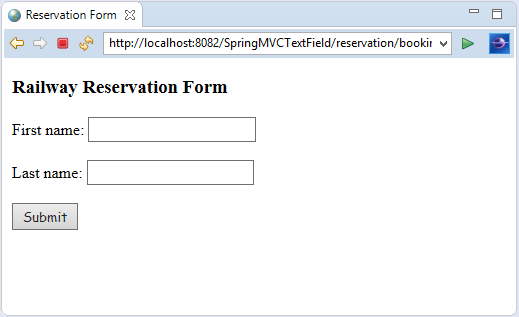
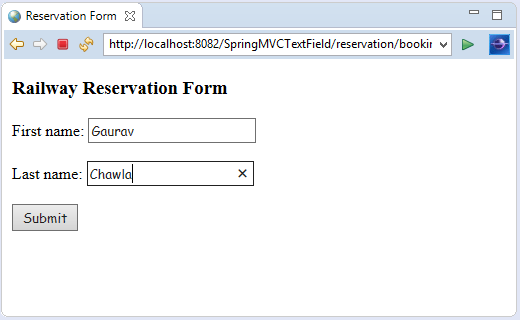
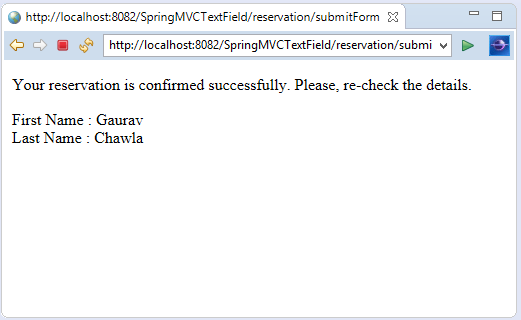
1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. **<input** type="submit" value="Submit" **/>**
15. **</form:form>**
16. **</body>**
17. **</html>**

### Note - The value passed with the @ModelAttribute annotation should be the same to the modelAttribute value present in the view page.

**confirmation-page.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<body>**
4. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
5. First Name : ${reservation.firstName} **<br>**
6. Last Name : ${reservation.lastName}
7. **</body>**
8. **</html>**

**Output:**

# **Spring MVC Form Radio Button**

The Spring MVC form radio button allows us to choose only one option at a time. This tag renders an HTML input tag of type radio.

### Syntax

1. **<form:radiobutton** path="abc" value="xyz"**/>**

Apart from radio button tag, Spring MVC form tag library also contains **radiobuttons** tag. This tag renders multiple HTML input tags with type radio.

1. **<form:radiobuttons** path="abc" items="${xyz}"**/>**

## **Example of Spring MVC Form Radio Button**

### 1. Add dependencies to pom.xml

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

**Reservation.java**

1. package com.javatpoint;
3. public class Reservation {
5. private String firstName;
6. private String lastName;
7. private String Gender;
9. public Reservation()
10. {
11. }
12. public String getFirstName() {
13. return firstName;
14. }
15. public void setFirstName(String firstName) {
16. this.firstName = firstName;
17. }
18. public String getLastName() {
19. return lastName;
20. }
21. public void setLastName(String lastName) {
22. this.lastName = lastName;
23. }
24. public String getGender() {
25. return Gender;
26. }
27. public void setGender(String gender) {
28. Gender = gender;
29. }
30. }

### 3. Create the controller class

**ReservationController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.ModelAttribute;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @RequestMapping("/reservation")
8. @Controller
9. **public** **class** ReservationController {
10. @RequestMapping("/bookingForm")
11. **public** String bookingForm(Model model)
12. {
13. //create a reservation object
14. Reservation res=**new** Reservation();
15. //provide reservation object to the model
16. model.addAttribute("reservation", res);
17. **return** "reservation-page";
18. }
19. @RequestMapping("/submitForm")
20. **public** String submitForm(@ModelAttribute("reservation") Reservation res)
21. {
22. **return** "confirmation-form";
23. }
24. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create the view components

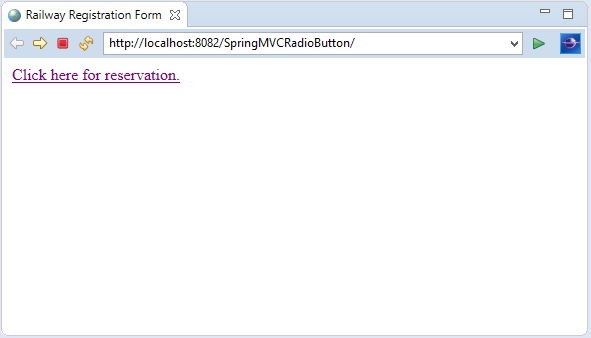
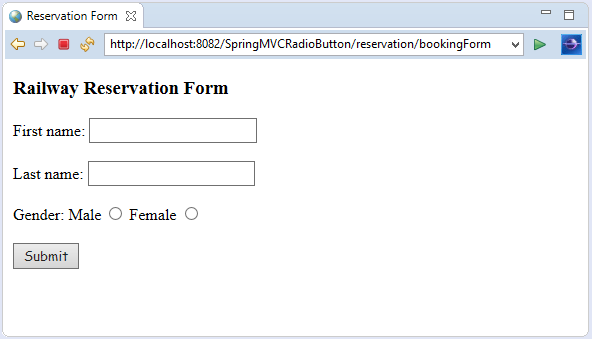
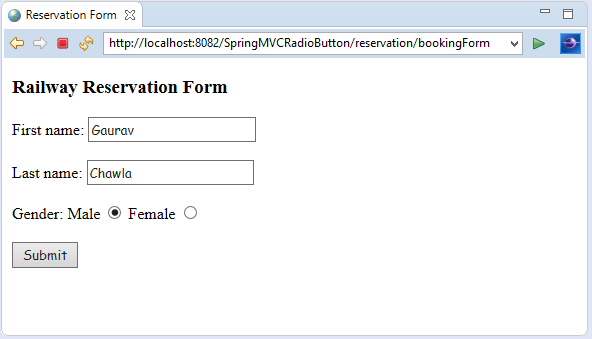
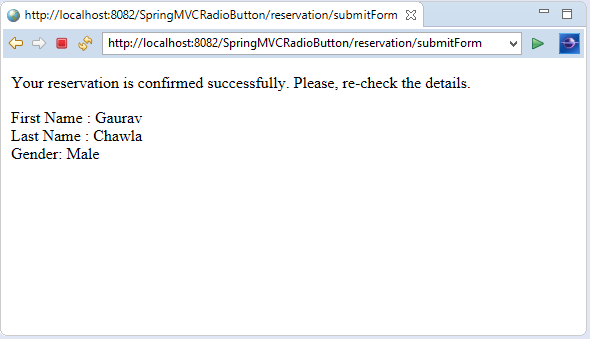
**reservation-page.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. Gender:
15. Male **<form:radiobutton** path="Gender" value="Male"**/>**
16. Female **<form:radiobutton** path="Gender" value="Female"**/>**
17. **<br><br>**
18. **<input** type="submit" value="Submit" **/>**
19. **</form:form>**
20. **</body>**
21. **</html>**

**confirmation-page.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<body>**
4. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
5. First Name : ${reservation.firstName} **<br>**
6. Last Name : ${reservation.lastName} **<br>**
7. Gender: ${reservation.gender}
8. **</body>**
9. **</html>**

**Output:**

# **Spring MVC Form Checkbox**

The Spring MVC form checkbox facilitates to choose multiple options at the same time. This tag renders an HTML input tag of type checkbox.

### Syntax

1. **<form:checkbox** path="abc" value="element"**/>**

Apart from checkbox tag, Spring MVC form tag library also contains **checkboxes** tag. This tag renders multiple HTML input tags with type checkbox. This tag is used only if you don't want to list all the elements in the view page. In such a case, you can provide the elements at runtime and pass it to the tag. As the user can select multiple options, you need to pass the elements of Array, a List or a Map type.

### Syntax

1. **<form:checkboxes** path="abc" items="${object.elementList}"**/>**

## **Example of Spring MVC Form Check Box**

### 1. Add dependencies to pom.xml file.

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

**Reservation.java**

1. **package** com.javatpoint;
3. **public** **class** Reservation {
5. **private** String firstName;
6. **private** String lastName;
7. **private** String Gender;
8. **private** String[] Food;
9. **public** Reservation()
10. {
11. }
12. **public** String getFirstName() {
13. **return** firstName;
14. }
15. **public** **void** setFirstName(String firstName) {
16. **this**.firstName = firstName;
17. }
18. **public** String getLastName() {
19. **return** lastName;
20. }
21. **public** **void** setLastName(String lastName) {
22. **this**.lastName = lastName;
23. }
24. **public** String getGender() {
25. **return** Gender;
26. }
27. **public** **void** setGender(String gender) {
28. Gender = gender;
29. }
30. **public** String[] getFood() {
31. **return** Food;
32. }
33. **public** **void** setFood(String[] food) {
34. Food = food;
35. }
36. }

### 3. Create the controller class

**ReservationController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.ModelAttribute;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @RequestMapping("/reservation")
8. @Controller
9. **public** **class** ReservationController {
10. @RequestMapping("/bookingForm")
11. **public** String bookingForm(Model model)
12. {
13. //create a reservation object
14. Reservation res=**new** Reservation();
15. //provide reservation object to the model
16. model.addAttribute("reservation", res);
17. **return** "reservation-page";
18. }
19. @RequestMapping("/submitForm")
20. **public** String submitForm(@ModelAttribute("reservation") Reservation res)
21. {
22. **return** "confirmation-form";
23. }
24. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create the view components

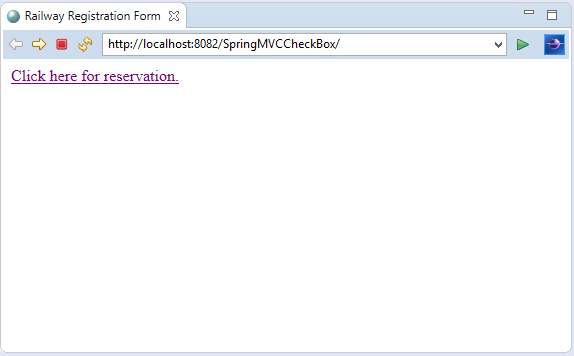
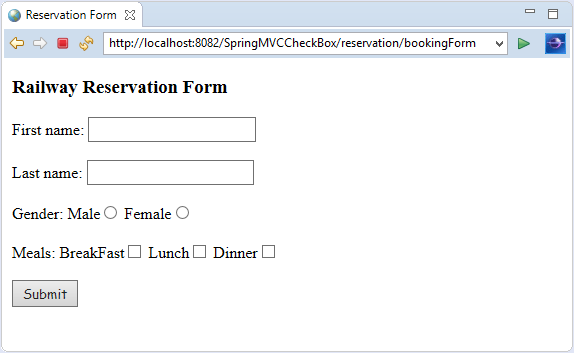
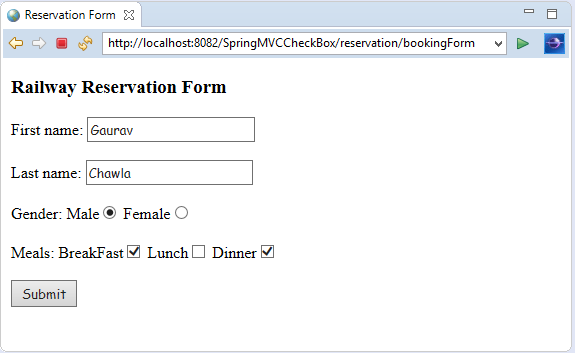
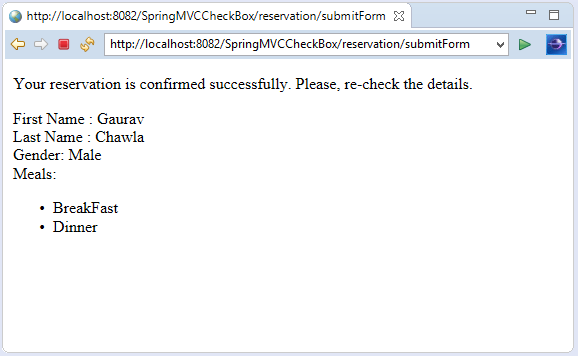
**reservation-page.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. Gender:
15. Male**<form:radiobutton** path="Gender" value="Male"**/>**
16. Female**<form:radiobutton** path="Gender" value="Female"**/>**
17. **<br><br>**
18. Meals:
19. BreakFast**<form:checkbox** path="Food" value="BreakFast"**/>**
20. Lunch**<form:checkbox** path="Food" value="Lunch"**/>**
21. Dinner**<form:checkbox** path="Food" value="Dinner"**/>**
22. **<br><br>**
23. **<input** type="submit" value="Submit" **/>**
24. **</form:form>**
25. **</body>**
26. **</html>**

**confirmation-page.jsp**

1. **<**%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<body>**
5. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
6. First Name : ${reservation.firstName} **<br>**
7. Last Name : ${reservation.lastName} **<br>**
8. Gender: ${reservation.gender}**<br>**
9. Meals:
10. **<ul>**
11. **<c:forEach** var="meal" items="${reservation.food}"**>**
12. **<li>**${meal}**</li>**
13. **</c:forEach>**
14. **</ul>**
15. **</body>**
16. **</html>**

**Output:**

# **Spring MVC Form Drop-Down List**

The Spring MVC form drop-down list contains the list of elements. This tag generates an HTML select element. It allows data binding to the selected element.

### Syntax

1. **<form:select** path="name"**>**

Here are some other tags used to select the options.

### option tag

This tag generates an HTML option tag. Each tag contains a value that can be selected by the user.

### Syntax

1. **<form:option** value="abc" label="xyz"**/>**

### options tag

This tag generates a list of HTML option tags. Each tag contains a list of elements selected by the user.

### Syntax

1. **<form:options** items="${elementList}" itemValue="abc" itemLabel="xyz"**/>**

## **Example of Spring MVC Form Drop-Down List**

### 1. Add dependencies to pom.xml file.

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
9. **<dependency>**
10. **<groupId>**javax.servlet**</groupId>**
11. **<artifactId>**servlet-api**</artifactId>**
12. **<version>**3.0-alpha-1**</version>**
13. **</dependency>**
15. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
16. **<dependency>**
17. **<groupId>**javax.servlet**</groupId>**
18. **<artifactId>**jstl**</artifactId>**
19. **<version>**1.2**</version>**
20. **</dependency>**
21. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
22. **<dependency>**
23. **<groupId>**org.apache.tomcat**</groupId>**
24. **<artifactId>**tomcat-jasper**</artifactId>**
25. **<version>**9.0.12**</version>**
26. **</dependency>**

### 2. Create the bean class

**Reservation.java**

1. **package** com.javatpoint;
3. **public** **class** Reservation {
5. **private** String firstName;
6. **private** String lastName;
7. **private** String Gender;
8. **private** String[] Food;
9. **private** String cityFrom;
10. **private** String cityTo;
11. **public** Reservation()
12. {
13. }
14. **public** String getFirstName() {
15. **return** firstName;
16. }
17. **public** **void** setFirstName(String firstName) {
18. **this**.firstName = firstName;
19. }
20. **public** String getLastName() {
21. **return** lastName;
22. }
23. **public** **void** setLastName(String lastName) {
24. **this**.lastName = lastName;
25. }
26. **public** String getGender() {
27. **return** Gender;
28. }
29. **public** **void** setGender(String gender) {
30. Gender = gender;
31. }
32. **public** String[] getFood() {
33. **return** Food;
34. }
35. **public** **void** setFood(String[] food) {
36. Food = food;
37. }
38. **public** String getCityFrom() {
39. **return** cityFrom;
40. }
41. **public** **void** setCityFrom(String cityFrom) {
42. **this**.cityFrom = cityFrom;
43. }
44. **public** String getCityTo() {
45. **return** cityTo;
46. }
47. **public** **void** setCityTo(String cityTo) {
48. **this**.cityTo = cityTo;
49. }
50. }

### 3. Create the controller class

**ReservationController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.ModelAttribute;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
7. @RequestMapping("/reservation")
8. @Controller
9. **public** **class** ReservationController {
10. @RequestMapping("/bookingForm")
11. **public** String bookingForm(Model model)
12. {
13. //create a reservation object
14. Reservation res=**new** Reservation();
15. //provide reservation object to the model
16. model.addAttribute("reservation", res);
17. **return** "reservation-page";
18. }
19. @RequestMapping("/submitForm")
20. **public** String submitForm(@ModelAttribute("reservation") Reservation res)
21. {
22. **return** "confirmation-form";
23. }
24. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<title>**Railway Registration Form**</title>**
5. **</head>**
6. **<body>**
7. **<a** href="reservation/bookingForm"**>**Click here for reservation.**</a>**
8. **</body>**
9. **</html>**

