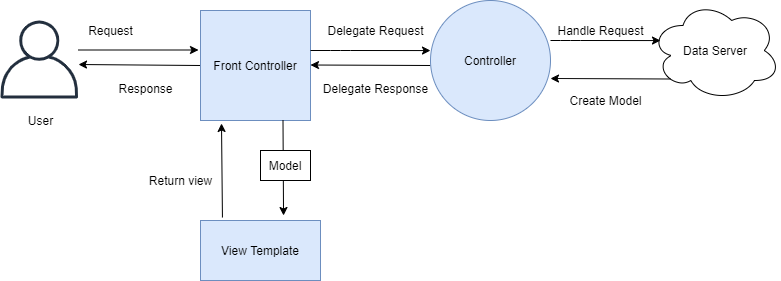
## Spring MVC

## What is MVC?

The MVC is a design pattern used to create applications. The MVC stands for Model, View, and Controller. It consists of three components Model, View, and Controller and in the below architecture, we can see how MVC components interact and work together. We will see each component of MVC in detail later in our tutorial but for now, we can understand it as the Controller is a request handler that handles the user request and responds requested view page back. The View is a page that can be an HTML or JSP page and rendered as a result while the Model represents the database table and used for data handling between application and database.

Spring MVC uses Servlet Dispatcher as a front controller that primarily works as a request handler and handles all user-defined controllers as well.



**User:** a user is a client in a client-server architecture that requests a resource.

[**View:**](https://www.studytonight.com/tutorials/preview?subject=spring-framework&url=spring-mvc-view) a view is a component in MVC architecture that represents an HTML/JSP file and generated randomly by the controller. For example,

**// hello.jsp**

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Hello JSP</title>

</head>

<body>

<h1>Hello, Welcome to JSP</h1>

</body>

</html>

**Front Controller:** it is a built-in controller of the Spring framework that is used to handle all requests for the web application. Spring's DispatcherServlet works as a front controller.

**Model:** It is a component and part of MVC architecture that represents a class. The model is used to write business logic and getter, setters methods. For example,

**// User.java**

package com.studytonight.models;

public class User {

int id;

String name;

String lastName;

String email;

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

//super();

this.id = id;

this.name = name;

this.lastName = lastName;

this.email = email;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

**Controller:** It is used to handle client requests in web applications. It receives a request from the user and sends the requested view after coordinating with models and the database. In the Spring application, a class that is marked with @Controller or @RestController annotation is treated as controller and con be used to handle requests and responses. For example,

package com.studytonight.controllers;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.ResponseBody;

import com.studytonight.models.User;

import com.studytonight.service.UserService;

@Controller

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/users")

@ResponseBody

public void users() {

List<User> users = userService.getUser();

for(User user: users) {

System.out.println(user.getFirstName()+" "+user.getLastName()+" "+user.getEmail());

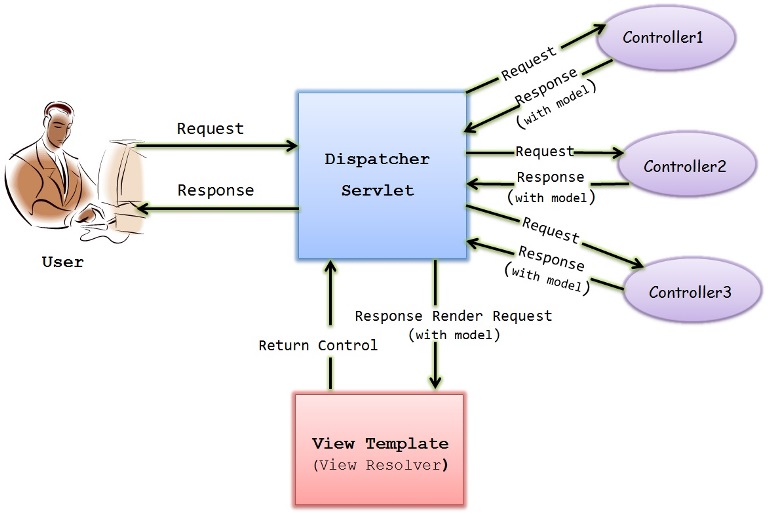
}

}

}

**Spring MVC (Model view controller)** is based on the MVC design pattern, it is a software architecture design pattern.  It provides solution to layer an application by separating three concerns business, presentation and control flow.

* The Model can be some DAO layer or some Service Layers which give some information about request or requested information or Model can be a POJO which encapsulates the application data given by the controller.
* The View is responsible for rendering the model data and in general it generates HTML output that the client’s browser can interpret.
* The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.



**Front Controller**

This is the Dispatcher servlet controller also called as Front controller which is the main controller which manages the flow of the Spring MVC application.

