## Spring:

Framework is a piece of software. This resolves the commonly and repeatedly occurred problems in multiple projects.

Spring framework is **an open source** Java platform and it was initially **written by Rod Johnson** and was first released under the Apache 2.0 license in June 2003.

Spring is more flexible framework when we compared with struts. By using Struts 1.3 and 2.3 we can develop only web based applications. But by using spring we can develop web apps as well as Standard alone applications also.

We can use any of the following frameworks to develop the web based applications.

The y are :

1. Struts 2.Spring 3.Jsf 4.Web work 5.OpenSymphony

**Spring framework** is a piece of software which **contains the solutions** to commonly repeatedly occurred problems across multiple projects.

**The following are the advantages of Frameworks like Struts and spring:-**.

- Frameworks resolve the problems of identifying the architecture. Every framework like Struts and Spring is delivered with MVC2 architecture.
- If we use Servlets and Jsps we have to develop our own Controllers. If we use frameworks like Struts and **Spring internally they came with Controllers**

Eg: **ActionServelt** in Struts

**DispatcherServlet** in Spring

- When we use any framework we no need to **use RequestDispatcher code** as well as we no need to hard code the resource path names. By using these frameworks we can configure them in the configuration files.
- If we use JDBC, Servlets, Jsps to develop the form based applications we have to write huge amount of code to take care of **Server side validations** and displaying errors in the same form.
- By using JDBC, Servlets, Jsps we have to provide huge amount of code to develop the **I18n applications**. (The programs which displays the output based on client regional Languages)

- The frameworks like Struts and spring delivered with set of **predefined tag libraries**. If we use Servlets and Jsps we have to develop our own tag library which is difficult.

- When We use the frameworks like Struts and spring we can use **pdf/velocity/jsf as view components**.

Most of the experienced guys are develops the new frameworks.

Every Company uses their own frameworks to develop the projects .all these frameworks internally uses the other frameworks.

**Benefits of Using Spring Framework:**

Following is the list of few of the great benefits of using Spring Framework:

- Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.

- Spring is organized in a **modular fashion.** Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.

- Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.

- Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBean-style POJOs, it becomes easier to use dependency injection for injecting test data.

- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks.

- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

- Lightweight **IoC containers** tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.

- Spring provides a **consistent transaction management** interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

The main feature of spring is **IOC.**

## What is IOC?

**IOC stands** for Inversion of Control. A person ' x' acts as supposed to do a work instead of that person perform that tasks some other person 'y' completed the task and the result is enjoyed by ' X' person. We will call this as Inversion of Control.

Spring supports **AOP**. AOP stands for Aspect Oriented Programming.

The advantages of Aop are all the common code is place in aspect before the business logic method is executed spring call the Aspect. The aspect internally called the business logic method.

As per spring they have provided the transaction Manager aspect. He will take care about the transaction managing in Spring.

Spring framework follows mVc2 architecture and it is an open source.

Spring frame work is divided into couple of modules. The advantage of this approach is we can include only required modules in our project.

The module is a collection classes and interfaces and enumerations.

Spring framework is divided into six modules :

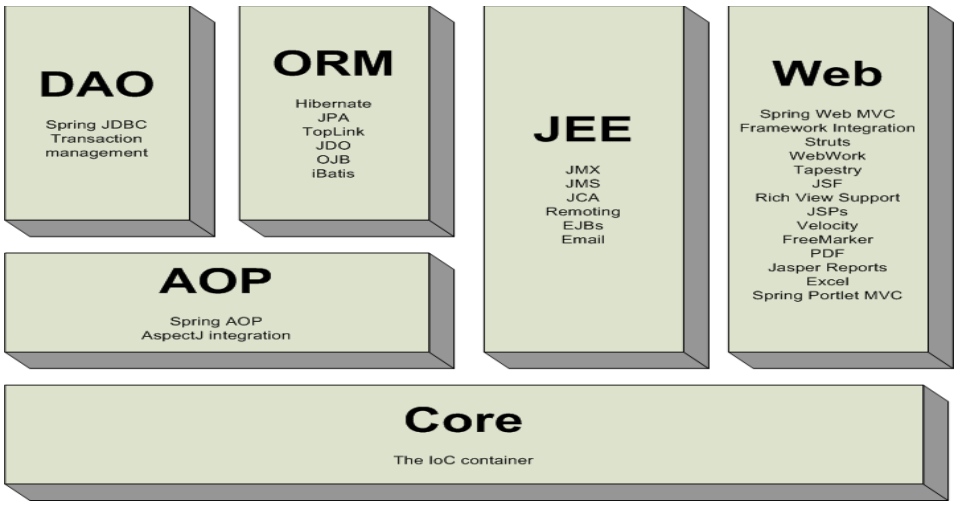
They are:

1.Core 2.DAO 3.ORM 4.AOP 5.JEE 6.WEB

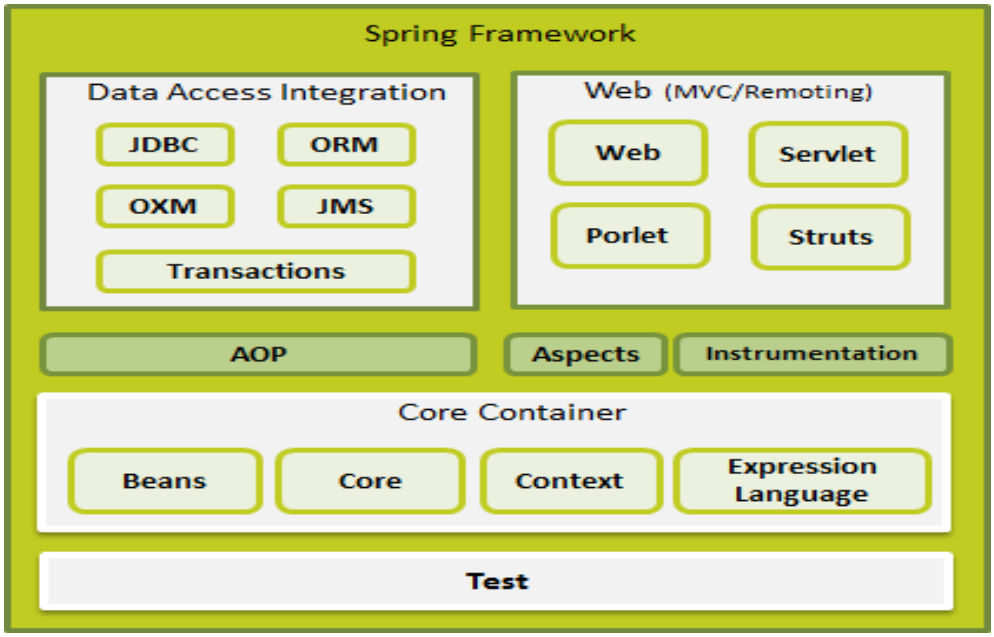
Sometime in the project we will use only core ,orm, aop only..

The following is the architecture of spring 2.0

**spring 2.0 modules**




The following is the diagram of **spring 3.0 modules:**




**Core module:**

As part of core module we have IOC.

The Core Container consists of the Core, Beans, Context, and Expression Language modules whose detail is as follows:

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

**Data Access/Integration:**

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

- The **DAO** module: When we use DAO module we can reduce a lot of code in project. DAO module resolves the all **problems of database specific errors**.
- The **JDBC** module provides a JDBC-abstraction layer that removes the need to do tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The advantage of spring orm is if we knowledge in one orm we can work any other orm module easily
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

**Web:**

The Web layer consists of the **Web, Web-Servlet, Web-Struts, and Web-Portlet** modules whose detail is as follows:

- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-Servlet** module contains Spring's model-view-controller (MVC) implementation for web applications.
- The **Web-Struts** module contains the support classes for integrating a classic Struts web tier within a Spring application.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

**Miscellaneous:**

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules whose detail is as follows:

**AOP:**

- AOP stands for Aspect oriented program. this resolves the problem of Oops.(inheritance between the multiple classes)
- The open source group people come with Programming model called AOP. The software AspectJ followed the Aop and developed the Aop module. Spring guys has developed some classes which follows the aop module.
- The **AOP** module provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ which is again a powerful and mature aspect oriented programming (AOP) framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

**JEE module in spring2.0:**

**JEE** stands for java environment enterprise.

As part of jee module spring guys has supported **EJB and MailAPI.**

## DepencyLookup:

If the resource of the application searches and gathers its dependent values from other resources of the application is called as dependency lookup. In dependency lookup resources pulls the values from other resources.

If values are assigned to variables only after calling the method explicitly is called as dependency lookup. The way the client app gathers **DataSource** object reference from registry software is called dependency lookup.

## Dependency injection:

If underlying container or framework or server or runtime environment is dynamically assigning dependent values to resources of the application then its called **Dependency injection or IOC**.

In **Dependency injection** underlying container/framework dynamically pushes the values to resources..

Eg: The way ActionServelt writes the form data to FormBean class object comes under Dependency injection.

Spring supports three types of **Dependency injections:**

1. Setter Injection (By using setXxx(xxx) methods)
2. Constructor injection (by using constructors)
3. Interface Injection (by implementing special interfaces)

Any predefined or userdefined java class can be called as Spring bean.

The java class that contains only Persistence logic and separate that logic from other logics of application is called as **DAO.**

Spring supports POJO class . POJO class is nothing but a class which is not extending or any other class or interface.

