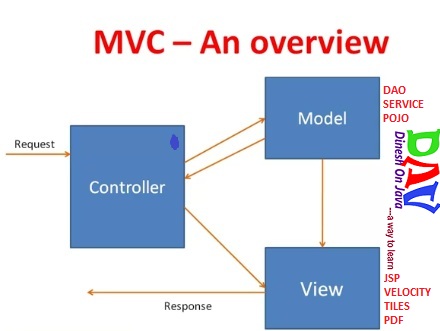
**Spring Web MVC Framework**

**Model view controller** 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.

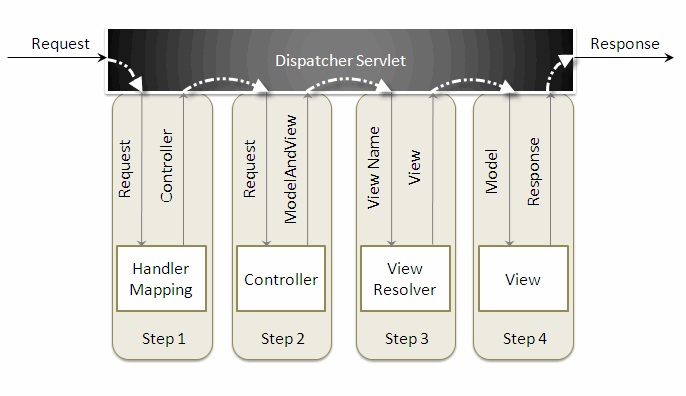


**Advantages of Spring MVC Framework-**

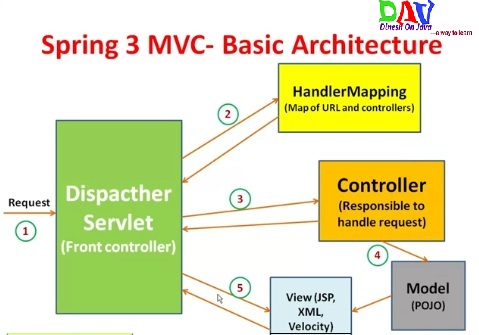
* Supports [RESTful URLs](http://en.wikipedia.org/wiki/Representational_state_transfer).
* Annotation based configuration(i.e. you may reduce the metadata file or less of configuration).
* Supports to plug with other MVC frameworks like Struts, Struts2, WebWorks etc.
* Flexible in supporting different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

