**Spring MVC**

**Overview**

Spring framework makes the development of web applications very easy by providing the*Spring MVC* module. Spring MVC module is based on two most popular design patterns - *Front controller* and *MVC*.

In this article, firstly we learn about the *Front controller* and *MVC* design pattern and then explore the details of Spring MVC module in detail, its architecture, and various components and finally we build a simple web application using Eclipse IDE.

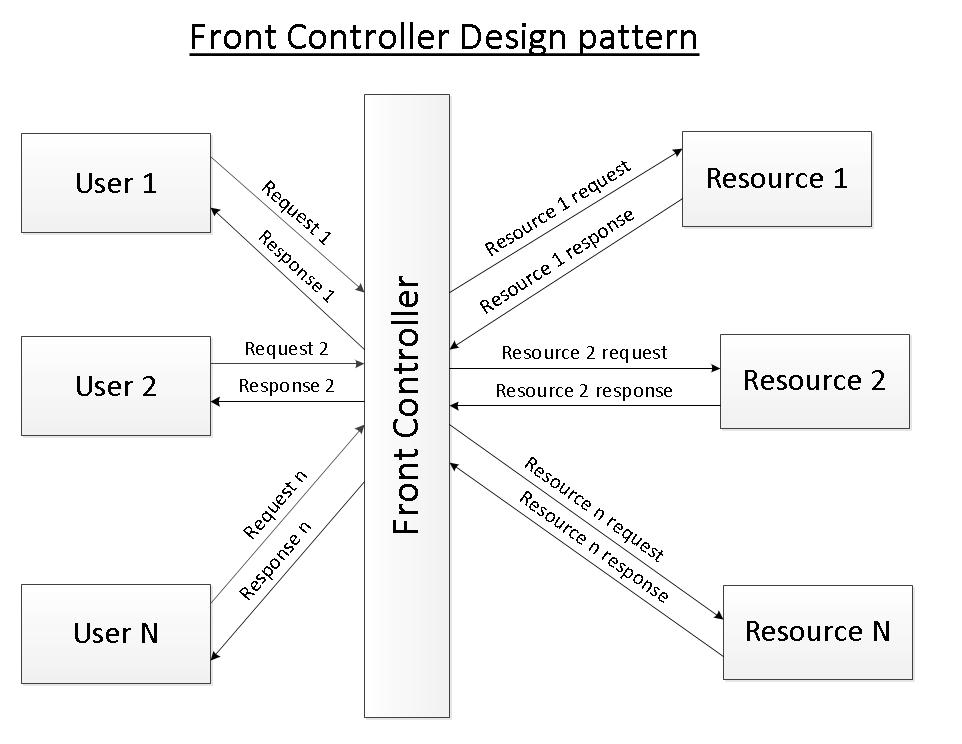
## 1. Architecture

* [Front Controller design pattern](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=1#frontcontroller)
* [MVC design pattern](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=1#mvc)
* [Spring's MVC architecture](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=1#springmvc)

Before going into details of Spring MVC architecture, let us first look at the two popular design patterns used for web development.

### Front Controller design pattern

This design pattern enforces a single point of entry for all the incoming requests. All the requests are handled by a single piece of code which can then further delegate the responsibility of processing the request to further application objects.