Front Controller is very important component one which route the all the requests into framework control that means when ever requests land on different controllers it queues that request to the controller of framework without this MVC framework will not may be able to take control of the request at landing at the application. So front controller is not only capture the request but also the following responsibility-

* It initialize the framework to cater to the requests.
* Load the map of all URLs and the components responsible to handle the request.
* Prepare the map for the views.

This is defined in the web.xml as below

1. <web-app>
2. <servlet>
3. <servlet-name>Dispatcher-Servlet</servlet-name>
4. <servlet-[class](http://i.viglink.com/?key=09de04bbca2b35471f6e5393bcef569d&insertId=9b3dc810c33dfb01&type=H&mid=7137&exp=60%3ACI1C55A%3A2&libId=k1siieu10101ysj1000DAmhk2ej9g&loc=http%3A%2F%2Fjavainsimpleway.com%2Fspring-mvc-architecture%2F&v=1&iid=9b3dc810c33dfb01&out=https%3A%2F%2Fwww.pntrs.com%2Ft%2F1-7137-43737-7137%3Furl%3Dhttps%253A%252F%252Fwww.decorplanet.com%252Fproducts%252Fdeltana-cl509frcnc-clarendon-3-12-ic-core-class-room-entry-lever-set-without-cylinder%252Fcl509frcnc.html&ref=http%3A%2F%2Fjavainsimpleway.com%2Fspring-bean-life-cycle%2F&title=Spring%20MVC%20Architecture%20%7C%20Javainsimpleway&txt=%3Cspan%3Eclass%3C%2Fspan%3E)>org.springframework.web.servlet.DispatcherServlet</servlet-[class](http://i.viglink.com/?key=09de04bbca2b35471f6e5393bcef569d&insertId=9b3dc810c33dfb01&type=H&mid=7137&exp=60%3ACI1C55A%3A2&libId=k1siieu10101ysj1000DAmhk2ej9g&loc=http%3A%2F%2Fjavainsimpleway.com%2Fspring-mvc-architecture%2F&v=1&iid=9b3dc810c33dfb01&out=https%3A%2F%2Fwww.pntrs.com%2Ft%2F1-7137-43737-7137%3Furl%3Dhttps%253A%252F%252Fwww.decorplanet.com%252Fproducts%252Fdeltana-cl509frcnc-clarendon-3-12-ic-core-class-room-entry-lever-set-without-cylinder%252Fcl509frcnc.html&ref=http%3A%2F%2Fjavainsimpleway.com%2Fspring-bean-life-cycle%2F&title=Spring%20MVC%20Architecture%20%7C%20Javainsimpleway&txt=%3Cspan%3Eclass%3C%2Fspan%3E)>
5. <load-on-startup>1</load-on-startup>
6. </servlet>
8. <servlet-mapping>
9. <servlet-name> Dispatcher-Servlet </servlet-name>
10. <url-pattern>/</url-pattern>
11. </servlet-mapping>
13. </web-app>

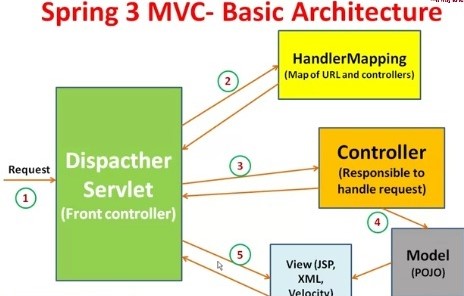
So Dispatcher servlet is defined and mapped to the url pattern /, which means all request must go through Dispatcher servlet.

## Spring MVC Basic Architecture

The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

In Spring MVC framework Dispatcher Servlet access Front Controller which handles all coming requests and queses for forward to the different controller.

1. Whenever request lands the dispatcher servlet consult with HandlerMapping  
(HandlerMapping– is a component which have the map of URL and Controller which need to be invoked for that particular request which lands with URL)  
2. then Dispatcher servlet has information about which is controller need to be invoked  
3. then that controller will be invoked  
4. and Controller can request the model for some information (about some DAO, Service layer or Data in POJO, or data in database using business logic)  
5. once process has been done then dispatcher servlet get the response then dispatcher servlet will get view resolver to build the view and view resolver look out what view has being configured it has been JSP, Velocity, XML etc. based this configuratin view has been prepared and the information from model i.e. POJO it will be put on the view and response will be send back to browser.



**Controller**

Controller is the one which handles the specific request which is mapped by its request mapping.

