Spring Tutorial

This spring tutorial provides in-depth concepts of Spring Framework with simplified examples. It was **developed by Rod Johnson in 2003**. Spring framework makes the easy development of JavaEE application.

It is helpful for beginners and experienced persons.

Spring Framework

Spring is a *lightweight* framework.

It can be thought of as a *framework of frameworks* because
it provides support to various frameworks such as Struts
, Hibernate
, Tapestry, EJB
, JSF
, 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.

Inversion Of Control (IOC) and Dependency Injection

These are the design patterns that are used to remove dependency from the programming code

. They make the code easier to test and maintain. Let's understand this with the following code:

```
1. class Employee{
2. Address address;
3. Employee(){
```

```
4. address=new Address();


5. }
6. }
```

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

```
1. class Employee{
2. Address address;
3. Employee(Address address){
4. this.address=address;
5. }
6. }
```

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

.

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

.

Advantage of Dependency Injection

- makes the code loosely coupled so

  easy to maintain

- makes the code easy to test

Advantages of Spring Framework

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

1) Predefined Templates

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

.

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

.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

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

4) Lightweight

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

.

5) Fast Development

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

6) Powerful abstraction

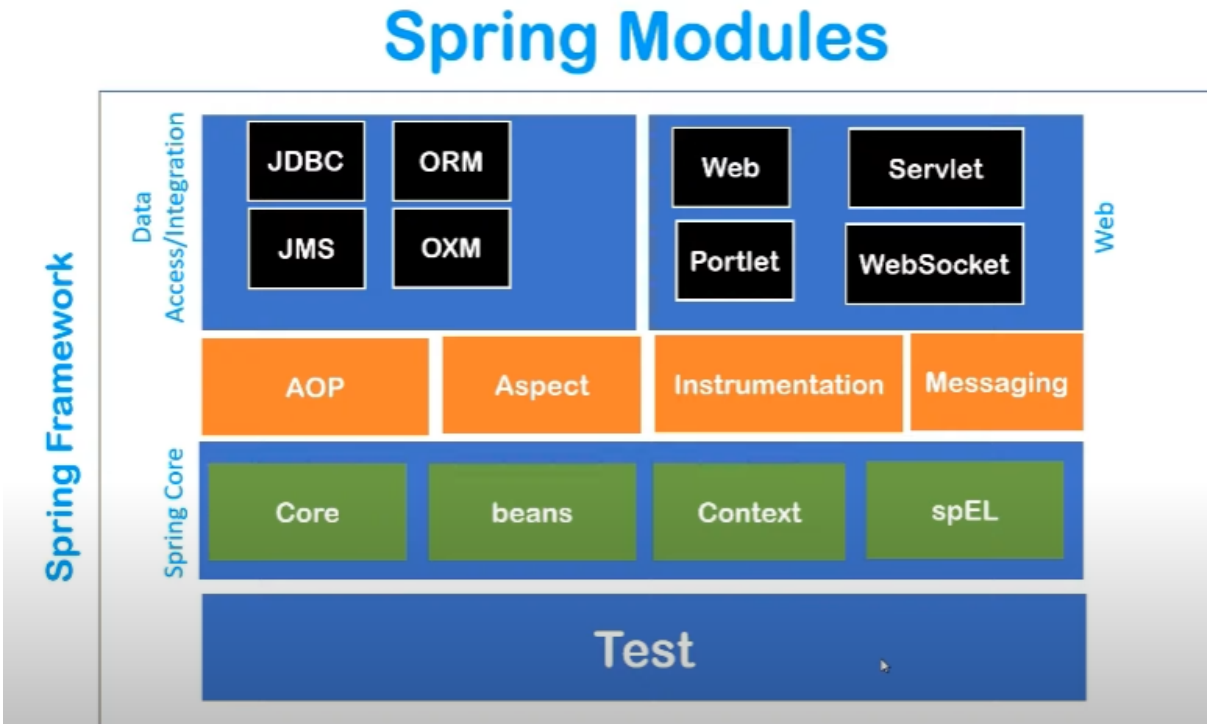
It provides powerful abstraction to JavaEE specifications such as JMS

, JDBC

, JPA and JTA.

7) Declarative support

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



Test

This layer provides support of testing with JUnit and TestNG.

Spring Core Container

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

Core and Beans

These modules provide IOC and Dependency Injection features.

Context

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

Expression Language

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

AOP, Aspects and Instrumentation

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

The aspects module provides support to integration with AspectJ.

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

Data Access / Integration

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

Web

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

Spring Example

Here, we are going to learn the simple steps to create the first spring application. To run this application, we are not using any IDE. We are simply using the command prompt. Let's see the simple steps to create the spring application

- **create the class**

- **create the xml file to provide the values**
- **create the test class**
- **Load the spring jar files**
- **Run the test class**

Steps to create spring application

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

1) Create Java class

This is the simple java bean class containing the name property only.

```
1. package com.javatpoint;
2.
3. public class Student {
4. private String name;
5.
6. public String getName() {
7.     return name;
8. }
9.
10. public void setName(String name) {
11.     this.name = name;
12. }
13.
14. public void displayInfo(){
15.     System.out.println("Hello: "+name);
16. }
17. }
```

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.

2) Create the xml file

In case of myeclipse IDE, you don't need to create the xml file as myeclipse does this for yourselves. 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="com.javatpoint.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** subelement 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.

3) 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 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;
7.
8. public class Test {
9. public static void main(String[] args) {
10.     Resource resource=new ClassPathResource("applicationContext.xml");
11.     BeanFactory factory=new XmlBeanFactory(resource);
12.
13.     Student student=(Student)factory.getBean("studentbean");
14.     student.displayInfo();
15. }
16. }
```

The **Resource** object represents the information of applicationContext.xml file. The Resource is the interface and the **ClassPathResource** is the implementation class of the Reource interface. The **BeanFactory** is responsible to return the bean. The **XmlBeanFactory** is the implementation class of the BeanFactory. There are many methods in the BeanFactory interface. One method is **getBean()**, which returns the object of the associated class.

4) Load the jar files required for spring framework

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.

Creating spring application in Eclipse IDE

1. Creating spring application in Eclipse
2. Steps to create spring application in Eclipse

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

Go to **File** menu - **New** - **project** - **Java Project**. Write the project name e.g. firstspring - **Finish**. Now the java project is created.

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

download the all jar files for spring including aop, mvc, j2ee, remoting, oxm, etc.

To run this example, you need to load only spring core jar files.

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

In such case, we are simply creating the Student class have name property. The name of the student will be provided by the xml file. It is just a simple example not the actual use of spring. We will see the actual use in Dependency Injection chapter. To create the java class, **Right click on src - New - class - Write the class name e.g. Student - finish**. Write the following code:

```
1. package com.javatpoint;
2.
3. public class Student {
4. private String name;
5.
6. public String getName() {
7.     return name;
8. }
9.
10. public void setName(String name) {
11.     this.name = name;
12. }
13.
14. public void displayInfo(){
15.     System.out.println("Hello: "+name);
16. }
17. }
```

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:

```
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">
8.
9.  <bean id="studentbean" class="com.javatpoint.Student">
10. <property name="name" value="Vimal Jaiswal"></property>
11. </bean>
12.
13. </beans>
```

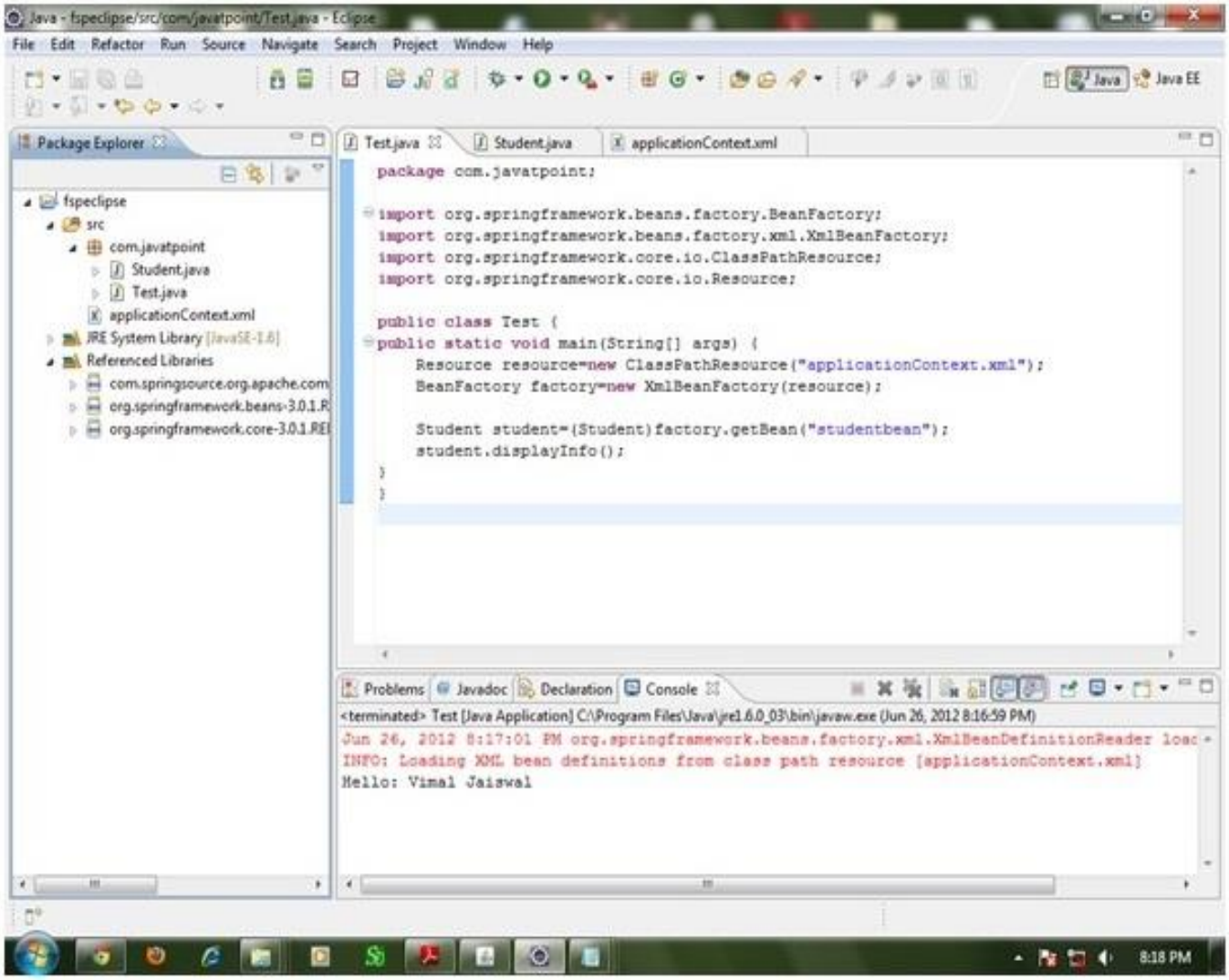
The **bean** element is used to define the bean for the given class. The **property** subelement of bean specifies the property of the Student class named name. The value specified in the property element will be set in the Student class object by the IOC container.

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.

```
1.  package com.javatpoint;
2.
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;
7.
8.  public class Test {
9.  public static void main(String[] args) {
10.    Resource resource=new ClassPathResource("applicationContext.xml");
11.    BeanFactory factory=new XmlBeanFactory(resource);
12.
13.    Student student=(Student)factory.getBean("studentbean");
14.    student.displayInfo();
15. }
16. }
```

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



IoC Container

1. IoC Container
2. Using BeanFactory
3. Using ApplicationContext

The IoC container

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

- to instantiate the application class
- to configure the object

- to assemble the dependencies between the objects

There are two types of IoC containers. They are:

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

Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.**BeanFactory** and the org.springframework.context.**ApplicationContext** interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface

.
It adds some extra functionality than BeanFactory
such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

Using BeanFactory

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

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

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

Using ApplicationContext

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

```
1. ApplicationContext context =
```

```
2.      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

1. Dependency Injection in Spring
2. Dependency Lookup
3. Dependency Injection

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. D

ependency 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;
3.
4. Employee(Address address){
5. this.address=address;
6. }
7. public void setAddress(Address address){
8. this.address=address;
9. }
10.
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 Example

1. Dependency Injection by constructor

2. Injecting primitive and string-based values

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.

```
1.  package com.javatpoint;
2.
3.  public class Employee {
4.  private int id;
5.  private String name;
6.
7.  public Employee() {System.out.println("def cons");}
8.
9.  public Employee(int id) {this.id = id;}
10.
11. public Employee(String name) {  this.name = name;}
12.
13. public Employee(int id, String name) {
14.     this.id = id;
15.     this.name = name;
16. }
```

```
17.
18. void show(){
19.     System.out.println(id+" "+name);
20. }
21.
22. }
```

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.

```
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">
8.
9. <bean id="e" class="com.javatpoint.Employee">
10. <constructor-arg value="10" type="int"></constructor-arg>

11. </bean>
12.
13. </beans>
```

Test.java

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

```
1. package com.javatpoint;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.*;
```

```
6.
7.  public class Test {
8.      public static void main(String[] args) {
9.
10.         Resource r=new ClassPathResource("applicationContext.xml");
11.         BeanFactory factory=new XmlBeanFactory(r);
12.
13.         Employee s=(Employee)factory.getBean("e");
14.         s.show();
15.
16.     }
17. }
```

Output:10 null

Dependency Injection by setter method

1. Dependency Injection by constructor
2. Injecting primitive and string-based values

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.

```
1.  package com.javatpoint;
2.
3.  public class Employee {
4.  private int id;
5.  private String name;
6.  private String city;
7.
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.
21. public String getCity() {
22.     return city;
23. }
24. public void setCity(String city) {
25.     this.city = city;
26. }
27. void display(){
28.     System.out.println(id+" "+name+" "+city);
29. }
30.
31. }
```

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.

```
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">
8.
9. <bean id="obj" class="com.javatpoint.Employee">
10. <property name="id">
11. <value>20</value>
12. </property>
13. <property name="name">
14. <value>Arun</value>
15. </property>
16. <property name="city">
17. <value>ghaziabad</value>
18. </property>
19.
20. </bean>
21.
22. </beans>
```

Test.java

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

```
1. package com.javatpoint;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
```

```
5.  import org.springframework.core.io.*;
6.
7.  public class Test {
8.      public static void main(String[] args) {
9.
10.         Resource r=new ClassPathResource("applicationContext.xml");
11.         BeanFactory factory=new XmlBeanFactory(r);
12.
13.         Employee e=(Employee)factory.getBean("obj");
14.         s.display();
15.
16.     }
17. }
```

Output:20 Arun Ghaziabad

Constructor injection Vs. Setter injection

Setter Injection	Constructor Injection
In Setter Injection, partial injection of dependencies can possible, means if we have 3 dependencies like int, string, long, then its not necessary to inject all values if we use setter injection. If you are not inject it will takes default values for those primitives.	In constructor injection, partial injection of dependencies cannot possible, because for calling constructor we must pass all the arguments, if not so we may get error.
Setter Injection will overrides the constructor injection value, provided if we write setter and constructor injection for the same property	But, constructor injection cannot overrides the setter injected values
We can use Setter injection when a number of dependencies are more or you need readability.	If readability is not a concern then we can use Constructor injection when a number of dependencies are more.
Setter injection makes bean class object as mutable [We can change]	Constructor injection makes bean class object as immutable [We cannot change]

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.

```
1. package org.sssit;
2. public class B {
3. B(){System.out.println("b is created");}
4. void print(){System.out.println("hello b");}
5. }
```

A.java

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

```
1. package org.sssit;
2. public class A {
3. B b;
4. A(){System.out.println("a is created");}
5. public B getB() {
```

```
6.      return b;
7.  }
8.  public void setB(B b) {
9.      this.b = b;
10. }
11. void print(){System.out.println("hello a");}
12. void display(){
13.     print();
14.     b.print();
15. }
16. }
```

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">
8.
9.  <bean id="b" class="org.sssit.B"></bean>
10. <bean id="a" class="org.sssit.A" autowire="byName"></bean>
11.
12. </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the display 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("applicat
    ionContext.xml");
```

```
7.      A a=context.getBean("a",A.class);
8.      a.display();
9.  }
10. }
```

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>
```

Spring Annotation

@Required

- This is used to make setter injection as mandatory.
- This is method level annotation.