## Procedure to set up the first Spring application:

**Steps:**

1. Create a work space for spring application .
2. Create the java project .

3. Add the spring libraries to project by selecting add spring capabilities in MyEclipse. And add user library in the eclipse ide following way:
   **Project--→properties----------→JavaBuildpath---------→Add library-----→ UserLibrary**--------→new ---→ give name for lib (spring)-------→Add jars--→ **Then add all spring jar files.**

**For developing Spring app u have to create the following classes**:

1. Spring bean class
2. Spring bean configuration class
3. Client application to get the bean class object

**Developing Spring bean class:**

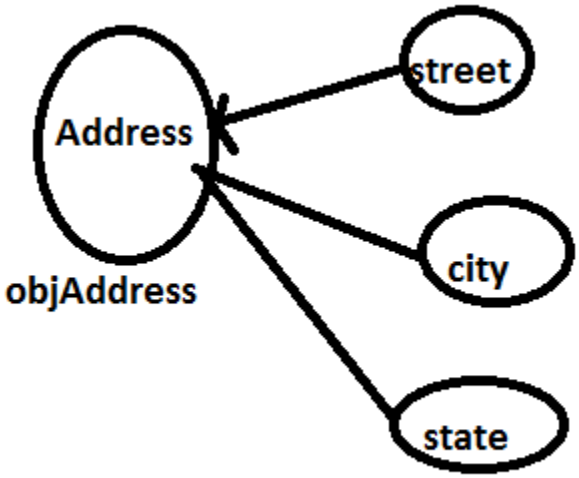
We have created the Address class with three properties

```
Public class Address {
        String street;
        String city;
        String state;
        //setter and getter for the above class like follow
        Public void setStreet(String street){
                this.street = street;
        }
        public String getStreet(){
                return street;
        }
}
```

we have created Address class object as follow as

**Address objAddress = new Address();**

we called a method **objAddress.setStreet("srnagar")** when this setter is called we have created a String object and given supplied input to that method. And it supply the value for instance variable.




**objAddress.setStreet("srnagar");**

**objAddress.setStreet("hyd");**

**objAddress.setStreet("ap");**

From above diagram we have observed that objAddress is depedent on String objects. The meaning this is Address object is using the String object.

when the setStreet() is executed it stabilize the dependency between String and Address object. As the Setter() method is stabilizes the dependency between two objects we are calling this as **Setter injection.**

Instead of creating an object and stabilize the dependency, Spring container will take care of creating object and stabilize the dependency or injecting the dependencies We will call this as Inversion of Control or dependency injection.

According to Spring documentation Ioc is called as Dependency injection.

In Spring we will develop the pojo or spring bean class . If spring Container want to take care of Ioc then we have to configure that in Spring configuration file.(ApplicationContext.xml)

The following is example of spring bean configuration file:

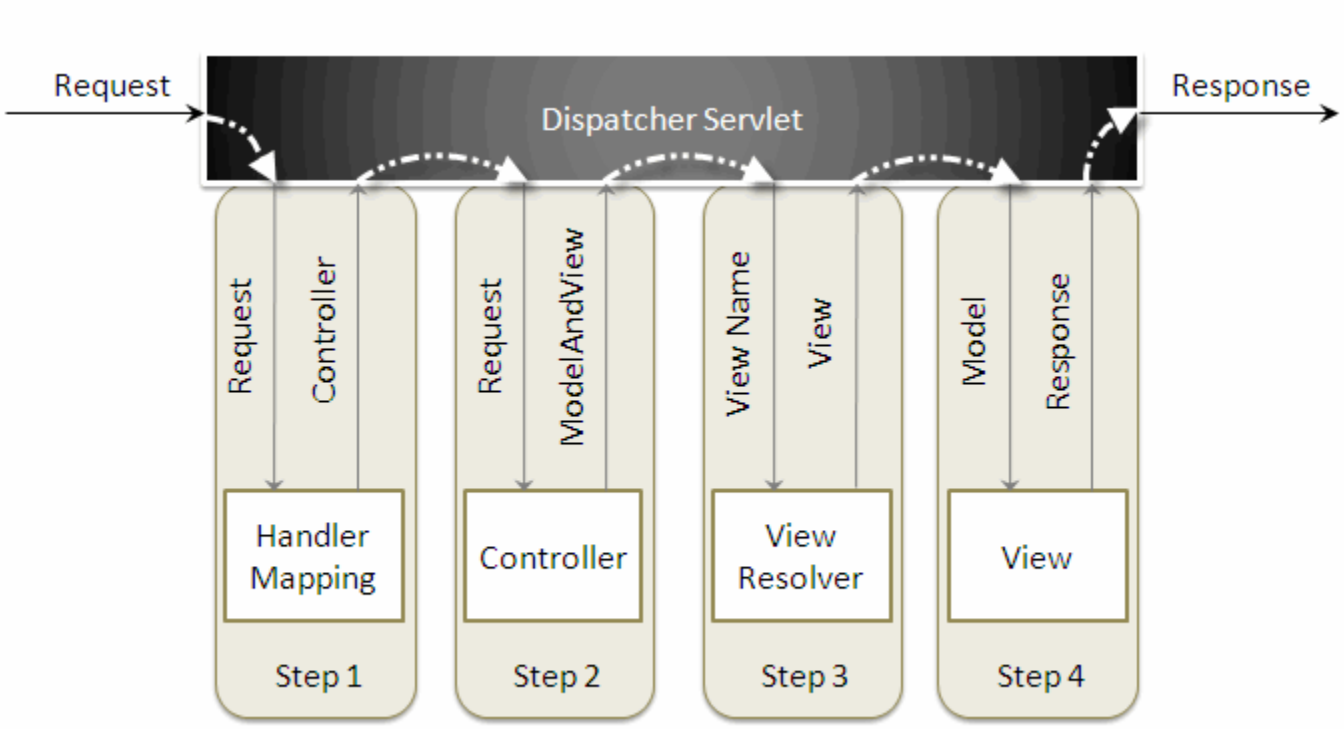
```
<beans>
        <bean id=””   class=”” lazy-init=””>
                <property > </property >
        </bean>
<beans>
```

**Procedure to configure spring bean in Spring configuration file in MyEclipse:**

# Spring MVC Framework

Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in **seperating the business logic**, presentation logic and navigation logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object . Controllers are responsible for receiving the request from the user and calling the back-end services.

The figure below shows the flow of request in the Spring MVC Framework.



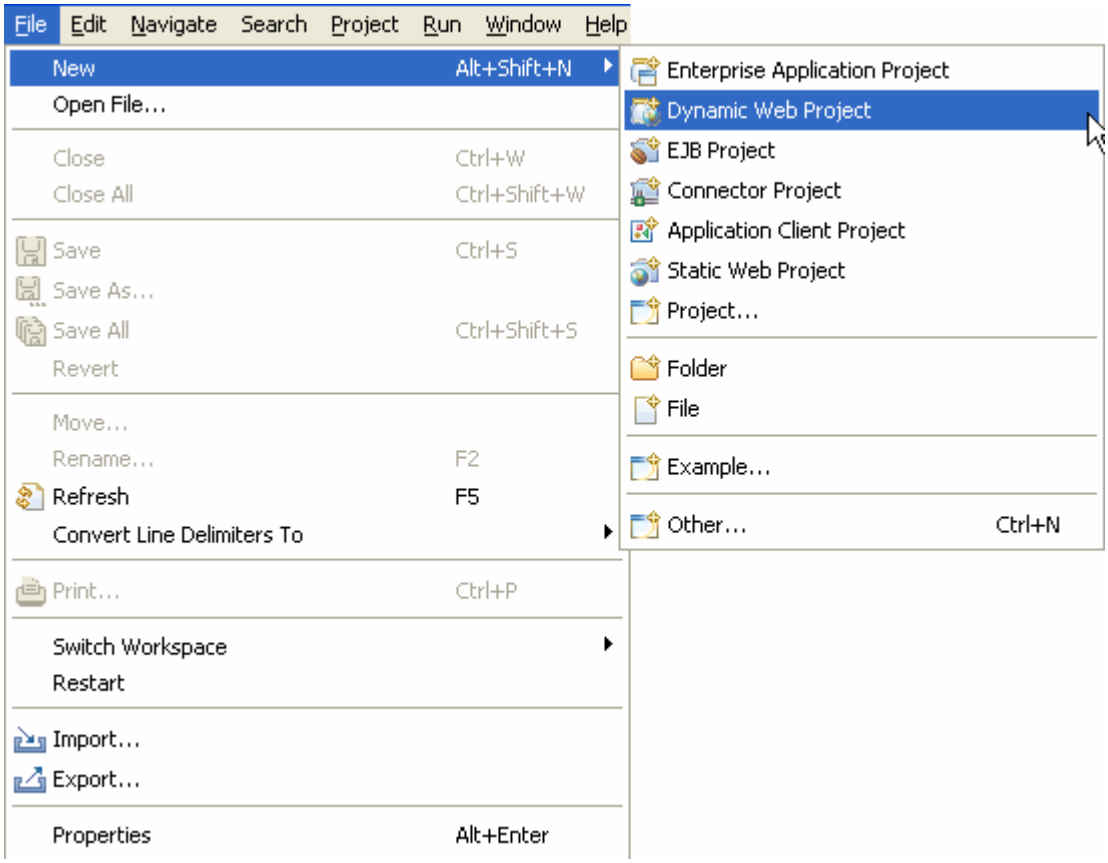

When a request is sent to the Spring MVC Framework the following sequence of events happen.

- The *DispatcherServlet* first receives the request.
- The *DispatcherServlet* consults the *HandlerMapping* and invokes the *Controller* associated with the request.
- The *Controller* process the request by calling the appropriate service methods and returns a*ModeAndView* object to the *DispatcherServlet*.
  The *ModeAndView* object contains the model data and the view name.