Example

1. @Controller
2. Public [class](http://i.viglink.com/?key=09de04bbca2b35471f6e5393bcef569d&insertId=9b3dc810c33dfb01&type=H&mid=7137&exp=60%3ACI1C55A%3A2&libId=k1siieu10101ysj1000DAmhk2ej9g&loc=http%3A%2F%2Fjavainsimpleway.com%2Fspring-mvc-architecture%2F&v=1&iid=9b3dc810c33dfb01&out=https%3A%2F%2Fwww.pntrs.com%2Ft%2F1-7137-43737-7137%3Furl%3Dhttps%253A%252F%252Fwww.decorplanet.com%252Fproducts%252Fdeltana-cl509frcnc-clarendon-3-12-ic-core-class-room-entry-lever-set-without-cylinder%252Fcl509frcnc.html&ref=http%3A%2F%2Fjavainsimpleway.com%2Fspring-bean-life-cycle%2F&title=Spring%20MVC%20Architecture%20%7C%20Javainsimpleway&txt=%3Cspan%3Eclass%3C%2Fspan%3E) A{
3. @RequestMapping(“/login”,)
4. Public void login(){
5. //do whatever you want
6. }
7. }

So every controller [class](http://i.viglink.com/?key=09de04bbca2b35471f6e5393bcef569d&insertId=9b3dc810c33dfb01&type=H&mid=7137&exp=60%3ACI1C55A%3A2&libId=k1siieu10101ysj1000DAmhk2ej9g&loc=http%3A%2F%2Fjavainsimpleway.com%2Fspring-mvc-architecture%2F&v=1&iid=9b3dc810c33dfb01&out=https%3A%2F%2Fwww.pntrs.com%2Ft%2F1-7137-43737-7137%3Furl%3Dhttps%253A%252F%252Fwww.decorplanet.com%252Fproducts%252Fdeltana-cl509frcnc-clarendon-3-12-ic-core-class-room-entry-lever-set-without-cylinder%252Fcl509frcnc.html&ref=http%3A%2F%2Fjavainsimpleway.com%2Fspring-bean-life-cycle%2F&title=Spring%20MVC%20Architecture%20%7C%20Javainsimpleway&txt=%3Cspan%3Eclass%3C%2Fspan%3E) will be annotated with @controller and each of its method is mapped to some request url.

**Model**

Model is the class which holds the data to transfer between view and controller.  
Anything in the view is transferred to controller using model class and vice versa.

**View Template**

View is the UI thing which is used for displaying the data,it can be jsp or any other view technology.  
Specific view for the corresponding request is displayed based on the view resolver, it helps in finding the exact view name.

## Advantages of Spring MVC Framework-

* Supports [RESTful URLs](http://en.wikipedia.org/wiki/Representational_state_transfer" \t "_blank).
* Annotation based configuration(i.e. you may reduce the metadata file or less of configuration).
* Supports to plug with other MVC frameworks like Struts, Struts2, WebWorks etc.
* Flexible in supporting different view types like JSP, velocity, XML, PDF, Tiles etc.

**Steps to create spring mvc application**

1) Define the dispatcher servlet in web.xml

2) Create spring configuration file and define the packages to scan for spring beans

3) Create a simple JSP page which will have a form request to be submitted.

4) Create the controller class which will have a method to map a particular request.

5) Create the model class which can hold the form data.

6) Define view resolver in spring configuration file

7) Create the view page(.jsp) to hold the response from the controller.

**Flow of spring mvc application will be as below**

1) User Makes a request through an URL

2) URL is passed to dispatcher servlet

3) Dispatcher servlet passes the request to the corresponding controller based on url mapping.

4) Controller performs the task and return the model and view.

5) Dispatcher servlet maps the view name to the corresponding jsp(any view technology) using view resolver.

6) View renders the model and display it.

