Spring – web mvc

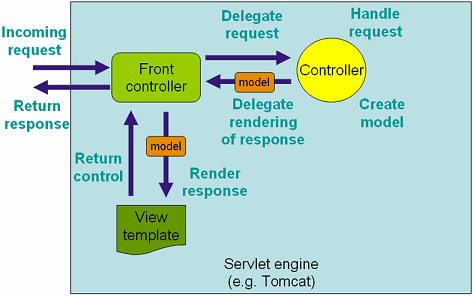
Define MVC:

* MVC (Model-View-Controller) is a software architecture pattern, which separates application into three parts: model, view, and controller.
* The model represents a Java object carrying data.
* The view visualizes the data that the model contains.
* The controller manages the data flow into model object and updates the view whenever data changes; it keeps view and model separate.

Spring MVC Framework

* *Spring MVC* is the original web framework built on the Servlet API.
* It is build on the MVC design pattern.
* Spring MVC is the web component of Spring’s framework.
* Provides a rich functionality for building robust Web Applications.
* It is architected and designed in such a way that every piece of logic and functionality is highly configurable as well as can integrate effortlessly with other popular Web Frameworks like Struts, JSF etc.
* 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 done.
* Spring’s web MVC framework is like other web MVC frameworks – means request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications.
* Spring’s **DispatcherServlet** is completely integrated with Spring IoC container.
* Spring’s **DispatcherServlet** allows to use every other feature of Spring.

Web MVC Request Processing Lifecycle



* + The client sends a request to web container in the form of http request.
  + This incoming request is intercepted by **Front controller** which is **DispatcherServlet** and it will then tries to find out appropriate **Handler Mappings**.
  + With help of Handler Mappings, DispatcherServlet will dispatch the request to appropriate **Controller**.
  + Now Controller tries to process the request and returns the **Model** and **View** object in form of **ModelAndView** instance to Front Controller.
  + Front Controller then tries to resolve the View, which can be JSP, Freemarker, Velocity etc. by consulting **View Resolver** object.
  + At end, the selected view is then rendered back to client.

Configuring Spring MVC

* + The entry point of Spring MVC is the **DispatcherServlet**.
  + DispatcherServlet is a normal servlet class which extends HttpServlet base class.
  + Need to configure it in web.xml.
  + Note: mapped \*.do url pattern with DispatcherServlet
  + Thus any url with \*.do pattern will call
    - Spring MVC Front controller
  + Once DispatcherServlet is initialized,
  + it will looks for a file names [servlet-name]-servlet.xml
  + in WEB-INF folder of web application.
  + Here, the framework will look for file called spring-servlet.xml

Web.xml is defined as :

**<servlet>**

**<servlet-name>spring</servlet-name>**

**<servlet-class>**

**org.springframework.web.servlet.DispatcherServlet**

**</servlet-class>**

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

**</servlet>**

**<servlet-mapping>**

**<servlet-name>spring</servlet-name>**

**<url-pattern>\*.do</url-pattern>**

**</servlet-mapping>**

spring-servlet.xml

* + <beans ......>
  + <context:annotation-config></context:annotation-config>
  + <context:component-scan base-package="com"></context:component-scan>
  + <bean id="jspViewResolver“ class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  + <property name="viewClass“ value="org.springframework.web.servlet.view.JstlView" />
  + <property name="prefix" value="/WEB-INF/jsp/" />
  + <property name="suffix" value=".jsp" />
  + </bean>
  + </beans>

**Model, ModelMap, ModelAndView**

* Model, ModelMap, and ModelAndView are used to define a model in a Spring MVC application.
* Model defines a holder for model attributes and is primarily designed for adding attributes to the model.
* ModelMap is an extension of Model with the ability to store attributes in a map and chain method calls.
* ModelAndView is a holder for a model and a view; it allows to return both model and view in one return value.
* **Model :**
* Simply put, the model can supply attributes used for rendering views.
* To provide a view with usable data, simply add this data to its *Model*object.
* It is an Interface. It defines a holder for model attributes and primarily designed for adding attributes to the model.
* @Controller
* public class MyController {
* @RequestMapping(value="/home1", method = RequestMethod.***GET)***
* **public String getMessage (Model model) {**
* Map<String, String> map = **new HashMap<String, String>();**
* map.put("spring", "mvc");
* map.put("uid", "ram");
* map.put("pwd", "sita");
* model.addAttribute("message", "Sample View with Model");
* model.addAttribute("message1", "Sample View with Model1");
* model.addAttribute("map",map);
* **return "home1";**
* }…….
* }
* In the getMessage() method, use the Model.
* It receives a message attribute with the addAttribute() method.
* The return keyword returns the name of the view, which will be resolved to show.