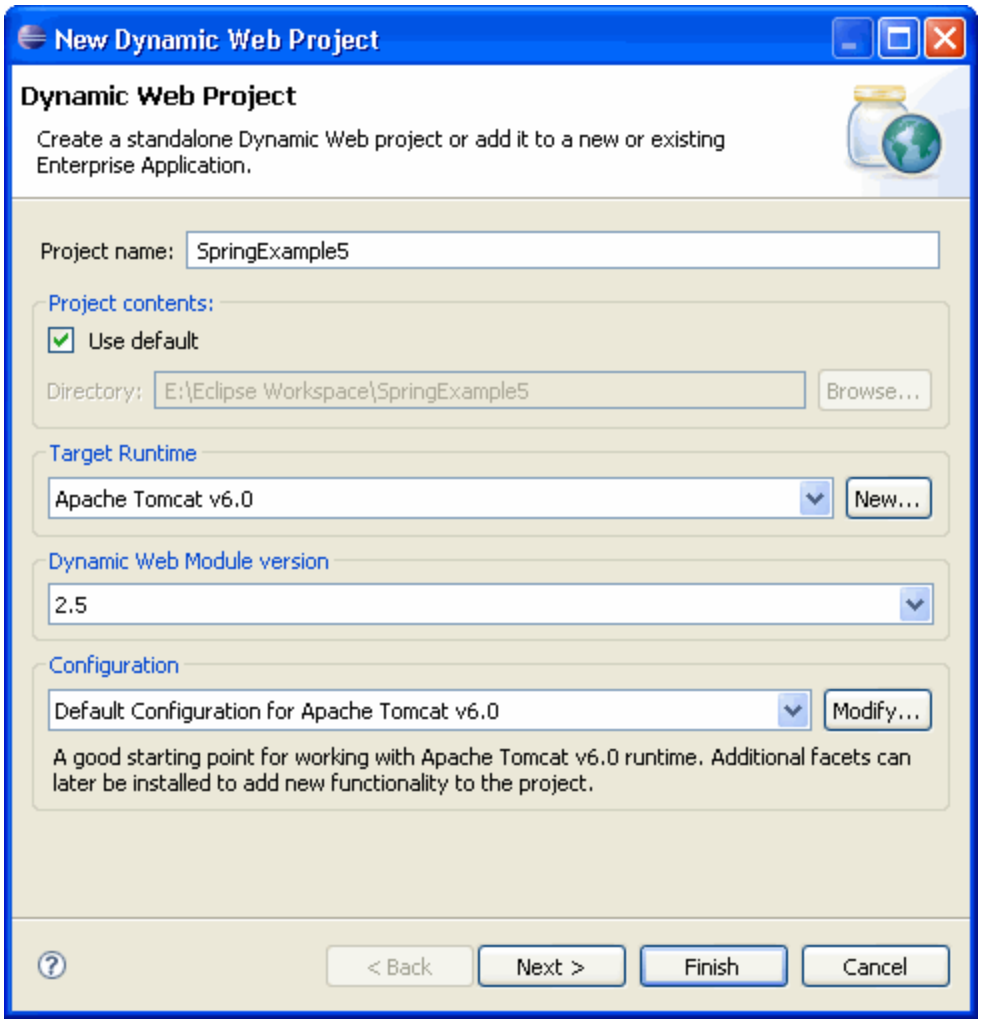
- The *DispatcherServlet* sends the view name to a *ViewResolver* to find the actual *View* to invoke.
- Now the *DispatcherServlet* will pass the model object to the *View* to render the result.
- The *View* with the help of the model data will render the result back to the user.

To understand the Spring MVC Framework we will now create a simple hello world example using the Eclipse IDE. I am using Exclipse IDE 3.4 , Spring IDE plugin, Tomcat 6.0 and Spring 3.0 to demonstrate this example.

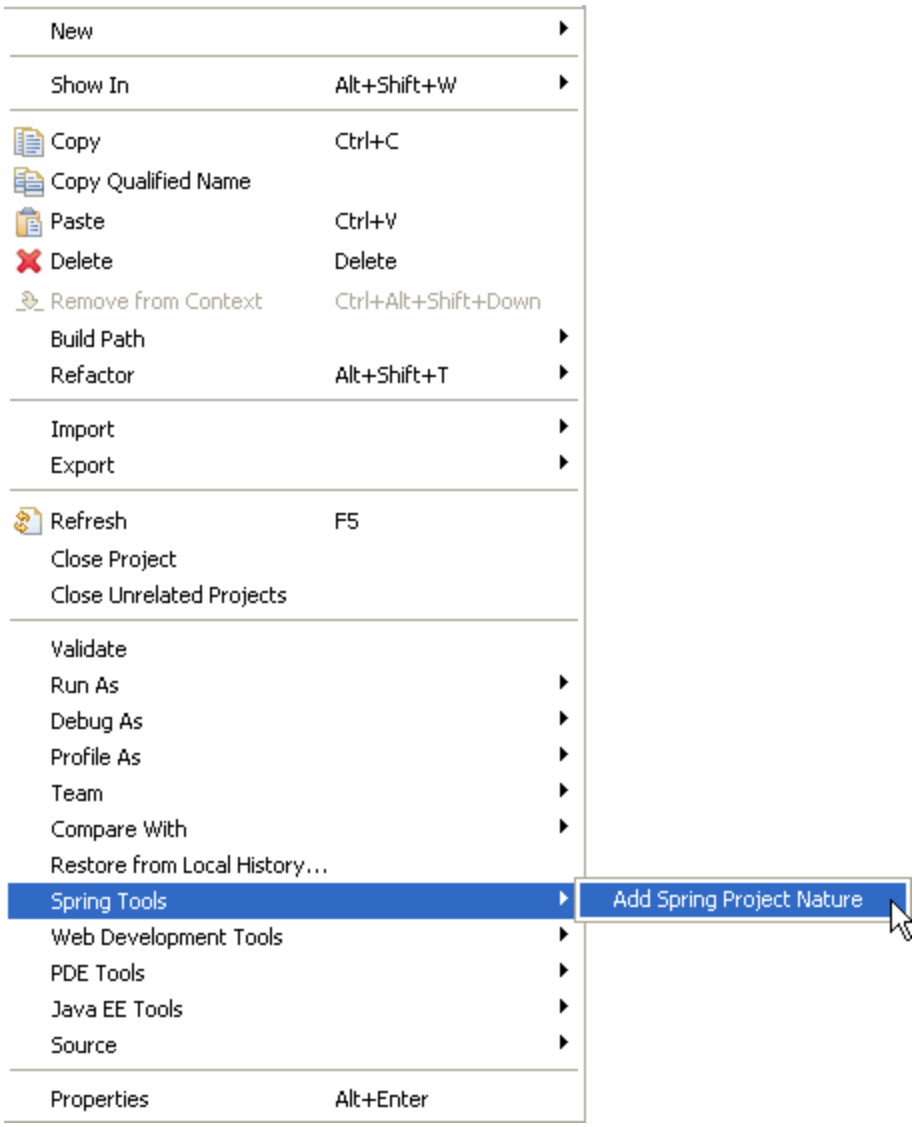
Go to *File -> New -> Dynamic Web Project*, to create a web project.



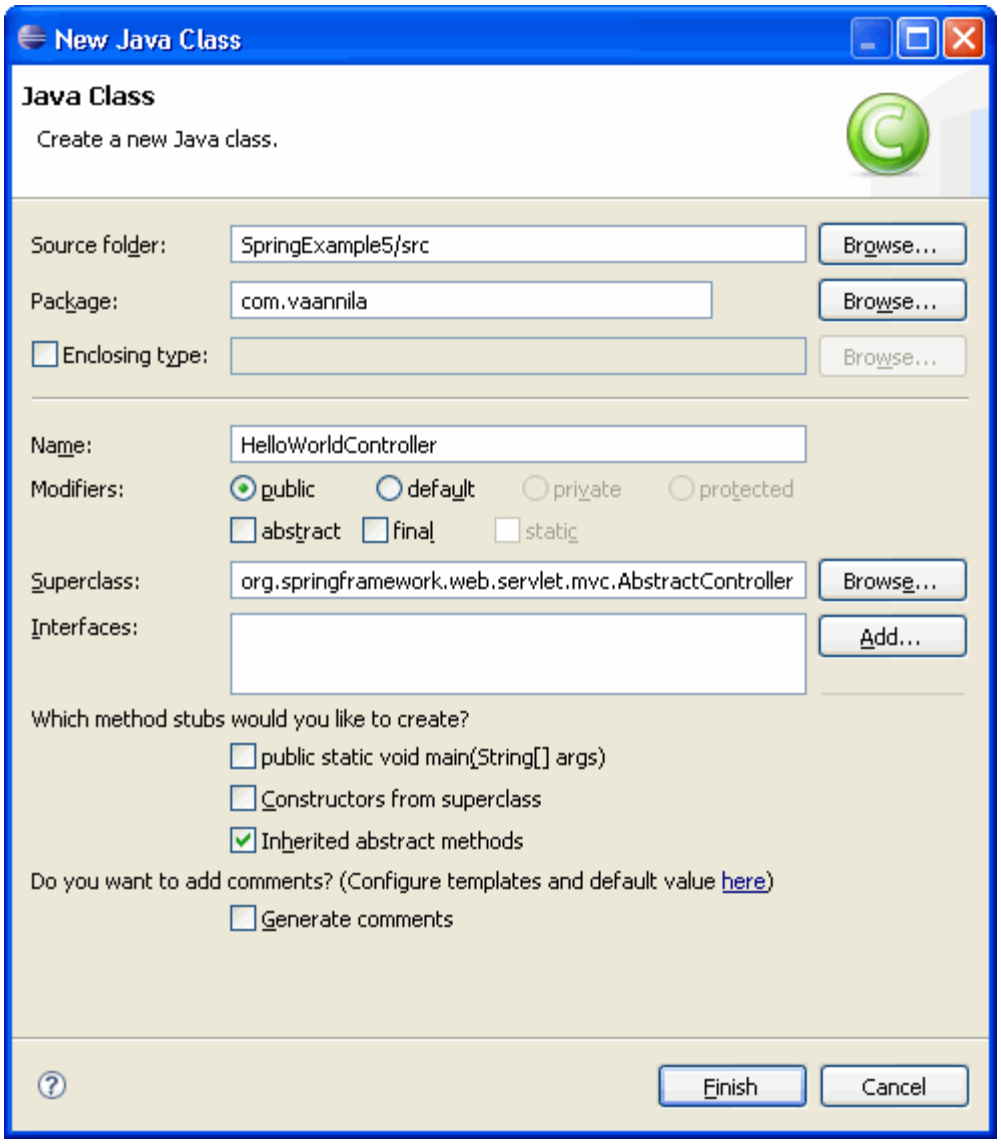

Enter the project name and click the *Finish* button.