**Spring MVC Flow Execution:-**

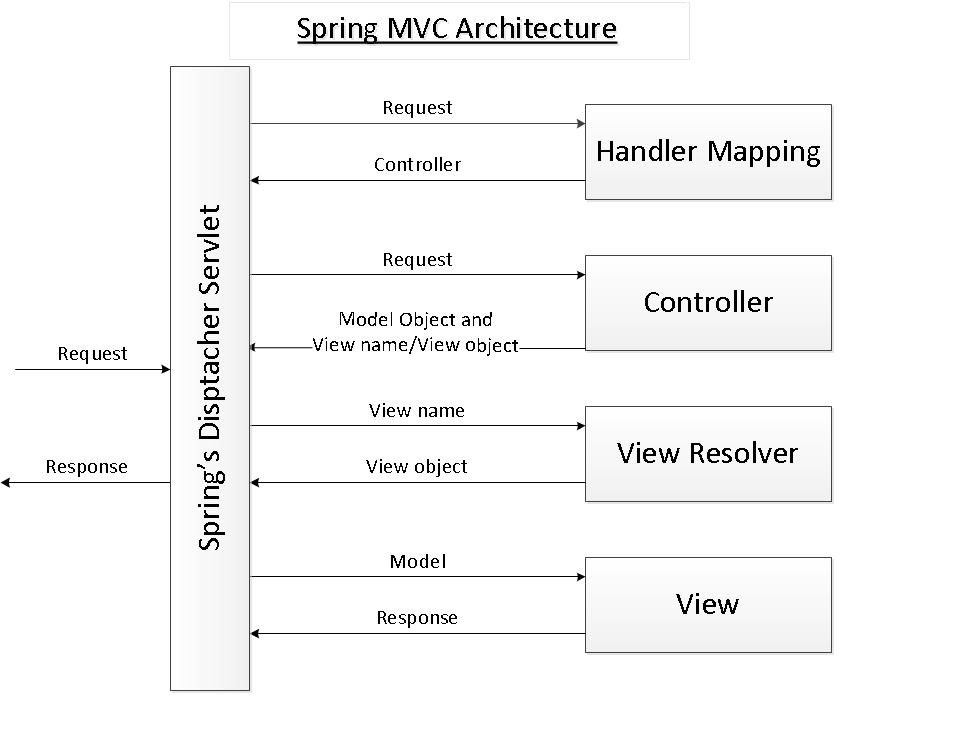


Step **1**: First request will be received by DispatcherServlet  
Step **2**: DispatcherServlet will take the help of HandlerMapping and get to know the Controller class name associated with the given request  
Step **3**: So request transfer to the Controller, and then controller will process the request by executing appropriate methods and returns ModeAndView object (contains Model data and View name) back to the DispatcherServlet  
Step 4: Now DispatcherServlet send the model object to the ViewResolver to get the actual view page  
Step **5**: Finally DispatcherServlet will pass the Model object to the View page to display the result

-------------------------------- ---------------------- - --------------------------------



In Spring 3 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.



1. **Spring’s dispatcher Servlet**: acts as a front controller between the spring application and its clients. The dispatcher Servlet intercepts all requests coming to the application and consults the Handler Mapping for which controller to be invoked to handle the requests.
2. **Handler mapping**: is responsible to find appropriate controllers that handle specific requests. The mapping between request URLs and controller classes is done via XML configuration or annotations.
3. **Controller:** is responsible to process the requests by calling other business/service classes. The output can be attached to model objects which will be sent to the view. To know which view will be rendered, the controller consults the View Resolver.
4. **View Resolver:** finds the physical view files from the logical names.
5. **View:** physical view files which can be JSP, HTML, XML, Velocity template, etc.

**<<Servlet-name>>-servlet.xml** file is loaded by the spring’s **DispatcherServlet** which receives all requests coming into the application and dispatches processing for controllers, based on the configuration specified in this **<<Servlet-name>>-servlet.**xml file. Let’s look at some default configurations:

1. **<annotation-driven />**: tells the framework to use annotations-based approach to scan files in the specified packages. Thus we can use the @Controller annotation for the controller class, instead of declaring XML elements.
2. **<resources mapping=…/>**: maps static resources directly with HTTP GET requests. For example images, JavaScript, CSS, and Resources do not have to go through controllers.
3. Bean **InternalResourceViewResolver**: this bean declaration tells the framework how to find physical JSP files according to logical view names returned by the controllers, by attaching the prefix and the suffix to a view name. For example, if a controller’s method returns “home” as logical view name, then the framework will find a physical file “home.jsp” under the /WEB-INF/views directory.
4. **<context: component-scan …/>**: tell the framework which packages to be scanned when using annotation-based strategy.

**Front Controller:-** 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.

**Handler Mappings:-**

The handler mapping maps the request with the corresponding request handler (e.g. controller, 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 (nireekshan) 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.

Interface to be implemented by objects that define a mapping between requests and handler objects.

**HandlerMapping** implementations can support mapped interceptors but do not have to (*kee jaroorat nahin hai*). A handler will always be wrapped in a **HandlerExecutionChain** instance, optionally accompanied (saath) by some **HandlerInterceptor** instances. The DispatcherServlet will first call each **HandlerInterceptor**’s **preHandle** method in the given order, finally invoking the handler itself if all preHandle methods have returned true.

HandlerExecutionChain **getHandler** (HttpServletRequest request) throws Exception

Return a handler and any interceptors for this request. The choice may be made on request URL, session state, or any factor the implementing class chooses.

The returned HandlerExecutionChain contains a handler Object, rather than even a tag interface, so that handlers are not constrained in any way. For example, a HandlerAdapter could be written to allow another framework's handler objects to be used.

Returns null if no match was found. This is not an error. The DispatcherServlet will query all registered HandlerMapping beans to find a match, and only decide there is an error if none can find a handler.

Exception - if there is an internal error

1. **BeanNameUrlHandlerMapping**

**My Words:** *BeanNameUrlHandlerMapping me se Bean Name Url ko alag kr lete h aur isse hum samj sakte h ki Bean ka Name hi URL hai, ya phir Bean ka name hi URL ko belong krta h, jaise jb koi request aati h to wo is tarha hoti h* [*http://localhost:8080/projectName/abc.htm*](http://localhost:8080/projectName/abc.htm) *, to agar humne BeanNameUrlHandlerMapping ko configure kr rakha h to bean ka name is tarha se hoga <bean name=”/abc.htm” class=”ControllerClass”/>*

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.