##### **Tools and Technologies used**

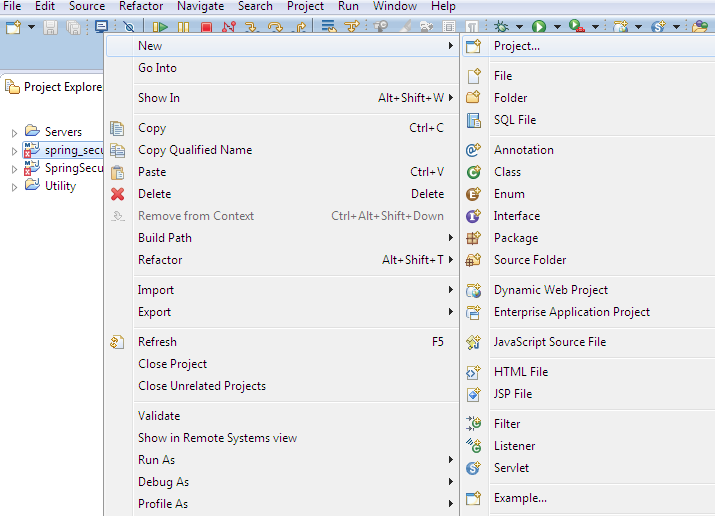
**1) Eclipse IDE Mars Release (4.5.0)**

**2) Java 8**

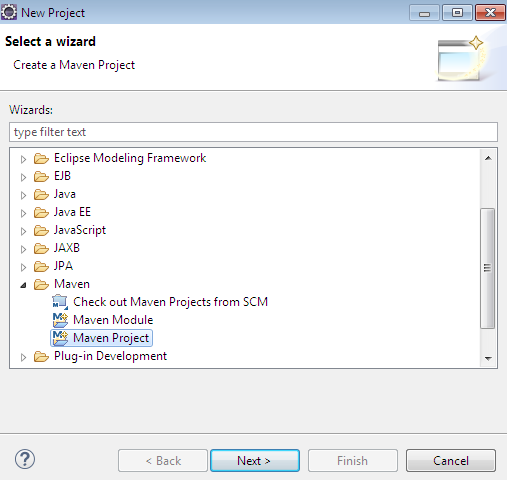
**3) Spring framework 4.2**

**4) Tomcat 9**

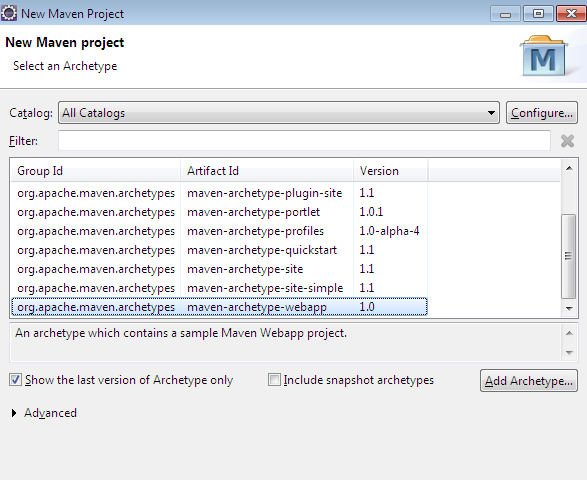
Right click on project explorer and select New Project



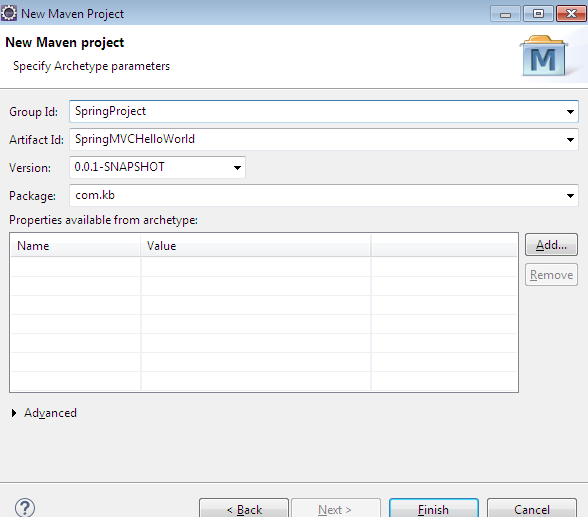
**Select maven project and click on Next**



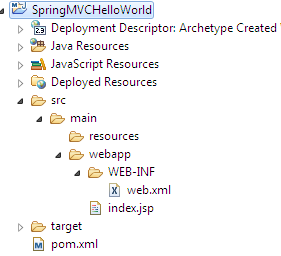
**Click on Next and select maven web app archetype**



**Enter archetype parameters as below and click on Finish**



**Now the Project Structure in eclipse should be**



**Create Spring configuration file**

<?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>"

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-3.2.xsd>

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

<http://www.springframework.org/schema/context/spring-context-3.2.xsd>

<http://www.springframework.org/schema/mvc>

<http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd>">

<context:component-scan base-package="com.ss.\*" />

<mvc:annotation-driven />

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/pages/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

**< context:component-scan >** tag specifies the packages to be scanned for the spring beans.  
So when spring gets loaded , it searches for all the spring beans inside the package specified in the base-package attribute.

Here I have given as com.kb.\* , here it represents all the classes which comes under any folder inside com/kb.  
So our controller is kept inside com/kb/controllers package to let the spring know it.

**View Resolver** is used to specifies how the absolute path of view returned in the controller can be obtained.  
It specifies **prefix**and **suffix**.  
In our case , prefix is **WEB-INF/pages** and suffix is **.jsp**

So any string returned from the controller is followed with this prefix and suffix before rendering.

From **HelloWorldController**we are returning **/helloworld**,  
So with the help of view resolver defined, it becomes **prefix+/helloworld+suffix**  
That is **WEB-INF/pages/helloworld.jsp**  
So this jsp page is going to be displayed when we call this controller.

**Modify web.xml as below**

<web-app xmlns="<http://xmlns.jcp.org/xml/ns/javaee>" xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"

xsi:schemaLocation="<http://xmlns.jcp.org/xml/ns/javaee> <http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd>"

version="3.1">

<display-name>Archetype Created Web Application</display-name>

<!-- Spring MVC dispatcher servlet -->

<servlet>

<servlet-name>mvc-dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>

/WEB-INF/spring-mvc.xml

</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>mvc-dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- Loads Spring Security configuration file -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

/WEB-INF/spring-mvc.xml

</param-value>

</context-param>

</web-app>

**Add**Spring mvc **dispatcher servlet** to enable the **spring mvc framework**  
and **add listener** and filename to **load the spring configuration file**.