Front Controller Design Pattern

### 

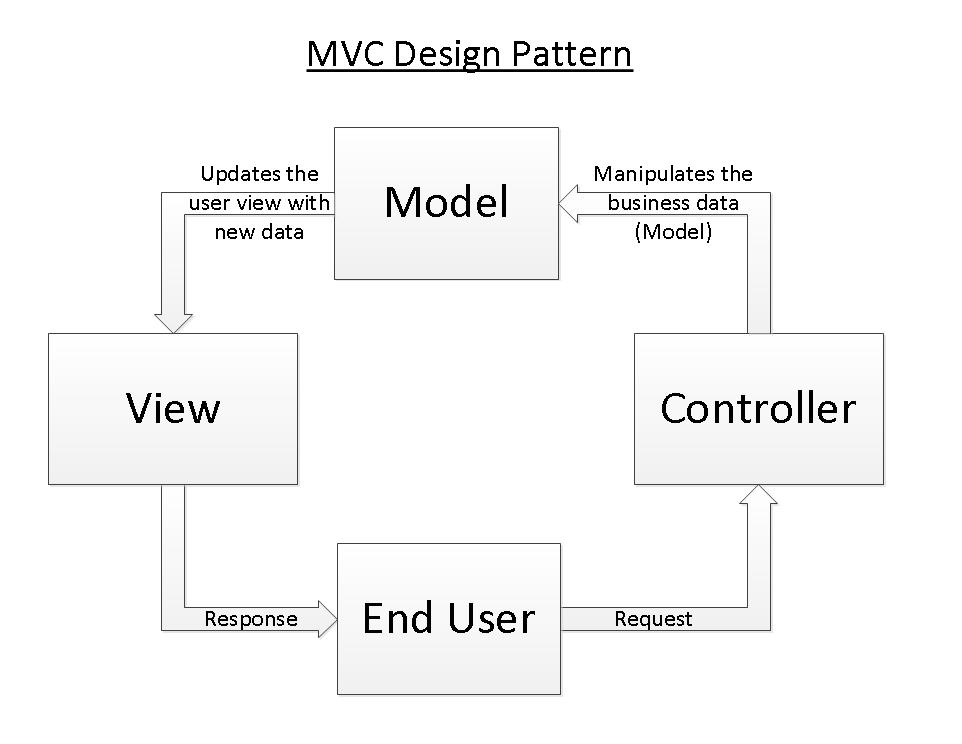
### MVC design pattern

This design pattern helps us develop loosely coupled application by segregating various concerns into different layers. MVC design pattern enforces the application to be divided into three layers, Model, View and Controller.

**Model:** This represents the application data.

**View:** This represents the application’s user interface. View takes model as the input and renders it appropriately to the end user.

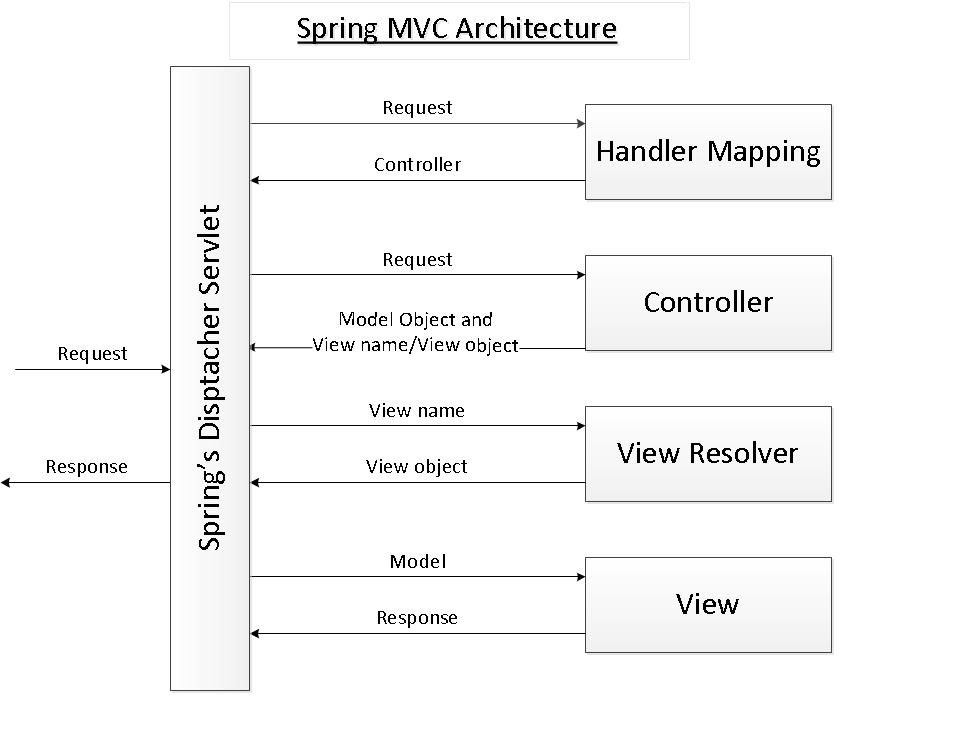
**Controller:** The controller is responsible for handling the request and generating the model and selecting the appropriate view for the request.