Employee.java

```
package com.cdac.beans;

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

```
public class Employee {
	private String empName;
	private String companyName;

	@Required
	public void setEmpName(String empName) {
		this.empName = empName;
	}

	public void setCompanyName(String companyName) {
		this.companyName = companyName;
	}

	@Override
	public String toString() {
		return "Employee [empName=" + empName + ", companyName=" +
companyName + "]";
	}

}
```

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

	<bean id="employee" class="com.cdac.beans.Employee"/>

	<bean
class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcess
or" />

	<context:annotation-config/>

</beans>
```

Test.java

```
package com.cdac.test;

import org.springframework.context.ApplicationContext;
```

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

import com.cdac.beans.Employee;

public class Test {
	public static void main(String[] args) {
		ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("com/cdac/common/application-context.xml");
		Employee employee =
applicationContext.getBean("employee",Employee.class);
		System.out.println(employee);
	}
}
```

@Autowired

- Instead of we declaring the dependency between the bean definition, if IOC container automatically detect and inject them then it is called auto-wiring.
- @Autowired is used for wiring object type dependencies into target class.
- @Autowired will identify dependencies by matching with type.
- The dependent object will be injected into target class based on place where we wrote.
- We can write it in 4 different places:
  - Attribute
  - Setter
  - Constructor

Employee.java

```
package com.cdac.beans;

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

public class Employee {
	@Autowired
	Address address;
	/*
	@Autowired
	public Employee(Address address) {
		this.address = address;
	}
	*/
	/*
	@Autowired
	public void setAddress(Address address) {
```

```
		this.address = address;
	}*/

	public void show() {
		address.showAddress();
	}
}
```

Address.java

```
package com.cdac.beans;

public class Address {
	public void showAddress() {
		System.out.println("Address class method");
	}
}
```

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

	<bean id="employee" class="com.cdac.beans.Employee"/>
	<bean id="address" class="com.cdac.beans.Address"/>

	<context:annotation-config/>

</beans>
```

Test.java

```
package com.cdac.test;

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

import com.cdac.beans.Employee;

public class Test {
	public static void main(String[] args) {
```

```
		ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("com/cdac/common/application-context.xml");
		Employee employee =
applicationContext.getBean("employee",Employee.class);
		employee.show();
	}
}
```

Stereotype Annotation

- These annotation are used for making a class as bean definition.
- There are 5 types stereotype annotation:
  - @Component
  - @Controller
  - @Service
  - @Repository
  - @RestController

@Component

Student.java

```
package com.cdac.beans;

import org.springframework.stereotype.Component;

@Component
public class Student {
	public Student() {
		System.out.println("student class constructor...");
	}
}
```

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
```

```
        <context:component-scan base-package="com.cdac.beans"/>

</beans>
```

Test.java

```
package com.cdac.test;

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

import com.cdac.beans.Student;

public class Test {
        public static void main(String[] args) {
                ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("com/cdac/common/application-context.xml");
                Student student = applicationContext.getBean("student",Student.class);
        }
}
```

Annotation without XML configuration

Student.java

```
package com.cdac.beans;

import org.springframework.stereotype.Component;

@Component
public class Student {
        public Student() {
                System.out.println("Student class constructor");
        }
}
```

Test.java

```
package com.cdac.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.cdac.beans.Student;

public class Test {
        public static void main(String[] args) {
                ApplicationContext applicationContext = new
AnnotationConfigApplicationContext("com.cdac.beans");
                Student s = applicationContext.getBean("student",Student.class);
```

```
	}
}
```

Note

- If we don't have source of the class. It is not possible to go for stereotype annotation. In this case we can use XML. Here alternate to XML is ***Java Configuration Approach***.

Student.java

```
package com.cdac.beans;

public class Student {
	public Student() {
		System.out.println("Student class constructor");
	}
}
```

JavaConfig.java

```
package com.cdac.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.cdac.beans.Student;

@Configuration
public class JavaConfig {
	@Bean
	public Student student() {
		Student s = new Student();
		return s;
	}
}
```

Test.java

```
package com.cdac.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.cdac.beans.Student;
import com.cdac.config.JavaConfig;
```

```
public class Test {
	public static void main(String[] args) {
		ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(JavaConfig.class);
		Student s = applicationContext.getBean("student",Student.class);
	}
}
```

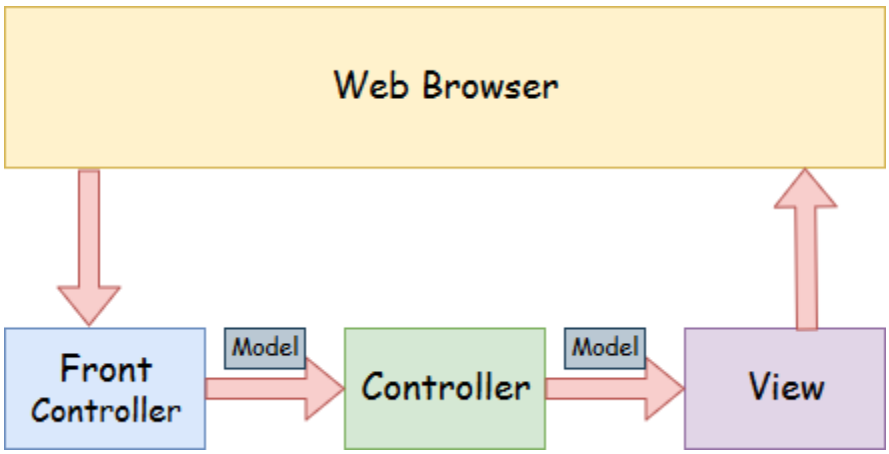
Spring MVC Tutorial

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

.

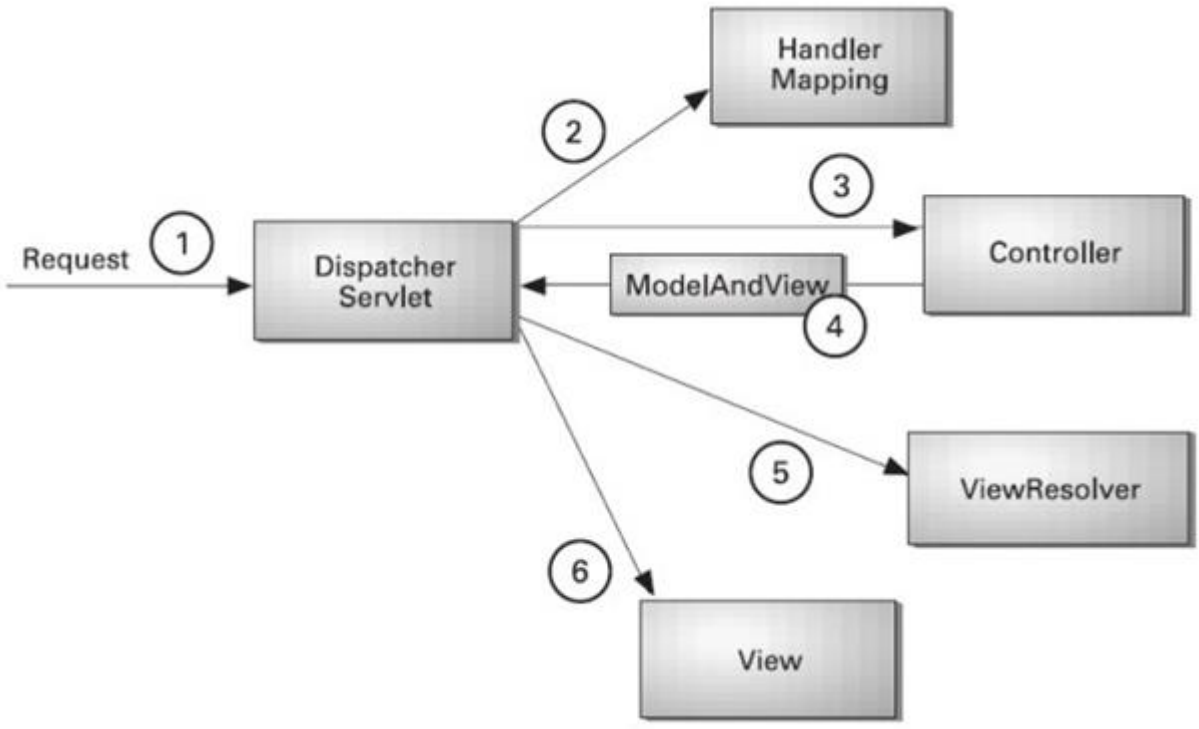
- **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker

  .

- **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.

Dispatcher Servlet

- Dispatcher servlet acts as a front controller in spring MVC based application.
- It takes care of common processing logic that should apply for the entire requests that are coming to application.
- Dispatcher servlet don't know about business logic. After applying common processing it should dispatch request to the Controller.
- Dispatcher servlet don't know for which request which controller is required.
- It will take help of Handle Mapping for identifying for which request which controller is required.

Handler Mapping

- It takes request from the dispatcher and helps in identifying the controller to be used for the request.

Controller

- The programmer should write request specific logic inside the controller.
- After processing it returns logical view name of the view.

View Resolver

- It takes logical view name from the request dispatcher and identifies which view should be used for rendering the request.

  -
  - 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.

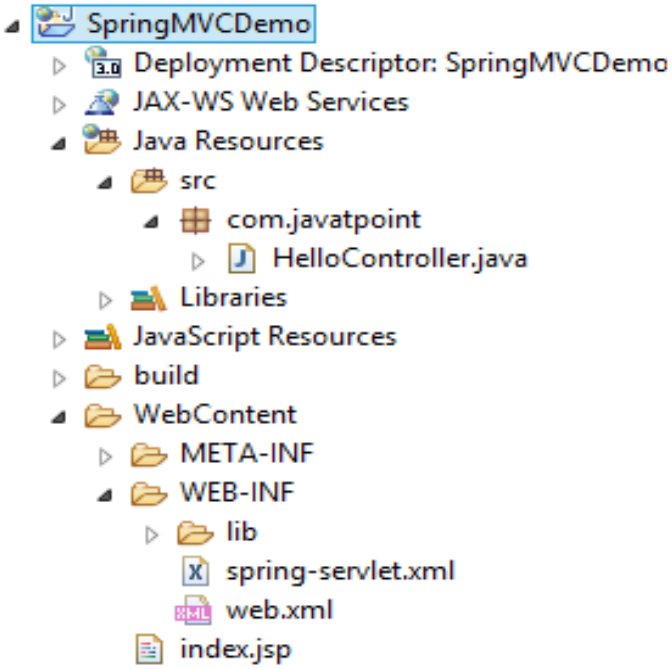
- **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
- **Rapid development** - The Spring MVC facilitates fast and parallel development.
- **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

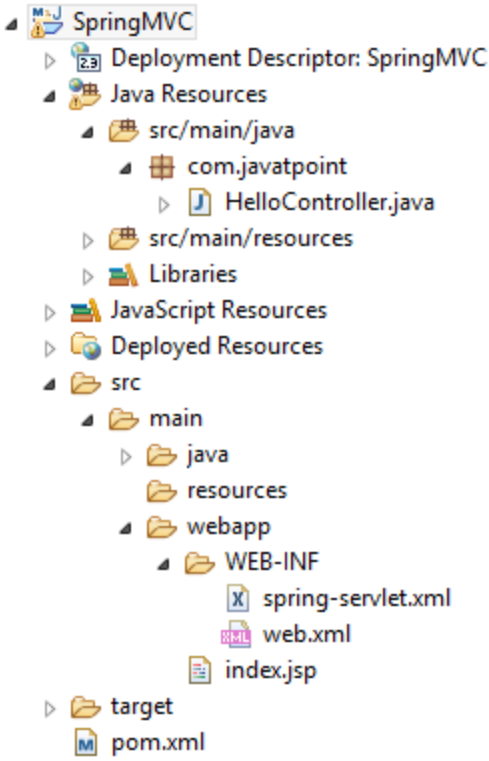
Let's see the simple example of a Spring Web MVC framework. The steps are as follows:

- 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.

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

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatpoint</groupId>
  <artifactId>SpringMVC</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringMVC Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
```

```
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>

  </dependencies>
  <build>
    <finalName>SpringMVC</finalName>
  </build>
</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";
```

```
    }
}
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>spring</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
</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

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />

    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>

</beans>
```