### 7. Create the view components

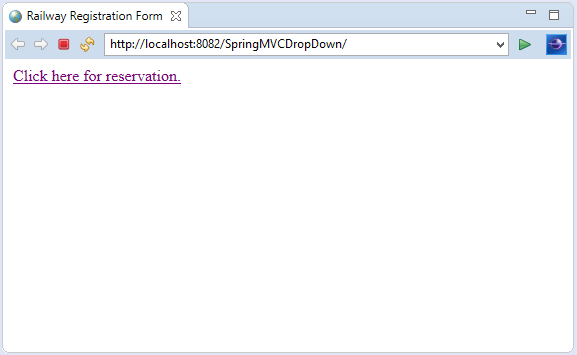
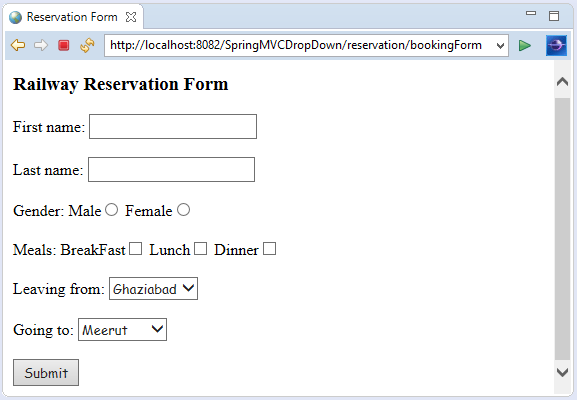
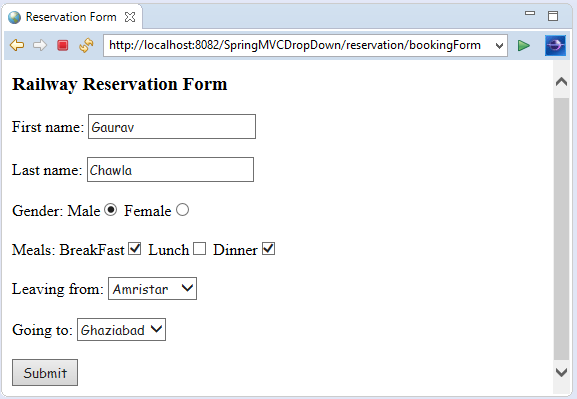
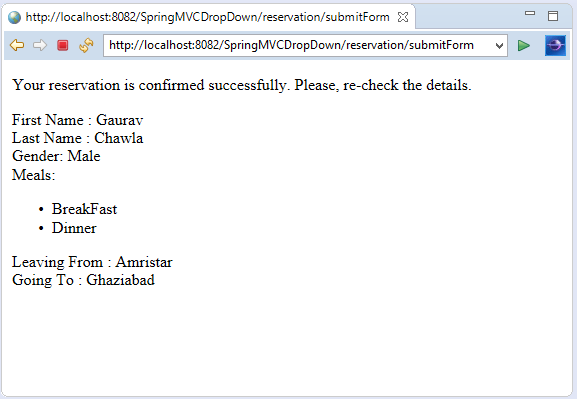
**reservation-page.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<head>**
5. **<title>**Reservation Form**</title>**
6. **</head>**
7. **<h3>**Railway Reservation Form**</h3>**
8. **<body>**
9. **<form:form** action="submitForm" modelAttribute="reservation"**>**
10. First name: **<form:input** path="firstName" **/>**
11. **<br><br>**
12. Last name: **<form:input** path="lastName" **/>**
13. **<br><br>**
14. Gender:
15. Male**<form:radiobutton** path="Gender" value="Male"**/>**
16. Female**<form:radiobutton** path="Gender" value="Female"**/>**
17. **<br><br>**
18. Meals:
19. BreakFast**<form:checkbox** path="Food" value="BreakFast"**/>**
20. Lunch**<form:checkbox** path="Food" value="Lunch"**/>**
21. Dinner**<form:checkbox** path="Food" value="Dinner"**/>**
22. **<br><br>**
23. Leaving from: **<form:select** path="cityFrom"**>**
24. **<form:option** value="Ghaziabad" label="Ghaziabad"**/>**
25. **<form:option** value="Modinagar" label="Modinagar"**/>**
26. **<form:option** value="Meerut" label="Meerut"**/>**
27. **<form:option** value="Amristar" label="Amristar"**/>**
28. **</form:select>**
29. **<br><br>**
30. Going to: **<form:select** path="cityTo"**>**
31. **<form:option** value="Ghaziabad" label="Ghaziabad"**/>**
32. **<form:option** value="Modinagar" label="Modinagar"**/>**
33. **<form:option** value="Meerut" label="Meerut"**/>**
34. **<form:option** value="Amristar" label="Amristar"**/>**
35. **</form:select>**
36. **<br><br>**
37. **<input** type="submit" value="Submit" **/>**
38. **</form:form>**
39. **</body>**
40. **</html>**

**confirmation-page.jsp**

1. **<**%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<body>**
5. **<p>**Your reservation is confirmed successfully. Please, re-check the details.**</p>**
6. First Name : ${reservation.firstName} **<br>**
7. Last Name : ${reservation.lastName} **<br>**
8. Gender: ${reservation.gender}**<br>**
9. Meals:
10. **<ul>**
11. **<c:forEach** var="meal" items="${reservation.food}"**>**
12. **<li>**${meal}**</li>**
13. **</c:forEach>**
14. **</ul>**
15. Leaving From : ${reservation.cityFrom} **<br>**
16. Going To : ${reservation.cityTo}
17. **</body>**
18. **</html>**

**Output:**

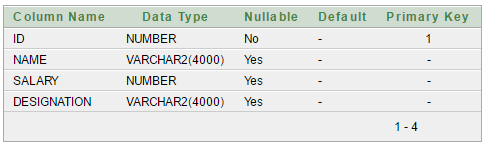
# **Spring MVC CRUD Example**

CRUD (Create, Read, Update and Delete) application is the most important application for creating any project. It provides an idea to develop a large project. In spring MVC, we can develop a simple CRUD application.

Here, we are using **JdbcTemplate** for database interaction.

## **Create a table**

Here, we are using emp99 table present in the MySQL database. It has 4 fields: id, name, salary, and designation. Here, id is auto incremented which is generated by the sequence.



## **Spring MVC CRUD Example**

### 1. Add dependencies to pom.xml file.

**pom.xml**

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
9. **<dependency>**
10. **<groupId>**org.apache.tomcat**</groupId>**
11. **<artifactId>**tomcat-jasper**</artifactId>**
12. **<version>**9.0.12**</version>**
13. **</dependency>**
14. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
15. **<dependency>**
16. **<groupId>**javax.servlet**</groupId>**
17. **<artifactId>**servlet-api**</artifactId>**
18. **<version>**3.0-alpha-1**</version>**
19. **</dependency>**
20. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
21. **<dependency>**
22. **<groupId>**javax.servlet**</groupId>**
23. **<artifactId>**jstl**</artifactId>**
24. **<version>**1.2**</version>**
25. **</dependency>**
26. <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
27. **<dependency>**
28. **<groupId>**mysql**</groupId>**
29. **<artifactId>**mysql-connector-java**</artifactId>**
30. **<version>**8.0.11**</version>**
31. **</dependency>**
32. <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
33. **<dependency>**
34. **<groupId>**org.springframework**</groupId>**
35. **<artifactId>**spring-jdbc**</artifactId>**
36. **<version>**5.1.1.RELEASE**</version>**
37. **</dependency>**

### 2. Create the bean class

Here, the bean class contains the variables (along setter and getter methods) corresponding to the fields exist in the database.

**Emp.java**

1. **package** com.javatpoint.beans;
3. **public** **class** Emp {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. **private** String designation;
9. **public** **int** getId() {
10. **return** id;
11. }
12. **public** **void** setId(**int** id) {
13. **this**.id = id;
14. }
15. **public** String getName() {
16. **return** name;
17. }
18. **public** **void** setName(String name) {
19. **this**.name = name;
20. }
21. **public** **float** getSalary() {
22. **return** salary;
23. }
24. **public** **void** setSalary(**float** salary) {
25. **this**.salary = salary;
26. }
27. **public** String getDesignation() {
28. **return** designation;
29. }
30. **public** **void** setDesignation(String designation) {
31. **this**.designation = designation;
32. }
34. }

### 3. Create the controller class

**EmpController.java**

1. **package** com.javatpoint.controllers;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.stereotype.Controller;
5. **import** org.springframework.ui.Model;
6. **import** org.springframework.web.bind.annotation.ModelAttribute;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.RequestMapping;
9. **import** org.springframework.web.bind.annotation.RequestMethod;
10. **import** com.javatpoint.beans.Emp;
11. **import** com.javatpoint.dao.EmpDao;
12. @Controller
13. **public** **class** EmpController {
14. @Autowired
15. EmpDao dao;//will inject dao from XML file
17. /\*It displays a form to input data, here "command" is a reserved request attribute
18. \*which is used to display object data into form
19. \*/
20. @RequestMapping("/empform")
21. **public** String showform(Model m){
22. m.addAttribute("command", **new** Emp());
23. **return** "empform";
24. }
25. /\*It saves object into database. The @ModelAttribute puts request data
26. \*  into model object. You need to mention RequestMethod.POST method
27. \*  because default request is GET\*/
28. @RequestMapping(value="/save",method = RequestMethod.POST)
29. **public** String save(@ModelAttribute("emp") Emp emp){
30. dao.save(emp);
31. **return** "redirect:/viewemp";//will redirect to viewemp request mapping
32. }
33. /\* It provides list of employees in model object \*/
34. @RequestMapping("/viewemp")
35. **public** String viewemp(Model m){
36. List<Emp> list=dao.getEmployees();
37. m.addAttribute("list",list);
38. **return** "viewemp";
39. }
40. /\* It displays object data into form for the given id.
41. \* The @PathVariable puts URL data into variable.\*/
42. @RequestMapping(value="/editemp/{id}")
43. **public** String edit(@PathVariable **int** id, Model m){
44. Emp emp=dao.getEmpById(id);
45. m.addAttribute("command",emp);
46. **return** "empeditform";
47. }
48. /\* It updates model object. \*/
49. @RequestMapping(value="/editsave",method = RequestMethod.POST)
50. **public** String editsave(@ModelAttribute("emp") Emp emp){
51. dao.update(emp);
52. **return** "redirect:/viewemp";
53. }
54. /\* It deletes record for the given id in URL and redirects to /viewemp \*/
55. @RequestMapping(value="/deleteemp/{id}",method = RequestMethod.GET)
56. **public** String delete(@PathVariable **int** id){
57. dao.delete(id);
58. **return** "redirect:/viewemp";
59. }
60. }

### 4. Create the DAO class

Let's create a DAO class to access the required data from the database.

**EmpDao.java**

1. **package** com.javatpoint.dao;
2. **import** java.sql.ResultSet;
3. **import** java.sql.SQLException;
4. **import** java.util.List;
5. **import** org.springframework.jdbc.core.BeanPropertyRowMapper;
6. **import** org.springframework.jdbc.core.JdbcTemplate;
7. **import** org.springframework.jdbc.core.RowMapper;
8. **import** com.javatpoint.beans.Emp;
10. **public** **class** EmpDao {
11. JdbcTemplate template;
13. **public** **void** setTemplate(JdbcTemplate template) {
14. **this**.template = template;
15. }
16. **public** **int** save(Emp p){
17. String sql="insert into Emp99(name,salary,designation) values('"+p.getName()+"',"+p.getSalary()+",'"+p.getDesignation()+"')";
18. **return** template.update(sql);
19. }
20. **public** **int** update(Emp p){
21. String sql="update Emp99 set name='"+p.getName()+"', salary="+p.getSalary()+",designation='"+p.getDesignation()+"' where id="+p.getId()+"";
22. **return** template.update(sql);
23. }
24. **public** **int** delete(**int** id){
25. String sql="delete from Emp99 where id="+id+"";
26. **return** template.update(sql);
27. }
28. **public** Emp getEmpById(**int** id){
29. String sql="select \* from Emp99 where id=?";
30. **return** template.queryForObject(sql, **new** Object[]{id},**new** BeanPropertyRowMapper<Emp>(Emp.**class**));
31. }
32. **public** List<Emp> getEmployees(){
33. **return** template.query("select \* from Emp99",**new** RowMapper<Emp>(){
34. **public** Emp mapRow(ResultSet rs, **int** row) **throws** SQLException {
35. Emp e=**new** Emp();
36. e.setId(rs.getInt(1));
37. e.setName(rs.getString(2));
38. e.setSalary(rs.getFloat(3));
39. e.setDesignation(rs.getString(4));
40. **return** e;
41. }
42. });
43. }
44. }

### 5. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 6. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. **<context:component-scan** base-package="com.javatpoint.controllers"**></context:component-scan>**
15. **<bean** class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
16. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
17. **<property** name="suffix" value=".jsp"**></property>**
18. **</bean>**
20. **<bean** id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>**
21. **<property** name="driverClassName" value="com.mysql.jdbc.Driver"**></property>**
22. **<property** name="url" value="jdbc:mysql://localhost:3306/test"**></property>**
23. **<property** name="username" value=""**></property>**
24. **<property** name="password" value=""**></property>**
25. **</bean>**
27. **<bean** id="jt" class="org.springframework.jdbc.core.JdbcTemplate"**>**
28. **<property** name="dataSource" ref="ds"**></property>**
29. **</bean>**
31. **<bean** id="dao" class="com.javatpoint.dao.EmpDao"**>**
32. **<property** name="template" ref="jt"**></property>**
33. **</bean>**
34. **</beans>**

### 7. Create the requested page

**index.jsp**

1. **<a** href="empform"**>**Add Employee**</a>**
2. **<a** href="viewemp"**>**View Employees**</a>**

### 8. Create the other view components

**empform.jsp**

1. **<**%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%**>**
2. **<**%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%**>**
4. **<h1>**Add New Employee**</h1>**
5. **<form:form** method="post" action="save"**>**
6. **<table** **>**
7. **<tr>**
8. **<td>**Name : **</td>**
9. **<td><form:input** path="name"  **/></td>**
10. **</tr>**
11. **<tr>**
12. **<td>**Salary :**</td>**
13. **<td><form:input** path="salary" **/></td>**
14. **</tr>**
15. **<tr>**
16. **<td>**Designation :**</td>**
17. **<td><form:input** path="designation" **/></td>**
18. **</tr>**
19. **<tr>**
20. **<td>** **</td>**
21. **<td><input** type="submit" value="Save" **/></td>**
22. **</tr>**
23. **</table>**
24. **</form:form>**

**empeditform.jsp**

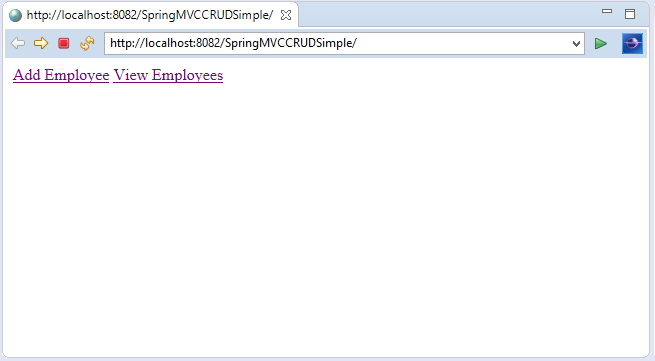
Here "/SpringMVCCRUDSimple" is the project name, change this if you have different project name. For live application, you can provide full URL.

