1. Spring MVC framework

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

The Model encapsulates the application data and in general they will consist of POJO.

The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.

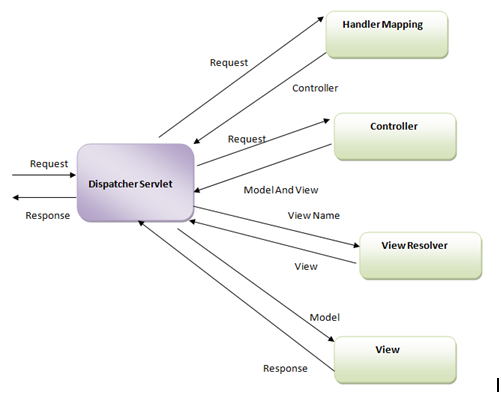
The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram:



<http://www.wideskills.com/spring/spring-mvc-framework>



Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

(1)After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.

(2)The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.

(3)The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.

(4)Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

All the above mentioned components ie. HandlerMapping, Controller and ViewResolver are parts of WebApplicationContext which is an extension of the plain ApplicationContext with some extra features necessary for web applications.

WebApplicationContext

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that it knows which servlet it is associated with.

Required Configuration

ou need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the web.xml file. The following is an example to show declaration and mapping for HelloWeb DispatcherServlet example:

<web-app id="WebApp\_ID" version="2.4"

xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

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

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

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

</servlet-mapping>

</web-app>

The web.xml file will be kept WebContent/WEB-INF directory of your web application. OK, upon initialization of HelloWeb DispatcherServlet, the framework will try to load the application context from a file named [servlet-name]-servlet.xml located in the application's WebContent/WEB-INF directory. In this case our file will be HelloWeb-servlet.xml.

Next, <servlet-mapping> tag indicates what URLs will be handled by the which DispatcherServlet. Here all the HTTP requests ending with .jsp will be handled by the HelloWeb DispatcherServlet.

Now, let us check the required configuration for HelloWeb-servlet.xml file, placed in your web application's WebContent/WEB-INF directory:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

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

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

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

</bean>

</beans>

Following are the important points about HelloWeb-servlet.xml file:

The [servlet-name]-servlet.xml file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.

The <context:component-scan...> tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.

The InternalResourceViewResolver will have rules defined to resolve the view names. As per the above defined rule, a logical view named hello is delegated to a view implementation located at /WEB-INF/jsp/hello.jsp .

Define a controller

DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The @Controller annotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

@Controller

@RequestMapping("/hello")

public

class HelloController{

@RequestMapping(method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The @Controller annotation defines the class as a Spring MVC controller. Here, the first usage of @RequestMapping indicates that all handling methods on this controller are relative to the /hello path. Next annotation @RequestMapping(method = RequestMethod.GET) is used to declare the printHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write above controller in another form where you can add additional attributes in @RequestMapping as follows:

@Controller

public class HelloController{

@RequestMapping(value = "/hello", method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The value attribute indicates the URL to which the handler method is mapped and the method attribute defines the service method to handle HTTP GET request. There are following important points to be noted about the controller defined above:

You will defined required business logic inside a service method. You can call another methods inside this method as per requirement.

Based on the business logic defined, you will create a model within this method. You can setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".

A defined service method can return a String which contains the name of the view to be used to render the model. This example returns "hello" as logical view name.

Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports etc. But most commonly we use JSP templates written with JSTL. So let us write a simple hello view in /WEB-INF/hello/hello.jsp:

<html>

<head>

<title>Hello Spring MVC</title>

</head>

<body>

<h2>${message}</h2> 🡪 Hello Spring MVC Framework!

</body>

</html>

Here ${message} is the attribute which we have setup inside the Controller. You can have multiple attributes to be displayed inside your view.

E.g Spring helloworld example

<http://www.tutorialspoint.com/spring/spring_mvc_hello_world_example.htm>

web.xml

<web-app id="WebApp\_ID" version="2.4"

xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

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

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

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

</servlet-mapping>

</web-app>

Spring Web configuration file HelloWeb-servlet.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

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

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

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

</bean>

</beans>

Hello.jsp

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

<html>

<head>

<title>Hello World</title>

</head>

<body>

<h2>${message}</h2>

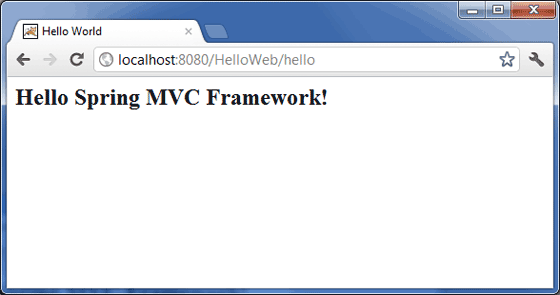
</body>

</html>

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in WebContent/WEB-INF/lib folder.