**Modify pom.xml to have spring mvc dependencies as below**

<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>SpringProject</groupId>

<artifactId>SpringMVCHelloWorld</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>SpringMVCHelloWorld Maven Webapp</name>

<url>[http://maven.apache.org](http://maven.apache.org/)</url>

<properties>

<org.springframework.version>4.2.0.RELEASE</org.springframework.version>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- Spring MVC depends on these modules spring-core, spring-beans, spring-context,

spring-web -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${org.springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${org.springframework.version}</version>

</dependency>

</dependencies>

<build>

<finalName>SpringHelloWorld</finalName>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.5.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

Now **create**a new folder ‘**java**’ inside **src/main folder**

Spring MVC internally uses a view resolver to fetch the requested view to the user. There are several ways to configure the view in the Spring application. We can use XML, Java code, and property file to configure the view.

Spring MVC defines the ViewResolver and View interfaces that let you render models in a browser without tying you to specific view technology. ViewResolver provides a mapping between view names and actual views. View addresses the preparation of data before handing over to the specific view of technology.

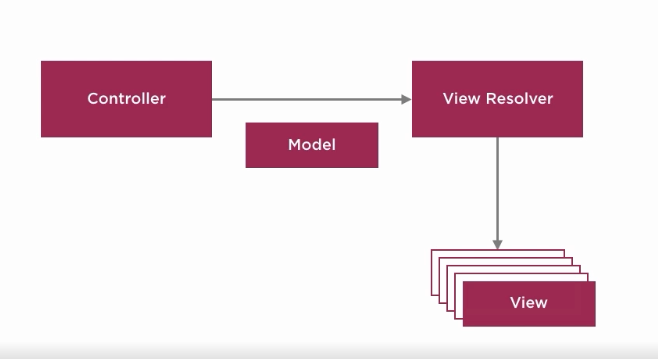
## ViewResolver Implementations

| ViewResolver | Description |
| --- | --- |
| AbstractCachingViewResolver | This class is used to make a cache of the view instance. Caching improves the performance of certain view technologies. Although we can turn off the cache by setting the cache property to false. |
| XmlViewResolver | This class is used to handle the XML view files. It accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. |
| ResourceBundleViewResolver | It uses bean definitions in a ResourceBundle, specified by the bundle base name. For each view it is supposed to resolve, it uses the value of the property [viewname].(class) as the view class and the value of the property [viewname].url as the view URL. |

|  |  |
| --- | --- |
| UrlBasedViewResolver | It used to affect the direct resolution of logical view names to URLs without an explicit mapping definition. |
| InternalResourceViewResolver | It is a convenient subclass of UrlBasedViewResolver that supports InternalResourceView and subclasses such as JstlView and TilesView. |
| FreeMarkerViewResolver | This class is a subclass of UrlBasedViewResolver that supports FreeMarkerView. |
| ContentNegotiatingViewResolver | It is used to resolve a view based on the request file name. |

## How Spring ViewResolver Works?

When a user requests a view then the controller interacts with the ViewResolver. The ViewResolver returns the requested view back to the controller and then the controller back to the user.



Configure views in Spring

There are three ways to configure views in the Spring application. We can use any of these ways to configure view:

XML

Java Code

Property file

1 Configure View using XML Tags

We can use <bean> tag to set class InternalResourceViewResolver and <property> tag to set prefix and suffix for the views page. For example, prefix tells the package location where views files are stored and suffix tells the extension of the view file. See the below code.

<!-- Define Spring MVC view resolver -->

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/views/" />

<property name="suffix" value=".jsp" />

</bean>

Configure View using Java Code

We can configure view handling in Java by adding some code to the configuration class. A class that is marked with @Configuration annotation is treated as a configuration class. You can see, we used setPrefix() and setSuffix() methods as substitutes for <property> tag of the XML file.

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.ViewResolver;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration

public class AppConfig {

@Bean

public ViewResolver viewResolver() {

InternalResourceViewResolver irvr = new InternalResourceViewResolver();

irvr.setPrefix("WEB-INF/views");

irvr.setSuffix(".jsp");

irvr.setOrder(0);

return irvr;

}

}

# Adding Data to Spring Model

The Spring Model is an interface in Spring that is used to add attributes to the model. It provides several methods to handle the data from controller to view.

It mainly used to pass data from the controller to the view page

## Methods of Model Interface

The following are the methods of the Model interface.