For this , there is an properties file, where we mention those things:

server.port=8085

spring.mvc.view.prefix = /WEB-INF/jsp/

spring.mvc.view.suffix = .jsp

spring.mvc.static-path-pattern=/resources/\*\*

\*\*It means home1.jsp page will be placed in location : /WEB-INF/jsp/

* **View: home1.jsp**
* <%@ page language=*"java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false“* pageEncoding=*"ISO-8859-1"%>*
* <%@taglib uri=*"http://java.sun.com/jsp/jstl/core" prefix="c"%>*
* <!DOCTYPE html>
* <html><head><meta charset=*"ISO-8859-1">*
* <title>Insert title here</title>
* </head>
* <body>
* <b>God Goddess are good......</b>
* <p>This is my message: ${message}</p>
* <OL>
* <c:forEach items=*"${map}" var="m" >*
* <li>
* Key=${m.key}: Value=${m.value}
* </li>
* </c:forEach>
* </OL>
* </body></html>
* **ModelMap :**
* Just like *Model* interface,*ModelMap* is also used to pass values to render a view.
* Implementation of Map for use when building model data for use with UI tools. Supports chained calls and generation of model attribute names.
* The advantage of *ModelMap* is it gives us the ability to pass a collection of values and treat these values as if they were within a *Map*:
* @Controller
* public class MyController {
* @RequestMapping(value="/home2", method = RequestMethod.***GET)***
* **public String passParametersWithModelMap(ModelMap map) {**
* map.addAttribute("welcomeMessage", "welcome");
* map.addAttribute("message", "Sample View with ModelMap");
* **return "home2";**
* }
* …….
* }
* In the getMessage() method, use the Model.
* It receives a message attribute with the addAttribute() method.
* The return keyword returns the name of the view, which will be resolved to show.

**ModelMap : home2.jsp**

* <%@ page language=*"java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false“* pageEncoding=*"ISO-8859-1"%>*
* <!DOCTYPE html>
* <html>
* <head>
* <meta charset=*"ISO-8859-1">*
* <title>Insert title here</title>
* </head>
* <body>
* <b>${welcomeMessage}</b><br>
* <b>God Goddess are good......</b>
* <p>This is my message: ${message}</p>
* </body>
* </html>
* **ModelAndView:**
* This class merely holds both to make it possible for a controller to return both model and view in a single return value.
* This allows to pass all the information required by Spring MVC in one return:
* @Controller
* public class MyController {
* @RequestMapping(value = "/home3", method = RequestMethod.***GET)***
* **public ModelAndViewhandleRequest(HttpServletRequestrequest,HttpServletResponse response) throws ServletException, IOException {**
* String aMessage = "Sample View with ModelAndView";
* ModelAndViewmodelAndView = **new ModelAndView("home3");**
* modelAndView.addObject("message", aMessage);
* //modelAndView.setViewName("home");
* Map<String, String> map = **new HashMap<String, String>();**
* map.put("spring", "mvc");
* map.put("uid", "ram");
* map.put("pwd", "sita");
* modelAndView.addObject("map", map);
* **return modelAndView;** }
* …….
* }
* Here, use addObject() and setViewName() to add the model data and the view name.
* The method returns ModelAndView object.
* **View: home3.jsp**
* <%@ page language=*"java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false“* pageEncoding=*"ISO-8859-1"%>*
* <%@taglib uri=*"http://java.sun.com/jsp/jstl/core" prefix="c"%>*
* <!DOCTYPE html>
* <html><head><meta charset=*"ISO-8859-1">*<title>Insert title here</title></head>
* <body>
* <b>God Goddess are good......</b>
* <p>This is my message: ${message}</p>
* <OL>
* <c:forEach items=*"${map}" var="m" >*
* <li>
* Key=${m.key}: Value=${m.value}
* </li>
* </c:forEach>
* </OL>
* </body>
* </html>

Spring MVC - @Config

* To configure Spring’s Dispatcher Servlet , need an xml file as :
* Spring-servlet.xml where we map requests for servlets.
* First, in **spring-servlet.xml** where define a view-resolver for identifying the real view , and location to search for beans via component-scanning.
* Second, in **web.xml**, define the front-controller configuration and the url pattern for request be looking on server.
* Without designing XML, may use Configuration by Java as a class.