1. **<**%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%**>**
2. **<**%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%**>**
4. **<h1>**Edit Employee**</h1>**
5. **<form:form** method="POST" action="/SpringMVCCRUDSimple/editsave"**>**
6. **<table** **>**
7. **<tr>**
8. **<td></td>**
9. **<td><form:hidden**  path="id" **/></td>**
10. **</tr>**
11. **<tr>**
12. **<td>**Name : **</td>**
13. **<td><form:input** path="name"  **/></td>**
14. **</tr>**
15. **<tr>**
16. **<td>**Salary :**</td>**
17. **<td><form:input** path="salary" **/></td>**
18. **</tr>**
19. **<tr>**
20. **<td>**Designation :**</td>**
21. **<td><form:input** path="designation" **/></td>**
22. **</tr>**
24. **<tr>**
25. **<td>** **</td>**
26. **<td><input** type="submit" value="Edit Save" **/></td>**
27. **</tr>**
28. **</table>**
29. **</form:form>**

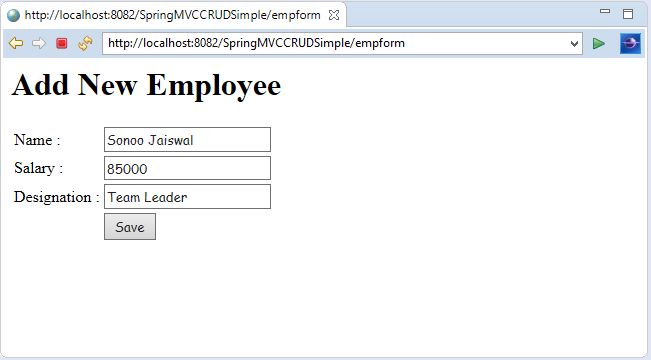
**viewemp.jsp**

1. **<**%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%**>**
2. **<**%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%**>**
4. **<h1>**Employees List**</h1>**
5. **<table** border="2" width="70%" cellpadding="2"**>**
6. **<tr><th>**Id**</th><th>**Name**</th><th>**Salary**</th><th>**Designation**</th><th>**Edit**</th><th>**Delete**</th></tr>**
7. **<c:forEach** var="emp" items="${list}"**>**
8. **<tr>**
9. **<td>**${emp.id}**</td>**
10. **<td>**${emp.name}**</td>**
11. **<td>**${emp.salary}**</td>**
12. **<td>**${emp.designation}**</td>**
13. **<td><a** href="editemp/${emp.id}"**>**Edit**</a></td>**
14. **<td><a** href="deleteemp/${emp.id}"**>**Delete**</a></td>**
15. **</tr>**
16. **</c:forEach>**
17. **</table>**
18. **<br/>**
19. **<a** href="empform"**>**Add New Employee**</a>**

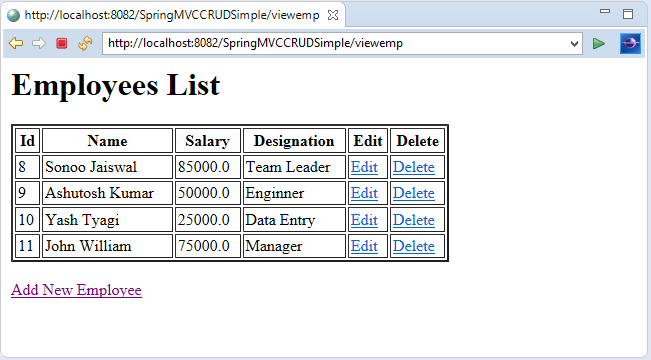
**Output:**



On clicking **Add Employee**, you will see the following form.



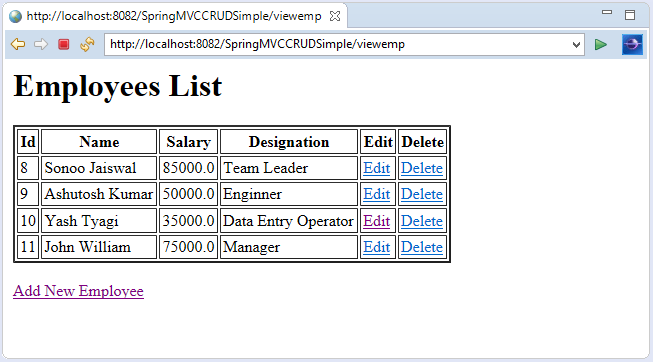
Fill the form and **click Save** to add the entry into the database.



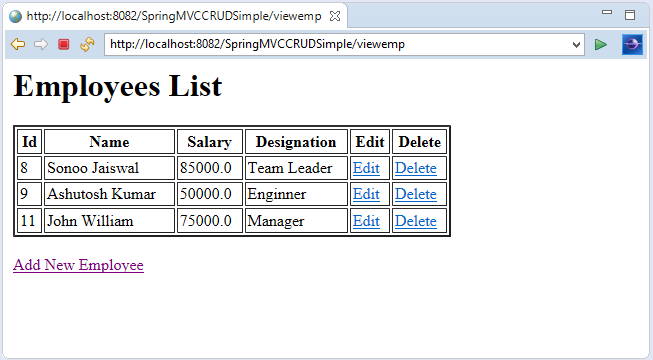
Now, click **Edit** to make some changes in the provided data.



Now, click **Edit Save** to add the entry with changes into the database.



Now, click **Delete** to delete the entry from the database.



# **Spring MVC Pagination Example**

Pagination is used to display a large number of records in different parts. In such case, we display 10, 20 or 50 records in one page. For remaining records, we provide links.

We can simply create pagination example in Spring MVC. In this pagination example, we are using MySQL database to fetch records.

## **Create a table or import SQL file**

Here, we have created "emp" table in "test" database. The emp table has three fields: id, name, and salary. Either create a table and insert records manually or import our SQL file.

## **Spring MVC Pagination Example**

### 1. Add dependencies to pom.xml file.

**pom.xml**

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
8. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
9. **<dependency>**
10. **<groupId>**org.apache.tomcat**</groupId>**
11. **<artifactId>**tomcat-jasper**</artifactId>**
12. **<version>**9.0.12**</version>**
13. **</dependency>**
14. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
15. **<dependency>**
16. **<groupId>**javax.servlet**</groupId>**
17. **<artifactId>**servlet-api**</artifactId>**
18. **<version>**3.0-alpha-1**</version>**
19. **</dependency>**
20. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
21. **<dependency>**
22. **<groupId>**javax.servlet**</groupId>**
23. **<artifactId>**jstl**</artifactId>**
24. **<version>**1.2**</version>**
25. **</dependency>**
26. <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
27. **<dependency>**
28. **<groupId>**mysql**</groupId>**
29. **<artifactId>**mysql-connector-java**</artifactId>**
30. **<version>**8.0.11**</version>**
31. **</dependency>**
32. <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
33. **<dependency>**
34. **<groupId>**org.springframework**</groupId>**
35. **<artifactId>**spring-jdbc**</artifactId>**
36. **<version>**5.1.1.RELEASE**</version>**
37. **</dependency>**

### 2. Create the bean class

Here, the bean class contains the variables (along setter and getter methods) corresponding to the fields exist in the database.

**Emp.java**

1. **package** com.javatpoint.beans;
3. **public** **class** Emp {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
8. **public** **int** getId() {
9. **return** id;
10. }
11. **public** **void** setId(**int** id) {
12. **this**.id = id;
13. }
14. **public** String getName() {
15. **return** name;
16. }
17. **public** **void** setName(String name) {
18. **this**.name = name;
19. }
20. **public** **float** getSalary() {
21. **return** salary;
22. }
23. **public** **void** setSalary(**float** salary) {
24. **this**.salary = salary;
25. }
27. }

### 3. Create the controller class

In Controller class, the **@PathVariable** annotation bounds the method parameter with a temporary URL. For example:

1. @RequestMapping(value="/viewemp/{pageid}")

Here, {} bracket contains the temporary value.

**EmpController.java**

1. **package** com.javatpoint.controllers;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.stereotype.Controller;
5. **import** org.springframework.ui.Model;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.RequestMapping;
8. **import** com.javatpoint.beans.Emp;
9. **import** com.javatpoint.dao.EmpDao;
10. @Controller
11. **public** **class** EmpController {
12. @Autowired
13. EmpDao dao;
15. @RequestMapping(value="/viewemp/{pageid}")
16. **public** String edit(@PathVariable **int** pageid,Model m){
17. **int** total=5;
18. **if**(pageid==1){}
19. **else**{
20. pageid=(pageid-1)\*total+1;
21. }
22. System.out.println(pageid);
23. List<Emp> list=dao.getEmployeesByPage(pageid,total);
24. m.addAttribute("msg", list);
25. **return** "viewemp";
26. }
27. }

### 4. Create the DAO class

Let's create a DAO class to access the required data from the database.

**EmpDao.java**

1. **package** com.javatpoint.dao;
3. **import** java.sql.ResultSet;
4. **import** java.sql.SQLException;
5. **import** java.util.List;
6. **import** org.springframework.jdbc.core.JdbcTemplate;
7. **import** org.springframework.jdbc.core.RowMapper;
8. **import** com.javatpoint.beans.Emp;
10. **public** **class** EmpDao {
11. JdbcTemplate template;
13. **public** **void** setTemplate(JdbcTemplate template) {
14. **this**.template = template;
15. }
17. **public** List<Emp> getEmployeesByPage(**int** pageid,**int** total){
18. String sql="select \* from emp limit "+(pageid-1)+","+total;
19. **return** template.query(sql,**new** RowMapper<Emp>(){
20. **public** Emp mapRow(ResultSet rs, **int** row) **throws** SQLException {
21. Emp e=**new** Emp();
22. e.setId(rs.getInt(1));
23. e.setName(rs.getString(2));
24. e.setSalary(rs.getFloat(3));
25. **return** e;
26. }
27. });
28. }
29. }

### 5. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 6. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. **<context:component-scan** base-package="com.javatpoint.controllers"**></context:component-scan>**
15. **<bean** class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
16. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
17. **<property** name="suffix" value=".jsp"**></property>**
18. **</bean>**
20. **<bean** id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>**
21. **<property** name="driverClassName" value="com.mysql.jdbc.Driver"**></property>**
22. **<property** name="url" value="jdbc:mysql://localhost:3306/test"**></property>**
23. **<property** name="username" value=""**></property>**
24. **<property** name="password" value=""**></property>**
25. **</bean>**
27. **<bean** id="jt" class="org.springframework.jdbc.core.JdbcTemplate"**>**
28. **<property** name="dataSource" ref="ds"**></property>**
29. **</bean>**
31. **<bean** id="dao" class="com.javatpoint.dao.EmpDao"**>**
32. **<property** name="template" ref="jt"**></property>**
33. **</bean>**
34. **</beans>**

### 7. Create the requested page

**index.jsp**

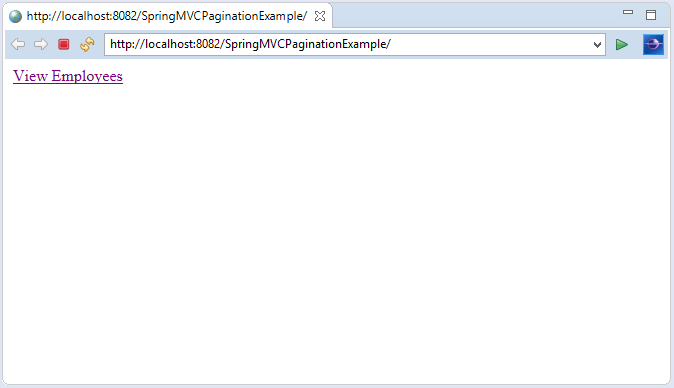
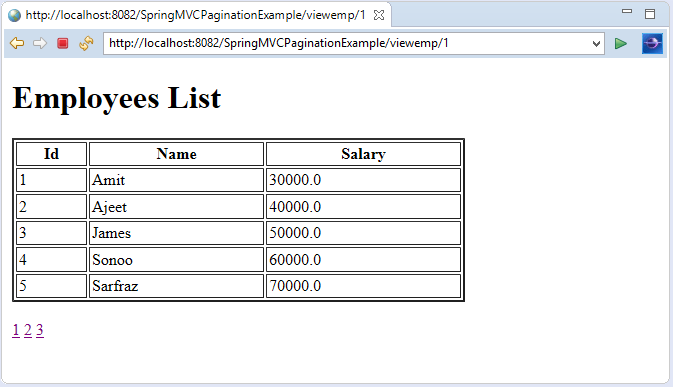
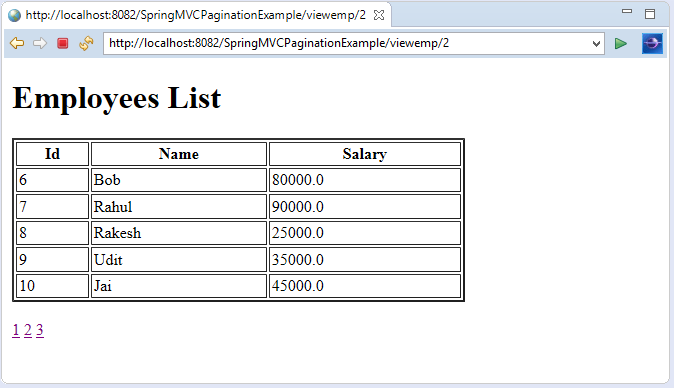
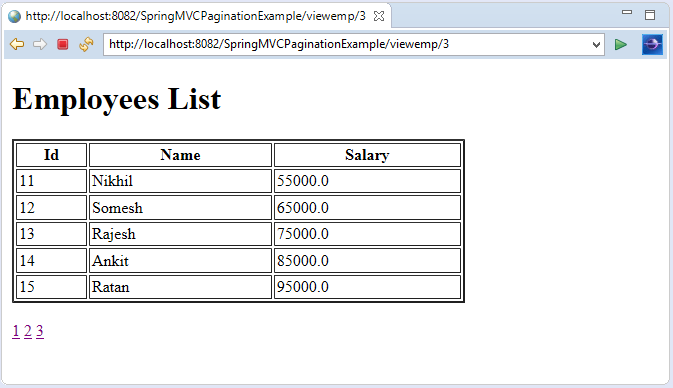
1. <!DOCTYPE html**>**
2. **<html>**
3. **<body>**
4. **<a** href="viewemp/1"**>**View Employees**</a>**
5. **</body>**
6. **</html>**

### 8. Create view component

**viewemp.jsp**

1. **<**%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%**>**
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"**>**
3. **<html>**
4. **<body>**
5. **<h1>**Employees List**</h1>**
6. **<table** border="2" width="70%" cellpadding="2"**>**
7. **<tr><th>**Id**</th><th>**Name**</th><th>**Salary**</th></tr>**
8. **<c:forEach** var="emp" items="${msg}"**>**
9. **<tr>**
10. **<td>**${emp.id}**</td>**
11. **<td>**${emp.name}**</td>**
12. **<td>**${emp.salary}**</td>**
13. **</tr>**
14. **</c:forEach>**
15. **</table>**
16. **<br/>**
17. **<a** href="/SpringMVCPaginationExample/viewemp/1"**>**1**</a>**
18. **<a** href="/SpringMVCPaginationExample/viewemp/2"**>**2**</a>**
19. **<a** href="/SpringMVCPaginationExample/viewemp/3"**>**3**</a>**
20. **</body>**
21. **</html>**

**Output:**

# **Spring MVC File Upload Example**

Spring MVC provides easy way to upload files, it may be image or other files. Let's see a simple example to upload file using Spring MVC.

#### Required Jar files

To run this example, you need to load:

* **Spring Core jar files**
* **Spring Web jar files**
* **commons-fileupload.jar and commons-io.jar file**

1) [Download all the jar files for spring including core, web, aop, mvc, j2ee, remoting, oxm, jdbc, orm etc.](https://static.javatpoint.com/src/sp/springjars.zip)

2) [Download commons-io.jar](https://static.javatpoint.com/sppages/download/commons-io-1.2.jar)

3) [Download commons-fileupload.jar](https://static.javatpoint.com/sppages/download/commons-fileupload-1.3.jar)

## **Spring MVC File Upload Steps (Extra than MVC)**

1) Add commons-io and fileupload.jar files

2) Add entry of CommonsMultipartResolver in spring-servlet.xml

1. **<bean** id="multipartResolver"
2. class="org.springframework.web.multipart.commons.CommonsMultipartResolver"**/>**

3) Create form to submit file. Method name must be "post" and enctype "multiple/form-data".

1. **<form** action="savefile" method="post" enctype="multipart/form-data"**>**
2. Select File: **<input** type="file" name="file"**/>**
3. **<input** type="submit" value="Upload File"**/>**
4. **</form>**

4) Use CommonsMultipartFile class in Controller.

1. @RequestMapping(value="/savefile",method=RequestMethod.POST)
2. **public** ModelAndView upload(@RequestParam CommonsMultipartFile file,HttpSession session){
3. String path=session.getServletContext().getRealPath("/");
4. String filename=file.getOriginalFilename();
6. System.out.println(path+" "+filename);
7. **try**{
8. **byte** barr[]=file.getBytes();
10. BufferedOutputStream bout=**new** BufferedOutputStream(
11. **new** FileOutputStream(path+"/"+filename));
12. bout.write(barr);
13. bout.flush();
14. bout.close();
16. }**catch**(Exception e){System.out.println(e);}
17. **return** **new** ModelAndView("upload-success","filename",path+"/"+filename);
18. }

5) Display image in JSP.

1. **<h1>**Upload Success**</h1>**
2. **<img** src="${filename}"**/>**

## **Spring MVC File Upload Example**

**Create images directory**

Create "images" directory in your project because we are writing the code to save all the files inside "/images" directory.