MVC Design Pattern

### Spring’s MVC module

Spring’s MVC module is based on front controller design pattern followed by MVC design pattern. All the incoming requests are handled by the single servlet named DispatcherServlet which acts as the front controller in Spring’s MVC module. The DispatcherServlet then refers to the  HandlerMapping to find a controller object which can handle the request.DispatcherServlet then dispatches the request to the controller object so that it can actually perform the business logic to fulfil the user request. (Controller may delegate the responsibility to further application objects known as service objects). The controller returns an encapsulated object containing the model object and the view object (or a logical name of the view). In Spring’s MVC, this encapsulated object is represented by class ModelAndView. In case ModelAndView contains the logical name of the view, the  DispatcherServlet refers the ViewResolver to find the actual View object based on the logical name. DispatcherServlet then passes the model object to the view object which is then rendered to the end user.



**Spring MVC Architecture**

## 2. Dispatcher Servlet

DispatcherServlet acts as the front controller in the Spring’s MVC module. All the user requests are handled by this servlet. Since this is like any other servlet, it must be configured in the application’s web deployment descriptor file i.e. web.xml.

|  |  |
| --- | --- |
| <web-app xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"  id="WebApp\_ID"  version="2.5"  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">     <display-name>Library</display-name>     <servlet>        <servlet-name>myLibraryAppFrontController</servlet-name>        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>        <load-on-startup>1</load-on-startup>     </servlet>     <servlet-mapping>        <servlet-name>myLibraryAppFrontController</servlet-name>        <url-pattern>\*.htm</url-pattern>     </servlet-mapping>     <welcome-file-list>        <welcome-file>welcome.htm</welcome-file>     </welcome-file-list>  </web-app> |  |

We have named the servlet as “myLibraryAppFrontController”. The URI pattern in the servlet mapping section is “\*.htm”. Thus all the requests matching the URI pattern will be handled by myLibraryAppFrontController.

## 3. Spring Application Context

* [Default Application context file](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=3#defaultappcontext)
* [User defined application context file](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=3#userdefinedappcontext)
* [Multiple application context files](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=3#multipleappcontext)

### Default Application context file

By default the dispatcher servlet loads the Spring application context from XML file with name [servlet name]-servlet.xml. Thus when our servletmyLibraryAppFrontController is loaded by the container, it will load the Spring application context from XML file “/WEB-INF/myLibraryAppFrontController-servlet.xml”.

### User defined application context file

We can override the name and location of the default XML file by providing the initialization parameters to the dispatcher servlet. The name of the initialization parameter is contextConfigLocation. The parameter value specifies the name and location of the application context which needs to be loaded by the container.

|  |  |
| --- | --- |
| <servlet>     <servlet-name>myLibraryAppFrontController</servlet-name>     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>     <init-param>        <param-name>contextConfigLocation</param-name>        <param-value>classpath:libraryAppContext.xml</param-value>     </init-param>     <load-on-startup>1</load-on-startup>  </servlet> |  |

In the above configuration of myLibraryAppFrontController, when the container initializes the dispatcher servlet, it will load the Spring application context from XML file “classpath:libraryAppContext.xml” instead of “/WEB-INF/myLibraryAppFrontController-servlet.xml”.

### Multiple application context files

It is a good practice to split the application into multiple logical units and have multiple application context file. Thus on servlet initialization we need to load all these application context files. It is possible to load the Spring application context from multiple XML file as shown below:

|  |  |
| --- | --- |
| <servlet>     <servlet-name>myLibraryAppFrontController</servlet-name>     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>     <init-param>        <param-name>contextConfigLocation</param-name>        <param-value>classpath:libraryAppContext.xml                     classpath:books.xml                     classpath:chapters.xml                     classpath:titles.xml</param-value>     </init-param>     <load-on-startup>1</load-on-startup>  </servlet> |  |

In the above servlet configuration, we have provided multiple XML files as initialization parameter value. All these XML files will be loaded by the container on initialization of the servlet myLibraryAppFrontController.