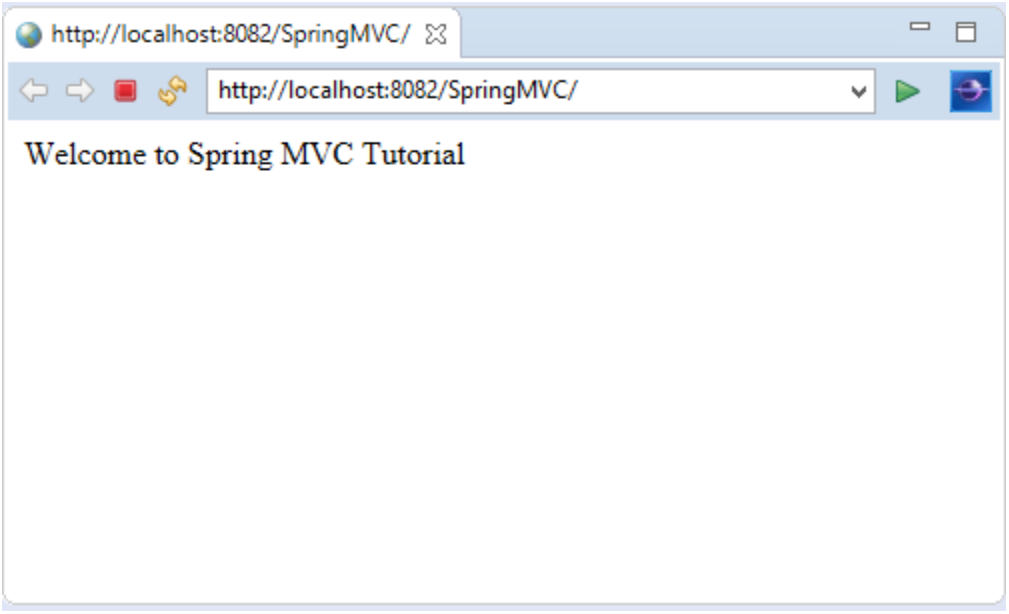
5. Display the message in the JSP page

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

index.jsp

```
<html>
<body>
<p>Welcome to Spring MVC Tutorial</p>
</body>
</html>
```

Output:



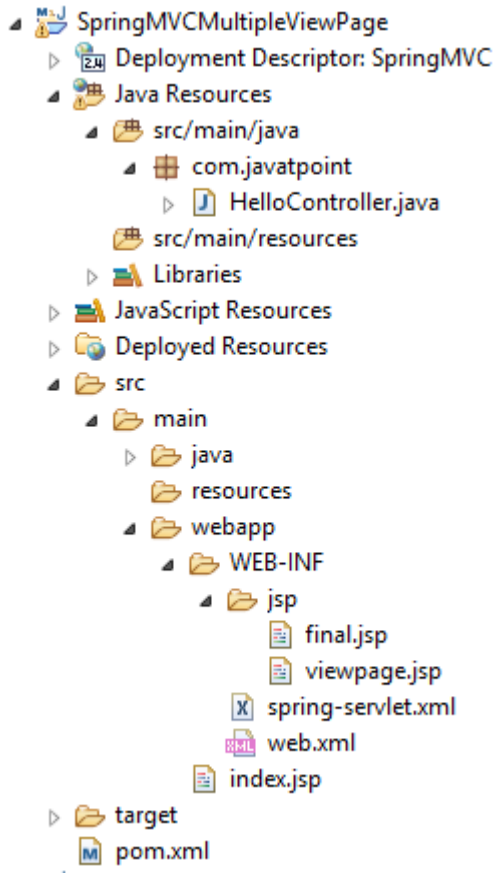
Spring MVC Multiple View page Example

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>
7.
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

```
<html>
<body>
<a href="hello">Click here...</a>
</body>
</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

```
package com.javatpoint;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HelloController {
@RequestMapping("/hello")
    public String redirect()
    {
        return "viewpage";
    }
@RequestMapping("/helloagain")
public String display()
{
    return "final";
}
}
```

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

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<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">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
```

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

    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />

    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>
<!-- Define Spring MVC view resolver -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

6. Create the other view components

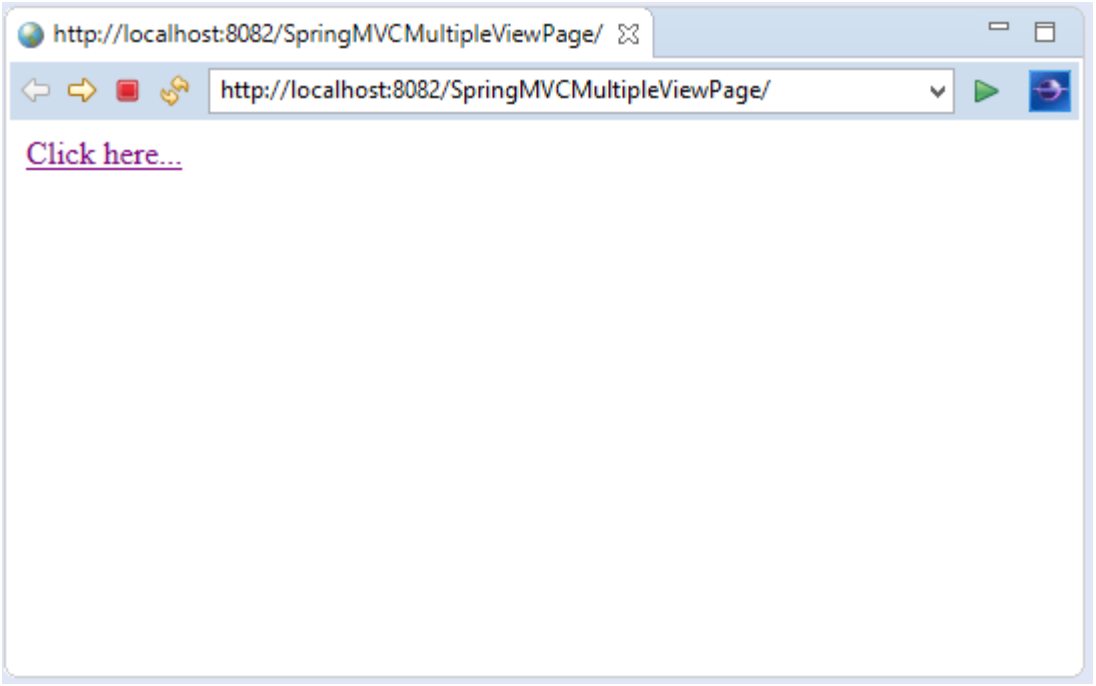
viewpage.jsp

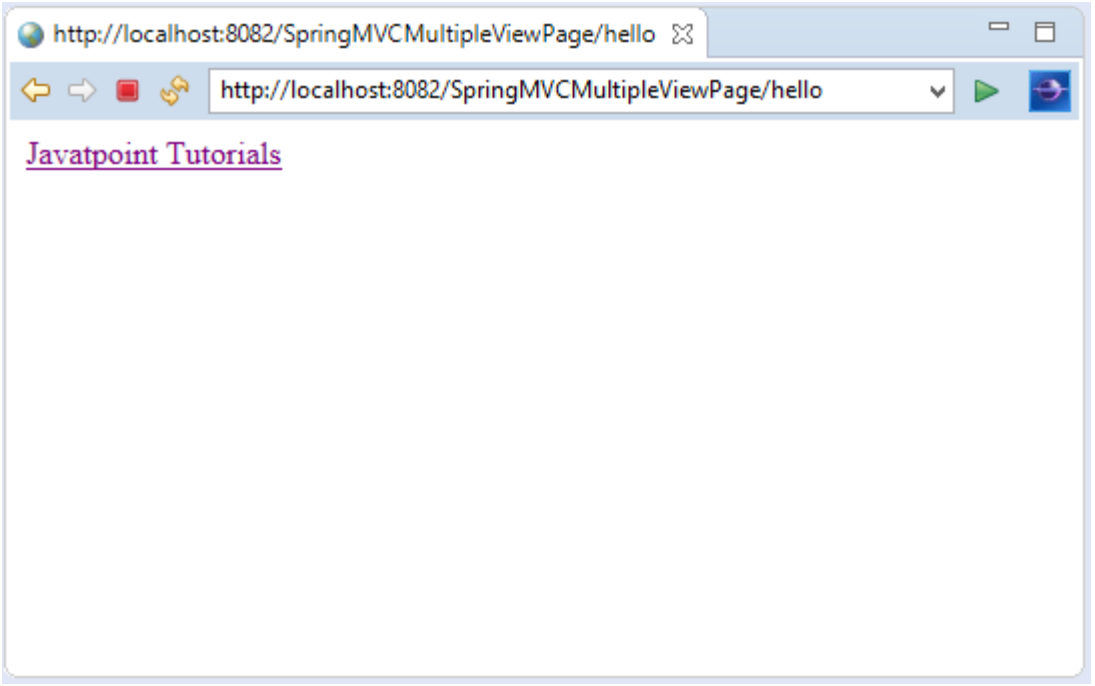
```
<html>
<body>
<a href="helloagain">Javatpoint Tutorials</a>
</body>
</html>
```

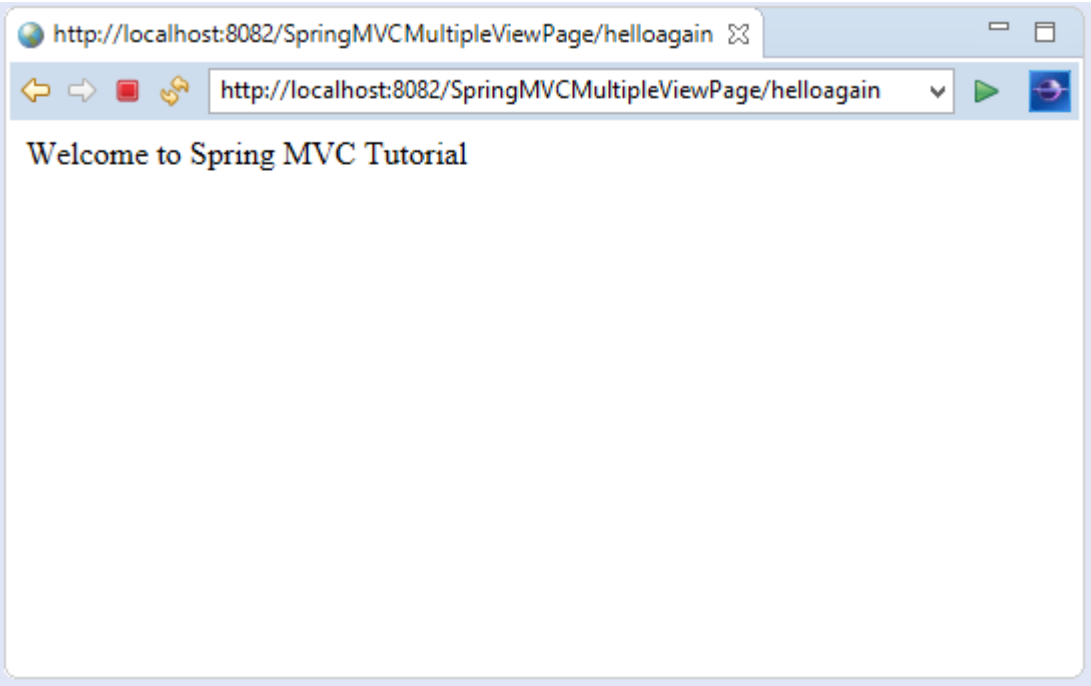
final.jsp

```
<html>
<body>
<p>Welcome to Spring MVC Tutorial</p>
</body>
</html>
```

Output:


http://localhost:8082/SpringMVCMultipleViewPage/
http://localhost:8082/SpringMVCMultipleViewPage/
Click here...


http://localhost:8082/SpringMVCMultipleViewPage/hello
http://localhost:8082/SpringMVCMultipleViewPage/hello
Javatpoint Tutorials

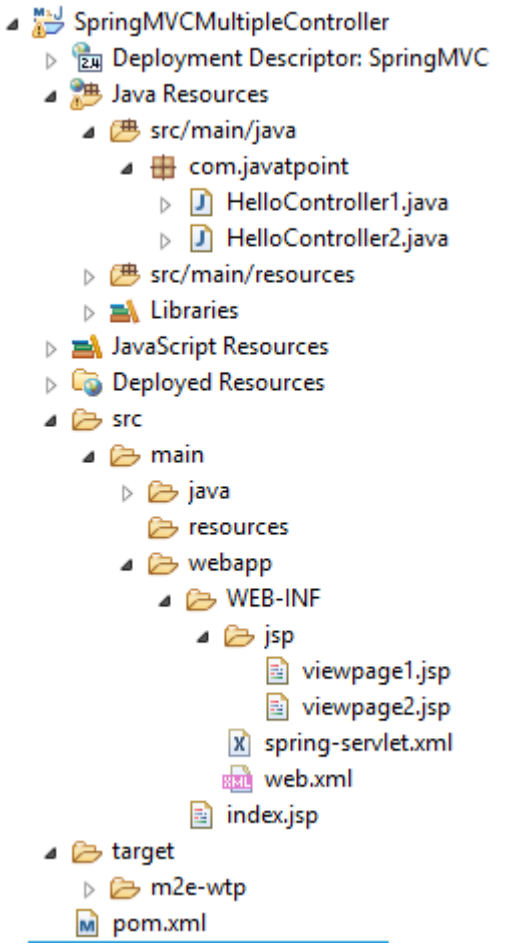


Spring MVC Multiple Controller Example

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>
7.
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

```
<html>
<body>
<a href="hello1">Spring MVC</a> ||
<a href="hello2">Spring Boot</a>
</body>
</html>
```

3. Create the controller class

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

HelloController1.java

```
package com.javatpoint;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HelloController1 {
@RequestMapping("/hello1")
    public String display()
    {
        return "viewpage1";
    }
}
```

HelloController2.java

```
package com.javatpoint;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HelloController2 {
@RequestMapping("/hello2")
    public String display()
```

```
    {
        return "viewpage2";
    }
}
```