**index.jsp**

1. **<a** href="uploadform"**>**Upload Image**</a>**

**Emp.java**

1. **package** com.javatpoint;
2. **import** java.io.BufferedOutputStream;
3. **import** java.io.File;
4. **import** java.io.FileOutputStream;
5. **import** javax.servlet.ServletContext;
6. **import** javax.servlet.http.HttpServletRequest;
7. **import** javax.servlet.http.HttpServletResponse;
8. **import** javax.servlet.http.HttpSession;
9. **import** org.apache.commons.fileupload.disk.DiskFileItemFactory;
10. **import** org.apache.commons.fileupload.servlet.ServletFileUpload;
11. **import** org.springframework.stereotype.Controller;
12. **import** org.springframework.web.bind.annotation.ModelAttribute;
13. **import** org.springframework.web.bind.annotation.RequestMapping;
14. **import** org.springframework.web.bind.annotation.RequestMethod;
15. **import** org.springframework.web.bind.annotation.RequestParam;
16. **import** org.springframework.web.multipart.commons.CommonsMultipartFile;
17. **import** org.springframework.web.servlet.ModelAndView;
19. @Controller
20. **public** **class** HelloController {
21. **private** **static** **final** String UPLOAD\_DIRECTORY ="/images";
23. @RequestMapping("uploadform")
24. **public** ModelAndView uploadForm(){
25. **return** **new** ModelAndView("uploadform");
26. }
28. @RequestMapping(value="savefile",method=RequestMethod.POST)
29. **public** ModelAndView saveimage( @RequestParam CommonsMultipartFile file,
30. HttpSession session) **throws** Exception{
32. ServletContext context = session.getServletContext();
33. String path = context.getRealPath(UPLOAD\_DIRECTORY);
34. String filename = file.getOriginalFilename();
36. System.out.println(path+" "+filename);
38. **byte**[] bytes = file.getBytes();
39. BufferedOutputStream stream =**new** BufferedOutputStream(**new** FileOutputStream(
40. **new** File(path + File.separator + filename)));
41. stream.write(bytes);
42. stream.flush();
43. stream.close();
45. **return** **new** ModelAndView("uploadform","filesuccess","File successfully saved!");
46. }
47. }

**web.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app version="2.5"
3. xmlns="http://java.sun.com/xml/ns/javaee"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">
7. <servlet>
8. <servlet-name>spring</servlet-name>
9. <servlet-**class**>org.springframework.web.servlet.DispatcherServlet</servlet-**class**>
10. <load-on-startup>1</load-on-startup>
11. </servlet>
12. <servlet-mapping>
13. <servlet-name>spring</servlet-name>
14. <url-pattern>/</url-pattern>
15. </servlet-mapping>
16. </web-app>

**spring-servlet.xml**

Here, you need to create a bean for CommonsMultipartResolver.

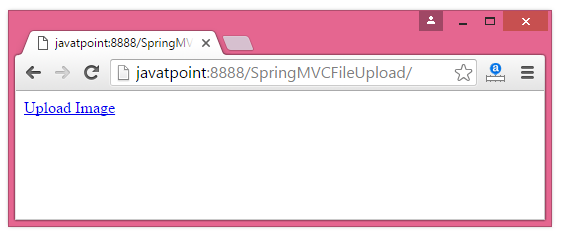
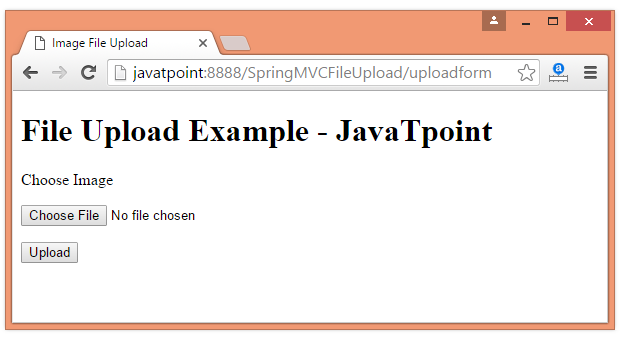
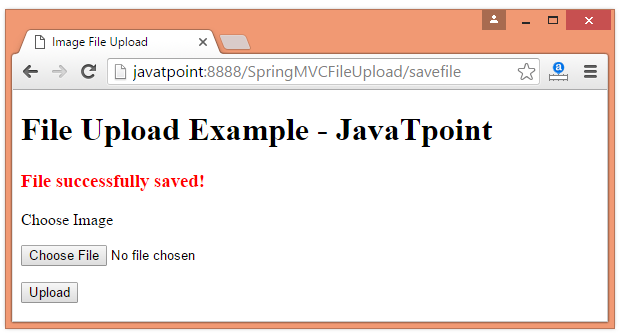
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5. xmlns:context="http://www.springframework.org/schema/context"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8. http://www.springframework.org/schema/context
9. http://www.springframework.org/schema/context/spring-context-3.0.xsd">
11. <context:component-scan base-**package**="com.javatpoint"></context:component-scan>
13. <bean **class**="org.springframework.web.servlet.view.InternalResourceViewResolver">
14. <property name="prefix" value="/WEB-INF/jsp/"></property>
15. <property name="suffix" value=".jsp"></property>
16. </bean>
18. <bean id="multipartResolver"
19. **class**="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
21. </beans>

**uploadform.jsp**

Here form must be method="post" and enctype="multipart/form-data".

1. <%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
3. <!DOCTYPE html>
4. <html>
5. <head>
6. <title>Image File Upload</title>
7. </head>
8. <body>
9. <h1>File Upload Example - JavaTpoint</h1>
11. <h3 style="color:red">${filesuccess}</h3>
12. <form:form method="post" action="savefile" enctype="multipart/form-data">
13. <p><label **for**="image">Choose Image</label></p>
14. <p><input name="file" id="fileToUpload" type="file" /></p>
15. <p><input type="submit" value="Upload"></p>
16. </form:form>
17. </body>
18. </html>

#### Output

Go to the path printed on the server console, to see the uploaded file.

.

# **Spring MVC Validation**

The Spring MVC Validation is used to restrict the input provided by the user. To validate the user's input, the Spring 4 or higher version supports and use Bean Validation API. It can validate both server-side as well as client-side applications.

## **Bean Validation API**

The Bean Validation API is a Java specification which is used to apply constraints on object model via annotations. Here, we can validate a length, number, regular expression, etc. Apart from that, we can also provide custom validations.

As Bean Validation API is just a specification, it requires an implementation. So, for that, it uses Hibernate Validator. The Hibernate Validator is a fully compliant JSR-303/309 implementation that allows to express and validate application constraints.

## **Validation Annotations**

Let's see some frequently used validation annotations.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| @NotNull | It determines that the value can't be null. |
| @Min | It determines that the number must be equal or greater than the specified value. |
| @Max | It determines that the number must be equal or less than the specified value. |
| @Size | It determines that the size must be equal to the specified value. |
| @Pattern | It determines that the sequence follows the specified regular expression. |

## **Spring MVC Validation Example**

In this example, we create a simple form that contains the input fields. Here, (\*) means it is mandatory to enter the corresponding field. Otherwise, the form generates an error.

### 1. Add dependencies to pom.xml file.

**pom.xml**

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
7. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
8. **<dependency>**
9. **<groupId>**org.apache.tomcat**</groupId>**
10. **<artifactId>**tomcat-jasper**</artifactId>**
11. **<version>**9.0.12**</version>**
12. **</dependency>**
13. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
14. **<dependency>**
15. **<groupId>**javax.servlet**</groupId>**
16. **<artifactId>**servlet-api**</artifactId>**
17. **<version>**3.0-alpha-1**</version>**
18. **</dependency>**
19. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
20. **<dependency>**
21. **<groupId>**javax.servlet**</groupId>**
22. **<artifactId>**jstl**</artifactId>**
23. **<version>**1.2**</version>**
24. **</dependency>**
25. <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
26. **<dependency>**
27. **<groupId>**org.hibernate.validator**</groupId>**
28. **<artifactId>**hibernate-validator**</artifactId>**
29. **<version>**6.0.13.Final**</version>**
30. **</dependency>**

### 2. Create the bean class

**Employee.java**

1. **package** com.javatpoint;
2. **import** javax.validation.constraints.Size;
4. **public** **class** Employee {
6. **private** String name;
7. @Size(min=1,message="required")
8. **private** String pass;
10. **public** String getName() {
11. **return** name;
12. }
13. **public** **void** setName(String name) {
14. **this**.name = name;
15. }
16. **public** String getPass() {
17. **return** pass;
18. }
19. **public** **void** setPass(String pass) {
20. **this**.pass = pass;
21. }
22. }

### 3. Create the controller class

**In controller class:**

* The **@Valid** annotation applies validation rules on the provided object.
* The **BindingResult** interface contains the result of validation.

1. **package** com.javatpoint;
3. **import** javax.validation.Valid;
4. **import** org.springframework.stereotype.Controller;
5. **import** org.springframework.ui.Model;
6. **import** org.springframework.validation.BindingResult;
7. **import** org.springframework.web.bind.annotation.ModelAttribute;
8. **import** org.springframework.web.bind.annotation.RequestMapping;
10. @Controller
11. **public** **class** EmployeeController {
13. @RequestMapping("/hello")
14. **public** String display(Model m)
15. {
16. m.addAttribute("emp", **new** Employee());
17. **return** "viewpage";
18. }
19. @RequestMapping("/helloagain")
20. **public** String submitForm( @Valid @ModelAttribute("emp") Employee e, BindingResult br)
21. {
22. **if**(br.hasErrors())
23. {
24. **return** "viewpage";
25. }
26. **else**
27. {
28. **return** "final";
29. }
30. }
31. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. **<html>**
3. **<body>**
4. **<a** href="hello"**>**Click here...**</a>**
5. **</body>**
6. **</html>**

### 7. Create the other view components

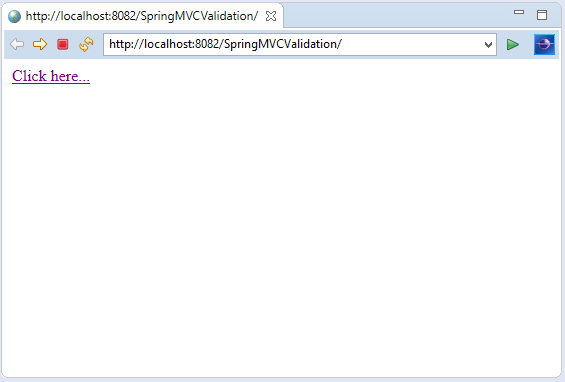
**viewpage.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. **<html>**
3. **<head>**
4. **<style>**
5. .error{color:red}
6. **</style>**
7. **</head>**
8. **<body>**
9. **<form:form** action="helloagain" modelAttribute="emp"**>**
10. Username: **<form:input** path="name"**/>** **<br><br>**
11. Password(\*): **<form:password** path="pass"**/>**
12. **<form:errors** path="pass" cssClass="error"**/><br><br>**
13. **<input** type="submit" value="submit"**>**
14. **</form:form>**
15. **</body>**
16. **</html>**

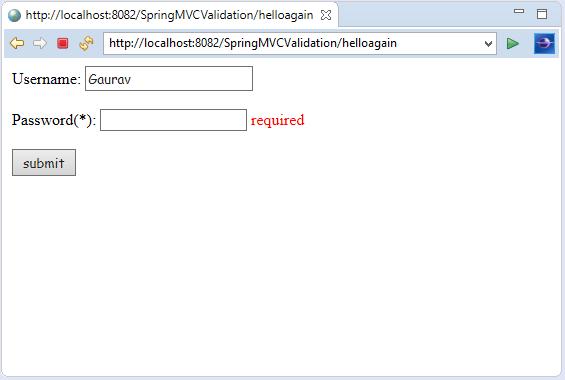
**final.jsp**

1. **<html>**
2. **<body>**
3. Username: ${emp.name} **<br><br>**
4. Password: ${emp.pass}
5. **</body>**
6. **</html>**

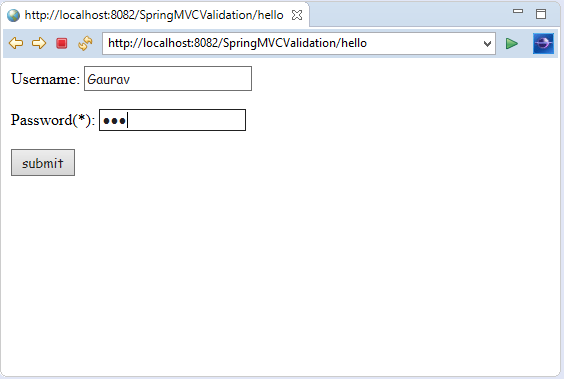
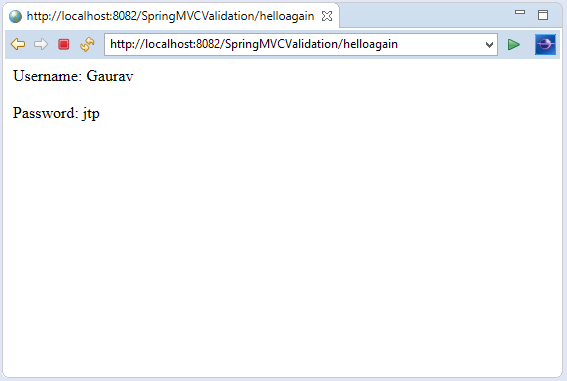
**Output:**



Let's submit the form without entering the password.



Now, we entered the password and then submit the form.

# **Spring MVC Regular Expression Validation**

The Spring MVC Validation allows us to validate the user input in a particular sequence (i.e., regular expression). The **@Pattern** annotation is used to achieve regular expression validation. Here, we can provide the required regular expression to **regexp** attribute and pass it with the annotation.

## **Spring MVC Regular Expression Validation Example**

### 1. Add dependencies to pom.xml file.

**pom.xml**

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
7. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
8. **<dependency>**
9. **<groupId>**org.apache.tomcat**</groupId>**
10. **<artifactId>**tomcat-jasper**</artifactId>**
11. **<version>**9.0.12**</version>**
12. **</dependency>**
13. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
14. **<dependency>**
15. **<groupId>**javax.servlet**</groupId>**
16. **<artifactId>**servlet-api**</artifactId>**
17. **<version>**3.0-alpha-1**</version>**
18. **</dependency>**
19. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
20. **<dependency>**
21. **<groupId>**javax.servlet**</groupId>**
22. **<artifactId>**jstl**</artifactId>**
23. **<version>**1.2**</version>**
24. **</dependency>**
25. <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
26. **<dependency>**
27. **<groupId>**org.hibernate.validator**</groupId>**
28. **<artifactId>**hibernate-validator**</artifactId>**
29. **<version>**6.0.13.Final**</version>**
30. **</dependency>**

### 2. Create the bean class

**Employee.java**

1. **package** com.javatpoint;
3. **import** javax.validation.constraints.Pattern;
4. **public** **class** Employee {
6. **private** String name;
7. @Pattern(regexp="^[a-zA-Z0-9]{3}",message="length must be 3")
8. **private** String pass;
10. **public** String getName() {
11. **return** name;
12. }
13. **public** **void** setName(String name) {
14. **this**.name = name;
15. }
16. **public** String getPass() {
17. **return** pass;
18. }
19. **public** **void** setPass(String pass) {
20. **this**.pass = pass;
21. }
22. }

### 3. Create the controller class

1. **package** com.javatpoint;
3. **import** javax.validation.Valid;
4. **import** org.springframework.stereotype.Controller;
5. **import** org.springframework.ui.Model;
6. **import** org.springframework.validation.BindingResult;
7. **import** org.springframework.web.bind.annotation.ModelAttribute;
8. **import** org.springframework.web.bind.annotation.RequestMapping;
10. @Controller
11. **public** **class** EmployeeController {
13. @RequestMapping("/hello")
14. **public** String display(Model m)
15. {
16. m.addAttribute("emp", **new** Employee());
17. **return** "viewpage";
18. }
19. @RequestMapping("/helloagain")
20. **public** String submitForm(@Valid @ModelAttribute("emp") Employee e, BindingResult br)
21. {
22. **if**(br.hasErrors())
23. {
24. **return** "viewpage";
25. }
26. **else**
27. {
28. **return** "final";
29. }
30. }
31. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. index.jsp
2. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
3. **<html>**
4. **<body>**
5. **<a** href="hello"**>**Click here...**</a>**
6. **</body>**
7. **</html>**