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| Model addAllAttributes(Collection<?> attributeValues) | This method is used to copy all attributes in the supplied collection into this Map, using attribute-name generation for each element. |
| Model addAllAttributes(Map<String,?> attributes) | It copies all attributes in the supplied Map into this Map. |
| Model addAttribute(Object attributeValue) | It adds the supplied attribute to this Map using a generated name. |
| Model addAttribute(String attributeName, Object attributeValue) | It adds the supplied attribute under the supplied name. |
| Map<String,Object> asMap() | It returns the current set of model attributes as a Map. |
| boolean containsAttribute(String attributeName) | It returns true if this model contains an attribute of the given name, false otherwise. |
| Object getAttribute(String attributeName) | It returns the attribute value for the given name if any. |
| Model mergeAttributes(Map<String,?> attributes) | It copies all attributes in the supplied Map into this Map, with existing objects of the same name taking precedence. |

# Spring @RequestParam Annotation

Spring @RequestParam annotation is used to fetch the value of a parameter in the form request. In Spring MVC, "request parameters" map to query parameters, form data.

For example, if we want to get parameter(user\_name) value from a requested URL then we can use @RequestParam annotation. We just need to provide a parameter name.

<http://localhost:8080/spring-mvc-app/showdata?username=s>arika

@RequestMapping("showdata")

public String userForm(@RequestParam("user\_name") String name, Model model)

{

model.addAttribute("name", name);

return "user-data";

}

The @RequestParam annotation uses several attributes like **name**, **required**, **defaultvalue**, etc. We can use them in our application based on the use cases.

### DefaultValue Attribute

Set DefaultValue for the parameter, to avoid null value. We can use this attribute to handle missing values. If the parameter does not contain any value then this default value will be supplied.

@RequestMapping("showdata")

public String userForm(@RequestParam(defaultValue = "No\_value") String name, Model model)

{

model.addAttribute("name", name);

return "user-data";

}

### Required Attribute

The required attribute is used to handle exceptions in case of missing data. It throws **HTTP Status 400 – Bad Request** (Required String parameter 'user\_name' is not present). To avoid this exception, we can use the required attribute.

@RequestMapping("showdata")

public String userForm(@RequestParam(required = false) String name, Model model)

{

model.addAttribute("name", name);

return "user-data";

}

### Java 8 Optional Class

We can use Optional class to avoid exceptions in case of missing data by providing a value. It is similar to the default attribute but can be used if you are working with java 8 or higher version.

@RequestMapping("showdata")

public String userForm(@RequestParam Optional<String> name, Model model)

{

String name1 = name.orElseGet(()->"no value found");

model.addAttribute("name", name1);

return "user-data";

}

## @RequestParam Attributes

The following are the attributes list of @RequestParam annotation.

|  |  |
| --- | --- |
| **Type and Element** | **Description** |
| String defaultValue | This element is used to set a default value to the parameter. |
| String name | It indicates the name of the request parameter to bind to. |
| boolean required | It is used to set whether the parameter is required. |
| String value | It is similar to name elements and can be used as an alias. |

**Spring @ModelAttribute Annotation**

Spring @ModelAttribute annotation is used to bind a method parameter or method return value to a named model attribute which further can be used in the view page.

It can be used either at the method level or method parameter level.

**@ModelAttribute at a Method Level**

We can use this annotation at the method level to specify that this method may add one or model attributes.

@PostMapping("save")

@ModelAttribute("user")

public String save(User user, Model model) {

model.addAttribute("user", user);

return "response";

}

@ModelAttribute at a Method Argument

We can use it with the method argument that indicates the argument should be retrieved from the model. If the argument is not present then first it should be instantiated and then added to the model.

See in this example, we used @ModelAttribute with the argument.

@PostMapping("save")

public String save(@ModelAttribute("user") User user, Model model) {

model.addAttribute("user", user);

return "response";

}

**Spring MVC Form Binding:**

**Spring form tags** are defined in the **spring-form.tld** and they are bundled in **spring.jar**

To use spring form tags, we need to add below **taglib directive** on top of jsp page.

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

where ‘**form**’ is the tag name **prefix**you want to use for the tags from this library.  
This tag acts as an **html form tag + binding the form data**

@ModelAttribute= “user”

List of SpringMVCFormTags:

Input <formlinput>

Checkbox <form:checkbox>

Radiobutton

Select and option

Textarea

Hidden

##### Lets understand about different form tags of spring through example



HttpServletRequest

@RequestParam

**Spring MVC Form Validation**

to validate user data on the server-side. It is one of the essential tasks that we need to perform while creating a web application.

To get user data in the valid form we need to validate it either on the client-side or server or both. Spring provides a @Valid annotation to get valid form data and use **Hibernate Validator API** to validate data.

Hibernate Validator is an implementation API for Bean validation in the Spring framework. It provides several annotations to make a clean and simple validation integration with Java code. It works on both server and client application programming.