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

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

5. Define the bean in the xml file

spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
```

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

    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />

    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

6. Create the other view components

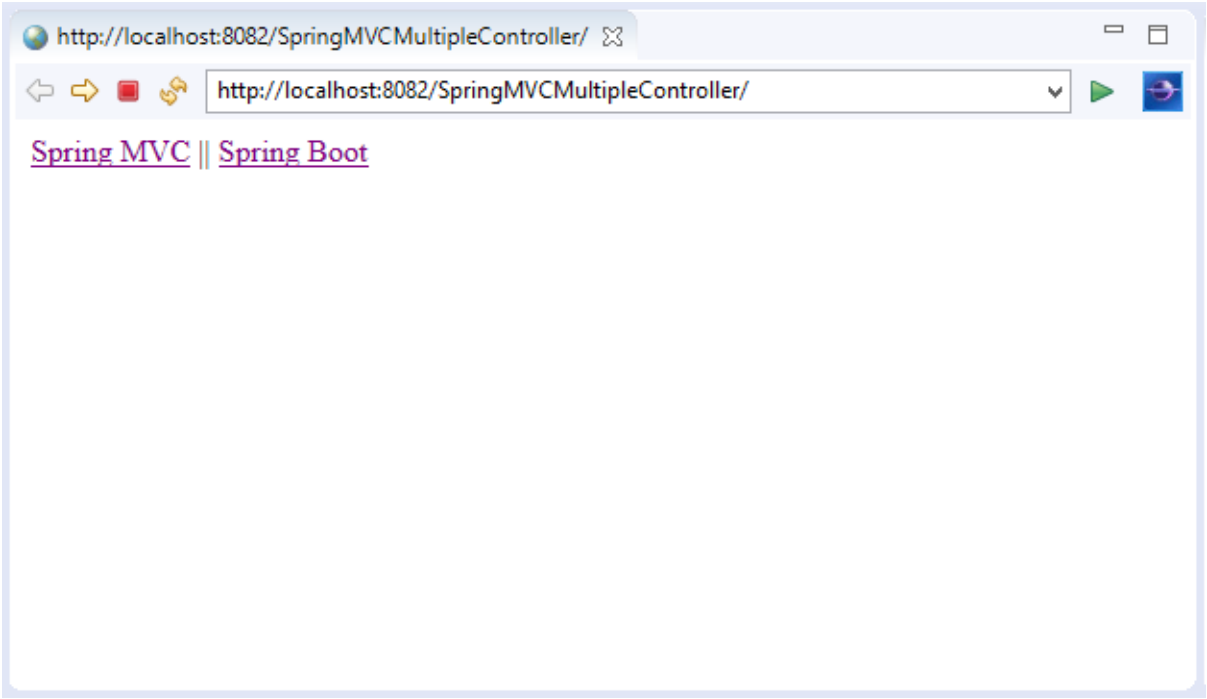
viewpage1.jsp

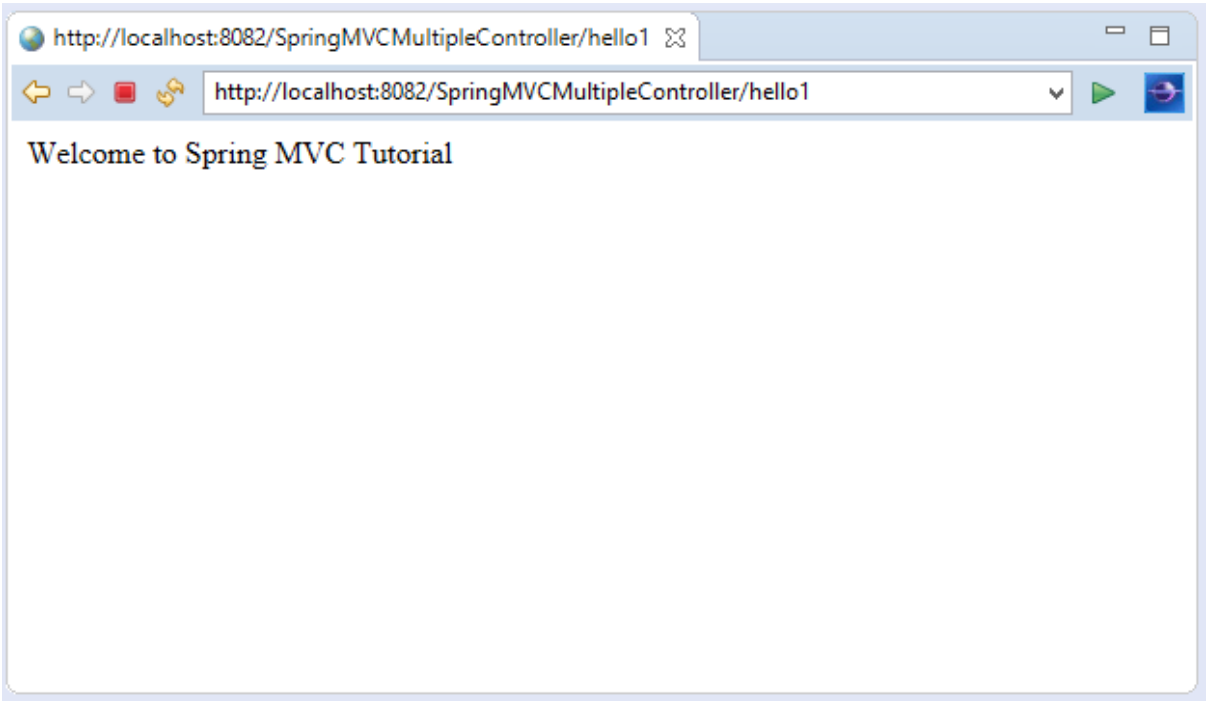
```
<html>
<body>
<p>Welcome to Spring MVC Tutorial</p>
</body>
</html>
```

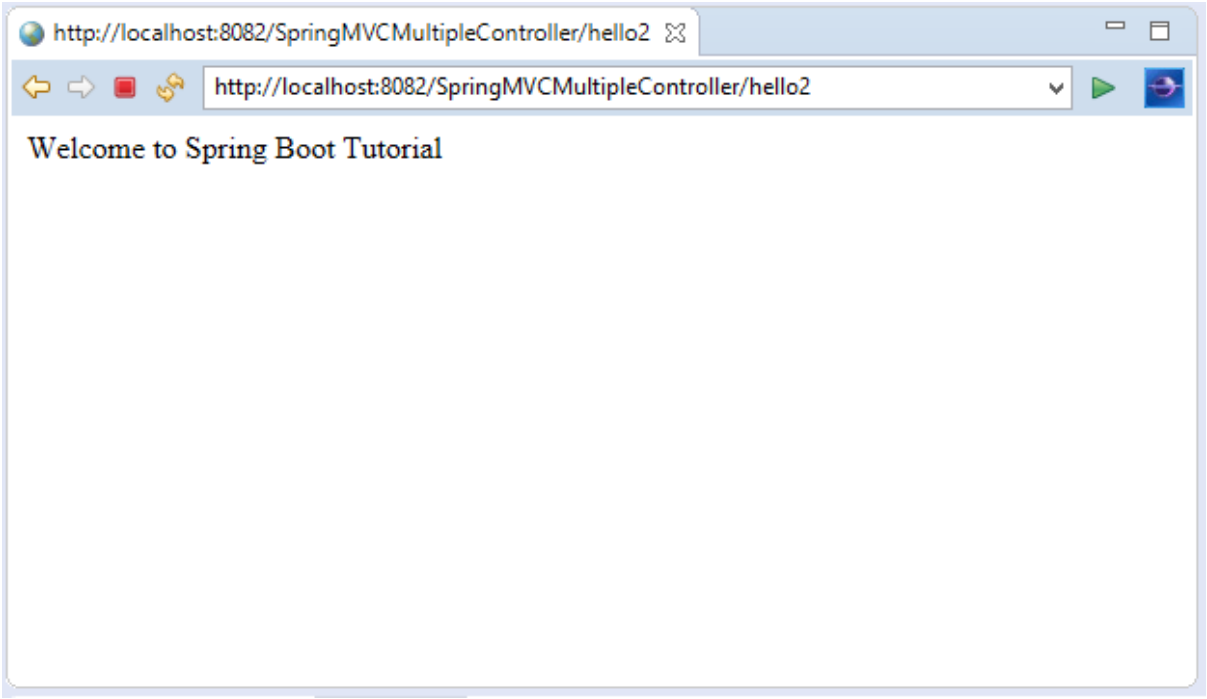
viewpage1.jsp

```
<html>
<body>
<p>Welcome to Spring Boot Tutorial</p>
</body>
</html>
```

Output:


http://localhost:8082/SpringMVCMultipleController/
http://localhost:8082/SpringMVCMultipleController/
Spring MVC || Spring Boot


http://localhost:8082/SpringMVCMultipleController/hello1
http://localhost:8082/SpringMVCMultipleController/hello1
Welcome to Spring MVC Tutorial



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

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
```

```
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
 <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

2. Create the bean class

Employee.java

```
package com.javatpoint;
import javax.validation.constraints.Size;

public class Employee {

    private String name;
    @Size(min=1,message="required")
    private String pass;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPass() {
        return pass;
    }
```

```
    public void setPass(String pass) {
        this.pass = pass;
    }
}
```

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.

```
package com.javatpoint;

import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class EmployeeController {

    @RequestMapping("/hello")
    public String display(Model m)
    {
        m.addAttribute("emp", new Employee());
        return "viewpage";
    }
    @RequestMapping("/helloagain")
    public String submitForm( @Valid @ModelAttribute("emp") Employee e, Bi
    ndingResult br)
    {
        if(br.hasErrors())
        {
            return "viewpage";
        }
```

```
        else
        {
        return "final";
        }
    }
}
```

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

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

5. Define the bean in the xml file

spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />
    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>
    <!-- Define Spring MVC view resolver -->
     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

6. Create the requested page

index.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hello">Click here...</a>
</body>
</html>
```

7. Create the other view components

viewpage.jsp

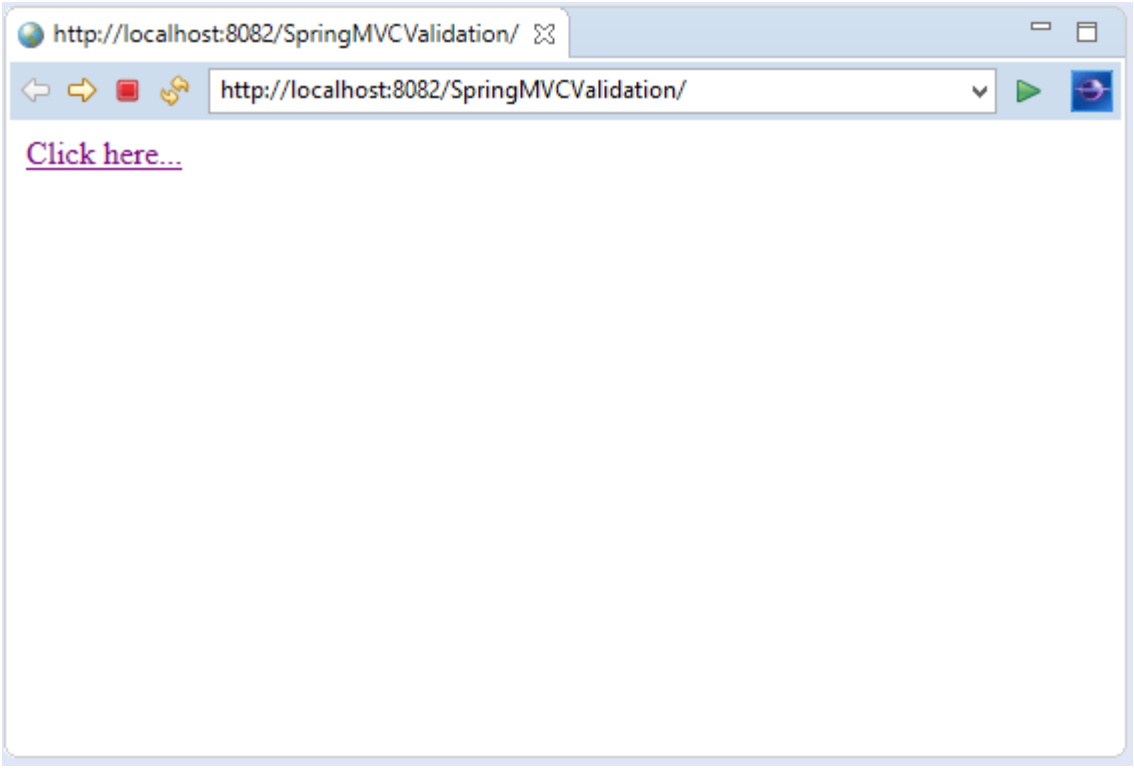
```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
```

```
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>
```

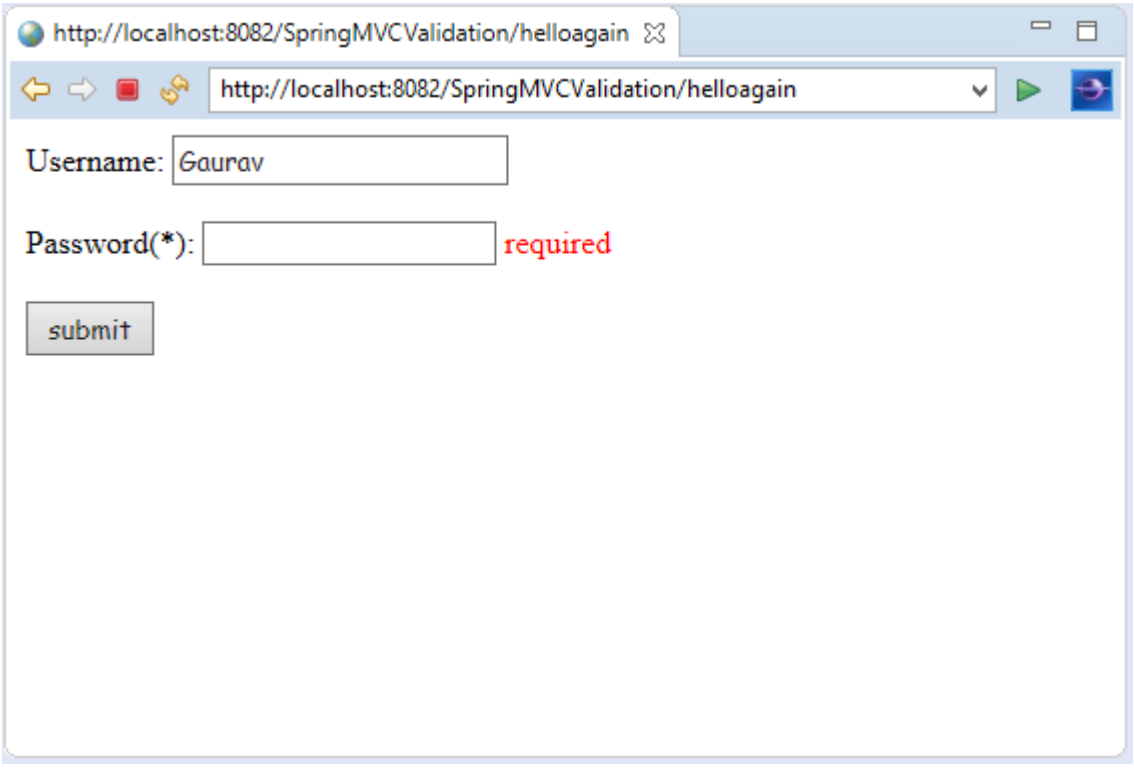
final.jsp

```
1. <html>
2. <body>
3. Username: ${emp.name} <br><br>
4. Password: ${emp.pass}
5. </body>
6. </html>
```