1. **DefaultAnnotationHandlerMapping**

**My Words:** *DefaultAnnotationHandlerMapping BeanNameUrlHandlerMapping ki tarha hi default handler mapping h, yanike agar koi handler mapping configure nahi h to ya to by default ye kam karegi ya BeanNameUrlHandlerMapping. Lekin DefaultAnnotationHandlerMapping tab kam ktri h jab controller banane ke liye annotation ko use kiya ho.*

Mappings declared via the **@RequestMapping** annotation are mapped via this Handler. The annotation may be expressed at the method level or type level. This mapping is registered by default, however if you register a custom Handler then the defaults are overridden. **@RequestMappings** may be defined at the type level and the method level. At the type level they map requests to the handler bean. The method level mappings help in narrowing requests. To identify that a class is a Controller use the **@Controller** annotation at type level. However if you have used **@RequestMapping** at the type level then the **@Controller** annotation may be omitted. However, if the **@RequestMapping** annotation is present at the method level and not at the type level then the annotation is required. The mappings expressed at the method level are processed via the **AnnotationMethodHandlerAdapter** and the mappings expressed at the type level are processed via specific Handlers.

<context:component-scan base-package="spring.first.controller" />

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>

1. **ControllerBeanNameHandlerMapping**

**My Words:** *ControllerBeanNameHandlerMapping BeanNameUrlHandlerMapping ki tarh hi h likin hum isko is tarh se samj sakte hai ki reqeuest URL ki mapping bean ke name se ho chahe usme slash ‘/’ nahi laga h, yanike url agar controller ke bean name se match hota h to configured controller execute ho jayega,*

This is similar to BeanNameUrlHandlerMapping in that it maps a URL path to a bean name by looking at the registered Controller beans or annotated beans. However, unlike BeanNameUrlHanlderMapping, the bean name may not be a URL. For example, it can map the URL "/app" to a Handler named "app". You can also specify a prefix and a suffix to the bean name to map the URL.

This is similar to BeanNameUrlHandlerMapping but doesn’t expect bean names to follow the URL convention: It turns plain bean names into URLs by prep ending a slash and optionally applying a specified prefix and/or suffix. For Example:

<bean id="handlerMapping" class="**org.springframework.web.servlet.mvc.support.ControllerBeanNameHandlerMapping**"/>

<bean name="helloworld.html" class="com.kruders.controller.HelloWorldController" />

1. **ControllerClassNameHandlerMapping**

**My Words:** *isko hm is tarh se samj sakte h isme request URL ki mapping controller class ke name se hoti hai aur hume controller class ki bean declare krte time bean name dene ki jarurat nahi h, isme request URL jb aayega to wo configured controller ke name ko lower case me convert krke aur slash ‘/’ laga kr match krta h agar ye requested URL se match hota h to controller execute ho jayega.*

Here the mapping is generated by comparing the URL path to the class name of the registered or annotated controllers. For simple controllers, the logic is to use ClassUtils.getShortName to obtain the class name, remove 'Controller' suffix if present, convert the remaining string to lower case and add '/' at the beginning.

<bean id="handlerMapping" class="**org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping**">

<property name=”caseSensitive” value=”true”>

< property name=”pathPrefix” value=”/prefixPathInURL”>

</bean>

<bean class="com.kruders.controller.HelloWorldController" />

1. **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="spring.mvc.controller.WelcomeController"/>**

**<bean name="listBooksController" class="spring.mvc.controller.ListBooksController"/>**

**<bean name="displayBookTOCController" class="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 fulfill the request. SimpleUrlHandlerMapping is one of the most commonly used handlers mapping.

There are two ways of defining **SimpleUrlHandlerMapping,**using**<value>** tag and **<props>** tag. **SimpleUrlHandlerMapping**has a property called **mappings** we will be passing the URL pattern to it. First way defined above and second way defined below:

<beans ...>

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<value>

/welcome.htm=welcomeController

/\*/welcome.htm=welcomeController

/helloGuest.htm=helloGuestController

</value>

</property>

</bean>

<bean id="welcomeController" class="com.common.controller.WelcomeController" />

<bean id="helloGuestController" class="com.common.controller.HelloGuestController" />

</beans>

The left side is the URL patterns while the right side is the handler IDs or names, separate by a equal symbol “=”.