## Hibernate Validator Annotations

The following are commonly used annotations for data validation in the Spring application.

|  |  |
| --- | --- |
| **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. |
| @Email | It checks whether the specified character sequence is a valid email address. |
| @NotEmpty | It checks whether the annotated element is not null nor empty. |
| @Null | It checks that the annotated value is null. |
| @Negative | It checks if the element is strictly negative. Zero values are considered invalid. |
| @NegativeOrZero | It checks if the element is negative or zero. |

If you are using the maven project then add the following dependency to the **pom.xml** file. For the latest hibernate validator dependency, you can visit its [official site](https://hibernate.org/validator/documentation/getting-started/).

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>6.1.1.Final</version>

</dependency>

## Spring MVC Form Validation Example

Let's understand by example and create a maven-based Spring project and add these files. After that run this example using web server (Apache Tomcat). See the source code of the project below.

**// HomeController.java**

It is a controller class file that uses @Controller annotation. It has two methods, one is used to show the user form, and the second is used to validate the submitted user form.

package com.studytonight.controllers;

import javax.validation.Valid;

import org.springframework.stereotype.Controller;

import org.springframework.validation.BindingResult;

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

import org.springframework.web.bind.annotation.PostMapping;

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

import com.studytonight.models.UserForm;

@Controller

public class HomeController {

@GetMapping("/userform")

public String index(@ModelAttribute("registration") UserForm registration) {

return "index";

}

@PostMapping("/userform")

public String register(@Valid @ModelAttribute("registration") UserForm registration, BindingResult result) {

if (result.hasErrors()) {

System.out.println("error");

return "index";

}

return "success";

}

}

**// MainApp.java**

This file contains code to create an IOC container for our application. The AnnotationConfigWebApplicationContext class is used to create an object for application context.

package com.studytonight;

import javax.servlet.ServletContext;

import javax.servlet.ServletException;

import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;

import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

import org.springframework.web.servlet.DispatcherServlet;

public class MainApp implements WebApplicationInitializer {

@Override

public void onStartup(ServletContext servletContext) throws ServletException {

AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();

context.register(AppConfig.class);

context.setServletContext(servletContext);

ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher", new DispatcherServlet(context));

servlet.setLoadOnStartup(1);

servlet.addMapping("/");

context.close();

}

}

**// AppConfig.java**

**This is a configuration file which is an alternate of the applicationContext.xml file that we created for the XML-based configuration example. The @Configuration annotation indicates that this is not a simple class but a configuration class and the @ComponentScan annotation is used to indicate the location of the component class in our spring project.**

**package com.studytonight;**

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

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

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

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

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

**import org.springframework.web.servlet.ViewResolver;**

**import org.springframework.web.servlet.config.annotation.EnableWebMvc;**

**import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;**

**import org.springframework.web.servlet.view.InternalResourceViewResolver;**

**@EnableWebMvc**

**@Configuration**

**@ComponentScan("com.studytonight.controllers")**

**public class AppConfig implements WebMvcConfigurer{**

**@Autowired**

**ApplicationContext applicationContext;**

**@Bean**

**public ViewResolver viewResolver() {**

**InternalResourceViewResolver ivr = new InternalResourceViewResolver();**

**ivr.setPrefix("/WEB-INF/views/");**

**ivr.setSuffix(".jsp");**

**ivr.setOrder(0);**

**return ivr;**

**}**

**}**

**// index.jsp**

**This is a JSP file that contains an HTML form that will be shown when the application runs.**

**<%@ page language="java" contentType="text/html; charset=ISO-8859-1"**

**pageEncoding="ISO-8859-1"%>**

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

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

**<!DOCTYPE html>**

**<html>**

**<head>**

**<meta charset="ISO-8859-1">**

**<title>User Form</title>**

**<style type="text/css">**

**.error {**

**color: #ff0000;**

**}**

**.errorblock {**

**color: #000;**

**background-color: #ffEEEE;**

**border: 3px solid #ff0000;**

**padding: 8px;**

**margin: 16px;**

**}**

**</style>**

**</head>**

**<body>**

**<form:form modelAttribute="registration">**

**<form:errors path="\*" cssClass="errorblock" element="div" />**

**<br>**

**<label>Enter User Name </label>**

**<br>**

**<br>**

**<form:input path="name" />**

**<br>**

**<br>**

**<label>Enter Email </label>**

**<br>**

**<br>**

**<form:input path="email" />**

**<br>**

**<br>**

**<input type="submit" value="Register">**

**<br>**

**</form:form>**

**</body>**

**</html>**

**// success.jsp**

**This JSP file will be shown as a response if the user form is validated successfully.**

**<%@ page language="java" contentType="text/html; charset=ISO-8859-1"**

**pageEncoding="ISO-8859-1"%>**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<meta charset="ISO-8859-1">**

**<title>Insert title here</title>**

**</head>**

**<body>**

**<h2>Registered Successfully</h2>**

**</body>**

**</html>**

**// pom.xml**

**This file contains all the dependencies of this project such as spring jars, servlet jars, etc. Put these dependencies into your project to run the application.**