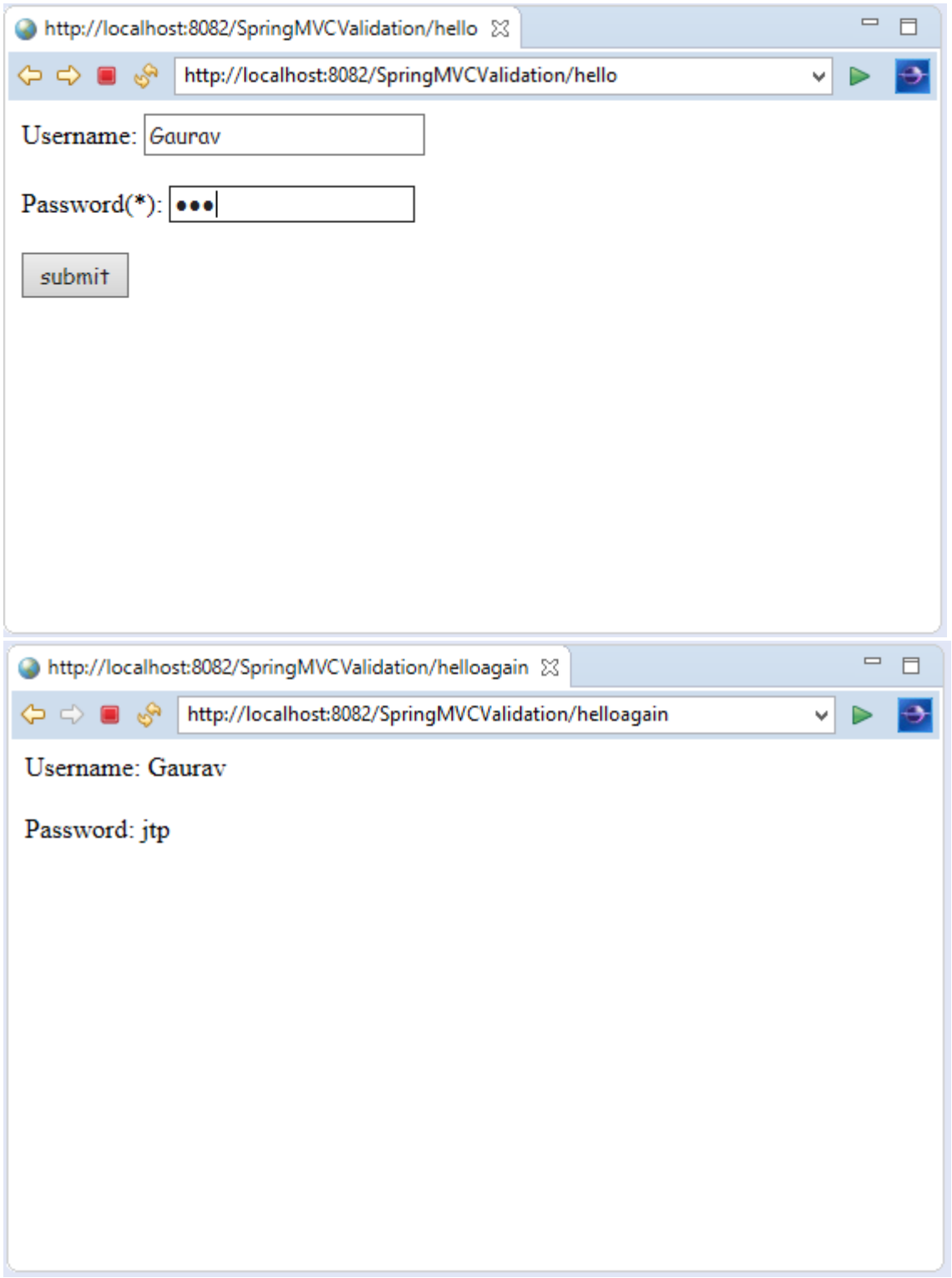
Output:



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

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
 <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
```

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

2. Create the bean class

Employee.java

```
package com.javatpoint;

import javax.validation.constraints.Pattern;
public class Employee {

    private String name;
    @Pattern(regexp="^[a-zA-Z0-9]{3}",message="length must be 3")
    private String pass;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPass() {
        return pass;
    }
    public void setPass(String pass) {
        this.pass = pass;
    }
}
```

3. Create the controller class

```
package com.javatpoint;

import javax.validation.Valid;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class EmployeeController {

    @RequestMapping("/hello")
    public String display(Model m)
    {
        m.addAttribute("emp", new Employee());
        return "viewpage";
    }
    @RequestMapping("/helloagain")
    public String submitForm(@Valid @ModelAttribute("emp") Employee e, Bin
dingResult br)
    {
        if(br.hasErrors())
        {
            return "viewpage";
        }
        else
        {
        return "final";
        }
    }
}
```

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

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="htt
```

```
p://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

5. Define the bean in the xml file

spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />
    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>
    <!-- Define Spring MVC view resolver -->
     <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

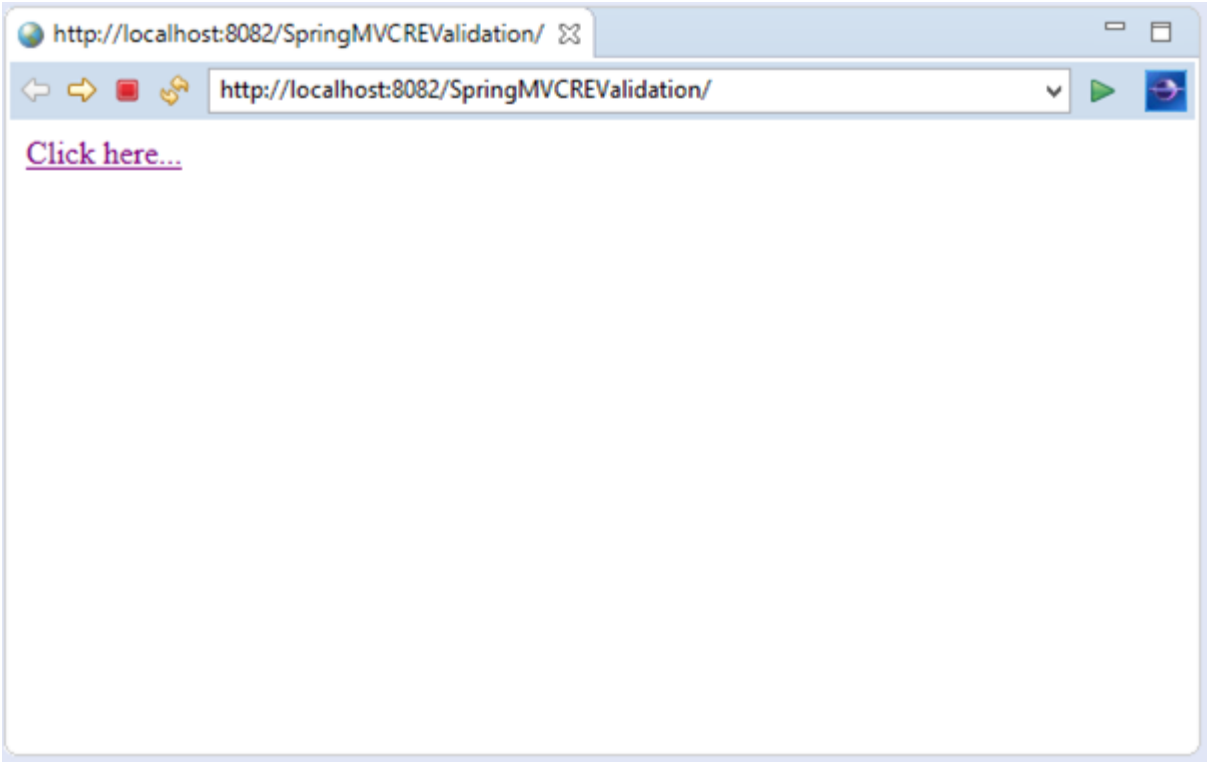
viewpage.jsp

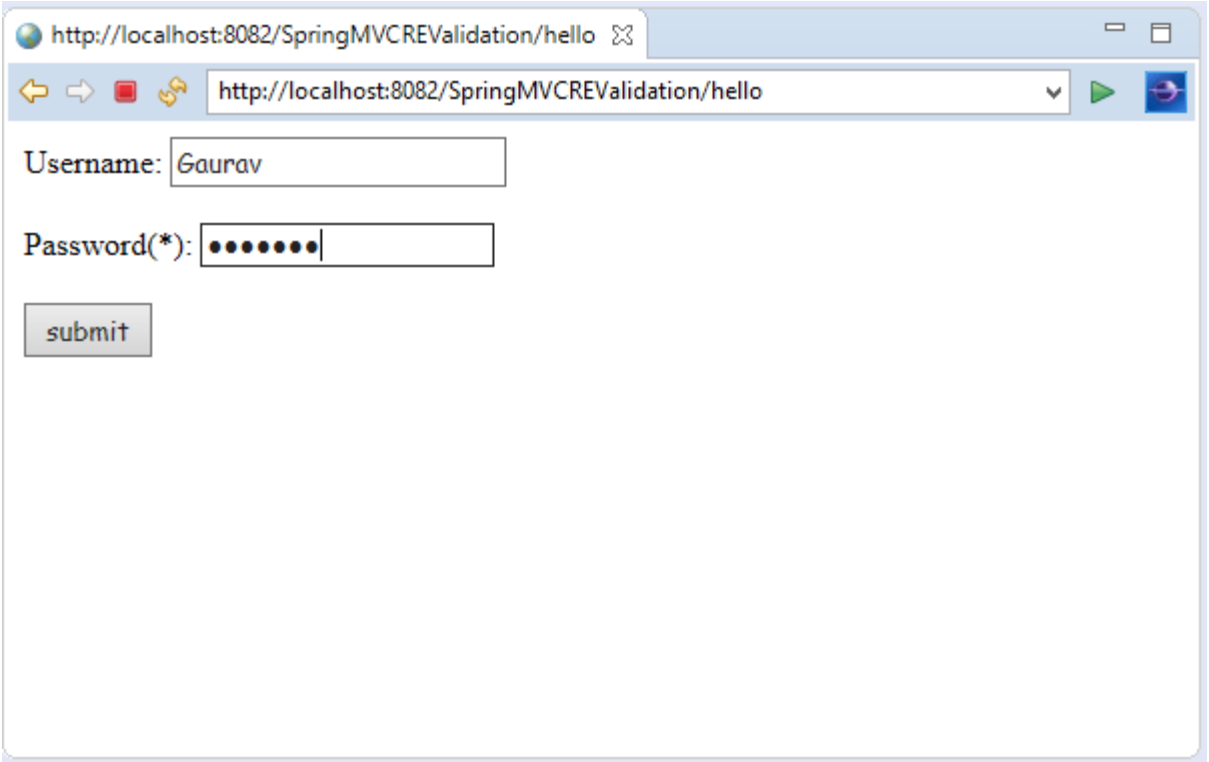
```
1. <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
   %>
2. <html>
3. <head>
4. <style>
5. .error{color:red}
6. </style>
7. </head>
8. <body>
9. <form:form action="helloagain" modelAttribute="emp">
10. Username: <form:input path="name"/> <br><br>
11. Password(*): <form:password path="pass"/>
12. <form:errors path="pass" cssClass="error"/><br><br>
13. <input type="submit" value="submit">
14. </form:form>
15. </body>
16. </html>
```

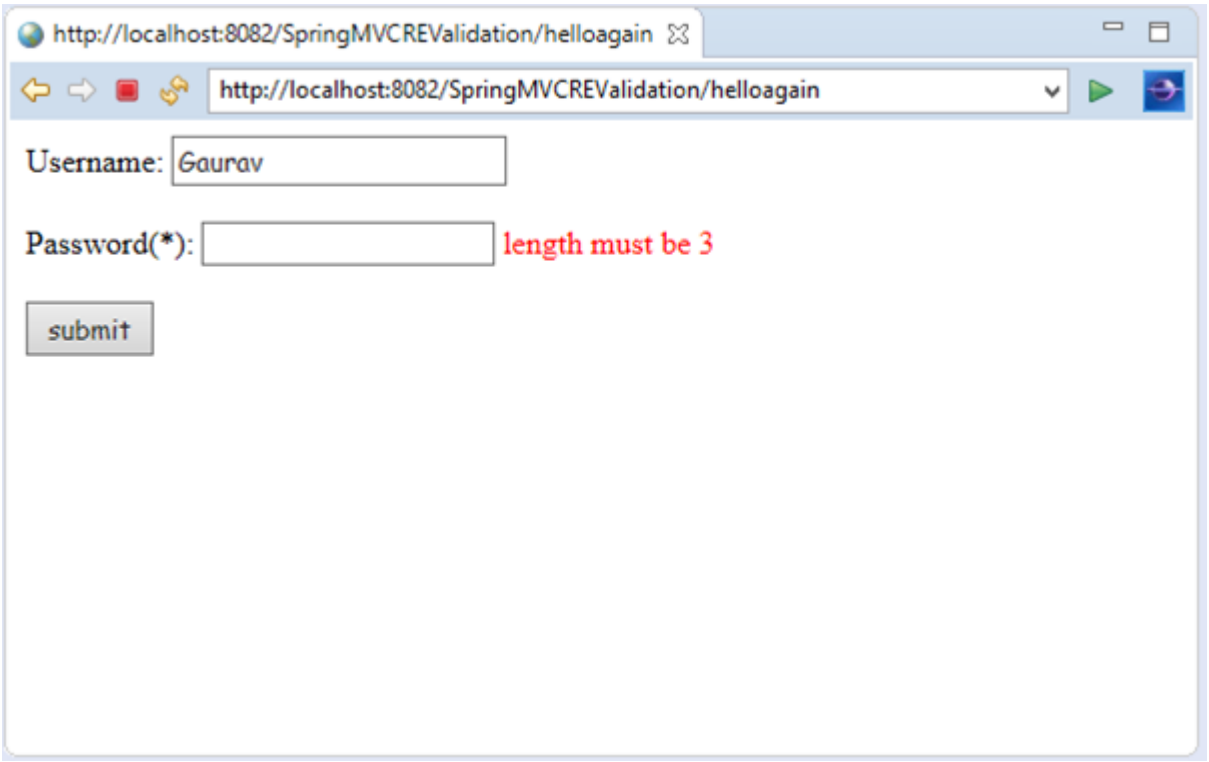
final.jsp

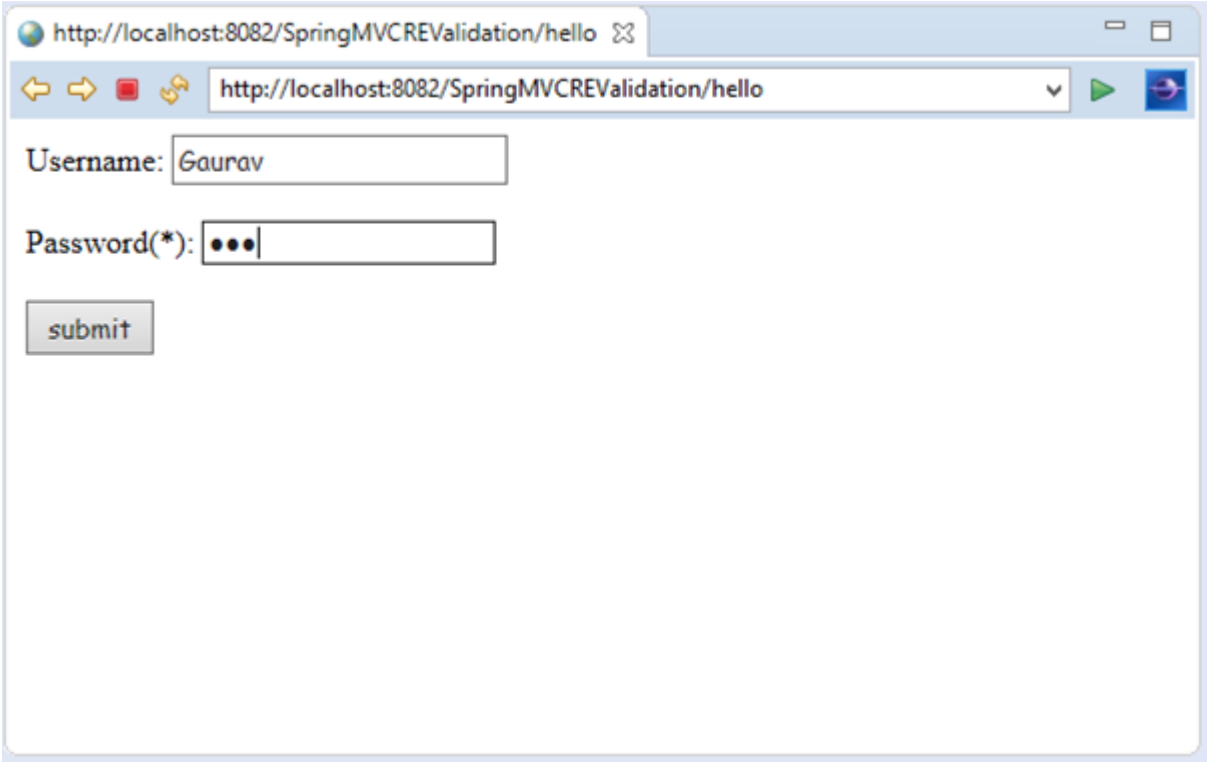
```
<html>
<body>
Username: ${emp.name} <br><br>
Password: ${emp.pass}
</body>
</html>
```

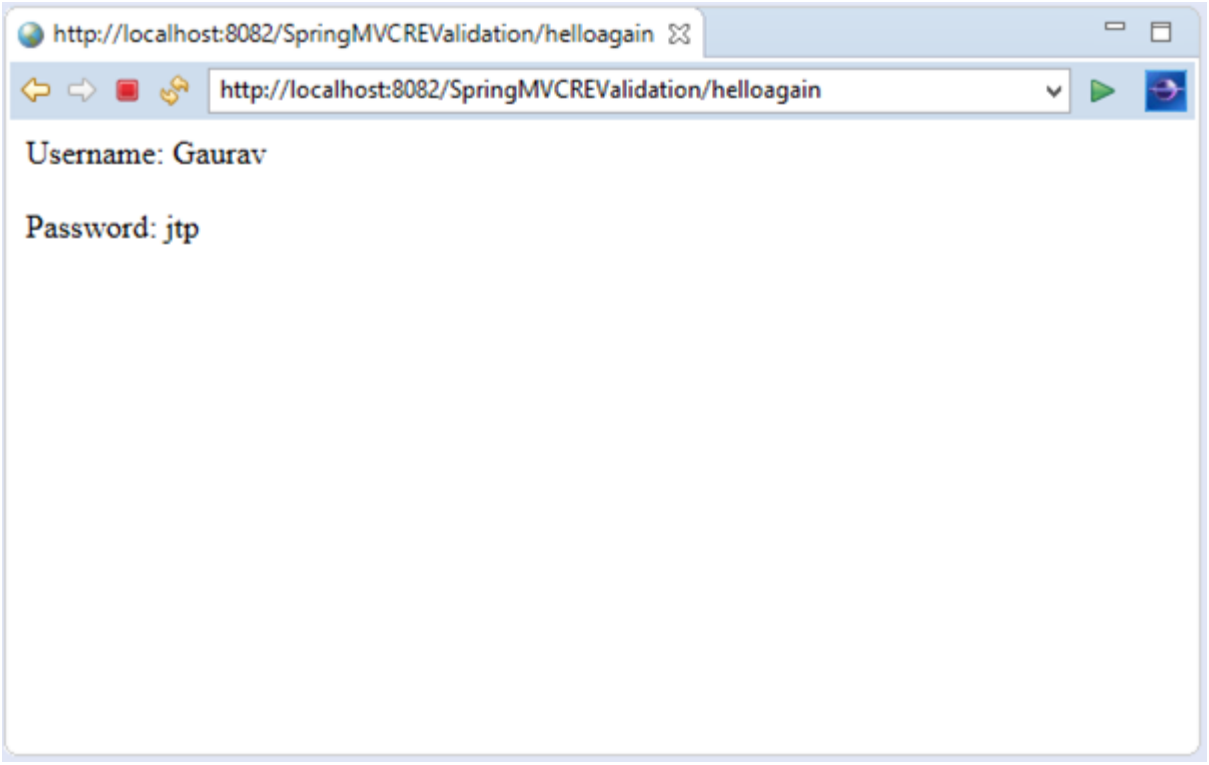
Output:


http://localhost:8082/SpringMVCREValidation/
http://localhost:8082/SpringMVCREValidation/
Click here...


http://localhost:8082/SpringMVCREValidation/hello
http://localhost:8082/SpringMVCREValidation/hello
Username: Gaurav
Password(*):
submit


http://localhost:8082/SpringMVCREValidation/helloagain
http://localhost:8082/SpringMVCREValidation/helloagain
Username: Gaurav
Password(*):
length must be 3
submit


http://localhost:8082/SpringMVCREValidation/hello
http://localhost:8082/SpringMVCREValidation/hello
Username: Gaurav
Password(*):
submit



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>
```

```
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
 <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

2. Create the bean class

Employee.java