1. **RequestMappingHandlerMapping**

**What is handler execution chain in spring?**

Handler execution chain in spring is a kind of chain of responsibility design pattern composed by handler mapping and handler interceptors. Handler mapper is used to match current request to appropriated controller. Interceptors are the objects invoked before and after some of dispatching actions (controller resolving, view rendering etc.).

We can resume that a handler execution chain is a group of elements used by dispatcher Servlet to process received request. However, all execution chain calls are made by dispatcher Servlet class. **The execution chain is only a kind of container which:**  
**-** defines handler mappings and interceptors   
**-** defines the methods to apply on some moments of the dispatch (after handler adapter retrieving, after controller's method invocation etc.)

## HandlerExecutionChain class

The handler execution chain is represented by **org.springframework.web.servlet.HandlerExecutionChain** class. **Its main two private fields**, **Object handler** and **HandlerInterceptor[] interceptors**, are used in **request's dispatching process**. The first one contains a handler object used to find the handler adapter's instance. The second one is an array containing interceptors to apply to treated request.

In DispatcherServlet class, HandlerExecutionChain retrieval is done through protected **HandlerExecutionChain getHandler (HttpServletRequest request).** It iterates through all available handler mappings and returns the first handler able to treat the request.

The second thing done in DispatcherServlet with HandlerExecutionChain instance is the applying of interceptors pre- and post- calls. That is translated by DispatcherServlet's methods like applyPreHandle, applyPostHandle, applyAfterConcurrentHandlingStarted and triggerAfterCompletion.

**HandlerAdapter**

HandlerAdapter is responsible for actually invoking the handler. The handler is of type Object and hence the dispatcher Servlet can handle any Handler type using the HandlerAdapter.

A HandlerMapping maps a method to a URL, so the DispatcherServlet knows which method should be invoked by a specific request. Then the DispatcherServlet use a HandlerAdapter to invoke the method. The question is why DispatcherServlet does not invoke the method directly? Because there are many ways to invoke a method, like annotation or xml etc. handlerAdapter de-coupled the DispatcherServlet and the invoked action.

You can find Adapter in HandlerAdapter and that part of name comes from Adapter pattern. Adapter is like a bridge between two objects and HandlerAdapter is a bridge between handler object and dispatcher Servlet.

**boolean supports(Object handler)**

Given a handler instance return, whether or not this HandlerAdapter can support it. Typical HandlerAdapters will base the decision on the handler type. HandlerAdapters will usually only support one handler types each.

A typical implementation:

return (handler instanceof MyHandler);

**ModelAndView handle(HttpServletRequest req, HttpServletResponse resp, Object handler)throws Exception**

Use the given handler to handle this request. The workflow that is required may vary widely.

**Parameters:**

req - current HTTP request

resp - current HTTP response

handler - handler to use. This object must have previously been passed to the supports method of this interface, which must have returned true.

**Returns:**

ModelAndView object with the name of the view and the required model data, or null if the request has been handled directly

**Throws:**

Exception - in case of errors

**long getLastModified(HttpServletRequest request, Object handler)**

Same contract as for HttpServlet's getLastModified method. Can simply return -1 if there's no support in the handler class.

**Parameters:**

request - current HTTP request

handler - handler to use

**Returns:**

The last Modified value for the given handler

# SimpleServletHandlerAdapter

# SimpleControllerHandlerAdapter

# HttpRequestHandlerAdapter

# AnnotationMethodHandlerAdapter

# RequestMappingHandlerAdapter

**Handler Interceptor**

As you know about Servlet filters that they can pre-handle and post-handle every web request they serve - before and after it’s handled by that Servlet. In the similar way, you can use HandlerInterceptor interface in your spring web application to pre-handle and post-handle web requests that are handled by Spring MVC handlers. These handlers are mostly used to manipulate the model attributes returned/submitted they are passed to the views/handlers.

## HandlerInterceptor Interface

Handler interceptors are configured in spring’s web application context, so they can make use of any container features and refer to any beans declared in the container. A handler interceptor can be registered for particular URL mappings, so it only intercepts requests mapped to certain URLs.