## 4. Handler mappings

* [BeanNameUrlHandlerMapping](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=4#beannameurlhm)
* [SimpleUrlHandlerMapping](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=4#simpleurlhm)

As the name specifies, the handler mapping maps the request with the corresponding request handler (in fact handler execution chain).  When a request comes to Spring’s dispatcher servlet, it hands over the request to the handler mapping. Handler mapping then inspects the request and identifies the appropriate handler execution chain and delivers it to dispatcher servlet. The handler execution chain contains handler that matches the incoming request and optionally contains the list of interceptors that are applied for the request. Dispatcher servlet then executes the handlers and any associated handler interceptor.

There are number of implementation of hander mapping provided by Spring’s MVC module. Some of these are described below. All the handler mappings classes implement the interface org.springframework.web.servlet.HandlerMapping.

### BeanNameUrlHandlerMapping

This implementation of handler mapping matches the URL of the incoming request with the name of the controller beans. The matching bean is then used as the controller for the request. This is the default handler mapping used by the Spring’s MVC module i.e. in case the dispatcher servlet does not find any handler mapping bean defined in Spring’s application context then the dispatcher servlet uses BeanNameUrlHandlerMapping.

Let us assume that we have three web pages in our application. The URL of the pages are:

1. [http://servername:portnumber/ApplicationContext/welcome.htm](http://servernameportnumber/)
2. [http://servername:portnumber/ApplicationContext/listBooks.htm](http://servernameportnumber/)
3. [http://servername:portnumber/ApplicationContext/displayBookContent.htm](http://servernameportnumber/)

The controllers which will perform the business logic to fulfil the request made to the above pages are:

1. net.codejava.frameorks.spring.mvc.controller.WelcomeController
2. net.codejava.frameorks.spring.mvc.controller.ListBooksController
3. net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController

Thus we need to define the controllers in Spring’s application context file such that the name of the controller matches the URL of the request. The controller beans in XML configuration file will look as below.

|  |  |
| --- | --- |
| <bean      name="/welcome.htm"      class="net.codejava.frameorks.spring.mvc.controller.WelcomeController" />  <bean      name="/listBooks.htm"      class="net.codejava.frameorks.spring.mvc.controller.ListBooksController"/>  <bean      name="/displayBookTOC.htm"      class="net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController"/> |  |

Note that we need not define the BeanNameUrlHandlerMapping in Spring’s application context file because this is the default one being used.

### SimpleUrlHandlerMapping

The BeanNameUrlHandlerMapping puts a restriction on the name of the controller beans that they should match the URL of the incoming request. SimpleUrlHandlerMapping removes this restriction and maps the controller beans to request URL using a property “mappings”.

|  |  |
| --- | --- |
| <bean   id="myHandlerMapping"   class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">     <property name="mappings">        <props>           <prop key="/welcome.htm">welcomeController</prop>           <prop key="/listBooks.htm">listBooksController</prop>           <prop key="/displayBookTOC.htm">displayBookTOCController</prop>        </props>     </property>  </bean>  <bean name="welcomeController"   class="net.codejava.frameorks.spring.mvc.controller.WelcomeController"/>  <bean name="listBooksController"   class="net.codejava.frameorks.spring.mvc.controller.ListBooksController"/>  <bean name="displayBookTOCController"   class="net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController"/> |  |

The key of the <prop> element is the URL pattern of the incoming request. The value of the <prop> element is the name of the controller bean which will perform the business logic to fulfil the request. SimpleUrlHandlerMapping is one of the most commonly used handler mapping.

## 5. Controllers

* [MultiActionController](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=5#multiactioncontroller)

Controller is the actual piece of code which performs the business logic to fulfil the incoming request. Controllers may delegate this responsibility to further service objects as well. All the user defined controllers must either implement the interface Controller or extend the abstract class AbstractController. The user defined controllers need to override the method handleRequestInternal. The method handleRequestInternaltakes HttpServletRequest and HTTPServletResponse as the input and returns an object of ModelAndView.