```
package com.javatpoint;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
```

```
5.  import javax.validation.constraints.Size;
6.
7.  public class Employee {
8.
9.      private String name;
10.     @Size(min=1,message="required")
11.     private String pass;
12.
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;
16.
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.     }
35.
36. }
```

3. Create the controller class

EmployeeController.java

```
1.   package com.javatpoint;
```

```
2.
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;
9.
10. @Controller
11. public class EmployeeController {
12.
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, Bi
    ndingResult 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"?>
```

```
<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">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

5. Define the bean in the xml file

spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />
    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>
    <!-- Define Spring MVC view resolver -->
```

```
18.     <bean id="viewResolver" class="org.springframework.web.servlet.view.Inte
        rnalResourceViewResolver">
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

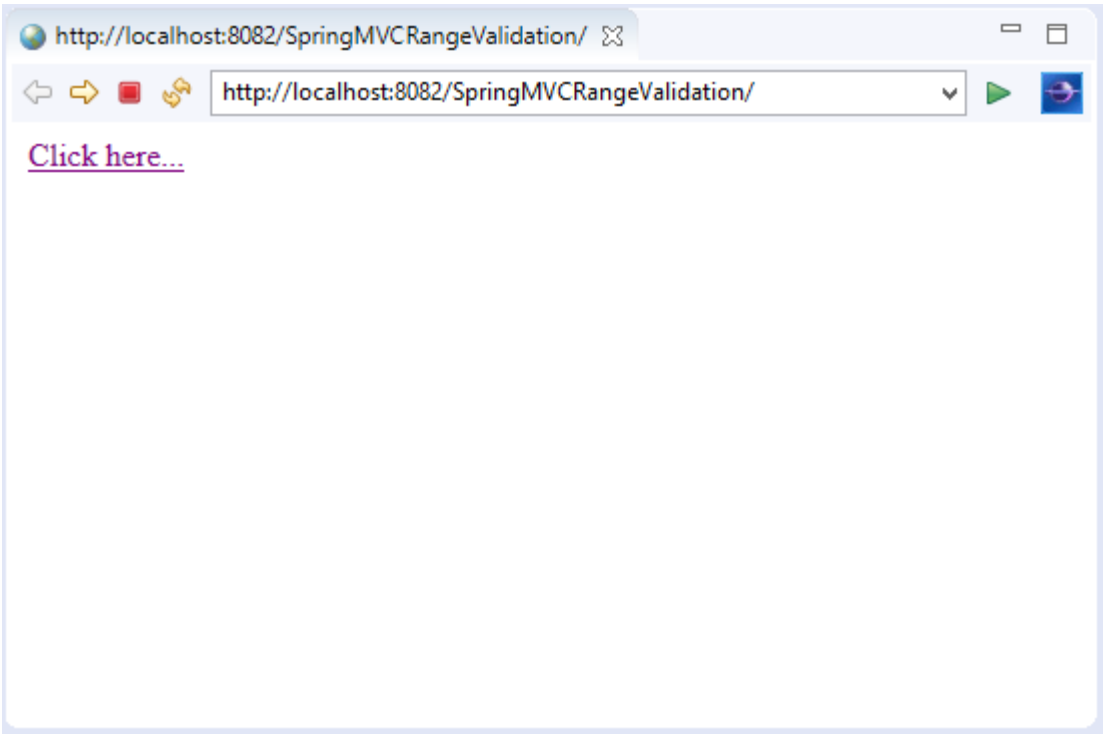
```
1. <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
   %>
2. <html>
3. <head>
4. <style>
5. .error{color:red}
6. </style>
7. </head>
8. <body>
9. <form:form action="helloagain" modelAttribute="emp">
10. Username: <form:input path="name"/> <br><br>
11. Password: <form:password 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">
```

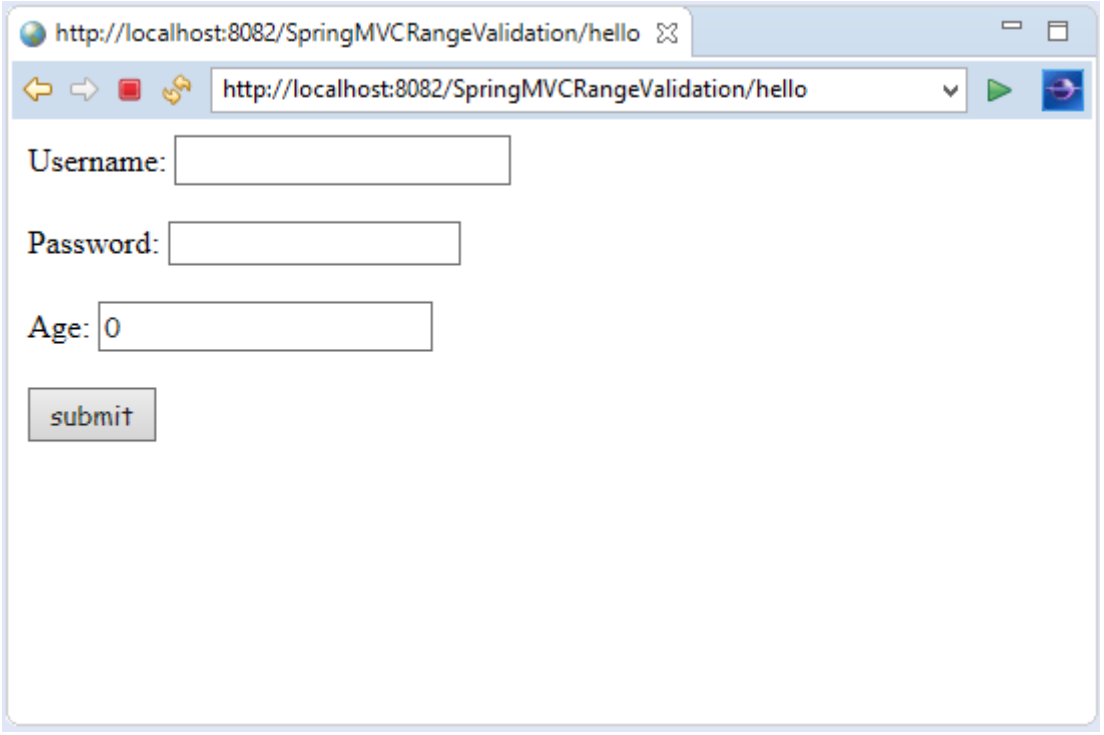
16. **</form:form>**
17. **</body>**
18. **</html>**