### 7. Create the other view components

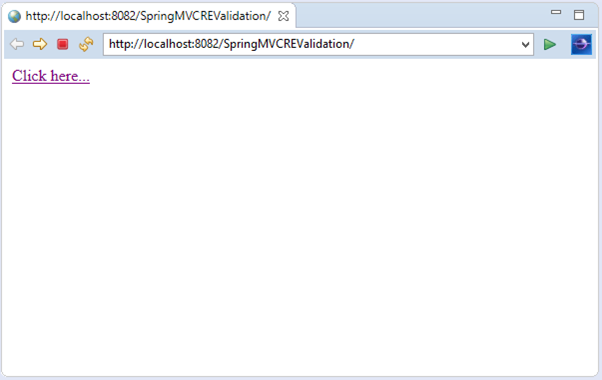
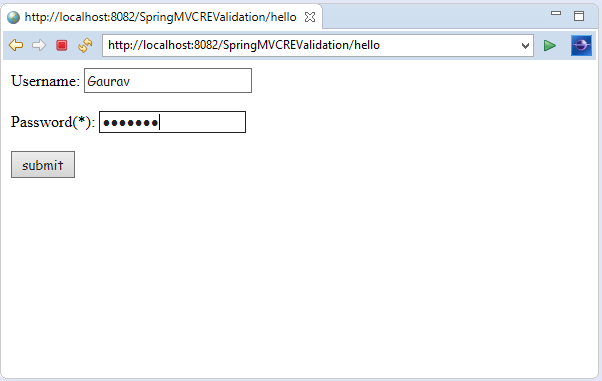
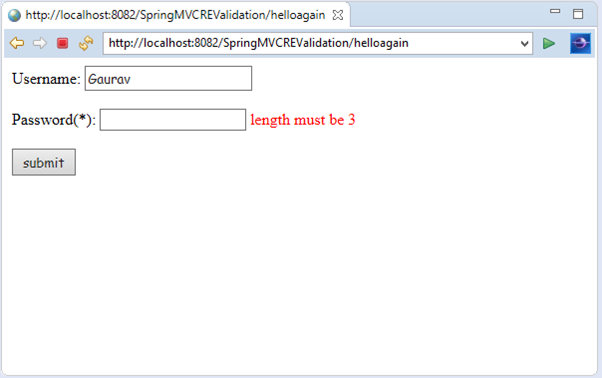
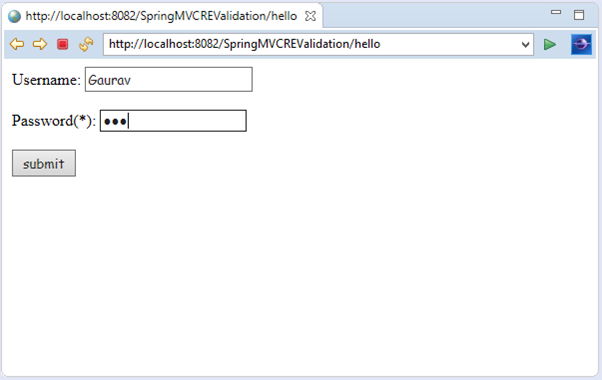
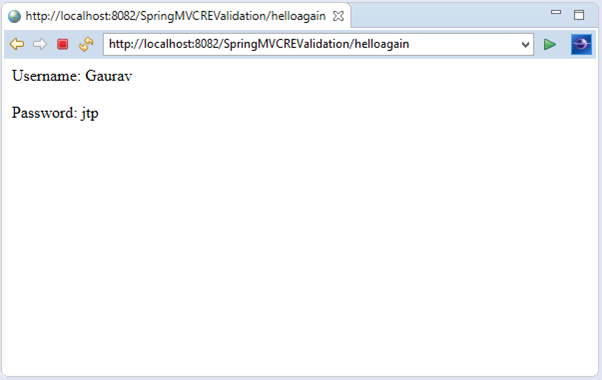
**viewpage.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. **<html>**
3. **<head>**
4. **<style>**
5. .error{color:red}
6. **</style>**
7. **</head>**
8. **<body>**
9. **<form:form** action="helloagain" modelAttribute="emp"**>**
10. Username: **<form:input** path="name"**/>** **<br><br>**
11. Password(\*): **<form:password** path="pass"**/>**
12. **<form:errors** path="pass" cssClass="error"**/><br><br>**
13. **<input** type="submit" value="submit"**>**
14. **</form:form>**
15. **</body>**
16. **</html>**

**final.jsp**

1. **<html>**
2. **<body>**
3. Username: ${emp.name} **<br><br>**
4. Password: ${emp.pass}
5. **</body>**
6. **</html>**

**Output:**

# **Spring MVC Number Validation**

In Spring MVC Validation, we can validate the user's input within a number range. The following annotations are used to achieve number validation:

* **@Min annotation** - It is required to pass an integer value with @Min annotation. The user input must be equal to or greater than this value.
* **@Max annotation** - It is required to pass an integer value with @Max annotation. The user input must be equal to or smaller than this value.

## **Spring MVC Number Validation Example**

### 1. Add dependencies to pom.xml file.

**pom.xml**

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
7. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
8. **<dependency>**
9. **<groupId>**org.apache.tomcat**</groupId>**
10. **<artifactId>**tomcat-jasper**</artifactId>**
11. **<version>**9.0.12**</version>**
12. **</dependency>**
13. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
14. **<dependency>**
15. **<groupId>**javax.servlet**</groupId>**
16. **<artifactId>**servlet-api**</artifactId>**
17. **<version>**3.0-alpha-1**</version>**
18. **</dependency>**
19. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
20. **<dependency>**
21. **<groupId>**javax.servlet**</groupId>**
22. **<artifactId>**jstl**</artifactId>**
23. **<version>**1.2**</version>**
24. **</dependency>**
25. <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
26. **<dependency>**
27. **<groupId>**org.hibernate.validator**</groupId>**
28. **<artifactId>**hibernate-validator**</artifactId>**
29. **<version>**6.0.13.Final**</version>**
30. **</dependency>**

### 2. Create the bean class

**Employee.java**

1. **package** com.javatpoint;
3. **import** javax.validation.constraints.Max;
4. **import** javax.validation.constraints.Min;
5. **import** javax.validation.constraints.Size;
7. **public** **class** Employee {
9. **private** String name;
10. @Size(min=1,message="required")
11. **private** String pass;
13. @Min(value=18, message="must be equal or greater than 18")
14. @Max(value=45, message="must be equal or less than 45")
15. **private** **int** age;
17. **public** String getName() {
18. **return** name;
19. }
20. **public** **void** setName(String name) {
21. **this**.name = name;
22. }
23. **public** String getPass() {
24. **return** pass;
25. }
26. **public** **void** setPass(String pass) {
27. **this**.pass = pass;
28. }
29. **public** **int** getAge() {
30. **return** age;
31. }
32. **public** **void** setAge(**int** age) {
33. **this**.age = age;
34. }
36. }

### 3. Create the controller class

**EmployeeController.java**

1. **package** com.javatpoint;
3. **import** javax.validation.Valid;
4. **import** org.springframework.stereotype.Controller;
5. **import** org.springframework.ui.Model;
6. **import** org.springframework.validation.BindingResult;
7. **import** org.springframework.web.bind.annotation.ModelAttribute;
8. **import** org.springframework.web.bind.annotation.RequestMapping;
10. @Controller
11. **public** **class** EmployeeController {
13. @RequestMapping("/hello")
14. **public** String display(Model m)
15. {
16. m.addAttribute("emp", **new** Employee());
17. **return** "viewpage";
18. }
19. @RequestMapping("/helloagain")
20. **public** String submitForm( @Valid @ModelAttribute("emp") Employee e, BindingResult br)
21. {
22. **if**(br.hasErrors())
23. {
24. **return** "viewpage";
25. }
26. **else**
27. {
28. **return** "final";
29. }
30. }
31. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 6. Create the requested page

**index.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. **<html>**
3. **<body>**
4. **<a** href="hello"**>**Click here...**</a>**
5. **</body>**
6. **</html>**

### 7. Create the other view components

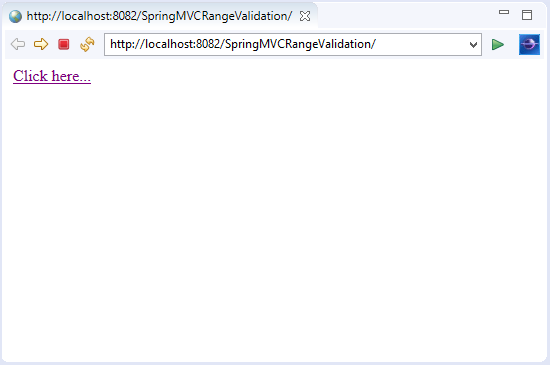
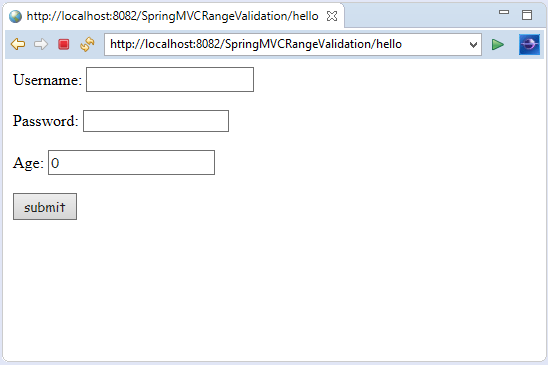
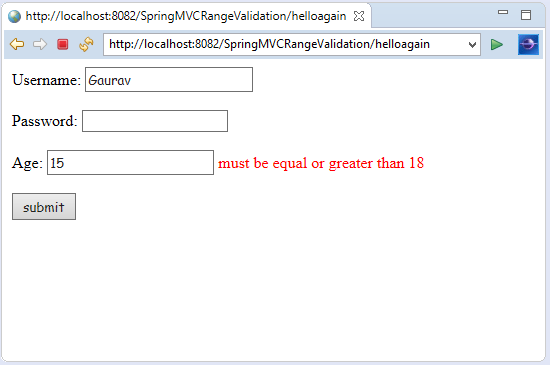
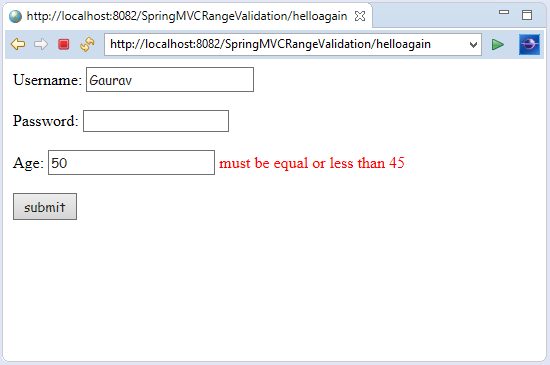
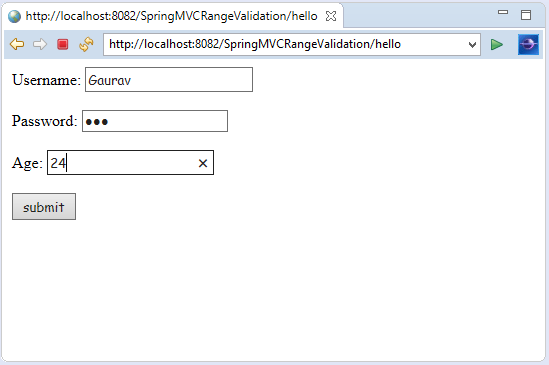
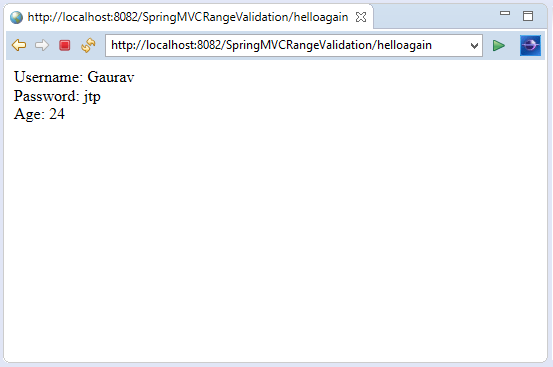
**viewpage.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. **<html>**
3. **<head>**
4. **<style>**
5. .error{color:red}
6. **</style>**
7. **</head>**
8. **<body>**
9. **<form:form** action="helloagain" modelAttribute="emp"**>**
10. Username: **<form:input** path="name"**/>** **<br><br>**
11. Password: **<form:password** path="pass"**/>**
12. **<form:errors** path="pass" cssClass="error"**/><br><br>**
13. Age: **<form:input** path="age"**/>**
14. **<form:errors** path="age" cssClass="error"**/><br><br>**
15. **<input** type="submit" value="submit"**>**
16. **</form:form>**
17. **</body>**
18. **</html>**

**final.jsp**

1. **<html>**
2. **<body>**
3. Username: ${param.name} **<br>**
4. Password: ${param.pass} **<br>**
5. Age: ${param.age } **<br>**
6. **</body>**
7. **</html>**

**Output:**

# **Spring MVC Custom Validation**

The Spring MVC framework allows us to perform custom validations. In such case, we declare own annotations. We can perform validation on the basis of own business logic.

## **Spring MVC Custom Validation Example**

In this example, we use both pre-defined annotations as well as custom annotations to validate user input.

### 1. Add dependencies to pom.xml file.

**pom.xml**

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
7. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
8. **<dependency>**
9. **<groupId>**org.apache.tomcat**</groupId>**
10. **<artifactId>**tomcat-jasper**</artifactId>**
11. **<version>**9.0.12**</version>**
12. **</dependency>**
13. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
14. **<dependency>**
15. **<groupId>**javax.servlet**</groupId>**
16. **<artifactId>**servlet-api**</artifactId>**
17. **<version>**3.0-alpha-1**</version>**
18. **</dependency>**
19. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
20. **<dependency>**
21. **<groupId>**javax.servlet**</groupId>**
22. **<artifactId>**jstl**</artifactId>**
23. **<version>**1.2**</version>**
24. **</dependency>**
25. <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
26. **<dependency>**
27. **<groupId>**org.hibernate.validator**</groupId>**
28. **<artifactId>**hibernate-validator**</artifactId>**
29. **<version>**6.0.13.Final**</version>**
30. **</dependency>**

### 2. Create the bean class

**Employee.java**

1. **package** com.javatpoint;
3. **import** javax.validation.constraints.Max;
4. **import** javax.validation.constraints.Min;
5. **import** com.javatpoint.customvalidation.Password;
7. **public** **class** Employee {
8. **private** String name;
9. //Custom annotation
10. @Password
11. **private** String password;
12. //Predefined annotation
13. @Min(value=18, message="must be equal or greater than 18")
14. @Max(value=45, message="must be equal or less than 45")
15. **private** **int** age;
17. **public** String getName() {
18. **return** name;
19. }
21. **public** **void** setName(String name) {
22. **this**.name = name;
23. }
25. **public** String getPassword() {
26. **return** password;
27. }
29. **public** **void** setPassword(String password) {
30. **this**.password = password;
31. }
33. **public** **int** getAge() {
34. **return** age;
35. }
37. **public** **void** setAge(**int** age) {
38. **this**.age = age;
39. }
40. }

### 3. Create the controller class

**EmployeeController.java**

1. **package** com.javatpoint;
3. **import** javax.validation.Valid;
4. **import** org.springframework.stereotype.Controller;
5. **import** org.springframework.ui.Model;
6. **import** org.springframework.validation.BindingResult;
7. **import** org.springframework.web.bind.annotation.ModelAttribute;
8. **import** org.springframework.web.bind.annotation.RequestMapping;
10. @Controller
11. **public** **class** EmployeeController {
13. @RequestMapping("/hello")
14. **public** String showForm(Model theModel) {
16. theModel.addAttribute("emp", **new** Employee());
18. **return** "viewpage";
19. }
21. @RequestMapping("/helloagain")
22. **public** String processForm(
23. @Valid @ModelAttribute("emp") Employee emp,
24. BindingResult br) {
26. **if** (br.hasErrors()) {
27. **return** "viewpage";
28. }
29. **else** {
30. **return** "final";
31. }
32. }
33. }

### 4. Create the validator annotation

**Password.java**

1. **package** com.javatpoint.customvalidation;
3. **import** java.lang.annotation.ElementType;
4. **import** java.lang.annotation.Retention;
5. **import** java.lang.annotation.RetentionPolicy;
6. **import** java.lang.annotation.Target;
8. **import** javax.validation.Constraint;
9. **import** javax.validation.Payload;
11. @Constraint(validatedBy = PasswordConstraintValidator.**class**)
12. @Target( { ElementType.METHOD, ElementType.FIELD } )
13. @Retention(RetentionPolicy.RUNTIME)
14. **public** **@interface** Password {
15. //error message
16. **public** String message() **default** "must contain jtp";
17. //represents group of constraints
18. **public** Class<?>[] groups() **default** {};
19. //represents additional information about annotation
20. **public** Class<? **extends** Payload>[] payload() **default** {};
21. }

### 5. Create the validator class

**PasswordConstraintValidator.java**

1. **package** com.javatpoint.customvalidation;
3. **import** javax.validation.ConstraintValidator;
4. **import** javax.validation.ConstraintValidatorContext;
6. **public** **class** PasswordConstraintValidator **implements** ConstraintValidator<Password,String> {
8. **public** **boolean** isValid(String s, ConstraintValidatorContext cvc) {
10. **boolean** result=s.contains("jtp");
11. **return** result;
12. }
13. }