Spring MVC - **Configuration Class**

* **package com;**
* **importorg.springframework.context.annotation.\*;**
* **importorg.springframework.web.servlet.view.\*;**
* @EnableWebMvc
* @Configuration
* @ComponentScan({ "com" })
* **public class AppConfig {**
* @Bean
* **public InternalResourceViewResolverviewResolver() {**
* InternalResourceViewResolverviewResolver = **new InternalResourceViewResolver();**
* viewResolver.setViewClass(JstlView.**class);**
* viewResolver.setPrefix("/WEB-INF/jsp/");
* viewResolver.setSuffix(".jsp");
* **return viewResolver;**
* } }

Equivalent :**spring-servlet.xml**

<**context:component-scan** base-package=“com" />

    <**mvc:annotation-driven** />

    <**bean**

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

        <**property** name="prefix">

            <**value**>/WEB-INF/views/</**value**>

        </**property**>

        <**property** name="suffix">

            <**value**>.jsp</**value**>

        </**property**>

    </**bean**>

* Explanation:
* @EnableWebMvc :
  + equivalent to mvc:annotation-driven in XML.
  + enables support for @Controller-annotated classes that use
    - @RequestMapping to map incoming requests to specific method.
* @ComponentScan :
  + equivalent to context:component-scan base-package="..."
  + providing with where to look for spring managed beans/classes.

|  |  |  |
| --- | --- | --- |
| XML Tag | Annotation | Description |
| <context:component-scan/> | @ComponentScan | Scan starts from base package and registers all controllers, repositories,  service, beans, etc. |
| <mvc:annotation-driven/> | @EnableWebMvc | Enable Spring MVC-specific annotations like @Controller |
| Spring config file | @Configuration | Treat as the configuration file for Spring MVC-enabled applications. |

Spring MVC - **Initialization class**

* Add an initializer class implementing WebApplicationInitializer interface under src/main/java.
* During Servlet 3.0 Container startup, this class will be loaded and instantiated and its onStartup method will be called by servlet container.
* public interface WebApplicationInitializer {
* public void onStartup(ServletContext container);
* }
* Interface to be implemented in Servlet 3.0+ environments
  + in order to configure ServletContext programmatically .

Need to design a class by implementing this interface

& override onStartUp() to initialize the Web App Environment.

No need to define anything in web.xml file.

**package com;**

**importjavax.servlet.ServletContext;**

**importjavax.servlet.ServletException;**

**importjavax.servlet.ServletRegistration;**

**importorg.springframework.web.WebApplicationInitializer;**

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

**importorg.springframework.web.servlet.DispatcherServlet;**

**public class MvcWebApplicationInitializer implements WebApplicationInitializer {**

**public void onStartup(ServletContext container) throws ServletException {**

AnnotationConfigWebApplicationContextctx = **new AnnotationConfigWebApplicationContext();**

ctx.register(AppConfig.**class);**

ctx.setServletContext(container);

ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", **new DispatcherServlet(ctx));**

servlet.setLoadOnStartup(1);

servlet.addMapping("\*.do");

}

}

Spring MVC – **Controller class**

* @Controller
* **public class MainServletController {**
* @RequestMapping(value="/home1", method = RequestMethod.***GET)***
* **public String passParametersWithModel(Model model) {**
* Map<String, String> map = **new HashMap<String, String>();**
* map.put("spring", "mvc");
* map.put("uid", "ram");
* map.put("pwd", "sita");
* model.addAttribute("message", "Sample View with Model");
* model.addAttribute("message1", "Sample View with Model1");
* model.addAttribute("map",map);
* **return "home1";**
* }

**Index.jsp**

<html>

<body>

<h2>Hello World!</h2>

<a href=*'home1.do'>Home1</a>|*

<hr>

</body>

</html>

home1.jsp

<b>God Goddess are good......</b>

<p>This is my message: ${message}</p>

<p>This is my message: ${message1}</p>

<p>This is my message: ${map}</p>

<OL>

<c:forEach items=*"${map}" var="m" >*

<li>

Key=${m.key}: Value=${m.value}

</li>

</c:forEach>