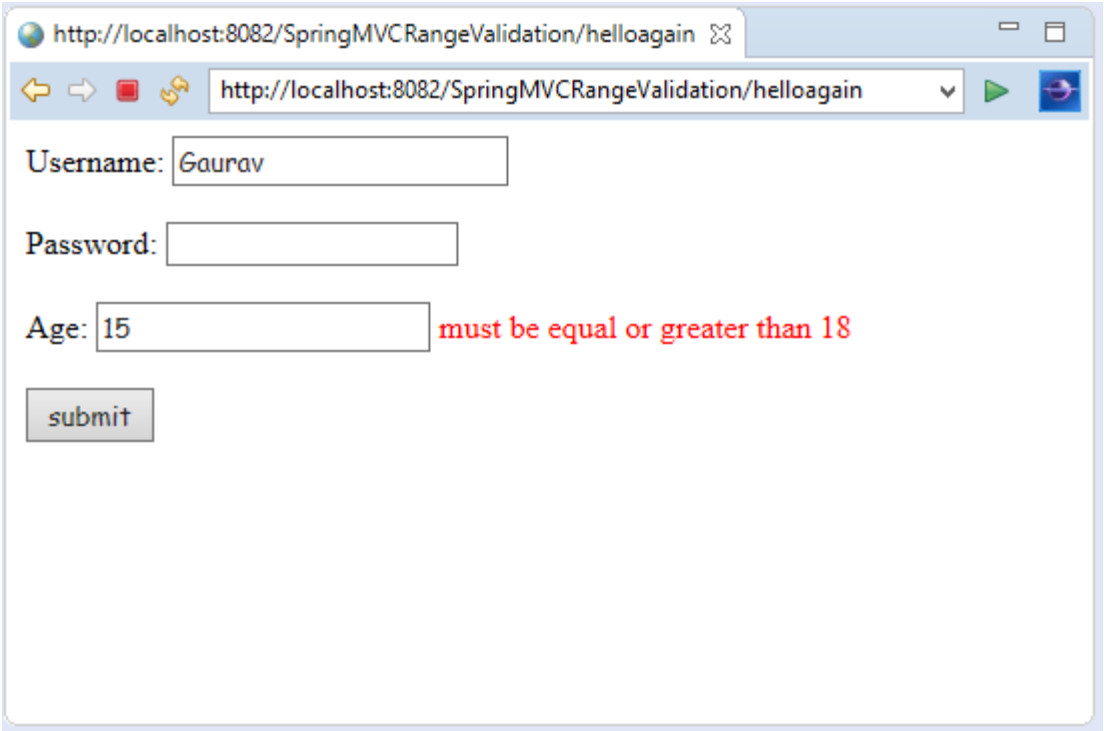
final.jsp

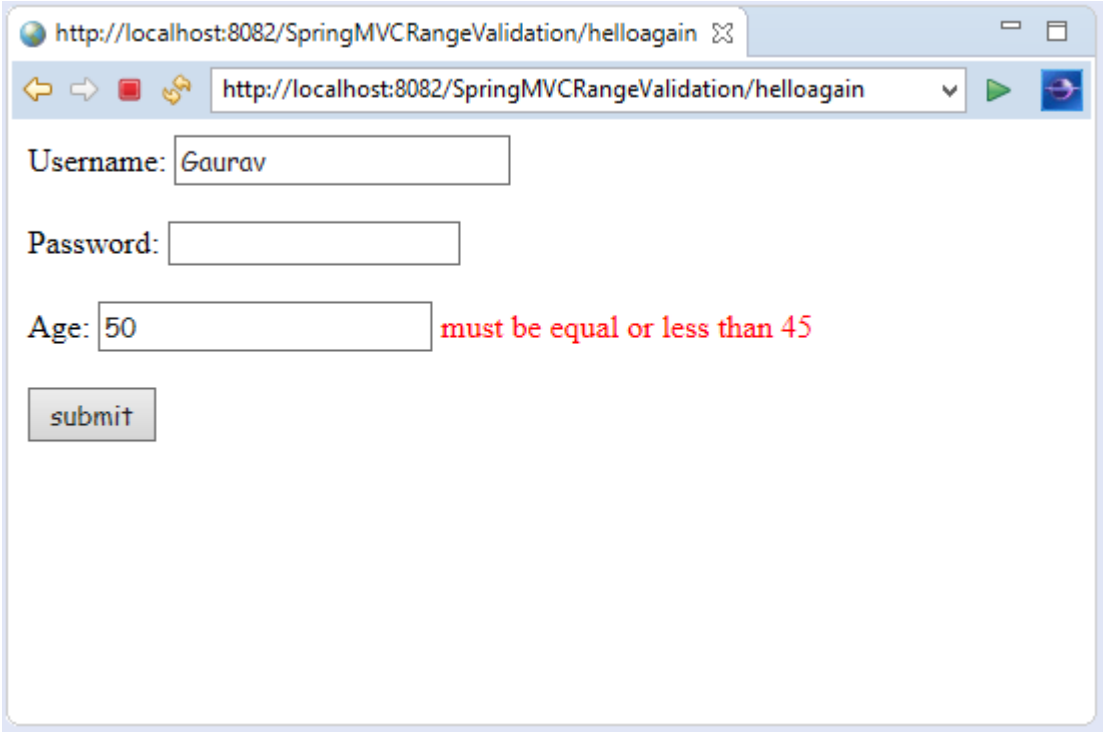
1. **<html>**
2. **<body>**
3. Username: ${param.name} **
**
4. Password: ${param.pass} **
**
5. Age: ${param.age } **
**
6. **</body>**
7. **</html>**

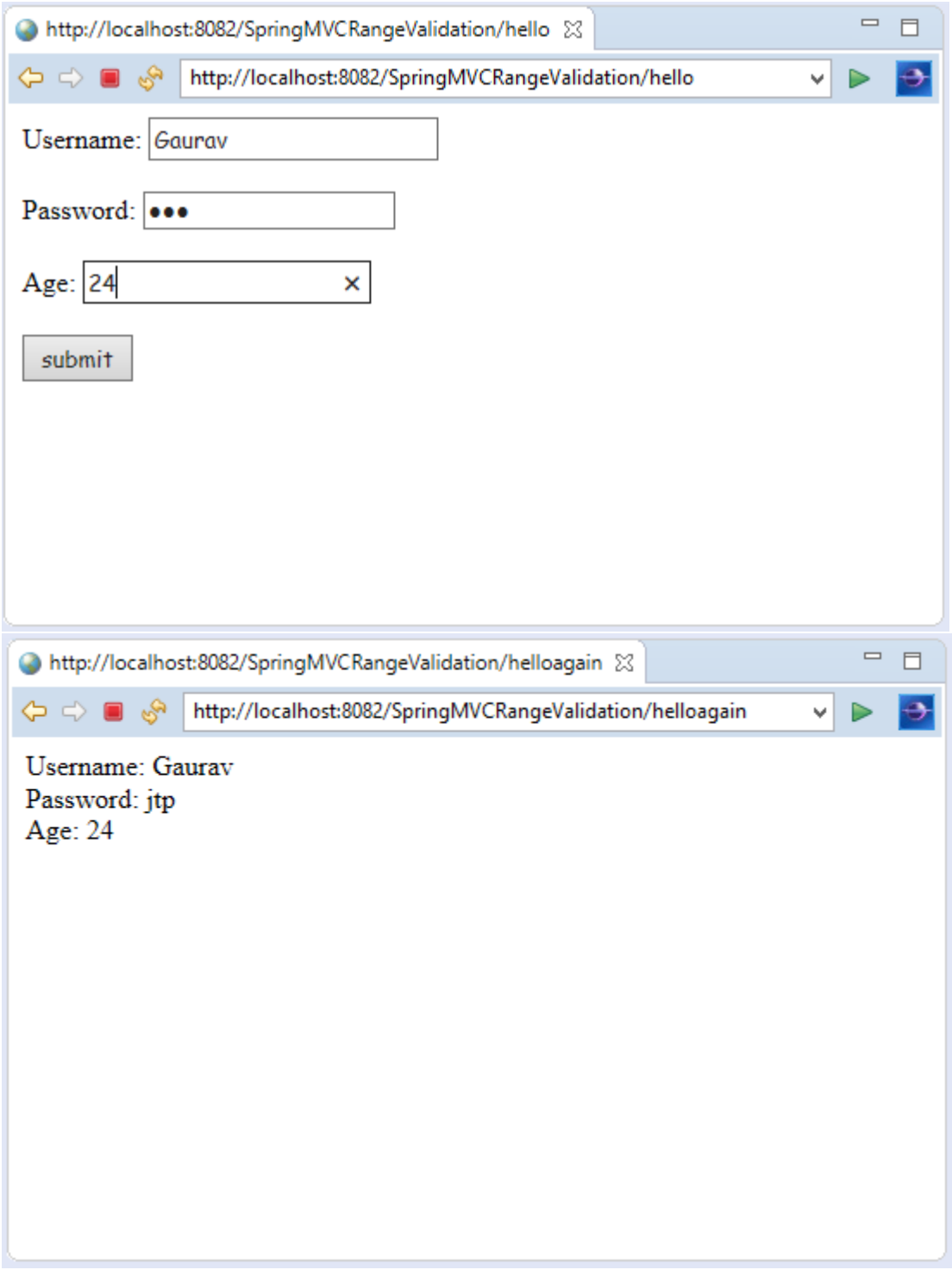
Output:


http://localhost:8082/SpringMVCRangeValidation/
http://localhost:8082/SpringMVCRangeValidation/
Click here...


http://localhost:8082/SpringMVCRangeValidation/hello
http://localhost:8082/SpringMVCRangeValidation/hello
Username:
Password:
Age: 0
submit


http://localhost:8082/SpringMVCRangeValidation/helloagain
http://localhost:8082/SpringMVCRangeValidation/helloagain
Username: Gaurav
Password:
Age: 15
must be equal or greater than 18
submit


http://localhost:8082/SpringMVCRangeValidation/helloagain
http://localhost:8082/SpringMVCRangeValidation/helloagain
Username: Gaurav
Password:
Age: 50
must be equal or less than 45
submit



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

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
 <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
```

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

2. Create the bean class

Employee.java

```
package com.javatpoint;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import com.javatpoint.customvalidation.Password;

public class Employee {
    private String name;
    //Custom annotation
    @Password
    private String password;
    //Predefined annotation
    @Min(value=18, message="must be equal or greater than 18")
    @Max(value=45, message="must be equal or less than 45")
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

```

```
    public void setPassword(String password) {
        this.password = password;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

3. Create the controller class

EmployeeController.java

```
 package com.javatpoint;

import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class EmployeeController {

    @RequestMapping("/hello")
    public String showForm(Model theModel) {

        theModel.addAttribute("emp", new Employee());

        return "viewpage";
    }

    @RequestMapping("/helloagain")
```

```
    public String processForm(
            @Valid @ModelAttribute("emp") Employee emp,
            BindingResult br) {

        if (br.hasErrors()) {
            return "viewpage";
        }
        else {
            return "final";
        }
    }
}
```

4. Create the validator annotation

Password.java

```
package com.javatpoint.customvalidation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Constraint(validatedBy = PasswordConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface Password {
    //error message
        public String message() default "must contain jtp";
    //represents group of constraints
        public Class<?>[] groups() default {};
    //represents additional information about annotation
        public Class<? extends Payload>[] payload() default {};
}
```

5. Create the validator class

PasswordConstraintValidator.java

```
1. package com.javatpoint.customvalidation;
2. 
3. import javax.validation.ConstraintValidator;
4. import javax.validation.ConstraintValidatorContext;
5. 
6. public class PasswordConstraintValidator implements ConstraintValidator<P
   assword,String> {
7. 
8.     public boolean isValid(String s, ConstraintValidatorContext cvc) {
9. 
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="htt
   p://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.Inte
    rnalResourceViewResolver">
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>
```

```
</html>
```

9. Create the other view components

viewpage.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
<html>
<head>
   <style>
      .error {color:red}
   </style>
</head>
<body>
   <form:form action="helloagain" modelAttribute="emp">
      Username: <form:input path="name" />
      <br><br>

      Password (*): <form:password path="password" />
      <form:errors path="password" cssClass="error" />
      <br><br>

      Age (*): <form:input path="age" />
      <form:errors path="age" cssClass="error" />
      <br><br>
      <input type="submit" value="Submit" />
   </form:form>
</body>
</html>
```

final.jsp

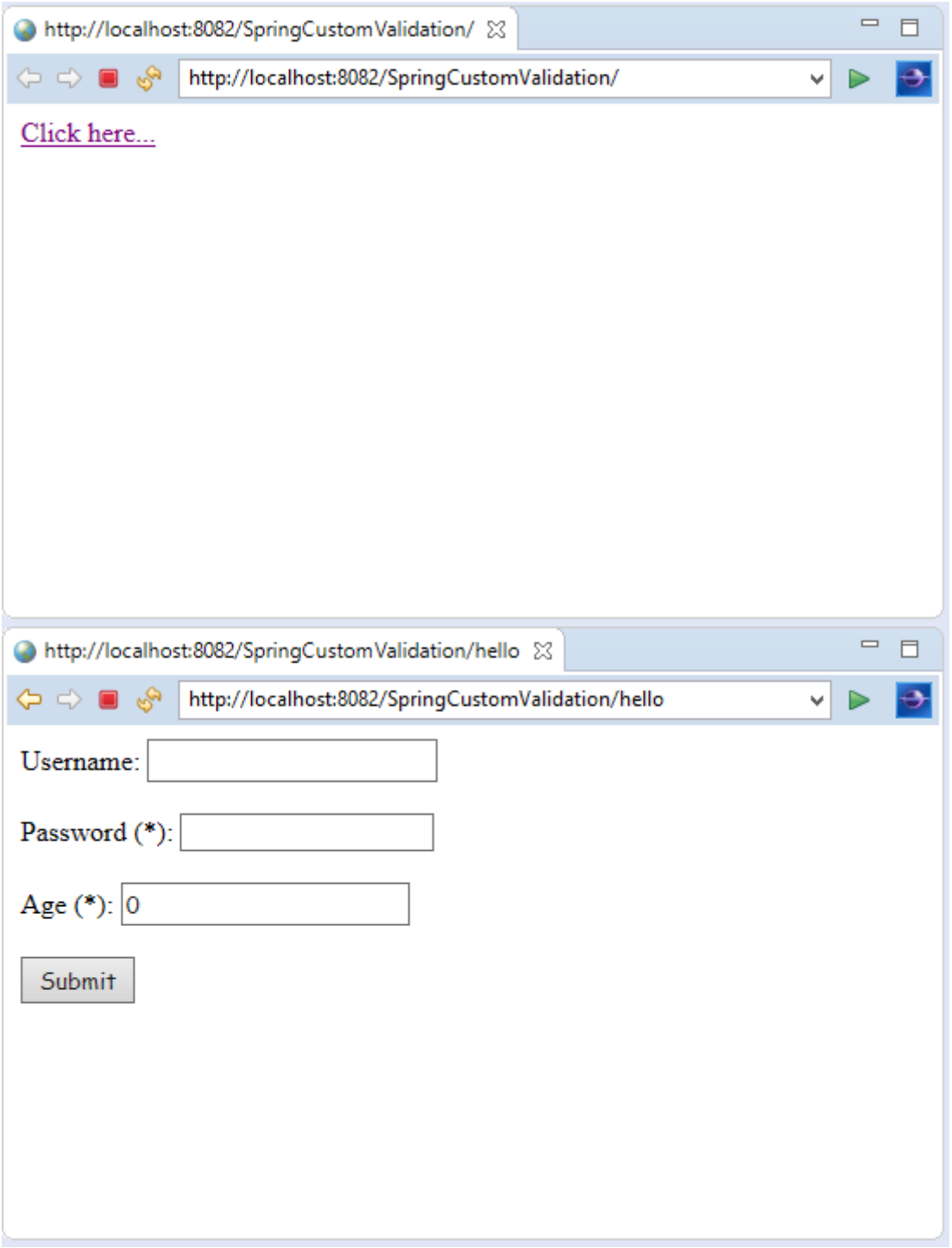
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<body>
Username: ${emp.name}<br><br>
```