Once you are done with creating source and configuration files, export your application. Right click on your application and use Export > WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try to access the URL http://localhost:8080/HelloWeb/hello and if everything is fine with your Spring Web Application, you should see the following result:



You should note that in the given URL, HelloWeb is the application name and hello is the virtual subfolder which we have mentioned in our controller using @RequestMapping("/hello"). You can use direct root while mapping your URL using @RequestMapping("/"), in this case you can access the same page using short URL http://localhost:8080/HelloWeb/ but it is advised to have different functionalities under different folders.

E.g Spring MVC form handling

<http://www.tutorialspoint.com/spring/spring_mvc_form_handling_example.htm>

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Create a *Dynamic Web Project* with a name *HelloWeb* and create a package *com.tutorialspoint* under the *src* folder in the created project. |
| 2 | Drag and drop below mentioned Spring and other libraries into the folder*WebContent/WEB-INF/lib*. |
| 3 | Create a Java classes *Student* and *StudentController* under the*com.tutorialspoint* package. |
| 4 | Create Spring configuration files *Web.xml* and *HelloWeb-servlet.xml* under the *WebContent/WEB-INF* folder. |
| 5 | Create a sub-folder with a name *jsp* under the *WebContent/WEB-INF* folder. Create a view files *student.jsp* and *result.jsp* under this sub-folder. |
| 6 | The final step is to create the content of all the source and configuration files and export the application as explained below. |

Here is the content of Student.java file:

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of StudentController.java file:

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

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.servlet.ModelAndView;

import org.springframework.ui.ModelMap;

@Controller

public class StudentController {

@RequestMapping(value = "/student", method = RequestMethod.GET)

public ModelAndView student() {

return new ModelAndView("student", "command", new Student());

}

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

public String addStudent(@ModelAttribute("SpringWeb")Student student,

ModelMap model) {

model.addAttribute("name", student.getName());

model.addAttribute("age", student.getAge());

model.addAttribute("id", student.getId());

return "result";

}

}

Here the first service method student(), we have passed a blank Student object in the ModelAndView object with name "command" because the spring framework expects an object with name "command" if you are using <form:form> tags in your JSP file. So when student() method is called it returns student.jsp view.

Second service method addStudent() will be called against a POST method on the HelloWeb/addStudent URL. You will prepare your model object based on the submitted information. Finally a "result" view will be returned from the service method, which will result in rendering result.jsp

Following is the content of Spring Web configuration file web.xml

<web-app id="WebApp\_ID" version="2.4"

xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Form Handling</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

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

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

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

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file HelloWeb-servlet.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

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

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

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

</bean>

</beans>

Following is the content of Spring view file student.jsp

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

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Student Information</h2>

<form:form method="POST" action="/HelloWeb/addStudent">

<table>

<tr>

<td><form:label path="name">Name</form:label></td>

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

</tr>

<tr>

<td><form:label path="age">Age</form:label></td>

<td><form:input path="age" /></td>

</tr>

<tr>

<td><form:label path="id">id</form:label></td>

<td><form:input path="id" /></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Submit"/>

</td>

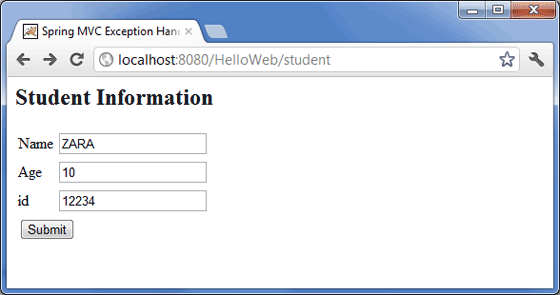
</tr>

</table>

</form:form>

</body>

</html>



Following is the content of Spring view file result.jsp

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

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Submitted Student Information</h2>

<table>

<tr>

<td>Name</td>

<td>${name}</td>

</tr>

<tr>

<td>Age</td>

<td>${age}</td>

</tr>

<tr>

<td>ID</td>

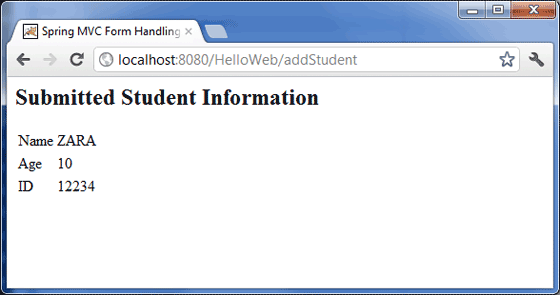
<td>${id}</td>

</tr>

</table>

</body>

</html>



1. What is difference between MVC pattern and 3-tier architecture ?

<http://programmers.stackexchange.com/questions/175950/in-mvc-dao-should-be-called-from-controller-or-model>

In my opinion, you have to distinguish between the MVC pattern and the 3-tier architecture. To sum up:

3-tier architecture:

data: persisted data; 🡪 DAO implementation with @Repository

service: logical part of the application; 🡪 Service implementation with @Service

presentation: hmi, webservice...