Right click the project folder, and select *Spring Tools -> Add Spring Project Nature*, to add Spring capabilities to the web project. This feature will be available once you install the Spring IDE.



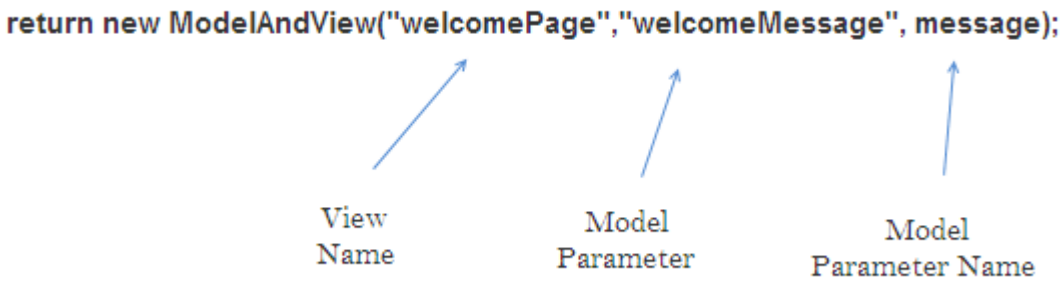

Create a new package *com.vaannila* inside the *src* directory. The Spring controller class extends *org.springframework.web.servlet.mvc.AbstractController* class. To create a new controller class right click the *src* directory and create a new java class, enter the controller class name and super class name and the *Finish* button.




Copy the following code inside the *HelloWorldController* class.

```
view source

print?

01.import javax.servlet.http.HttpServletRequest;

02.import javax.servlet.http.HttpServletResponse;

03.

04.import org.springframework.web.servlet.ModelAndView;
```

```
05.import org.springframework.web.servlet.mvc.AbstractController;
06.
07.public class HelloWorldController extends AbstractController {
08.
09.private String message;
10.
11.@Override
12.protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response) throws Exception {
13.return new ModelAndView("welcomePage","welcomeMessage", message);
14.}
15.
16.public void setMessage(String message) {
17.this.message = message;
18.}
19.
20.}
```



The *DispatcherSevlet*, as the name indicates, is a single servlet that manages the entire request-handling process. When a request is sent to the *DispatcherServlet* it delegates the job by invoking the appropriate controllers to process the request. Like any other servlet the *DispatcherServlet* need to be configured in the web deployment descriptor as shown.

view source

print?

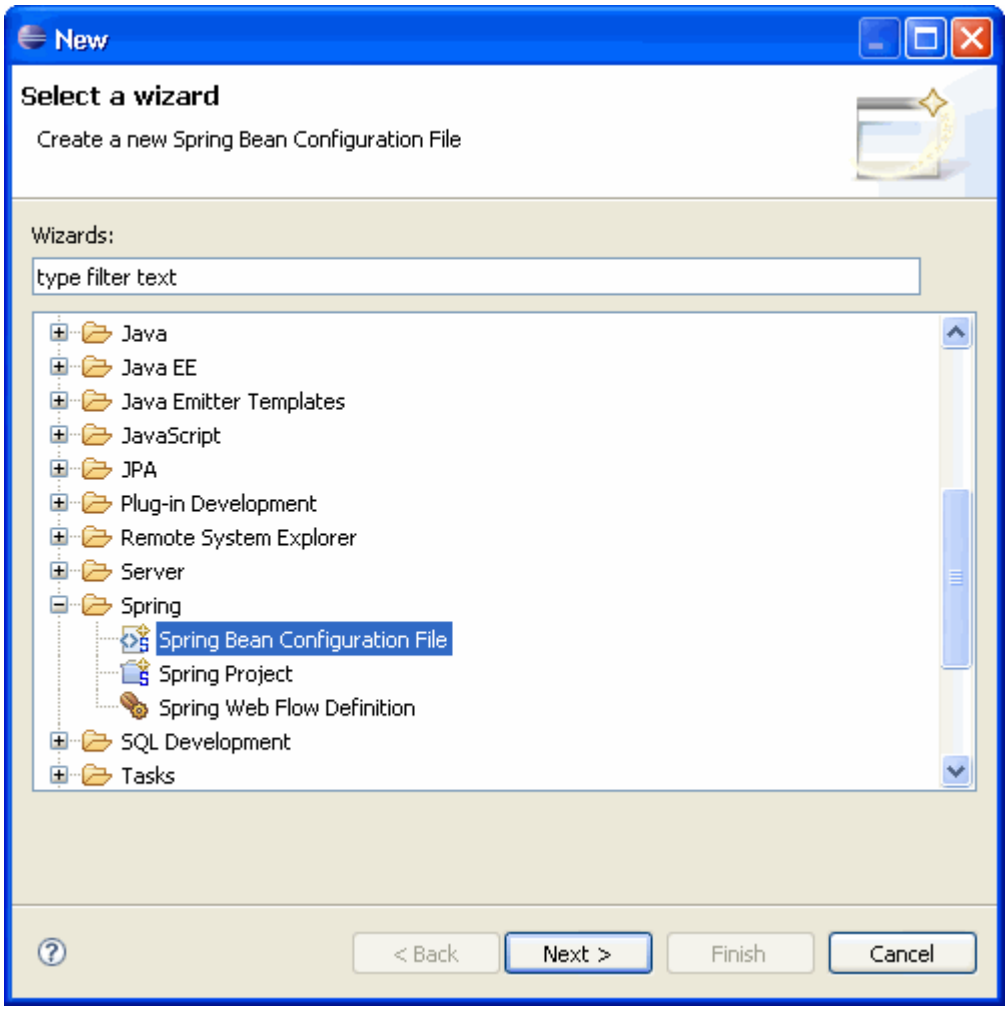
```
01.<?xml version="1.0" encoding="UTF-8"?>

02.<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"xmlns="http://java.sun.com/xml/ns/javaee"xmlns:web="http://java.sun.
com/xml/ns/javaee/web-
app_2_5.xsd"xsi:schemaLocation="http://java.sun.com/xml/ns/javaeehttp://java.
sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"version="2.5">

03.<servlet>

04.<servlet-name>dispatcher</servlet-name>

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

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

07.</servlet>

08.<servlet-mapping>

09.<servlet-name>dispatcher</servlet-name>

10.<url-pattern>*.htm</url-pattern>

11.</servlet-mapping>

12.<welcome-file-list>

13.<welcome-file>redirect.jsp</welcome-file>

14.</welcome-file-list>
```

```
15.</web-app>
```

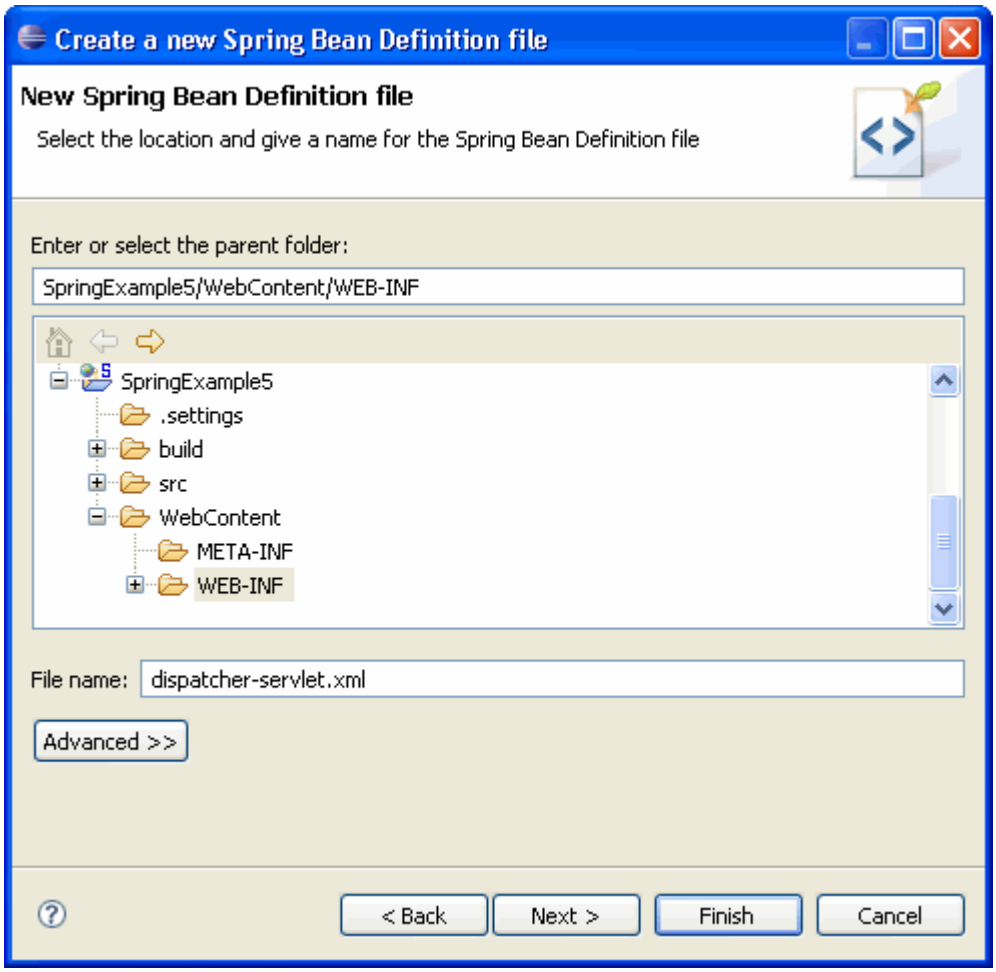
Here the servlet name is *dispatcher*. By default the *DispatcherServlet* will look for a file name *dispatcher-servlet.xml* to load the Spring MVC configuration. This file name is formed by concatenating the servlet name ("*dispatcher*") with "-*servlet.xml*". Here we user the the *url-pattern* as "*.htm*" inorder to hide the implementations technology to the users.