In the Spring’s application context file, we have defined a user defined custom controller named welcomeController. As per the SimpleUrlHandlerMApping, all the requests matching URL pattern /welcome.htm will be handled by this controller. The WelcomeController must extend AbstractController and provide the definition of method handleRequestInternal. Thus WelcomeController looks as below:

public class WelcomeController extends AbstractController {

    @Override

    protected ModelAndView handleRequestInternal(HttpServletRequest arg0,

            HttpServletResponse arg1) throws Exception {

        return new ModelAndView("welcome");

    }

}

### MultiActionController

In any medium to large size enterprise web application, there are quite a number of web pages. To fulfil the request for those web pages we need to define multiple controllers, one each for a web page. And sometimes the business logic is executed to fulfil those requests is similar. This creates redundancy of business logic in multiple controllers and makes the maintenance difficult.

Spring’s MVC module provides a way to deal with this scenario by providing a single controller fulfilling the request for multiple web pages. Such a controller is known as Multi Action Controller. A user defined multi action controller should extend the class org.springframework.web.servlet.mvc.multiaction.MultiActionController. Each method in user defined multi action controller contains the logic to fulfil the request for a particular web page.

By default, the URL of the incoming request (excluding the extension part) will be matched against the name of the method in multi action controller and the matching method will perform the business logic for the incoming request. So for the incoming request with URL /welcome.htm, the method name containing the business logic will be welcome.

Let us assume that the multi action controller in our application is MyMultiActionController which fulfils the request for the three web pages with URL /welcome.htm, /listBooks.htm and /displayBookTOC.htm. Thus the class should extend the MultiActionController and have three methods with name welcome, listBooks and displayBookTOC. The controller will look as below:

public class MyMultiActionController extends MultiActionController {

    // This method will server all the request matching URL pattern /welcome.htm

    public ModelAndView welcome(HttpServletRequest request,

            HttpServletResponse response) {

        // Business logic goes here

        // Return an object of ModelAndView to DispatcherServlet

        return new ModelAndView("Welcome");

    }

    // This method will server all the request matching URL pattern

    // /listBooks.htm

    public ModelAndView listBooks(HttpServletRequest request,

            HttpServletResponse response) {

        // Business logic goes here

        // Return an object of ModelAndView to DispatcherServlet

        return new ModelAndView("listBooks");

    }

    // This method will server all the request matching URL pattern

    // /displayBookTOC.htm

    public ModelAndView displayBookTOC(HttpServletRequest request,

            HttpServletResponse response) {

        // Business logic goes here

        // Return an object of ModelAndView to DispatcherServlet

        return new ModelAndView("displayBookTOC");

    }

}

### MethodNameResolver

Spring MVC provides a number of other method name resolvers that helps to resolve the multi action controller method name based on the request. Some of these are:

## ParameterMethodNameResolver

A particular parameter in the request contains the method name. The name of the parameter is defined in the Spring’s application context file while defining ParameterMethodNameResolver. In the example below, the parametercontrollerMethod in the request will determine the multi action controller method which will be executed to fulfil the request.

<bean name="parameterMethodNameResolver"

      class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">

   <property name="paramName">

      <value>controllerMethod</value>

   </property>

</bean>

Notes: The request for a particular web page should now contain an additional parameter with name “controllerMethod” and value as the multi action controller method name to be executed. The request URL will be as follow:

1. [http://servername:portnumber/ProjectWebContext/welcome.htm?controllerMethod=](http://servernameportnumber/) handleWelcomePage
2. [http://servername:portnumber/ProjectWebContext/listBooks.htm?controllerMethod=](http://servernameportnumber/) handleListBooksPage
3. [http://servername:portnumber/ProjectWebContext/displayBookTOC.htm?controllerMethod=](http://servernameportnumber/)handleDisplayBookTOCPage

In the above configuration, the request for URL /welcome.htm will be fulfilled by method handleWelcomePage of multi action controller. Request for URL /listBooks.htm will be fulfilled by method handleListBooksPage and request for URL/displayBookTOC.htm will be fulfilled by method handleDisplayBookTOCPage.