### 6. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 7. Define the bean in the xml file

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
13. <!-- Provide support for component scanning -->
14. **<context:component-scan** base-package="com.javatpoint" **/>**
15. <!--Provide support for conversion, formatting and validation -->
16. **<mvc:annotation-driven/>**
17. <!-- Define Spring MVC view resolver -->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix" value="/WEB-INF/jsp/"**></property>**
20. **<property** name="suffix" value=".jsp"**></property>**
21. **</bean>**
22. **</beans>**

### 8. Create the requested page

**index.jsp**

1. **<html>**
2. **<body>**
3. **<a** href="hello"**>**Click here...**</a>**
4. **</body>**
5. **</html>**

### 9. Create the other view components

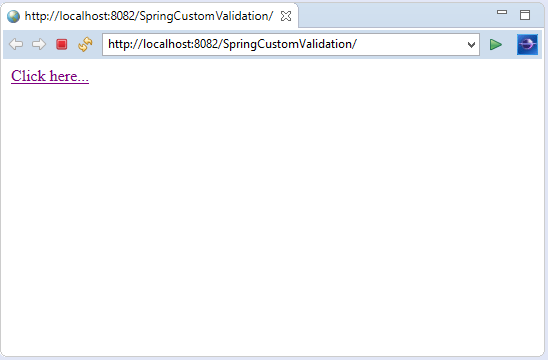
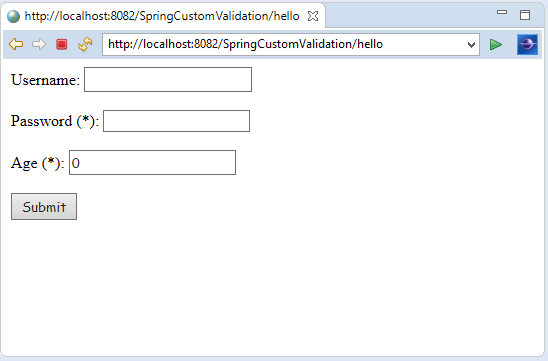
**viewpage.jsp**

1. **<**%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
2. **<html>**
3. **<head>**
4. **<style>**
5. .error {color:red}
6. **</style>**
7. **</head>**
8. **<body>**
9. **<form:form** action="helloagain" modelAttribute="emp"**>**
10. Username: **<form:input** path="name" **/>**
11. **<br><br>**
13. Password (\*): **<form:password** path="password" **/>**
14. **<form:errors** path="password" cssClass="error" **/>**
15. **<br><br>**
17. Age (\*): **<form:input** path="age" **/>**
18. **<form:errors** path="age" cssClass="error" **/>**
19. **<br><br>**
20. **<input** type="submit" value="Submit" **/>**
21. **</form:form>**
22. **</body>**
23. **</html>**

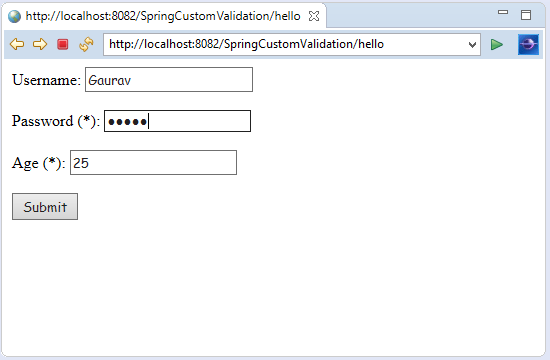
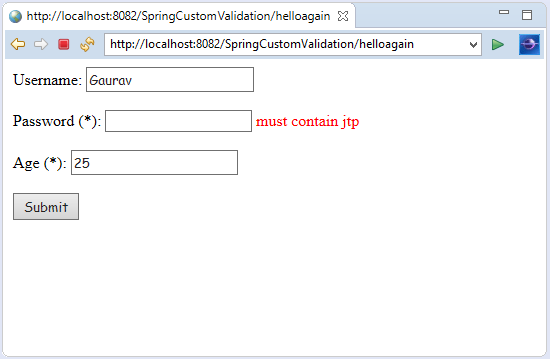
**final.jsp**

1. **<**%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %**>**
2. <!DOCTYPE html**>**
3. **<html>**
4. **<body>**
5. Username: ${emp.name}**<br><br>**
6. Password: ${emp.password}**<br><br>**
7. Age: ${emp.age}
8. **<br><br>**
9. **</body>**
10. **</html>**

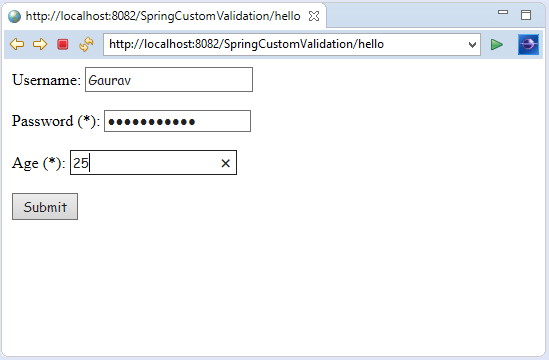
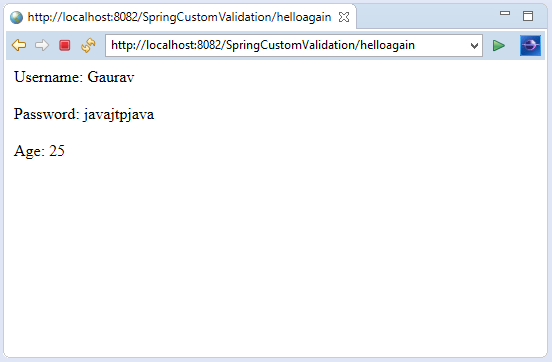
**Output:**

Here, we entered the password without having "jtp" sequence.

Now, we entered the password having "jtp" sequence.

# **Spring MVC Tiles Example**

Spring provides integration support with apache tiles framework. So we can simply manage the layout of the Spring MVC application with the help of spring tiles support.

## **Advantage of Tiles support in Spring MVC**

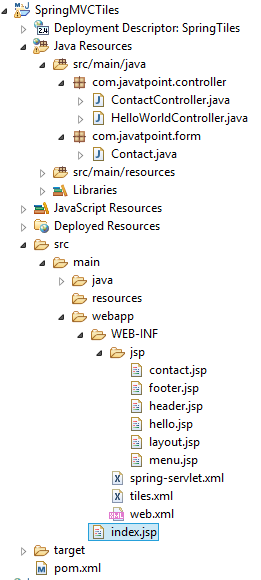
**Reusability:** We can reuse a single component in multiple pages like header and footer components.

**Centralized control:** We can control the layout of the page by a single template page only.

**Easy to change the layout:** By the help of single template page, we can change the layout of the page anytime. So your website can easily adopt new technologies such as bootstrap, jQuery, etc.

### Directory Structure

Let's see the files of spring tiles example in eclipse.



## **Spring MVC Tiles Example**

### 1. Add dependencies to pom.xml file.

**pom.xml**

1. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
2. **<dependency>**
3. **<groupId>**org.springframework**</groupId>**
4. **<artifactId>**spring-webmvc**</artifactId>**
5. **<version>**5.1.1.RELEASE**</version>**
6. **</dependency>**
7. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
8. **<dependency>**
9. **<groupId>**javax.servlet**</groupId>**
10. **<artifactId>**servlet-api**</artifactId>**
11. **<version>**3.0-alpha-1**</version>**
12. **</dependency>**
13. <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
14. **<dependency>**
15. **<groupId>**javax.servlet**</groupId>**
16. **<artifactId>**jstl**</artifactId>**
17. **<version>**1.2**</version>**
18. **</dependency>**
19. <!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
20. **<dependency>**
21. **<groupId>**org.apache.tomcat**</groupId>**
22. **<artifactId>**tomcat-jasper**</artifactId>**
23. **<version>**9.0.12**</version>**
24. **</dependency>**
25. <!-- https://mvnrepository.com/artifact/org.apache.tiles/tiles-jsp -->
26. **<dependency>**
27. **<groupId>**org.apache.tiles**</groupId>**
28. **<artifactId>**tiles-jsp**</artifactId>**
29. **<version>**3.0.5**</version>**
30. **</dependency>**
32. <!-- https://mvnrepository.com/artifact/org.apache.tiles/tiles-servlet -->
33. **<dependency>**
34. **<groupId>**org.apache.tiles**</groupId>**
35. **<artifactId>**tiles-servlet**</artifactId>**
36. **<version>**3.0.5**</version>**
37. **</dependency>**
38. <!-- https://mvnrepository.com/artifact/org.apache.tiles/tiles-core -->
39. **<dependency>**
40. **<groupId>**org.apache.tiles**</groupId>**
41. **<artifactId>**tiles-core**</artifactId>**
42. **<version>**3.0.5**</version>**
43. **</dependency>**
44. <!-- https://mvnrepository.com/artifact/org.apache.tiles/tiles-el -->
45. **<dependency>**
46. **<groupId>**org.apache.tiles**</groupId>**
47. **<artifactId>**tiles-el**</artifactId>**
48. **<version>**3.0.5**</version>**
49. **</dependency>**

### 2. Create the bean class

**Contact.java**

1. **package** com.javatpoint.form;
2. **public** **class** Contact {
3. **private** String firstname;
4. **private** String lastname;
5. **private** String email;
6. **private** String telephone;
8. **public** String getEmail() {
9. **return** email;
10. }
11. **public** String getTelephone() {
12. **return** telephone;
13. }
14. **public** **void** setEmail(String email) {
15. **this**.email = email;
16. }
17. **public** **void** setTelephone(String telephone) {
18. **this**.telephone = telephone;
19. }
20. **public** String getFirstname() {
21. **return** firstname;
22. }
23. **public** String getLastname() {
24. **return** lastname;
25. }
26. **public** **void** setFirstname(String firstname) {
27. **this**.firstname = firstname;
28. }
29. **public** **void** setLastname(String lastname) {
30. **this**.lastname = lastname;
31. }
33. }

### 3. Create the controller class

**HelloWorldController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.web.bind.annotation.RequestMapping;
5. @Controller
6. **public** **class** HelloWorldController {
7. @RequestMapping("/hello")
8. **public** String helloWorld(Model m) {
9. String message = "Hello World, Spring MVC @ Javatpoint";
10. m.addAttribute("message", message);
11. **return** "hello";
12. }
13. }

**ContactController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.ui.Model;
4. **import** org.springframework.validation.BindingResult;
5. **import** org.springframework.web.bind.annotation.ModelAttribute;
6. **import** org.springframework.web.bind.annotation.RequestMapping;
7. **import** org.springframework.web.bind.annotation.RequestMethod;
8. **import** org.springframework.web.bind.annotation.SessionAttributes;
9. **import** com.javatpoint.form.Contact;
10. @Controller
11. @SessionAttributes
12. **public** **class** ContactController {
13. @RequestMapping(value = "/addContact", method = RequestMethod.POST)
14. **public** String addContact(@ModelAttribute("contact") Contact contact, BindingResult result) {
15. //write the code here to add contact
16. **return** "redirect:contact.html";
17. }
19. @RequestMapping("/contact")
20. **public** String showContacts(Model m) {
21. m.addAttribute("command", **new** Contact());
22. **return** "contact";
23. }
24. }

### 4. Provide the entry of controller in the web.xml file

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" id="WebApp\_ID" version="2.5"**>**
3. **<display-name>**SpringTiles**</display-name>**
4. **<welcome-file-list>**
5. **<welcome-file>**index.jsp**</welcome-file>**
6. **</welcome-file-list>**
7. **<servlet>**
8. **<servlet-name>**spring**</servlet-name>**
9. **<servlet-class>**
10. org.springframework.web.servlet.DispatcherServlet
11. **</servlet-class>**
12. **<load-on-startup>**1**</load-on-startup>**
13. **</servlet>**
14. **<servlet-mapping>**
15. **<servlet-name>**spring**</servlet-name>**
16. **<url-pattern>**\*.html**</url-pattern>**
17. **</servlet-mapping>**
18. **</web-app>**

### 5. Define the bean in the xml file

**spring-servlet.xml**

1. **<beans** xmlns="http://www.springframework.org/schema/beans"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xmlns:p="http://www.springframework.org/schema/p"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/context
8. http://www.springframework.org/schema/context/spring-context-3.0.xsd"**>**
10. **<context:annotation-config** **/>**
11. **<context:component-scan** base-package="com.javatpoint.controller" **/>**
12. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.tiles3.TilesViewResolver"**/>**
13. **<bean** id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesConfigurer"**>**
14. **<property** name="definitions"**>**
15. **<list>**
16. **<value>**/WEB-INF/tiles.xml**</value>**
17. **</list>**
18. **</property>**
19. **</bean>**
20. **</beans>**

### 6. Provide the tiles.xml file

**tiles.xml**

1. **<?xml** version="1.0" encoding="UTF-8" **?>**
2. <!DOCTYPE tiles-definitions PUBLIC
3. "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
4. "http://tiles.apache.org/dtds/tiles-config\_2\_0.dtd"**>**
5. **<tiles-definitions>**
6. **<definition** name="base.definition"
7. template="/WEB-INF/jsp/layout.jsp"**>**
8. **<put-attribute** name="title" value="" **/>**
9. **<put-attribute** name="header" value="/WEB-INF/jsp/header.jsp" **/>**
10. **<put-attribute** name="menu" value="/WEB-INF/jsp/menu.jsp" **/>**
11. **<put-attribute** name="body" value="" **/>**
12. **<put-attribute** name="footer" value="/WEB-INF/jsp/footer.jsp" **/>**
13. **</definition>**
14. **<definition** name="contact" extends="base.definition"**>**
15. **<put-attribute** name="title" value="Contact Manager" **/>**
16. **<put-attribute** name="body" value="/WEB-INF/jsp/contact.jsp" **/>**
17. **</definition>**
19. **<definition** name="hello" extends="base.definition"**>**
20. **<put-attribute** name="title" value="Hello Spring MVC" **/>**
21. **<put-attribute** name="body" value="/WEB-INF/jsp/hello.jsp" **/>**
22. **</definition>**
24. **</tiles-definitions>**

### 7. Create the requested page

**index.jsp**

1. **<a** href="hello.html"**>**Hello Spring**</a>** |
2. **<a** href="contact.html"**>**Contact**</a>**

### 8. Create the other view components

**hello.jsp**

1. **<html>**
2. **<head>**
3. **<title>**Spring MVC Example**</title>**
4. **</head>**
5. **<body>**
6. **<h1>**Welcome to Spring MVC**</h1>**
7. **<p>**Message is: ${message}**</p>**
8. **</body>**
9. **</html>**

**contact.jsp**

1. **<**%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%**>**
2. **<html>**
3. **<head>**
4. **<title>**Spring Tiles Contact Form**</title>**
5. **</head>**
6. **<body>**
7. **<h2>**Contact Manager**</h2>**
8. **<form:form** method="post" action="addContact.html"**>**
10. **<table>**
11. **<tr>**
12. **<td><form:label** path="firstname"**>**First Name**</form:label></td>**
13. **<td><form:input** path="firstname" **/></td>**
14. **</tr>**
15. **<tr>**
16. **<td><form:label** path="lastname"**>**Last Name**</form:label></td>**
17. **<td><form:input** path="lastname" **/></td>**
18. **</tr>**
19. **<tr>**
20. **<td><form:label** path="lastname"**>**Email**</form:label></td>**
21. **<td><form:input** path="email" **/></td>**
22. **</tr>**
23. **<tr>**
24. **<td><form:label** path="lastname"**>**Telephone**</form:label></td>**
25. **<td><form:input** path="telephone" **/></td>**
26. **</tr>**
27. **<tr>**
28. **<td** colspan="2"**>**
29. **<input** type="submit" value="Add Contact"**/>**
30. **</td>**
31. **</tr>**
32. **</table>**
34. **</form:form>**
35. **</body>**
36. **</html>**