The *redirect.jsp* will be invoked first when we execute the Spring web application. This is the only *jsp*file outside the *WEB-INF* directory and it is here to provide a redirect to the *DispatcherServlet*. All the other views should be stored under the *WEB-INF* directory so that they can be invoked only through the controller process.

To create a bean configuration file right click the *WebContent* folder and select *New -> Other*. The following dialog box appears.



Select the Spring Bean Configuration file and click Next.




Enter the file name as "*dispatcher-servlet.xml*" and click the *Finish* button. Now the Spring bean configuration file is created, we need to configure the *Controller* and the*ViewResolver* classes. The following code shows how to do this.

```
view source

print?

01.<bean id="viewResolver"

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

03.<property name="prefix">
```

```
<value>/WEB-INF/jsp/</value>
</property>
<property name="suffix">
<value>.jsp</value>
</property>
</bean>

<bean name="/welcome.htm" class="com.vaannila.HelloWorldController" >
<property name="message" value="Hello World!" />
</bean>

</beans>
```

First let's understand how to configure the controller.

```
<bean name="/welcome.htm" class="com.vaannila.HelloWorldController" >
<property name="message" value="Hello World!" />
</bean>
```

Here the ***name*** attribute of the *bean* element indicates the **URL pattern to map** the request. Since the *id* attribute can't contain special characters like "/" , we specify the URL pattern using the *name* attribute of the *bean* element. By default the *DispatcherServlet* uses the *BeanNameUrlHandlerMapping* to map the incoming request. The *BeanNameUrlHandlerMapping* uses the bean name as the URL pattern. Since *BeanNameUrlHandlerMapping* is used by default, you need not do any seperate configuration for this.
We set the message attribute of the *HelloWorldController* class thru setter injection. The*HelloWorldController* class is configured just like an another

JavaBean class in the Spring application context, so like any other JavaBean we can set values to it through Dependency Injection(DI).
The *redirect.jsp* will redirect the request to the *DispatcherServlet*, which inturn consults with the*BeanNameUrlHandlerMapping* and invokes the *HelloWorldController*. The *handleRequestInternal()*method in the HelloWorldController class will be invoked. Here we return the *message* property under the name *welcomeMessage* and the view name *welcomePage* to the *DispatcherServlet*. As of now we only know the view name, and to find the actual view to invoke we need a *ViewResolver*.
The *ViewResolver* is configured using the following code.

view source

print?

```
01.<bean id="viewResolver"
02.class="
org.springframework.web.servlet.view.InternalResourceViewResolver">
03.<property name="prefix">
04.<value>/WEB-INF/jsp/</value>
05.</property>
06.<property name="suffix">
07.<value>.jsp</value>
08.</property>
09.</bean>
```

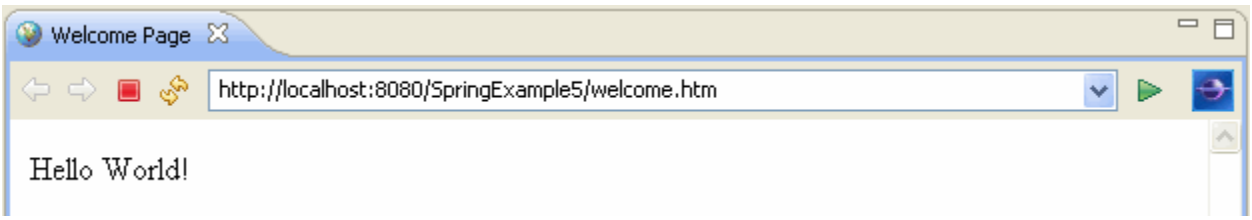
Here the *InternalResourceViewResolver* is used to resolve the view name to the actual view. The *prefix value + view name + suffix value* will give the actual view location. Here the actual view location is */WEB-INF/jsp/welcomePage.jsp*

The following library files are needed to run the example.

view source

```
print?

01.antlr-runtime-3.0

02.commons-logging-1.0.4

03.org.springframework.asm-3.0.0.M3

04.org.springframework.beans-3.0.0.M3

05.org.springframework.context-3.0.0.M3

06.org.springframework.context.support-3.0.0.M3

07.org.springframework.core-3.0.0.M3

08.org.springframework.expression-3.0.0.M3

09.org.springframework.web-3.0.0.M3

10.org.springframework.web.servlet-3.0.0.M3
```

To execute the example run the *redirect.jsp* file. The following page will be displayed.



# 1) Introduction

The ***Spring*** ***MVC*** provides rich functionality for building robust ***Web Applications*** and it is available as a separate module in the Distribution. As a pre-requisite, readers are advised to go through the introductory article on ***Spring Framework*** Introduction to Spring Framework. The***Spring MVC Framework*** is architected and designed in such a way that every piece of logic and functionality is highly configurable. Also Spring can integrate effortlessly with other popular Web Frameworks like ***Struts***, ***WebWork***, ***Java Server Faces*** and ***Tapestry***. It means that you can even instruct Spring to use any one of the Web Frameworks. More than that Spring is not tightly coupled with Servlets or Jsp to render the View to the Clients. Integration with other View technologies like ***Velocity***, ***Freemarker***, ***Excel*** or ***Pdf*** is also possible now. This article provides an introduction over the various components that are available in the ***Spring MVC*** for the Web Tier. Specifically the major Core Components like ***Dispatcher Servlet***, ***Handler Mappings***,***Controller***, ***Model***, ***View*** and ***View Resolver*** along with the appropriate Api are discussed briefly. Finally the article will conclude by presenting a Sample Application.

***also read:***

- *Spring Interview Questions*
- *Spring Framework Books (recommended)*
- *JSF Interview Questions*
- *Introduction to Spring MVC*


# 2) The Spring Workflow

Before taking a look over the various Components that are involved in the ***Spring MVC Framework***, let us have a look on the style of Spring Web Flow.

1. The Client requests for a ***Resource*** in the Web Application.

2. The ***Spring Front Controller***, which is implemented as a Servlet, will intercept the Request and then will try to find out the appropriate ***Handler Mappings***.
3. The ***Handle Mappings*** is used to map a request from the Client to its Controller object by browsing over the various Controllers defined in the Configuration file.
4. With the help of ***Handler Adapters***, the Dispatcher Servlet will dispatch the Request to the Controller.
5. The Controller processes the Client Request and returns the ***Model and the View*** in the form of `ModelAndView` object back to the Front Controller.
6. The ***Front Controller*** then tries to resolve the actual ***View*** (which may be Jsp, Velocity or Free marker) by consulting the ***View Resolver object***.
7. Then the selected View is rendered back to the Client.
   Let us look into the various Core Components that make up the Spring Web Tier. Following are the components covered in the next subsequent sections.

# 3) Dispatcher Servlet

The ***Dispatcher Servle***t as represented by org.springframework.web.servlet.DispatcherServlet, follows the ***Front Controller Design Pattern*** for handling Client Requests. It means that whatever Url comes from the Client, this Servlet will intercept the Client Request before passing the Request Object to the Controller. The ***Web Configuration file*** should be given definition in such a way that this Dispatcher Servlet should be invoked for Client Requests.
Following is the definition given in the `web.xml` to invoke ***Spring's Dispatcher Servlet***.

**web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
    <web-app version="2.4">

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>
```

```
10
11        <servlet-mapping>
12            <servlet-name>dispatcher</servlet-name>
13            <url-pattern>*.*</url-pattern>
14        </servlet-mapping>
15
16    </web-app>
17
```

Look into the definition of `servlet-mapping` tag. It tells that whatever be the Client Request (represented by *.* meaning any Url with any extension), invoke the Servlet by name `'dispatcher'`. In our case, the 'dispatcher' servlet is nothing but an instance of type `'org.springframework.web.servlet.DispatcherServlet'`.
Closing associated term with the Dispatcher Servlet is the ***Application Context***. An ***Application Context*** usually represents a set of ***Configuration Files*** that are used to provide Configuration Information to the Application. The Application Context is a Xml file that contain various ***Bean Definitions***. By default the Dispatcher Servlet will try to look for a file by name `<servlet-name>-servlet.xml` in the `WEB-INF` directory. So, in our case the Servlet will look for a file name called `dispatcher-servlet.xml` file in the WEB-INF directory.
It is wise sometimes to split all the Configuration information across multiple Configuration Files. In such a case we have to depend on a ***Listener Servlet*** called ***Context Loader*** represented by `org.springframework.web.context.ContextLoaderListener`.

```
1   <web-app>
2
3       <listener>
4           <listener-class>
5               org.springframework.web.context.ContextLoaderListener
6           </listener-class>
7       </listener>
8
```

```
</web-app>
```

By default, this Context Listener will try to look for the Configuration File by name `'applicationContext.xml'` in the `'/WEB-INF'` directory. But with the help of the parameter`'contextConfigLocation'` the default location can be overridden. Even multiple Configuration Files each containing separate piece of Information is also possible.

**web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4">

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

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/contacts.xml, /WEB-INF/resources.xml</param-value>
    </context-param>

</web-app>
```

The above definition instructs the Framework to look and load for the Configuration Files by name `'contacts.xml'` and `'resources.xml'` in the `WEB-INF` directory.