6. Password: ${emp.password}

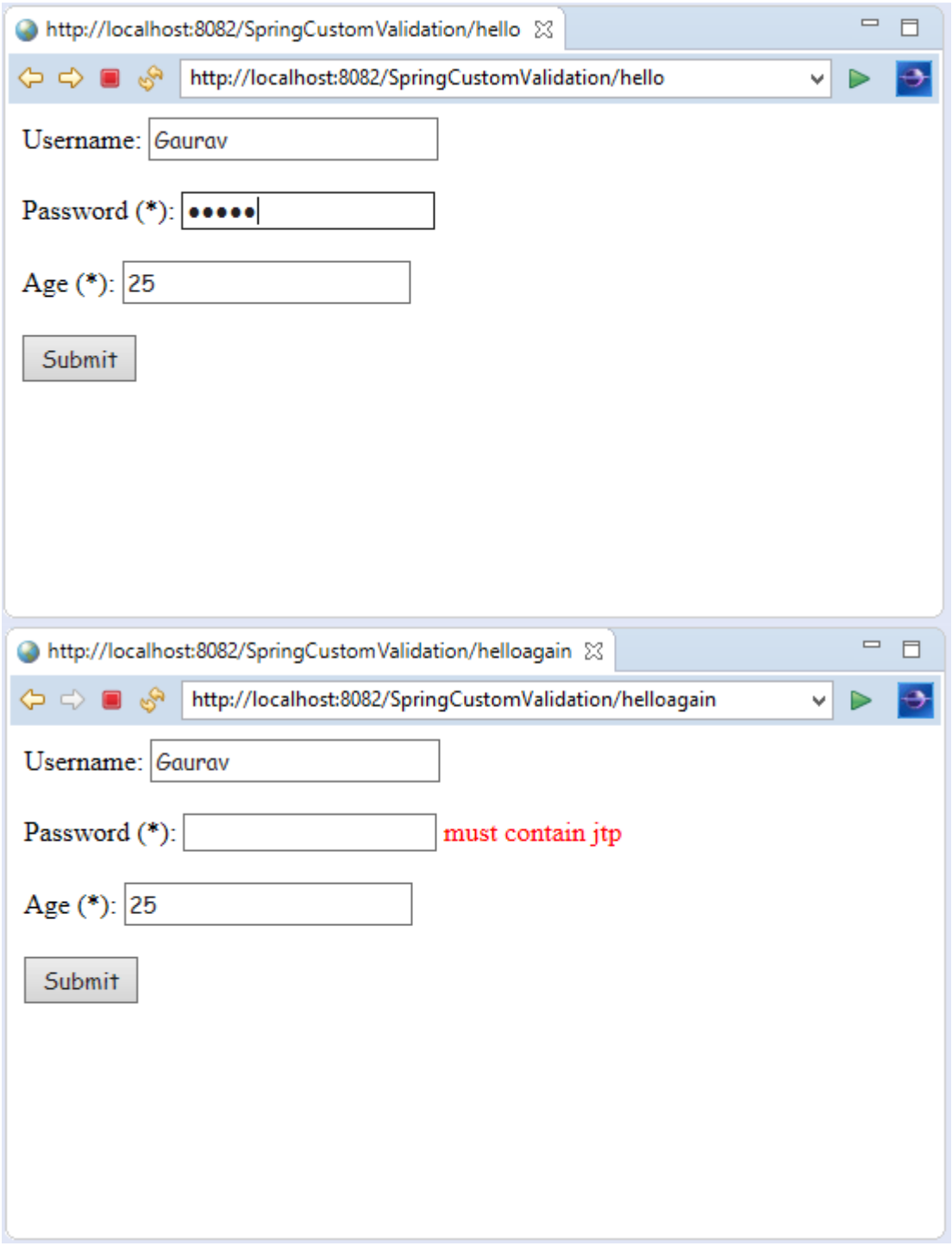
7. Age: ${emp.age}
8.

9. </body>
10. </html>

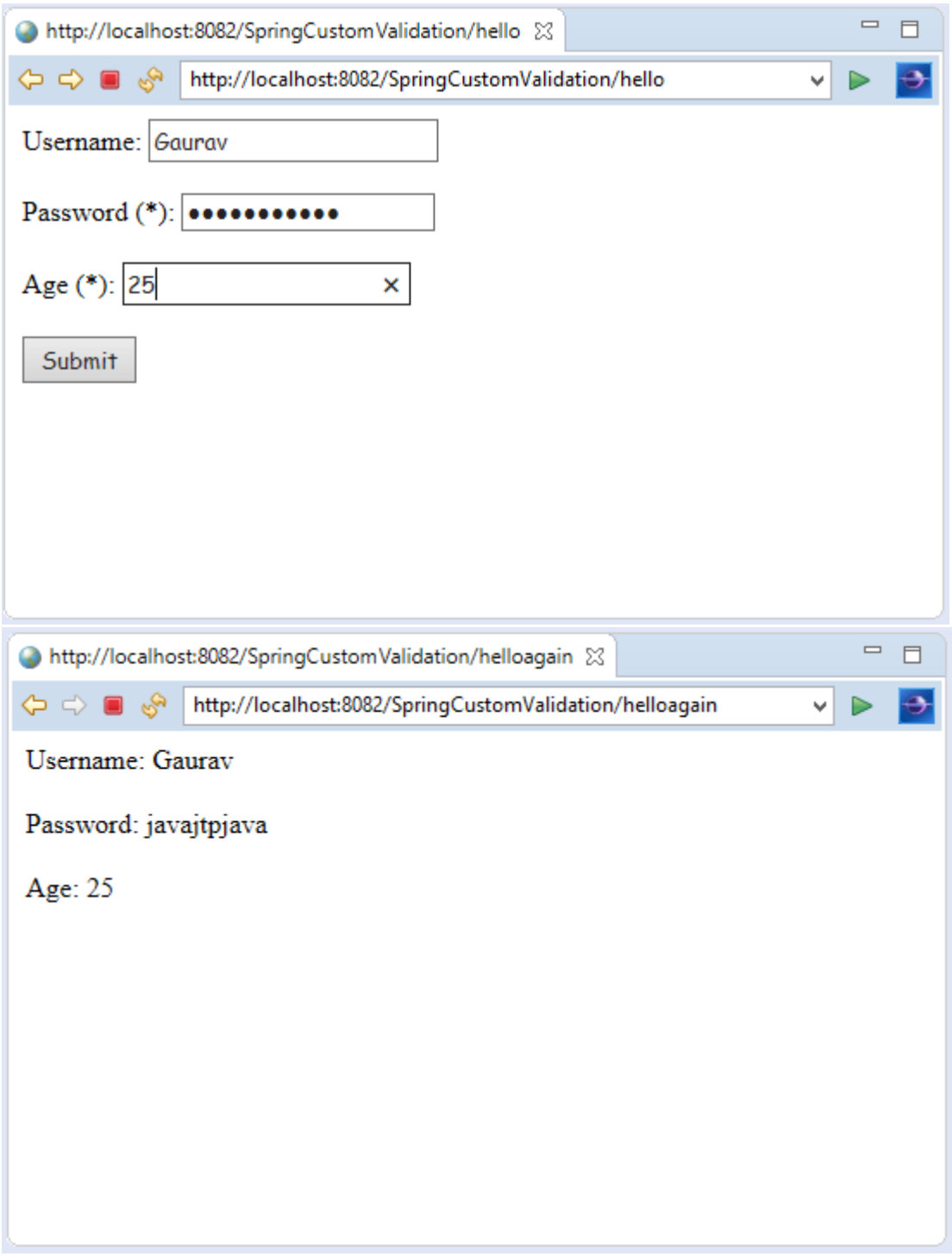
Output:



Here, we entered the password without having "jtp" sequence.



Now, we entered the password having "jtp" sequence.



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.

2) Download commons-io.jar

3) Download commons-fileupload.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

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
```

3) Create form to submit file. Method name must be "post" and enctype "multiple/form-data".

```
<form action="savefile" method="post" enctype="multipart/form-data">
Select File: <input type="file" name="file"/>
<input type="submit" value="Upload File"/>
</form>
```

4) Use CommonsMultipartFile class in Controller.

```
@RequestMapping(value="/savefile",method=RequestMethod.POST)
public ModelAndView upload(@RequestParam CommonsMultipartFile file,HttpSession session){
        String path=session.getServletContext().getRealPath("/");
        String filename=file.getOriginalFilename();
```

```
5.
6.          System.out.println(path+" "+filename);
7.          try{
8.          byte barr[]=file.getBytes();
9.
10.         BufferedOutputStream bout=new BufferedOutputStream(
11.                 new FileOutputStream(path+"/"+filename));
12.         bout.write(barr);
13.         bout.flush();
14.         bout.close();
15.
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;
18.
19. @Controller
20. public class HelloController {
21.     private static final String UPLOAD_DIRECTORY ="/images";
22.
23.     @RequestMapping("uploadform")
24.     public ModelAndView uploadForm(){
25.         return new ModelAndView("uploadform");
26.     }
27.
28.     @RequestMapping(value="savefile",method=RequestMethod.POST)
29.     public ModelAndView saveimage( @RequestParam CommonsMultipartFile
    file,
30.             HttpSession session) throws Exception{
31.
32.     ServletContext context = session.getServletContext();
33.     String path = context.getRealPath(UPLOAD_DIRECTORY);
34.     String filename = file.getOriginalFilename();
35.
36.     System.out.println(path+" "+filename);
37.
38.     byte[] bytes = file.getBytes();
39.     BufferedOutputStream stream =new BufferedOutputStream(new FileOutpu
    tStream(
40.         new File(path + File.separator + filename)));
41.     stream.write(bytes);
```

```
42.     stream.flush();
43.     stream.close();
44.
45.     return new ModelAndView("uploadform","filesuccess","File successfully sav
    ed!");
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
```

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

<context:component-scan base-package="com.javatpoint"></context:component-scan>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/jsp/"></property>
<property name="suffix" value=".jsp"></property>
</bean>

<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

</beans>
```

---

uploadform.jsp

Here form must be method="post" and enctype="multipart/form-data".

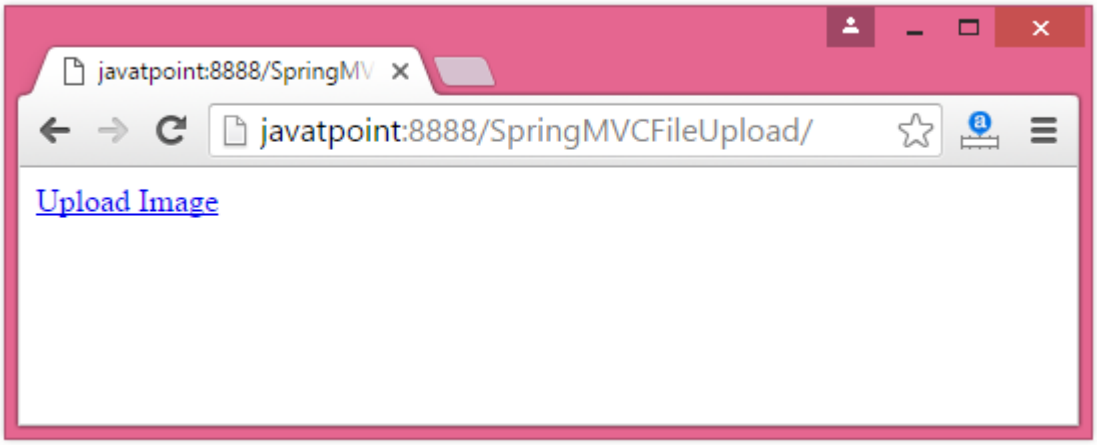
```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<!DOCTYPE html>
<html>
 <head>
 <title>Image File Upload</title>
 </head>
 <body>
<h1>File Upload Example - JavaTpoint</h1>

<h3 style="color:red">${filesuccess}</h3>
<form:form method="post" action="savefile" enctype="multipart/form-data">
<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



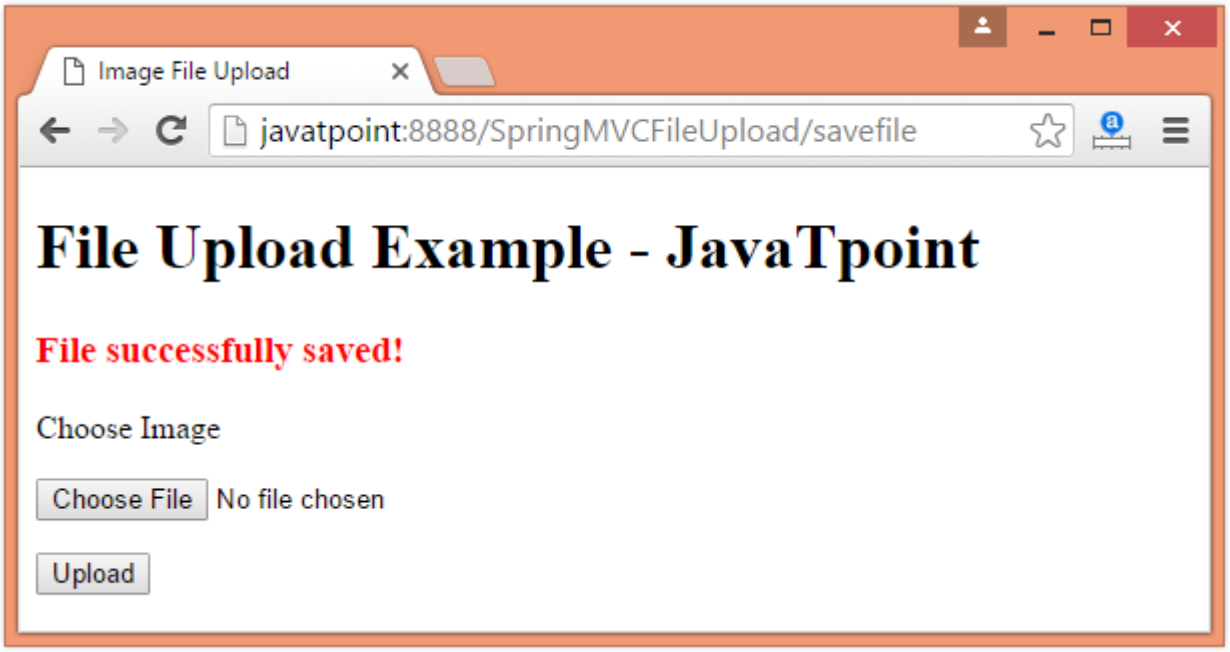


Image File Upload
javatpoint:8888/SpringMVCFileUpload/savefile
File Upload Example - JavaTpoint
File successfully saved!
Choose Image
Choose File
No file chosen
Upload