## PropertiesMethodNameResolver

The name of method is determined from the list of pre-defined properties supplied to the method name resolver in Spring’s application context file. The PropertiesMethodNameResolver in Spring’s application context file will look as below.

<bean name="propertiesMethodNameResolver"

      class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">

   <property name="mappings">

      <props>

         <prop key="/welcome.htm">handleWelcomePage</prop>

         <prop key="/listBooks.htm">handleListBooksPage</prop>

         <prop key="/displayBookTOC.htm">handleDisplayBookTOCPage</prop>

      </props>

   </property>

</bean>

Again, in the above configuration, the request for URL /welcome.htm will be fulfilled by method handleWelcomePage of multi action controller. Request for URL /listBooks.htm will be fulfilled by method handleListBooksPage and request for URL /displayBookTOC.htm will be fulfilled by method handleDisplayBookTOC.

We need to tell the multi action controller to use a particular method name resolver by setting its propertymethodNameResolver. Thus the configuration of multi action controller will look as below:

<bean name="myMultiActionController"

      class="net.codejava.frameworks.spring.mvc.controller.MyMultiActionController">

   <property name="methodNameResolver">

      <ref bean="propertiesMethodNameResolver"/>

   </property>

</bean>

## 6. ModelAndView and ViewResolver

* [ModelAndView](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=6#modelandview)
* [ViewResolver](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=6#vr)
  + [InternalResourceViewResolver](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=6#internalresourcevr)
  + [BeanNameViewResolver](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=6#beannamevr)
  + [XMLFileViewResolver](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=6#xmlfilevr)
  + [ResourceBundleViewResolver](http://www.codejava.net/frameworks/spring/understanding-spring-mvc?showall=&start=6#resourcebundlevr)

## ModelAndView

Spring’s MVC module encapsulates the model object and the view object in a single entity which is represented by the object of class ModelAndView. This object contains the model object and view object or the logical name of the view. The model object is the application data and the view is the object that renders the output to the user. The controller returns an object ofModelAndView to the dispatcher servlet for further processing.

## ViewResolver

In case ModelAndView object contains the logical name of the view then the DispatcherServlet needs resolving the view object based on its logical name. To resolve the view object, DispatcherServlet take the help of ViewResolver. There are number of implementation of view resolver provided by Spring. All the view resolvers implement the interfaceorg.springframework.web.servlet.ViewResolver.

### InternalResourceViewResolver

It resolves the logical name of the view to an internal resource by prefixing the logical view name with the resource path and suffixing it with the extension.

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

If the logical name of the view returned by the controller in ModelAndView object is Welcome then the view which is shown to the user is /WEB-INF/jsp/Welcome.jsp

### BeanNameViewResolver

It resolves the logical name of the view to the bean name which will render the output to the user. The bean should be defined in the Spring app context file. So if the logical name returned by the controller in ModelAndView object is Welcomethen the bean with name Welcome defined in the application context will be responsible to render the model to the user.

### XMLFileViewResolver

This view resolver is the same as BeanNameViewResolver with only difference is that instead of looking for the beans in Spring’s application context file it looks for beans defined in a separate XML file (/WEB-INF/views.xml by default). The location and file name can be overridden by providing a location property while defining the XMLFileViewResolver.

|  |  |
| --- | --- |
| <bean name="propertiesMethodNameResolver"        class="org.springframework.web.servlet.view.XMLFileViewResolver">        <propertyname="location">              <value>classpath:myViews.xml</value>        </property>  </bean> |  |

### ResourceBundleViewResolver

It resolves the logical name of the view to the actual view defined in the resource bundle. This view resolver takes basenameas the input which is the name of the property file where views can be located.

|  |  |
| --- | --- |
| <bean name="propertiesMethodNameResolver"        class="org.springframework.web.servlet.view.ResourceBundleViewResolver">     <property name="basename">        <value>myViews</value>     </property>  </bean> |  |

So if the logical name returned by the controller in ModelAndView object is Welcome then the view resolver will look for the property Welcome.class in properties file myViews.properties (or myViews\_en\_US.properties depending upon the user language and locale).