# 4) Handler Mappings

When the Client Request reaches the ***Dispatcher Servlet***, the ***Dispatcher Servlet*** tries to find the appropriate ***Handler Mapping*** Object to map between the Request and the Handling Object.
A ***Handler Mapping*** provides an abstract way that tell how the Client's Url has to be mapped to the Handlers. Four concrete variation of Handler Mapping are available. They are defined as follows

- BeanNameUrl HandlerMapping
- CommonsPathMap HandlerMapping
- ControllerClassName HandlerMapping
- SimpleUrl HandlerMapping

  All the above Handler Mapping objects are represented as `BeanNameUrlHandlerMapping`,`CommonsPathMapHandlerMapping`, `ControllerClassNameHandlerMapping` and`SimpleUrlHandlerMapping` in the `org.springframework.web.servlet` package respectively. Let us see the functionalities and the differences in usage one by one.

### 4.1) BeanNameUrl HandlerMapping

This is the simplest of the ***Handler Mapping*** and it is used to map the Url that comes from the Clients directly to the Bean Object. In the later section, we will see that the Bean is nothing but a Controller object. For example, consider that the following are the valid Url in a Web Application that a Client Application can request for.

```
1  http://myserver.com/eMail/showAllMails
2
3  http://myserver.com/eMail/composeMail
4
5
6  http://myserver.com/eMail/deleteMail
```

Note that the Url (excluding the Application Context) in the above cases are `'showAllMails','composeMail'` and `'deleteMail'`. This means that the Framework will look for Bean Definitions with Identifiers ‘showAllMails’, ‘composeMail’ and ‘deleteMail’. Consider the following Xml code snippet in the Configuration file,

```
    <beans>
1
2
        <bean name="/showAllMails.jsp"
3
        class="com.javabeat.net.ShowAllMailsController">
4
        </bean>
5
```

```
        <bean name="/composeMail.jsp"
        class="com.javabeat.net.ComposeMailController">
        </bean>

        <bean name="/ deleteMail.jsp"
        class="com.javabeat.net.DeleteMailController">
        </bean>

    </beans>
```

So, in ***BeanNameUrl Handler Mapping***, the Url of the Client is directly mapped to the Controller. To enable this kind of Handler Mapping in the Application, the Configuration file should have a similar kind of definition like the following,

```
<beans>

    …
   <bean id="beanNameUrl"
   class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
    …

</beans>
```

### 4.2) CommonsPathMap HandlerMapping

This is a rarely used Handler Mapping in which case, the name of the Url to which the Controller has to be mapped is specified directly in the Source file of the Controller. Considering the previous example, if we want to map ‘showAllMails’, ‘composeMail’ and ‘deleteMail’ to Controllers namely `ShowAllMailsController`, `ComposeMailController` and `DeleteMailController`,

then the mapping information must be specified in the form of ***meta-data in the source files***inside the Javadoc comments. Consider the following Controller Definitions,

```
1
2   /**
3   *@@ org.springframework.web.servlet.handler.commonsattributes.
4   *PathMap("/showAllMails.jsp")
5   */
6   public class ShowAllMailsController{
7   }
8
9   /**
10  *@@ org.springframework.web.servlet.handler.commonsattributes.
11  *PathMap("/composeMail.jsp")
12  */
13  public class ComposeMailController{
14  }
15
16  /**
17  *@@ org.springframework.web.servlet.handler.commonsattributes.
18  *PathMap("/deleteMail.jsp")
19  */
20  public class DeleteMailController {
    }
```

The attribute must point to`org.springframework.web.servlet.handler.commonsattributes.PathMap`. By defining Controllers in this way, one more additional compilation step is needed. That is to make the availability of this attribute in the Class files, this Java Source has to be compiled with the***Commons***

***Attribute Compiler*** which comes along with the Spring Distribution. As before, to enable this kind of mapping , the Configuration File should have an entry similar to this,

```
<beans>

    <bean id="metaHandlerMapping" class="org.springframework.web.servlet.handler.
    metadata.CommonsPathMapHandlerMapping"/>

</beans>

```

### 4.3) ControllerClassName HandlerMapping

In this kind of Handler Mapping, the name of the Controller is taking directly from the Url itself with slight modifications. For example, let us assume that the Client request ends with Url as shown below,

```
http://myserver.com/emailApp/showInbox.jsp

http://myserver.com/emailApp/showDeletedItems.jsp
```

And as such, we have a Controller definition by name ShowController as follows,

**ShowController.java**

```
public class ShowController{
}
```

Also the Configuration file is made to activate this kind of Handler Mapping by making the following definition,

```
<beans>

    <bean id="controllerClassName" class="org.springframework.web.servlet.handler.
```

```
4       metadata.ControllerClassNameHandlerMapping"/>
5
6   </beans>
```

The first thing the Framework does it, it will traverse through the List of Controllers defined in the Configuration File and perform these actions. For the Controller ShowController, then Framework will remove the Controller String and then lowercase the first letter. In our case the string now becomes show. Now whatever Client Request matches the pattern `/show*`, then the`ShowController` will be invoked.

### 4.4) SimpleUrl HandlerMapping

This is the Simplest of all the Handler Mappings as it directly maps the Client Request to some Controller object. Consider the following Configuration File,

```
1
    <bean id="simpleUrlMapping"
2
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
3
4
        <property name="mappings">
5           <props>
6               <prop key="/showAllMails.jsp">showController</prop>
7               <prop key="/composeMail.jsp">composeController</prop>
8               <prop key="/deleteMail.jsp">deleteController</prop>
9           </props>
        </property>
10
11
    </bean>
12
```

The set of mappings is encapsulated in the `'property'` tag with each defined in a `'prop'`element with the `'key'` attribute being the Url, the value being the Identifier of the Controller Objects. Note that the Beans for the above Identifiers should be defined somewhere in the Configuration File.

# 5) Handler Adapters

It is important to understand that the Spring Framework is so flexible enough to define what Components should be delegated the Request once the ***Dispatcher Servlet*** finds the appropriate ***Handler Mapping***. This is achieved in the form of ***Handler Adapters***. If you remember in the Spring Work flow section, that it is mentioned once the Dispatcher Servlet chooses the appropriate ***Handler Mapping***, the Request is then forwarded to the Controller object that is defined in the Configuration File. This is the default case. And this so happens because the ***Default Handler Adapter*** is ***Simple Controller Handler Adapter*** (represented by`org.springframework.web.servlet.SimpleControllerHandlerAdapter`), which will do the job of the Forwarding the Request from the Dispatcher to the Controller object.
Other types of ***Handler Adapters*** are ***Throwaway Controller HandlerAdapter***(`org.springframework.web.servlet.ThrowawayControllerHandlerAdapter`) and***SimpleServlet HandlerAdapter***(`org.springframework.web.servlet.SimpleServletHandlerAdapter`).
The ***Throwaway Controller HandlerAdapter***, for example, carries the Request from the Dispatcher Servlet to the ***Throwaway Controller*** (discussed later in the section on Controllers) and Simple Servlet Handler Adapter will carry forward the Request from the Dispatcher Servlet to a Servlet thereby making the `Servlet.service()` method to be invoked.
If, for example, you don't want the default ***Simple Controller Handler Adapter***, then you have to redefine the Configuration file with the similar kind of information as shown below,

```
<bean id="throwawayHandler" class = "org.springframework.web.servlet.mvc.throwaway.
    ThrowawayControllerHandlerAdapter"/>

or

<bean id="throwawayHandler" class="org.springframework.web.servlet.mvc.throwaway.
    SimpleServletHandlerAdapter"/>

```

Even, it is possible to write a ***Custom Handler Adapter*** by implementing the `HandlerAdapter`interface available in the `org.springframework.web.servlet` package.

# 6) Controller

Controllers are components that are being called by the Dispatcher Servlet for doing any kind of Business Logic. Spring Distribution already comes with a variety of ***Controller Components***each doing a specific purpose. All Controller Components in Spring implement the`org.springframework.web.servlet.mvc.Controller` interface. This section aimed to provide the commonly used Controllers in the Spring Framework. The following are the Controller Components available in the Spring Distribution.

- SimpleFormController
- AbstractController
- AbstractCommandController
- CancellableFormController
- AbstractCommandController
- MultiActionController
- ParameterizableViewController
- ServletForwardingController
- ServletWrappingController
- UrlFilenameViewController

The following section covers only on `AbstractController`, `AbstractCommandController`,`SimpleFormController` and `CancellableFormController` in detail.

## 6.1) Abstract Controller

If one wants to implement ***Custom Controller Component*** right from the scratch, then instead of implementing the Controller interface, extending `AbstractController` can be preferred as it provides the basic support for the ***GET and the POST*** methods. It is advised that only for simple purpose, this type of extensions should be used. The purpose may be as simple as returning a resource to the Client upon request without having the need to examine the Request Parameters or other Stuffs. For example, consider the following piece of code,

**MySimpleController.java**

```
1  public class MySimpleController extends AbstractController{
2
3      public void handleRequestInternal(HttpServletRequest request,
4      HttpServletResponse response){
```

```
5
6           return new ModelAndView("myView");
7
8       }
9   }
```

Note that the Dispatcher Servlet will call the `handleRequest()` method by passing the Request and the Response parameters. The implementation just returns a `ModelAndView` (discussed later) object with myView being the logical view name. There are Components called ***View Resolvers*** whose job is to provide a mapping between the ***Logical View Name*** and the actual***Physical Location of the View Resource***. For the time being, assume that somehow, myView is mapped to `myView.jsp`. So, whenever the Dispatcher Servlet invokes this MySimpleController object, finally myView.jsp will be rendered back to the Client.