* ***preHandle(…)*** – called just before the controller
* ***postHandle(…)*** – called immediately after the controller
* ***afterCompletion(…)*** – called just before sending response to view
* ***HandlerInterceptorAdaptor*** – an implementation class of ***HandlerInterceptor*** interface provided by spring as a convenient class. By extending this we can override only the necessary methods out of the three.
* Interceptor classes must be declared in spring context xml configuration file within the tag **<mvc:interceptors>**
* Interceptor can be configured to execute in two ways, execute for all requests and map to specific URL requests.
* ORDER: All global interceptors get executed first and then the mapped interceptor. Among them, the same order in which the interceptors are declared, the execution is also done.
* If true is returned, the execution chain continues and for false, the execution stops for that request with that interceptor.

**Configuration of Interceptor:**

public class CustomRequestHandler extends HandlerInterceptorAdapter

{

...

}

**Ex.1:**

<beans:beans>

……….

……….

<!-- configuring interceptors based on URI -->

<interceptors>

<interceptor>

<mapping path=”/home”>

<beans:bean class=”….CustomRequestHandler”/>

</interceptor>

</interceptor>

</beans:beans>

**Ex.2:**

<bean id="customRequestHandler" class="com.demo.handlers.CustomRequestHandler" />

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="interceptors">

<list>

<ref bean="customRequestHandler" />

</list>

</property>

</bean>

**What is 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.



**Controller classes:**

These are the classes whose methods are called by DispatcherServlet. A controller class is the one which provides the implementation of **org.springframework.web.servlet.mvc.Controller**interface directly or indirectly.

**Controller Interface:**

public interface **Controller {**

ModelAndView **handleRequest** (HttpServletRequest request, HttpServletResponse response) throws Exception

}

Base Controller interface, representing a component that receives **HttpServletRequest** and **HttpServletResponse** like an **HttpServlet** but is able to participate in an MVC workflow.

Any implementation of the Controller interface should be a *reusable, thread-safe* class, capable of handling multiple HTTP requests throughout the lifecycle of an application. To be able to configure Controller in an easy way, Controllers are usually JavaBeans.

**Notes on design and testing**

The Controller interface is explicitly designed to operate on HttpServletRequest and HttpServletResponse objects, just like an HttpServlet. It does not aim to decouple from the Servlet API, in contrast to, for example, WebWork, JSF or Tapestry. Instead, the full power of the Servlet API is available, allowing Controllers to be general-purpose: not only to handle web user interface requests but also to process remoting protocols or to generate reports on demand.

Controllers can easily be tested through passing in mock objects for Servlet request and response. For convenience, spring ships with a set of Servlet API mocks that are suitable for testing any kind of web components, but are particularly suitable for testing spring web controllers. In contrast to a Struts Action, there is no need to mock the ActionServlet or any other infrastructure; HttpServletRequest and HttpServletResponse are sufficient.

If Controllers need to be aware of specific environment references, they can choose to implement specific awareness interfaces, just like any other bean in a Spring (web) application context can do, for example:

1. **org.springframework.context.ApplicationContextAware**
2. **org.springframework.context.ResourceLoaderAware**
3. **org.springframework.web.context.ServletContextAware**

Such environment references can easily be passed in testing environments, through the corresponding setters defined in the respective awareness interfaces. In general, it is recommended to keep the dependencies as minimal as possible: for example, if all you need is resource loading, implement **ResourceLoaderAware** only. Alternatively, derive from the WebApplicationObjectSupport base class, which gives you all those references through convenient accessors - but requires an **ApplicationContext** reference on initialization.

**Type of Controller**

1. **AbstractController**
2. **AbstractCommandController**
3. **AbstractFormController**
4. **AbstractUrlViewController**
5. **AbstractWizardFromController**
6. **BaseCommandController**
7. **BurlapServiceExporter**
8. **CancellableFormController**
9. **HessianServiceExporter**
10. **HttpInvokerServiceExporter**
11. **MultiActionController**
12. **ParameterizableViewController**
13. **ServletForwardingController**
14. **ServletWrappingController**
15. **SipmpleFormController**

depricated

1. **UrlFilenameViewController**

There might be times when you do not need or want to write a controller, in this case we just need to configure UrlFilenameViewController and then we can access to view (index.html) by using the URL of this view file.

**<bean id=”urlFilenameController” class=”org.springframework.web.servlet.mvc.UrlFilenameViewController”/>**

Then we can reference urlFilenameController in our handler mapping (the urlMap bean in dispatcher-servlet.xml, for example):

**<prop key=”/help.htm”>urlFilenameController</prop>**

**Spring Annotation Based Controller**

First this about annotation based configuration we don’t need to extend defined base classes and need not implement some specific interfaces.