**<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<groupId>com.studytonight</groupId>**

**<artifactId>springmvc</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<packaging>war</packaging>**

**<dependencies>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-core</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-webmvc</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<dependency>**

**<groupId>org.hibernate.validator</groupId>**

**<artifactId>hibernate-validator</artifactId>**

**<version>6.1.1.Final</version>**

**</dependency>**

**</dependencies>**

**<properties>**

**<spring.version>5.2.8.RELEASE</spring.version>**

**</properties>**

**<build>**

**<sourceDirectory>src</sourceDirectory>**

**<plugins>**

**<plugin>**

**<artifactId>maven-compiler-plugin</artifactId>**

**<version>3.8.1</version>**

**<configuration>**

**<source>1.8</source>**

**<target>1.8</target>**

**</configuration>**

**</plugin>**

**<plugin>**

**<artifactId>maven-war-plugin</artifactId>**

**<version>3.2.3</version>**

**<configuration>**

**<warSourceDirectory>WebContent</warSourceDirectory>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

**</project>**

**// UserForm.java**

**import javax.validation.constraints.Email;**

**import javax.validation.constraints.NotEmpty;**

**public class UserForm {**

**@NotEmpty(message = "Name can not empty")**

**String name;**

**String lastName;**

**@NotEmpty(message = "Email can not empty")**

**@Email**

**String email;**

**public UserForm(String name, String lastName, String email) {**

**this.name = name;**

**this.lastName = lastName;**

**this.email = email;**

**}**

**public String getEmail() {**

**return email;**

**}**

**public void setEmail(String email) {**

**this.email = email;**

**}**

**public String getName() {**

**return name;**

**}**

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

**this.name = name;**

**}**

**public String getLastName() {**

**return lastName;**

**}**

**public void setLastName(String lastName) {**

**this.lastName = lastName;**

**}**

**}**

**Run the Application**

**After successfully completing the project and adding the dependencies run the application using the server**

**SpringMVCValidationAPI**

@NotNull

@Email

@Size

@NotEmpty

In spring MVC , we can validate the form in many ways , we can use validation API ,custom validator or directly validating in controller after the form submit.

Lets see how we can do the same using Java Validation API.

Java Validation API suggests us to use annotation on the model class for each attribute to specify the validation constraints.

Example:

public class Customer {

    @NotNull

    @Email

    private String emailId;

    @NotNull

    @Size(min=8,max=15)

    private String password;

//getters  and setters

}

So we are using **@NotNull** to specify that emailId and password fields should not be null.

**@Email** – specifies that , emailId field is getting validated as per the email standard provided by Hibernate validator.

**@Size** specifies that password field should be minimum of 8 characters and maximum of 15 characters.

We can use many other annotations to validate the fields in the form.

In the controller class, specify the form backing model object with @Valid annotation to ensure that form values gets binded with this model object and do the validation.

@Controller

public class LoginController {

     @RequestMapping(value = "/doLogin", method = RequestMethod.POST)

        public String doLogin(@Valid User user, BindingResult result) {

            // login logic here

        }

}

If any validation error occurs as per our validation defined in the model class, then BindingResult will have the errors.

if (result.hasErrors()) {

    // form validation error, return the same form

} else {

    // form input is fine, continue the flow

}

so we can check the binding results for any errors.  
If it has errors we can return the same form with error message, if not we can continue the flow.

We can display the error messages on the jsp by using spring’s form errors tag as follows

<form:errors path="emailId" />

Where **emailId**is the exact attribute name in the Customer Model class.

We can also specify the error message while specifying the validation constraints in the model class with annotation as below

@NotEmpty(message = "Email Id is Required.")

private String emailId;

since error message has to be localized, we will specify it in the properties file.

The way of specifying the key in the properties file is

ConstraintName.modelAttributeName.propertyName=validation error message

Example

NotEmpty.customer.emailId=Email Id is Required.

# Spring MVC form validation with custom validator

In the previous example, we have seen the form validation using java validations by annotating model class with constraint validation annotations.

Good part of it is, we can just add annotations on the fields of the model class and if any errors in the form as per the annotations , Binding Result will get the errors.

But bad part of it is , we will not get annotations for all our **business validations** on the form fields.

Example : If we want to Validate age field to be **greater than 18 and less than 60** to be allowed then we need to define our own validator.

**So lets see how we can write our custom validator**We need to do following things to define our custom validator

1)Create the custom validator class for model class on which we validate and implement the **validator**interface, make custom validator class as spring bean.  
2)Override **supports(Class clazz)** method  
3)Override **validate(Object target,Errors errors)** method  
4)After this we need to inject this custom validator in the controller class and call its **validate()** method  
5)Then check the binding result for any errors and return the appropriate view.

//==================================================================