## 6.2) Abstract Command Controller

The concept of ***Command Controller*** comes into picture when the Business Logic depends upon the values that are submitted by the User. Instead of depending on the Servlet Api to get the Request Parameter Values and other session Objects, we can depend on this ***Abstract Command Controller*** to take those pieces of Information. For example consider the following code snippet which has a simple business logic telling that, depending on the existence of username, display the form `success.jsp` or `failure.jsp`

**MySimpleController.java**

```
1   public class MySimpleController extends AbstractCommandController{
2
3       public MySimpleController(){
4           setCommandClass(UserInfo.class);
5       }
6
7       public void handle(HttpServletRequest request, HttpServletResponse response,
8       Object command){
9
```

```
10            UserInfo userInfo = (UserInfo)command;
11            if ( exists(userInfo.getUserName){
12                return new ModelAndView("success");
13            }else{
14                return new ModelAndView("failure");
15            }
16        }
17
18        private boolean exits(String username){
19            // Some logic here.
20        }
21    }
```

Note that the ***Client Parameters*** (username , in this case) is encapsulated in a simple Class called `UserInfo` which is given below. The value given by the Client for the username field will be directly mapped to the property called username in the UserInfo. In the Constructor of the`MySimpleController` class, we have mentioned the name of the Command Class which is going to hold the Client Request Parameters by calling the `setCommandClass()` method. Also note that in the case of Command Controller, the method that will be called by the Dispatcher Servlet will be `handle()` which is passed with the Command object apart from the Request and the Response objects.

**UserInfo.java**

```
1   public class UserInfo{
2
3       private String username;
4       // Getters and Setters here.
5
6   }
```

## 6.3) Simple Form Controller

Asking the User to fill in a Form containing various information and submitting the form normally happens in almost every Web Application. The ***Simple Form Controller*** is exactly used for that purpose. Let us give a simple example to illustrate this. Assume that upon Client Request a page called empInfo.jsp is rendered to the client containing `empName`, `empAge` and `empSalary`fields. Upon successful completion a Jsp Page called `empSuccess.jsp` is displayed back to the Client. Now let us see how we can make use of the Simple Form Controller to achieve this kind functionality.

The very first thing is that, to collect the Client Input Values, a Command object which contains getter and setters must be defined. Following the skeleton of the class called `EmpInfo`.

**EmpInfo.java**

```
1  public class EmpInfo{
2
3      private String empName;
4      private int empAge;
5      private double empSalary;
6
7      // Getters and setters for the above properties.
8
9  }
```

The next thing is to write a class that extends SimpleFormController. But this time, the`doSubmitAction()` method should be overridden. This is the method that will be called when the Client submits the form. Following is the definition of the Controller class.

**EmpFormController.java**

```
1   public class EmpFormController extends SimpleFormController{
2
3       public EmpFormController(){
4           setCommandClass(EmpInfo.class);
5       }
```

```
6
7       public void doSubmitAction(Object command){
8           EmpInfo info = (EmpInfo)command;
9           process(info);
10      }
11
12      private void process(EmpInfo info){
13          //Do some processing with this object.
14      }
15  }
16
```

As we mentioned previously, the form that collects information from the Client is `empInfo.jsp`and upon successful submission the view `empSuccess.jsp` should be displayed. This information is externalized from the Controller class and it is maintained in the Configuration File like the following,

```
1   <bean id = "empForm" class="EmpFormController">
2
3       <property name="formView">
4           <value>empInfo</value>
5       </property>
6
7       <property name="successView">
8           <value>empSuccess</value>
9       </property>
10
11  </bean>
```

Note the two property names `'formView'` and `'successView'` along with the values`'empInfo'` and `'empSuccess'`. These properties represent the initial View to be displayed and the final view (after successful Form submission) to be rendered to the Client.

### 6.4) Cancellable FormController

If you carefully notice with the implementation of Simple Form Controller, there are ways to provide the Initial and the Successful View to the Clients. But what happens when the Form is cancelled by the User? Who will process the Cancel operation of the Form?

The above issues can be given immediate solution with the usage of ***Cancellable FormController***. The good thing is that Cancellable FormController extends ***SimpleForm Controller*** so that all the functionalities are visible to this Controller also. Suppose say that the User clicks the cancel button, the Framework will check in the Request parameter for a key with name `'cancelParamKey'`. If it is so, then it will call the `onCancel()` method. Consider the following definition,

**MyCompleteFormController.java**

```
1  public class MyCompleteFormController extends CancellableFormController{
2
3      public ModelAndView onCancel(){
4          return new ModelAndView("cancelView");
5      }
6  }
```

# 7) Model And View

***Model and View*** (represented by the class`org.springframework.web.servlet.ModelAndView`) is returned by the Controller object back to the Dispatcher Servlet. This class is just a Container class for holding the Model and the View information. The Mode object represents some piece of information that can be used by the View to display the information. Both these Objects are given high degree of abstraction in the Spring Framework.

Any kind of ***View Technology*** (`org.springframework.web.servlet.View`) can be plugged into the Framework with ease. For example, Excel, Jasper Reports, Pdf, Xslt, Free Marker, Html, Tiles, Velocity etc. are the supported Frameworks as of now. The Model object (represented

by`org.springframework.ui.ModelMap`) is internally maintained as a Map for storing the Information.

Following are the ways to Construct the Model and the View object.

```
View pdfView = …;

Map modelData = new HashMap();

ModelAndView mv1 = new ModelAndView(pdfView, modelData);
```

The above constructs a `ModelAndView` object by passing the actual View object along with the Model object. Now consider the following code,

```
ModelAndView mv1 = new ModelAndView("myView", someData);
```

Note, in the above example, a string with “myView” is passed for the View. This way of specifying a ***View*** is called a ***Logical View***. It means that myView either can point to something called myView.jsp or myView.pdf or myView.xml. The ***Physical View Location*** corresponding to the***Logical View*** can be made configurable in the Configuration File.

# 8) View Resolver

In the previous section, we talked about ***Logical View*** and the ***Physical View Location*** for the Logical View. The mapping between the Logical name and the Physical View Location is taken care by the ***View Resolver*** object. Without any surprise, Spring comes with a set of ***Built-In Spring Resolvers***. It is even possible to write ***Custom View Resolvers*** by implementing the`org.springframework.web.servlet.ViewResolver` interface. Following are the available View Resolvers in the Spring Distribution.

- BeanNameViewResolver
- FreeMarkerViewResolver
- InternalResourceViewResolver
- JasperReportsViewResolver
- ResourceBundleViewResolver
- UrlBasedViewResolver
- VelocityLayoutViewResolver
- VelocityViewResolver
- XmlViewResolver
- XsltViewResolver

The following section concentrates only on ***Internal Resource View Resolver*** and ***Bean Name View Resolver*** in detail.

## 8.1) Internal Resource View Resolver

The ***Internal Resource View Resolver*** will try to map the Logical name of the Resource as returned by the Controller object in the form of ModelAndView object to the ***Physical View location***. For example, consider the following class definition which returns different ModelAndView objects.

**MyController.java**

```
public class MyController {


    public void handle(){
        if(condition1()){
            return new ModelAndView("myView1");
        }else if (condition2()){
            return new ModelAndView("myView2");
        }
        return new ModelAndView("myView3");
    }
}
```

Assume that if the Client Request satisfies `condition1()`, then the view myView.jsp which is present in the `/WEB-INF` folder should be displayed and for the client Requests satisfying`condition2()` and the other one, `myView2.jsp` and `myView3.jsp` should be displayed.

For this to happen, the following entry must be made in the Configuration File,

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

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

```
5       <property name="suffix"><value>.jsp</value></property>
6
7   </bean>
```

This is how the ***Internal Resource View Resolver*** will map the Logical View Name to the physical Location. When the logical View name is myView1, then it will construct a view name which is the summation of ***the prefix + the logical View Name + the suffix***, which is going to be /WEB-INF/myView.jsp. The same is the case for `myView2.jsp` and `myView3.jsp`.

### 8.2) Bean Name View Resolver

One of the dis-advantage of using ***Internal Resource View Resolver*** is that the name of the View file (whether it is a Jsp File or the Pdf File) must be present in the Web Application Context. Dynamically generated View files may not be possible. In such a case, we may use the ***Bean Name View Resolver*** which will dynamically generate View in Pdf or Excel Formats.
For the example, if the `ModelAndView` object represents a View by name “pdf” as shown in the following snippet,

```
1   return ModelAndView("pdf")
```

And, if we want to generate the Pdf file, then we should have defined the Configuration information in the file as follows,

```
1   <bean id="beanNameResolver"
2   class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
```

The above code configures the Framework to use BeanNameViewResolver. Since the logical name ‘pdf’ must resolve to a Bean Name, we should define a similar entry like the following in the Configuration File. Note that, in the following MyPdfGenerator may be the sub-class of`org.springframework.web.servlet.view.document.AbstractPdfView` for generating the Pdf File.

```
1   <bean id = " pdf " class = "MyPdfGenerator"/>
```

# 9) Sample Application

### 9.1) Introduction

The final Section of this article details a Simple Contact Application that has provisions for Creating, Deleting and Listing Contact Objects. The aim of this Application is to show the various use of

Controller Components like Abstract Controller, Abstract Command Controller and Form Controller along with Configuration Information.

## 9.2) The Web Descriptor File

As mentioned previously, since the ***Dispatcher Servlet*** acts as an ***Interceptor*** for the Client Request, an entry for the same has to be mentioned in the `web.xml` file. Follow is the code snippet for the same,

**web.xml**

```
1   web.xml
2
3   <?xml version="1.0" encoding="UTF-8"?>
4   <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
7
8   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
9
10      <servlet>
11          <servlet-name>dispatcher</servlet-name>
12          <servlet-class>
13              org.springframework.web.servlet.DispatcherServlet
14          </servlet-class>
15          <load-on-startup>2</load-on-startup>
16      </servlet>
17
18      <servlet-mapping>
19          <servlet-name>dispatcher</servlet-name>
            <url-pattern>*.htm</url-pattern>
```

```
20        </servlet-mapping>
21
22    </web-app>
23
24
```

### 9.3) Configuration File

The following represents the Configuration File for holding various piece of Configuration Information. The first thing to note is the type of Handler Mapping configured. In our case, it is the Bean Name Url Handler Mapping which means that the Url of the Client is tightly coupled with the class name of the Bean (Controller). Since all the Jsp files are maintained in the`'/WEB/contacts'` directory the `'prefix'` property is pointing to `'/WEB/contacts'`.
For the Create, Delete and List operation on Contacts, three different Controller Components have been defined. They
are `CreateContactController`, `DeleteContactController` and`ListContactsControlle r` respectively.