Spring 2.5 introduced support for annotation based MVC controllers. @RequestMapping, @RequestParam, @ModelAttribute are some of the annotations provided for this implementation.



package com.vaannila.web;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

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

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

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

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

import com.vaannila.domain.User;

import com.vaannila.service.UserService;

**@Controller**

**@RequestMapping("/userRegistration.htm")**

**@SessionAttributes("user")**

**public class UserController {**

private UserService userService;

**@Autowired**

**public void setUserService(UserService userService)** {

this.userService = userService;

}

**@RequestMapping(method = RequestMethod.GET)**

**public String showUserForm(ModelMap model)** {

User user = new User();

model.addAttribute(user);

return "userForm";

}

**@RequestMapping(method = RequestMethod.POST)**

**public String onSubmit(@ModelAttribute("user") User user)** {

userService.add(user);

return "redirect:userSuccess.htm";

}

}

The **@Controller** annotation is used to mark any java class as a controller class.

The **@RequestMapping** annotation is used to map the web request "/userRegistration.htm" to the UserController class.

The **@SessionAttributes** annotation is used to store the model object in the session. In our case the model object is user.

Using the **@Autowired** annotation the container can wire your beans automatically. By default autowire is done by type. Here any class that is compatible with UserService class will be chosen.

In the controller class you need to specify two methods, one for handling the HTTP GET request and the other for handling the HTTP POST request. These methods can have arbitrary names. When the form is first rendered the showUserForm() method will be invoked and when the form is submitted for processing the onSubmit() method will be invoked.

The **@RequestMapping** annotation is used to indicate the type of HTTP request.

The command object associated with the form is initialized in the showUserForm() method. The showUserForm() method has access to the ModelMap, in the showUserForm() method, create an instance of the command object and add it to the ModelMap. To add attribute to the ModelMap you can use the addAttribute() method. This method will automatically generate the attribute names for each element. In our case the user object will by default get the name as "user"; you can also override the default value.

After the form is filled and submitted the onSubmit() method will be called. You can access the command object using the **@ModelAttribute** annotation. Here the command object name is user, the one that we set in the ModelMap. After getting the command object, call the service method to register the user and redirect the control to the "userSuccess.jsp" page.

Yes, here we redirect to the "userSuccess.htm" instead of just returning "userSuccess" this is necessary because if we simply return "userSuccess" the userSuccess.jsp page will be displayed, but when you refresh the page the form will be resubmitted, this is something that we don't want.

The "userSuccess.htm" request is mapped to the UserSuccessController class, here the redirect()method will be called and the control will be transferd to the userSuccess.jsp page.

package com.vaannila.web;

import org.springframework.stereotype.Controller;

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

@Controller

public class UserSuccessController {

@RequestMapping("/userSuccess.htm")

public String redirect()

{

return "userSuccess";

}

}

To enable Spring to auto-detect the controllers you need to specify the base package of the controllers using the **<context:component-scan>** element in the Spring configuration file. **DefaultAnnotationHandlerMapping** and **AnnotationMethodHandlerAdapter** are preregistered in the web application context by default. But you need to explicitly register them, in case you have registered any other handler mappings or handler adapters explicitly.

<?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"

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

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

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

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

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

p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />

<bean id="userService" class="com.vaannila.service.UserServiceImpl" />

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

<bean class="org.springframework.web.servlet.mvc.annotation. DefaultAnnotationHandlerMapping" />

<bean class="org.springframework.web.servlet.mvc.annotation. AnnotationMethodHandlerAdapter" />

</beans>

**ModelAndView**

Model and View (***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.

**View Resolver**

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.

1. **BeanNameViewResolver**

One of the dis-advantages 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,

**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,

**<bean id='beanNameResolver' 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.

**<bean id = ' pdf ' class = 'MyPdfGenerator'/>**

1. **FreeMarkerViewResolver**
2. **InternalPathMehtodNameResolver**
3. **InternalResourceViewResolver**

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.

**public class MyController {**

**public ModelAndView 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 ***myView1.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>**

**<property name='suffix'><value>.jsp</value></property>**

**</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***.

1. **JasperReportsViewResolver**
2. **PropertiesMethodNameResolver**
3. **ParameterMethodNameResolver**
4. **ResourceBundleViewResolver**
5. **UrlBasedViewResolver**
6. **VelocityLayoutViewResolver**
7. **VelocityViewResolver**
8. **XmlViewResolver**
9. **XsltViewResolver**