**header.jsp**

1. **<h2>**Header**</h2>**
2. **<hr/>**

**footer.jsp**

1. **<hr/>**
2. **<p>**Copyright  2010-2014 javatpoint.com.**</p>**

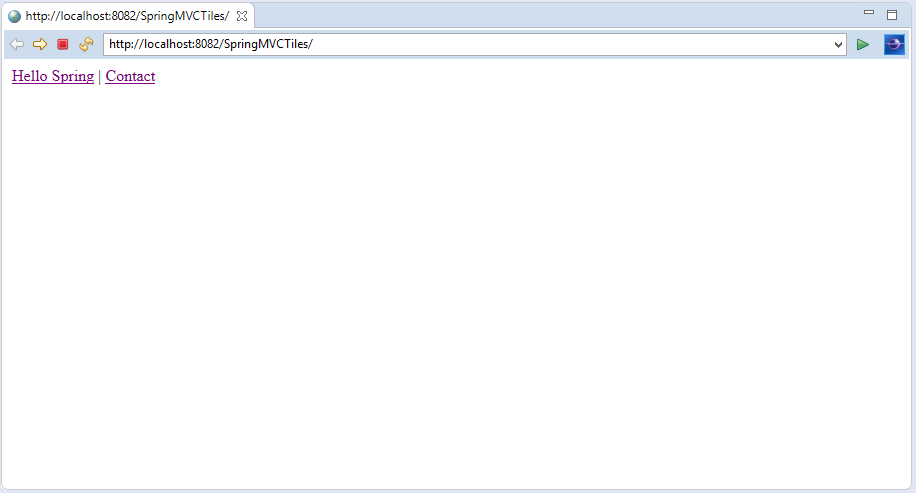
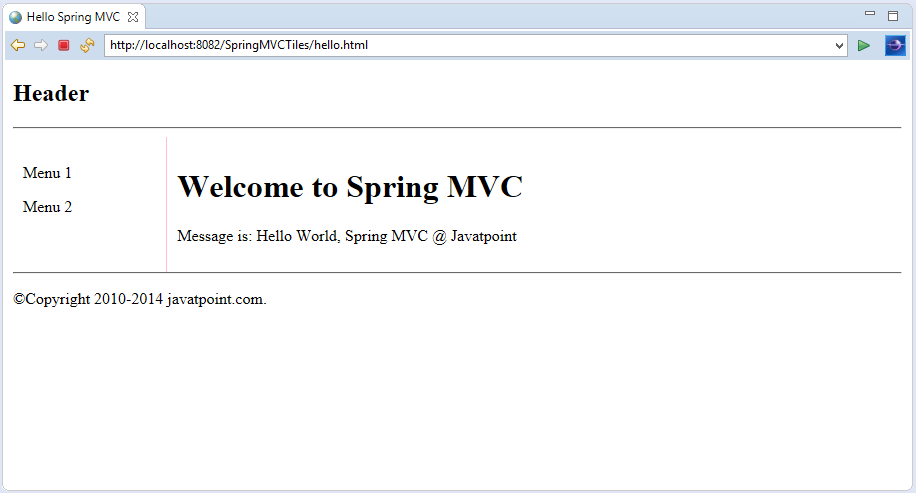
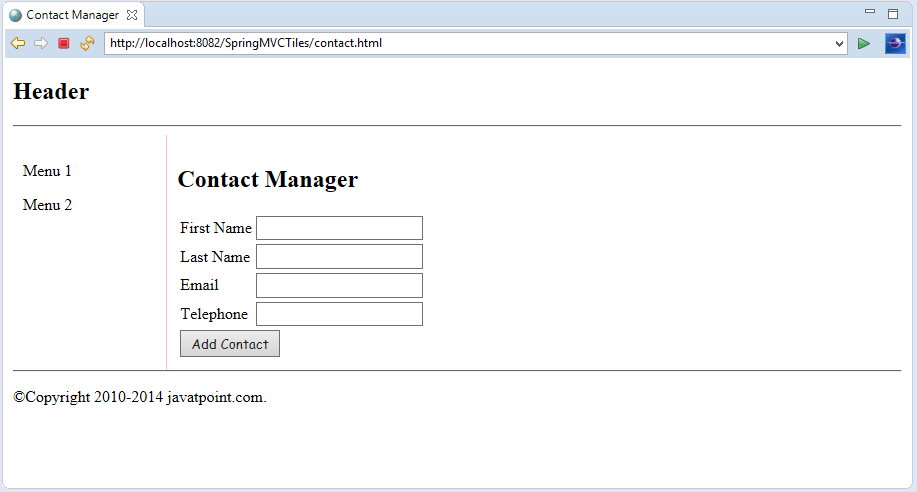
**menu.jsp**

1. **<p>**Menu 1**</p>**
2. **<p>**Menu 2**</p>**

**layout.jsp**

1. **<**%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%**>**
2. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
3. "http://www.w3.org/TR/html4/loose.dtd"**>**
4. **<html>**
5. **<head>**
6. **<meta** http-equiv="Content-Type" content="text/html; charset=UTF-8"**>**
7. **<title><tiles:insertAttribute** name="title" ignore="true" **/></title>**
8. **</head>**
9. **<body>**
10. **<div><tiles:insertAttribute** name="header" **/></div>**
11. **<div** style="float:left;padding:10px;width:15%;"**><tiles:insertAttribute** name="menu" **/></div>**
12. **<div** style="float:left;padding:10px;width:80%;border-left:1px solid pink;"**>**
13. **<tiles:insertAttribute** name="body" **/></div>**
14. **<div** style="clear:both"**><tiles:insertAttribute** name="footer" **/></div>**
16. **</body>**
17. **</html>**

**Output:**

# **Remoting in Spring Framework**

1. [Remoting in Spring Framework](https://www.javatpoint.com/remoting-in-spring-framework)
2. [Remoting Technologies](https://www.javatpoint.com/remoting-in-spring-framework#ex)

Spring framework makes the developement of remote-enabled services easy. It saves a lot of code by providing its own API.

#### Advantage of Spring Remoting

The programmer needs to concentrate on business logic only not plumbing activities such as starting and stopping the server.

Spring framework supports following remoting technologies:

* Remote Method Invocation (RMI)
* Spring's HTTP invoker
* Hessian
* Burlap
* JAX-RPC (J2EE 1.4 API)
* JAX-WS (Java EE 5 and Java EE 6 API)
* JMS

### Remote Method Invocation (RMI)

By the help of **RmiServiceExporter** and **RmiProxyFactoryBean** classes, spring framework supports RMI provided by Sun.

Click here to get details about [Spring and RMI Integration](https://www.javatpoint.com/spring-and-rmi-integration)

### Spring's HTTP invoker

Spring provides its own remoting service that allows serialization by HTTP. The classes used in HTTP Invoker are **HttpInvokerServiceExporter** and **HttpInvokerProxyFactoryBean**.

Click here to get details about [Spring Remoting by Http Invoker](https://www.javatpoint.com/spring-remoting-by-http-invoker-example)

### Hessian

It also provides remoting service by using http protocol. It is provided by Coucho. The classes used in Hessian are **HessianServiceExporter** and **HessianProxyFactoryBean**.

Click here to get details about [Spring Remoting by Hessian](https://www.javatpoint.com/spring-remoting-by-hessian-example)

### Burlap

It is same as Hessian but XML-based implementation provided by Coucho. The classes used in Burlap are **BurlapServiceExporter** and **BurlapProxyFactoryBean**.

Click here to get details about [Spring Remoting by Burlap](https://www.javatpoint.com/spring-remoting-by-burlap-example)

### JAX-RPC

Spring provides remoting support for web services using JAX-RPC. It uses J2EE 1.4 API.

### JAX-WS

It is the successor of JAX-RPC. It uses Java EE 5 and Java EE 6 API. The classes used in JAX-WS are **SimpleJaxWsServiceExporter** and **JaxWsPortProxyFactoryBean**.

### JMS

Spring supports remoting service using JMS. The classes used in JMS are **JmsInvokerServiceExporter** and **JmsInvokerProxyFactoryBean**.

# **Spring and RMI Integration**

1. [Spring and RMI Integration](https://www.javatpoint.com/spring-and-rmi-integration)
2. [Example of Spring and RMI Integration](https://www.javatpoint.com/spring-and-rmi-integration#ex)

Spring RMI lets you expose your services through the RMI infrastructure.

Spring provides an easy way to run RMI application by the help of org.springframework.remoting.rmi.**RmiProxyFactoryBean** and org.springframework.remoting.rmi.**RmiServiceExporter** classes.

#### RmiServiceExporter

It provides the exportation service for the rmi object. This service can be accessed via plain RMI or RmiProxyFactoryBean.

#### RmiProxyFactoryBean

It is the factory bean for Rmi Proxies. It exposes the proxied service that can be used as a bean reference.

### Example of Spring and RMI Integration

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

1. **Calculation.java**
2. **CalculationImpl.java**
3. **applicationContext.xml**
4. **client-beans.xml**
5. **Host.java**
6. **Client.java**

#### Required Jar files

To run this example, you need to load:

* **Spring Core jar files**
* **Spring Remoting jar files**
* **Spring AOP jar files**

[download all the jar files for spring including core, web, aop, mvc, j2ee, remoting, oxm, jdbc, orm etc.](https://static.javatpoint.com/src/sp/springjars.zip)

**1) Calculation.java**

It is the simple interface containing one method cube.

1. **package** com.javatpoint;
3. **public** **interface** Calculation {
4. **int** cube(**int** number);
5. }

**2) CalculationImpl.java**

This class provides the implementation of Calculation interface.

1. **package** com.javatpoint;
3. **public** **class** CalculationImpl **implements** Calculation{
5. @Override
6. **public** **int** cube(**int** number) {
7. **return** number\*number\*number;
8. }
10. }

**3) applicationContext.xml**

In this xml file we are defining the bean for CalculationImpl class and **RmiServiceExporter** class. We need to provide values for the following properties of RmiServiceExporter class.

1. service
2. serviceInterface
3. serviceName
4. replaceExistingBinding
5. registryPort
6. <?xml version="1.0" encoding="UTF-8"?>
7. <beans xmlns="http://www.springframework.org/schema/beans"
8. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9. xsi:schemaLocation="http://www.springframework.org/schema/beans
10. http://www.springframework.org/schema/beans/spring-beans.xsd">
12. <bean id="calculationBean" **class**="com.javatpoint.CalculationImpl"></bean>
13. <bean **class**="org.springframework.remoting.rmi.RmiServiceExporter">
14. <property name="service" ref="calculationBean"></property>
15. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
16. <property name="serviceName" value="CalculationService"></property>
17. <property name="replaceExistingBinding" value="true"></property>
18. <property name="registryPort" value="1099"></property>
19. </bean>
20. </beans>

**4) client-beans.xml**

In this xml file, we are defining bean for **RmiProxyFactoryBean**. You need to define two properties of this class.

1. serviceUrl
2. serviceInterface
3. <?xml version="1.0" encoding="UTF-8"?>
4. <beans xmlns="http://www.springframework.org/schema/beans"
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans.xsd">
9. <bean id="calculationBean" **class**="org.springframework.remoting.rmi.RmiProxyFactoryBean">
10. <property name="serviceUrl" value="rmi://localhost:1099/CalculationService"></property>
11. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
12. </bean>
13. </beans>

**5) Host.java**

It is simply getting the instance of ApplicationContext. But you need to run this class first to run the example.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Host{
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");
8. System.out.println("Waiting for requests");
9. }
10. }

**6) Client.java**

This class gets the instance of Calculation and calls the method.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Client {
6. **public** **static** **void** main(String[] args)  {
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("client-beans.xml");
8. Calculation calculation = (Calculation)context.getBean("calculationBean");
9. System.out.println(calculation.cube(7));
10. }
11. }

# **Spring Remoting by HTTP Invoker Example**

1. [Spring Remoting by HTTP Invoker](https://www.javatpoint.com/spring-remoting-by-http-invoker-example)
2. [Example of Spring HTTP Invoker](https://www.javatpoint.com/spring-remoting-by-http-invoker-example#ex)

Spring provides its own implementation of remoting service known as **HttpInvoker**. It can be used for http request than RMI and works well across the firewall.

By the help of **HttpInvokerServiceExporter** and **HttpInvokerProxyFactoryBean** classes, we can implement the remoting service provided by Spring's Http Invokers.

### Http Invoker and Other Remoting techniques

You can use many Remoting techniques, let's see which one can be best for you.

#### Http Invoker Vs RMI

RMI uses JRMP protocol whereas Http Invokes uses HTTP protocol. Since enterprise applications mostly use http protocol, it is the better to use HTTP Invoker. RMI also has some security issues than HTTP Invoker. HTTP Invoker works well across firewalls.

#### Http Invoker Vs Hessian and Burlap

HTTP Invoker is the part of Spring framework but Hessian and burlap are proprietary. All works well across firewall. Hessian and Burlap are portable to integrate with other languages such as .Net and PHP but HTTP Invoker cannot be.

### Example of Spring HTTP Invoker

To create a simple spring's HTTP invoker application, you need to create following files.

1. **Calculation.java**
2. **CalculationImpl.java**
3. **web.xml**
4. **httpInvoker-servlet.xml**
5. **client-beans.xml**
6. **Client.java**

**1) Calculation.java**

It is the simple interface containing one method cube.

1. **package** com.javatpoint;
2. **public** **interface** Calculation {
3. **int** cube(**int** number);
4. }

**2) CalculationImpl.java**

This class provides the implementation of Calculation interface.

1. **package** com.javatpoint;
2. **public** **class** CalculationImpl **implements** Calculation{
3. **public** **int** cube(**int** number) {
4. **return** number\*number\*number;
5. }
6. }

**3) web.xml**

In this xml file, we are defining DispatcherServlet as the front controller. If any request is followed by .http extension, it will be forwarded to DispatcherServlet.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app version="2.5"
3. xmlns="http://java.sun.com/xml/ns/javaee"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">
8. <servlet>
9. <servlet-name>httpInvoker</servlet-name>
10. <servlet-**class**>org.springframework.web.servlet.DispatcherServlet</servlet-**class**>
11. <load-on-startup>1</load-on-startup>
12. </servlet>
14. <servlet-mapping>
15. <servlet-name>httpInvoker</servlet-name>
16. <url-pattern>\*.http</url-pattern>
17. </servlet-mapping>
19. </web-app>

**4) httpInvoker-servlet.xml**

It must be created inside the WEB-INF folder. Its name must be servletname-servlet.xml. It defines bean for **CalculationImpl** and **HttpInvokerServiceExporter**.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans.xsd">
7. <bean id="calculationBean" **class**="com.javatpoint.CalculationImpl"></bean>
8. <bean name="/Calculation.http"
9. **class**="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
10. <property name="service" ref="calculationBean"></property>
11. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
12. </bean>
14. </beans>

**5) client-beans.xml**

In this xml file, we are defining bean for **HttpInvokerProxyFactoryBean**. You need to define two properties of this class.

1. serviceUrl
2. serviceInterface
3. <?xml version="1.0" encoding="UTF-8"?>
4. <beans xmlns="http://www.springframework.org/schema/beans"
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans.xsd">
9. <bean id="calculationBean"
10. **class**="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
11. <property name="serviceUrl"
12. value="http://localhost:8888/httpinvoker/Calculation.http"></property>
13. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
14. </bean>
15. </beans>

**6) Client.java**

This class gets the instance of Calculation and calls the method.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Client {
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("client-beans.xml");
8. Calculation calculation = (Calculation)context.getBean("calculationBean");
9. System.out.println(calculation.cube(5));
10. }
11. }

### Output

1. Output: 125

### How to run this example

Start and deploy the project, here we are assuming that server is running on 8888 port number. If the port number is different, change the serviceURL in client-beans.xml.

Then, Compile and Run the Client.java file.

[download this example (developed using Myeclipse IDE)](https://static.javatpoint.com/src/sp/httpinvoker.zip)

## **Web-based Client**

In the example given above, we used console based client. We can also use web based client. You need to create 3 additional files. Here, we are using following files:

1. ClientInvoker.java
2. index.jsp
3. process.jsp

**ClientInvoker.java**

It defines only one method getCube() that returns cube of the given number

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** ClientInvoker {
6. **public** **static** **int** getCube(**int** number){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("client-beans.xml");
8. Calculation calculation = (Calculation)context.getBean("calculationBean");
9. **return** calculation.cube(number);
10. }
11. }

**index.jsp**

It creates a form to get number.

1. <form action="process.jsp">
2. Enter Number:<input type="text" name="number"/>
3. <input type="submit" value="cube" />
4. </form>

**process.jsp**

It creates a form to get number.

1. <jsp:include page="index.jsp"></jsp:include>
2. <hr/>
3. <%@page **import**="com.javatpoint.ClientInvoker"%>
5. <%
6. **int** number=Integer.parseInt(request.getParameter("number"));
7. out.print("cube of "+number+" is: "+ClientInvoker.getCube(number));
8. %>

### Output

# A computer screen with a blue and white box Description automatically generated **Spring Remoting by Hessian Example**

1. [Spring Remoting by Hessian](https://www.javatpoint.com/spring-remoting-by-hessian-example)
2. [Example of Spring Hessian](https://www.javatpoint.com/spring-remoting-by-hessian-example#ex)

By the help of **HessianServiceExporter** and **HessianProxyFactoryBean** classes, we can implement the remoting service provided by hessian.

#### Advantage of Hessian

Hessian works well across firewall. Hessian is portable to integrate with other languages such as PHP and .Net.

### Example of Remoting by Hessian

You need to create following files for creating a simple hessian application:

1. **Calculation.java**
2. **CalculationImpl.java**
3. **web.xml**
4. **hessian-servlet.xml**
5. **client-beans.xml**
6. **Client.java**

**1) Calculation.java**

It is the simple interface containing one method cube.