**dispatcher-servlet.xml**

```
1         <?xml version="1.0" encoding="UTF-8" ?>
2     <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3     "http://www.springframework.org/dtd/spring-beans.dtd">
      <beans>
4
5
          <bean id="beanNameUrlMapping" class="org.springframework.web.servlet.handler.
6
          BeanNameUrlHandlerMapping"/>
7
8
          <bean name = "/CreateContact.htm" class="net.javabeat.articles.spring.mvc.
9
          contacts.CreateContactController">
10
11            <property name="formView">
```

```
12              <value>CreateContact</value>
13          </property>
14          <property name="successView">
15              <value>ContactCreated</value>
16          </property>
17
18      </bean>
19
20      <bean name = "/DeleteContact.htm" class= "net.javabeat.articles.spring.mvc.
        contacts.DeleteContactController">
21      </bean>
22
23      <bean name = "/ListContacts.htm" class= "net.javabeat.articles.spring.mvc.
24      contacts.ListContactsController">
25      </bean>
26
27      <bean id="viewResolver" class="org.springframework.web.servlet.view.
        InternalResourceViewResolver">
28          <property name="prefix" value="/WEB-INF/contacts/"/>
29          <property name="suffix" value=".jsp" />
30      </bean>
31
32  </beans>
33
34
35
```

## 9.4) CreateContact and ContactCreated Jsp Files

The following is the code for CreateContact.jsp file.

**CreateContact.jsp**

```
1
    <html>
2
    <head>
3
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
4

5
        <title>Create a Contact</title>
6
        </head>
7
        <body>
8

9
        <h1>Create a Contact</h1>
10

11
        <form name = "CreateContact" method = "get"">
12
            <input type = "text" name = "firstname" />
13
            <input type = "text" name = "lastname" />
14
            <br>
15
            <input type="submit" name = "Create Contact" value = "Create Contact"/>
       </form>
16

17
    </body>
18
    </html>
19
```

Note that since this is the page that will be shown to the user initially, in the Configuration file, the property `'formView'` is pointed to `'CreateContact'`. Following is the code for ContactCreated.jsp. Since this is the View that will be shown after the Form Submission the property `'successView'` is made to point to `'ContactCreated'`.

**ContactCreated.jsp**

```
<html>
    <head>
    <meta http-equiv = "Content-Type" content = "text/html; charset = UTF-8">
    <title>Contact is Created</title>
    </head>

    <body>

    <h1>Contact is successfully Created</h1>

   </body>
</html>
```

9.5) DeleteContact.jsp

Following is the complete listing for DeleteContact.jsp file. Note that this Jsp File is mapped to`DeleteContactController` in the Configuration File.

**DeleteContact.jsp**

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

    <title>Delete Contact</title>
    </head>
    <body>

    <h1>Delete Contact</h1>
```

```
10
11        <form name = "DeleteContact" method = "get">
12            <input type = "text" name = "firstname" />
13            <br>
14            <input type="submit" name = "DeleteContact" value = "Delete Contact"/>
15        </form>
16
17        </body>
18    </html>
```

9.6) ListContacts.jsp

This page is to list all the existing Contacts that were created before. It should be noted that the***Model Object*** that holds all the Contact Information in the form of List is available in the ListContactsController. The ***Model Information*** from the Controller after getting bound to the***Request Scope*** is being taken off from the View in the form of ***Expression Language***. Following is the listing for ListContacts.jsp

**ListContacts.jsp**

```
1     <html>
2     <head>
3     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
4
5         <title>Showing All Contacts</title>
6         </head>
7         <body>
8
9         <h1>Showing All Contacts</h1>
10
```

```
    <p> The following are the created Contacts </p>

        <c:forEach items = "${allContacts}" var="contact">
            <c:out value="${contact.firstname}"/><br>
            <c:out value="${contact.lastname}"/><br>
        </c:forEach>

    </body>
</html>
```

9.7) Contacts.java

The following is the Class structure for Contacts.java for encapsulating the properties firstname and lastname.

**Contact.java**

```
package net.javabeat.articles.spring.mvc.contacts;

public class Contact {

    private String firstName;
    private String lastName;

    public Contact() {
    }

    public Contact(String firstName, String lastName){
```

```
12          this.firstName = firstName;
13          this.lastName = lastName;
14      }
15
16      public String getFirstName() {
17          return firstName;
18      }
19
20      public void setFirstName(String firstName) {
21          this.firstName = firstName;
22      }
23
24      public String getLastName() {
25          return lastName;
26      }
27
28      public void setLastName(String lastName) {
29          this.lastName = lastName;
30      }
31
32      public int hashCode(){
33          return firstName.hashCode() + lastName.hashCode();
34      }
35
36      public boolean equals(Object object){
            if (object instanceof Contact){
                Contact second = (Contact)object;
```

```
            return (firstName.equals(second.getFirstName()) &&
                lastName.equals(second.getLastName()));
        }
        return false;
    }

    public String toString(){
        return "[First Name = " + firstName + ", Last Name = " + lastName + "]";
    }
}
```

9.8) ContactService.java

This simple service class provides functionalities for creating, deleting and listing the Contact information. All the Controller Components makes use of this class to achieve their respective functionalities.

**ContactService.java**

```
package net.javabeat.articles.spring.mvc.contacts;

import java.util.*;

public class ContactService {

    private static Map contacts = new HashMap();

    public ContactService() {
```

```
9       }
10
11      public static Contact createContact(Contact contact){
12          contacts.put(new Integer(contact.hashCode()), contact);
13          return contact;
14      }
15
16      public static Contact createContact(String firstName, String lastName){
17          return createContact(new Contact(firstName, lastName));
18      }
19
20      public static boolean deleteContact(String firstName){
21          Iterator iterator = contacts.entrySet().iterator();
22          while (iterator.hasNext()){
23              Map.Entry entry = (Map.Entry)iterator.next();
24              Contact contact = (Contact)entry.getValue();
25              if (contact.getFirstName().equals(firstName)){
26                  contacts.remove(new Integer(contact.hashCode()));
27                  return true;
28              }
29          }
30          return false;
31      }
32
33      public static List listContacts(){
34          return toList(contacts);
35      }
```

```
34
35      private static List toList(Map contacts){
36          List contactList = new ArrayList();
37          Iterator iterator = contacts.entrySet().iterator();
38          while (iterator.hasNext()){
39              Map.Entry entry = (Map.Entry)iterator.next();
40              Contact contact = (Contact)entry.getValue();
41              contactList.add(contact);
42          }
43          return contactList;
44      }
45  }
46
47
48
```

## 9.9) Controller Classes

Following is the listing for `CreateContact Controller`. Note that since the Model Information for creating a contact (for which the Client supplies the firstname and the lastname parameters) is the Contact class, call has been made in the Constructor to `setCommandClass()` by passing the class name of the `Contact` class.

**CreateContactController.java**

```
1   package net.javabeat.articles.spring.mvc.contacts;
2
3   import org.springframework.web.servlet.mvc.SimpleFormController;
4
5   public class CreateContactController extends SimpleFormController{
```

```
6
7         public CreateContactController() {
8             setCommandClass(Contact.class);
9         }
10
11        public void doSubmitAction(Object command){
12            Contact contact = (Contact)command;
13            ContactService.createContact(contact);
14        }
15    }
```

Note that the method `doSubmitAction()` doesn't return anything because the next Logical View to be displayed will be taken from the Configuration file which is represented by the property called `'successView'`.

Following two classes are the Controller Components for Deleting and Listing Contacts. Note that in the case of Delete Operation, a Jsp Page (`DeletedContact.jsp`) containing information telling that the Contact has been Deleted will displayed. But since for the Contact Listing operation, the model information containing a Collection of Contact Objects has to be passed from the Controller to the View and the same is achieved in the 3 argument constructor to ModelAndView.

**DeleteContactController.java**

```
1     package net.javabeat.articles.spring.mvc.contacts;
2
3     import javax.servlet.http.*;
4     import org.springframework.validation.BindException;
5     import org.springframework.web.servlet.ModelAndView;
6     import org.springframework.web.servlet.mvc.AbstractCommandController;
7
8     public class DeleteContactController extends AbstractCommandController{
9
```

```
10        public DeleteContactController(){
11            setCommandClass(Contact.class);
12        }
13
14        public ModelAndView handle(HttpServletRequest request,
15            HttpServletResponse response, Object command, BindException errors)
16                throws Exception {
17
18            Contact contact = (Contact)command;
19            ContactService.deleteContact(contact.getFirstName());
20            return new ModelAndView("DeletedContact");
21
22        }
23    }
```

Here is the listing for ListContactsController.java.

**ListContactsController.java**

```
1     package net.javabeat.articles.spring.mvc.contacts;
2
3     import java.util.List;
      import javax.servlet.http.*;
```

```
4   import org.springframework.web.servlet.ModelAndView;
5   import org.springframework.web.servlet.mvc.AbstractController;
6
7   public class ListContactsController extends AbstractController{
8
9       public ListContactsController() {
        }
10
11      public ModelAndView handleRequestInternal(HttpServletRequest request,
12          HttpServletResponse response) throws Exception{
13
14          List allContacts = ContactService.listContacts();
15          return new ModelAndView("ListContacts", "allContacts", allContacts);
16
17      }
18  }
19
20
21
```