1. **package** com.javatpoint;
2. **public** **interface** Calculation {
3. **int** cube(**int** number);
4. }

**2) CalculationImpl.java**

This class provides the implementation of Calculation interface.

1. **package** com.javatpoint;
2. **public** **class** CalculationImpl **implements** Calculation{
3. **public** **int** cube(**int** number) {
4. **return** number\*number\*number;
5. }
6. }

**3) web.xml**

In this xml file, we are defining DispatcherServlet as the front controller. If any request is followed by .http extension, it will be forwarded to DispatcherServlet.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app version="2.5"
3. xmlns="http://java.sun.com/xml/ns/javaee"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">
8. <servlet>
9. <servlet-name>hessian</servlet-name>
10. <servlet-**class**>org.springframework.web.servlet.DispatcherServlet</servlet-**class**>
11. <load-on-startup>1</load-on-startup>
12. </servlet>
14. <servlet-mapping>
15. <servlet-name>hessian</servlet-name>
16. <url-pattern>\*.http</url-pattern>
17. </servlet-mapping>
19. </web-app>

**4) hessian-servlet.xml**

It must be created inside the WEB-INF folder. Its name must be servletname-servlet.xml. It defines bean for **CalculationImpl** and **HessianServiceExporter**.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans.xsd">
7. <bean id="calculationBean" **class**="com.javatpoint.CalculationImpl"></bean>
8. <bean name="/Calculation.http"
9. **class**="org.springframework.remoting.caucho.HessianServiceExporter">
10. <property name="service" ref="calculationBean"></property>
11. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
12. </bean>
14. </beans>

**5) client-beans.xml**

In this xml file, we are defining bean for **HessianProxyFactoryBean**. You need to define two properties of this class.

1. serviceUrl
2. serviceInterface
3. <?xml version="1.0" encoding="UTF-8"?>
4. <beans xmlns="http://www.springframework.org/schema/beans"
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans.xsd">
9. <bean id="calculationBean"
10. **class**="org.springframework.remoting.caucho.HessianProxyFactoryBean">
11. <property name="serviceUrl"
12. value="http://localhost:8888/hessian/Calculation.http"></property>
13. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
14. </bean>
15. </beans>

In this example, our project name is hessian, i.e. used as the context root in the serviceURL.

**6) Client.java**

This class gets the instance of Calculation and calls the cube method.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Client {
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("client-beans.xml");
8. Calculation calculation = (Calculation)context.getBean("calculationBean");
9. System.out.println(calculation.cube(5));
10. }
11. }

### How to run this example

Start and deploy the project, here we are assuming that server is running on 8888 port number. If the port number is different, change the serviceURL in client-beans.xml.

Then, Compile and Run the Client.java file.

# **Spring Remoting by Burlap Example**

1. [Spring Remoting by Burlap](https://www.javatpoint.com/spring-remoting-by-burlap-example)
2. [Example of Spring Burlap](https://www.javatpoint.com/spring-remoting-by-burlap-example#ex)

Both, Hessian and Burlap are provided by Coucho. Burlap is the xml-based alternative of Hessian.

By the help of **BurlapServiceExporter** and **BurlapProxyFactoryBean** classes, we can implement the remoting service provided by burlap.

#### Example of Burlap is same as Hessian, you need to change Hessian to Burlap only.

### Example of Remoting by Burlap

You need to create following files for creating a simple burlap application:

1. **Calculation.java**
2. **CalculationImpl.java**
3. **web.xml**
4. **burlap-servlet.xml**
5. **client-beans.xml**
6. **Client.java**

**1) Calculation.java**

It is the simple interface containing one method cube.

1. **package** com.javatpoint;
2. **public** **interface** Calculation {
3. **int** cube(**int** number);
4. }

**2) CalculationImpl.java**

This class provides the implementation of Calculation interface.

1. **package** com.javatpoint;
2. **public** **class** CalculationImpl **implements** Calculation{
3. **public** **int** cube(**int** number) {
4. **return** number\*number\*number;
5. }
6. }

**3) web.xml**

In this xml file, we are defining DispatcherServlet as the front controller. If any request is followed by .http extension, it will be forwarded to DispatcherServlet.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app version="2.5"
3. xmlns="http://java.sun.com/xml/ns/javaee"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">
8. <servlet>
9. <servlet-name>burlap</servlet-name>
10. <servlet-**class**>org.springframework.web.servlet.DispatcherServlet</servlet-**class**>
11. <load-on-startup>1</load-on-startup>
12. </servlet>
14. <servlet-mapping>
15. <servlet-name>burlap</servlet-name>
16. <url-pattern>\*.http</url-pattern>
17. </servlet-mapping>
19. </web-app>

**4) burlap-servlet.xml**

It must be created inside the WEB-INF folder. Its name must be servletname-servlet.xml. It defines bean for **CalculationImpl** and **BurlapServiceExporter**.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans.xsd">
7. <bean id="calculationBean" **class**="com.javatpoint.CalculationImpl"></bean>
8. <bean name="/Calculation.http"
9. **class**="org.springframework.remoting.caucho.BurlapServiceExporter">
10. <property name="service" ref="calculationBean"></property>
11. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
12. </bean>
14. </beans>

**5) client-beans.xml**

In this xml file, we are defining bean for **BurlapProxyFactoryBean**. You need to define two properties of this class.

1. serviceUrl
2. serviceInterface
3. <?xml version="1.0" encoding="UTF-8"?>
4. <beans xmlns="http://www.springframework.org/schema/beans"
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans.xsd">
9. <bean id="calculationBean"
10. **class**="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
11. <property name="serviceUrl"
12. value="http://localhost:8888/burlap/Calculation.http"></property>
13. <property name="serviceInterface" value="com.javatpoint.Calculation"></property>
14. </bean>
15. </beans>

In this example, our project name is burlap, i.e. used as the context root in the serviceURL.

**6) Client.java**

This class gets the instance of Calculation and calls the cube method.

1. **package** com.javatpoint;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Client {
6. **public** **static** **void** main(String[] args){
7. ApplicationContext context = **new** ClassPathXmlApplicationContext("client-beans.xml");
8. Calculation calculation = (Calculation)context.getBean("calculationBean");
9. System.out.println(calculation.cube(3));
10. }
11. }

### How to run this example

Start and deploy the project, here we are assuming that server is running on 8888 port number. If the port number is different, change the serviceURL in client-beans.xml.

Then, Compile and Run the Client.java file.

# **Spring and JMS Integration**

To integrate spring with JMS, you need to create two applications.

1. JMS Receiver Application
2. JMS Sender Application

To create JMS application using spring, we are using **Active MQ Server** of Apache to create the Queue.

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

### Required Jar Files

1) You need to add **spring core**, **spring misc**,**spring aop**, **spring j2ee** and **spring persistence core** jar files.

[download the all jar files for spring including aop, mvc, j2ee, remoting, oxm, etc.](https://static.javatpoint.com/src/sp/springjars.zip)

2) Add **activemqall5.9.jar** file located inside the activemq directory.

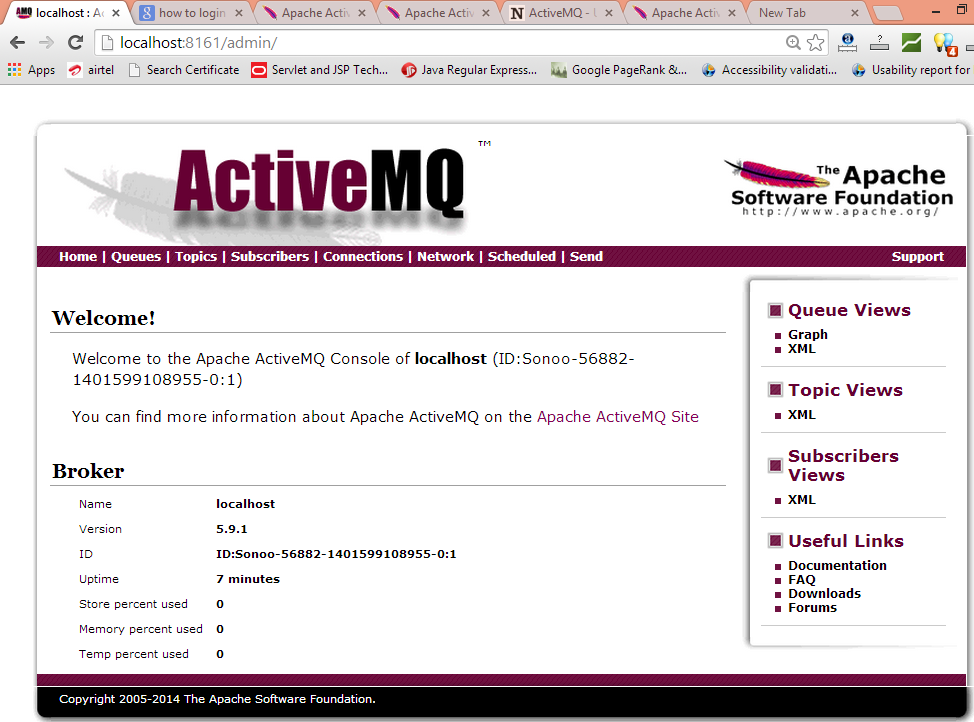
### Create a queue in ActiveMQ Server

Download the Active MQ Server [Download Active MQ](http://www.apache.org/dyn/closer.cgi?path=/activemq/5.9.1/apache-activemq-5.9.1-bin.zip)

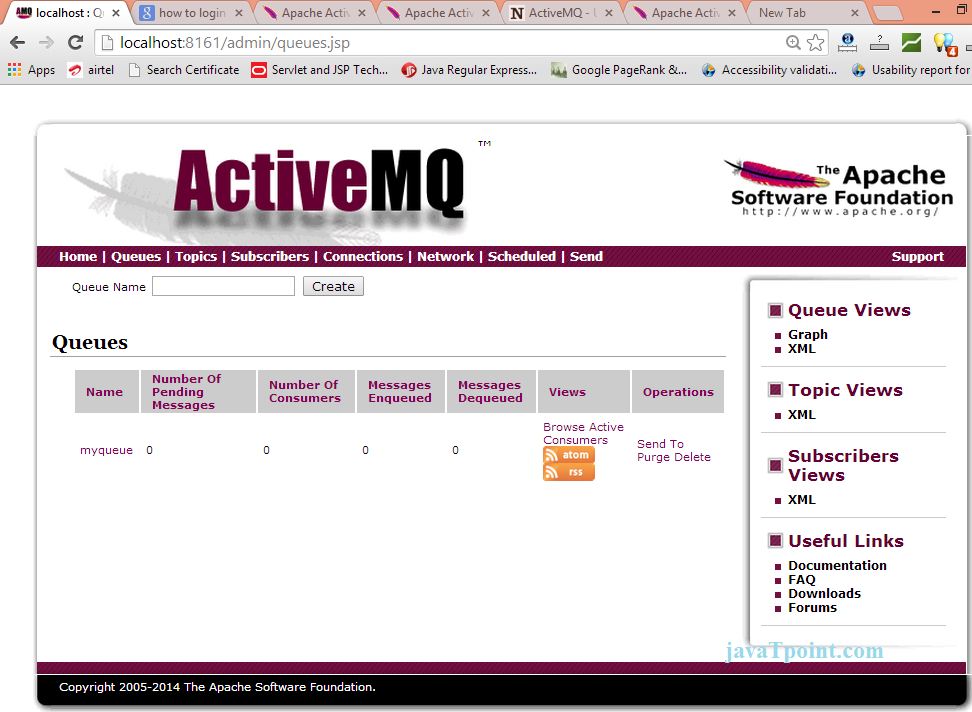
Double Click on the **activemq.bat** file located inside apache-activemq-5.9.1-bin\apache-activemq-5.9.1\bin\win64 or win32 directory.

Now activemq server console will open.

Access the admin console of activemq server by **http://localhost:8161/admin/** url.



Now, click on the **Queues link**, write **myqueue** in the textfield and click on the create button.



### 1) JMS Receiver Application

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

1. **MyMessageListener.java**
2. **TestListener.java**
3. **applicationContext.xml**

**1) MyMessageListener.java**

1. **package** com.javatpoint;
2. **import** javax.jms.Message;
3. **import** javax.jms.MessageListener;
4. **import** javax.jms.TextMessage;
5. **public** **class** MyMessageListener **implements** MessageListener{
6. @Override
7. **public** **void** onMessage(Message m) {
8. TextMessage message=(TextMessage)m;
9. **try**{
10. System.out.println(message.getText());
11. }**catch** (Exception e) {e.printStackTrace();  }
12. }
13. }

**2) TestListener.java**

1. **package** com.javatpoint;
2. **import** org.springframework.context.support.GenericXmlApplicationContext;
3. **public** **class** TestListener {
4. **public** **static** **void** main(String[] args) {
5. GenericXmlApplicationContext ctx=**new** GenericXmlApplicationContext();
6. ctx.load("classpath:applicationContext.xml");
7. ctx.refresh();
9. **while**(**true**){}
10. }
11. }

**3) applicationContext.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans**
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:jms="http://www.springframework.org/schema/jms"
7. xmlns:p="http://www.springframework.org/schema/p"
8. xsi:schemaLocation="http://www.springframework.org/schema/beans
9. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
10. http://www.springframework.org/schema/jms
11. http://www.springframework.org/schema/jms/spring-jms-3.0.xsd"**>**
13. **<bean** id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory"
14. p:brokerURL="tcp://localhost:61616" **/>**
16. **<bean** id="listener" class="com.javatpoint.MyMessageListener"**></bean>**
18. **<jms:listener-container** container-type="default" connection-factory="connectionFactory"
19. acknowledge="auto"**>**
20. **<jms:listener** destination="myqueue" ref="listener" method="onMessage"**></jms:listener>**
21. **</jms:listener-container>**
23. **</beans>**

### 2) JMS Sender Application

Let's see the files to create the JMS Sender application:

1. **MyMessageSender.java**
2. **TestJmsSender.java**
3. **applicationContext.xml**

**1) MyMessageListener.java**

1. **package** com.javatpoint;
2. **import** javax.jms.\*;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.jms.core.JmsTemplate;
5. **import** org.springframework.jms.core.MessageCreator;
6. **import** org.springframework.stereotype.Component;
8. @Component("messageSender")
9. **public** **class** MyMessageSender {
10. @Autowired
11. **private** JmsTemplate jmsTemplate;
12. **public** **void** sendMessage(**final** String message){
13. jmsTemplate.send(**new** MessageCreator(){
15. @Override
16. **public** Message createMessage(Session session) **throws** JMSException {
17. **return** session.createTextMessage(message);
18. }
19. });
20. }
21. }

**2) TestJmsSender.java**

1. **package** com.javatpoint;
2. **import** org.springframework.context.support.GenericXmlApplicationContext;
3. **public** **class** TestJmsSender {
4. **public** **static** **void** main(String[] args) {
5. GenericXmlApplicationContext ctx=**new** GenericXmlApplicationContext();
6. ctx.load("classpath:applicationContext.xml");
7. ctx.refresh();
9. MyMessageSender sender=ctx.getBean("messageSender",MyMessageSender.**class**);
10. sender.sendMessage("hello jms3");
12. }
13. }

**3) applicationContext.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans**
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:context="http://www.springframework.org/schema/context"
6. xmlns:jms="http://www.springframework.org/schema/jms"
8. xmlns:p="http://www.springframework.org/schema/p"
9. xsi:schemaLocation="http://www.springframework.org/schema/beans
10. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
11. http://www.springframework.org/schema/context
12. http://www.springframework.org/schema/context/spring-context-3.0.xsd
13. http://www.springframework.org/schema/jms
14. http://www.springframework.org/schema/jms/spring-jms-3.0.xsd"**>**
16. **<bean** id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory"
17. p:brokerURL="tcp://localhost:61616" **/>**
19. **<bean** id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate"**>**
20. **<constructor-arg** name="connectionFactory" ref="connectionFactory"**></constructor-arg>**
21. **<property** name="defaultDestinationName" value="myqueue"**></property>**
22. **</bean>**
24. **<context:component-scan** base-package="com.javatpoint"**></context:component-scan>**
26. **</beans>**

# **Spring Security Tutorial**



Spring Security Tutorial provides basic and advanced concepts of Spring Security. Our Spring Security Tutorial is designed for beginners and professionals both.

Our Spring Security Tutorial includes all topics of Spring Security such as spring security introduction, features, project modules, xml example, java example, login logout, spring boot etc.

## **Prerequisite**

To learn Spring Security, you must have the basic knowledge of HTML and CSS.

## **Audience**

Our Spring Security Tutorial is designed to help beginners and professionals.

## **Problem**

We assure that you will not find any problem in this Spring Security tutorial. But if there is any mistake, please post the problem in contact form.