The MVC pattern takes place in the presentation tier of the above architecture (for a webapp):

data: ...;

service: ...;

presentation:

controller: intercepts the HTTP request and returns the HTTP response;

model: stores data to be displayed/treated;

view: organize output/display.

Life cycle of a typical HTTP request: 🡪 Note: Critical answer in interview

(1)The user sends the HTTP request;

(2)The controller intercepts it;

(3)The controller calls the appropriate service;

(4)The service calls the appropriate dao, which returns some persisted data (for example);

(5)The service treats the data, and returns data to the controller;

(6)The controller stores the data in the appropriate model and calls the appropriate view;

(7)The view get instantiated with the model's data, and get returned as the HTTP response.

Following is the best example for above 7 steps

<http://howtodoinjava.com/spring/spring-mvc/spring-mvc-hello-world-example/>

<http://www.slideshare.net/guestd0cc01/3-tier-architecture>

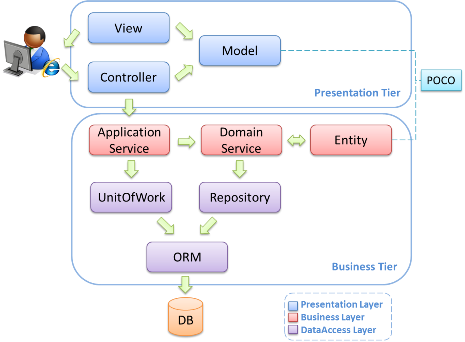
What is MVC Framework?

Model-view-controller (MVC) is a well known design pattern for designing UI based applications. It mainly decouples business logic from UIs by separating the roles of model, view, and controller in an application. Usually, models are responsible for encapsulating application data for views to present. Views should only present this data, without including any business logic. And controllers are responsible for receiving requests from users and invoking back-end services (manager or dao) for business logic processing. After processing, back-end services may return some data for views to present. Controllers collect this data and prepare models for views to present. The core idea of the MVC pattern is to separate business logic from UIs to allow them to change independently without affecting each other.



In a Spring MVC application, models usually consist of POJO objects that are processed by the service layer and persisted by the persistence layer. Views are usually JSP templates written with Java Standard Tag Library (JSTL). Controller part is played by dispatcher servlet which we will learn about in this tutorial in more detail.

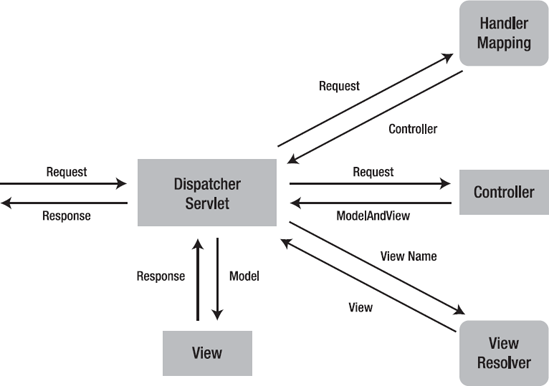
Some developers consider the service layer and DAO layers classes as part of model component in MVC. I have a different opinion on this. I do not consider service and DAO layers classes the part of MVC framework. Usually a web application is 3-tier architecture i.e. data-service-presentation. MVC is actually part of presentation layer



Dispatcher Servlet (Spring Controller)

In the simplest Spring MVC application, a controller is the only servlet you need to configure in a Java web deployment descriptor (i.e., the web.xml file). A Spring MVC controller—often referred to as a Dispatcher Servlet implements front controller design pattern and every web request must go through it so that it can manage the entire request life cycle.

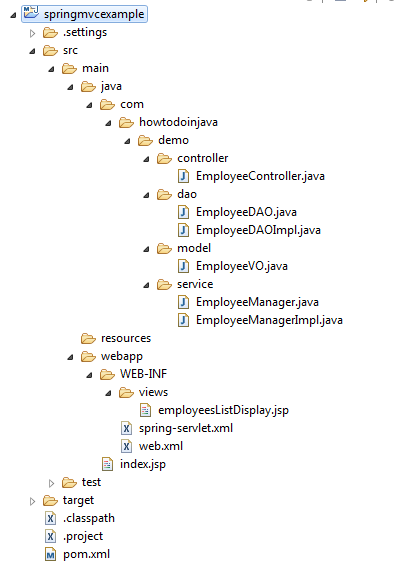
When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring’s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.



To define a controller class in Spring 3.0, a class has to be marked with the @Controller annotation. When a @Controller annotated controller receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class’s methods are decorated with the @RequestMapping annotation, making them handler methods.

After a handler method has finished processing the request, it delegates control to a view, which is represented as handler method’s return value. To provide a flexible approach, a handler method’s return value doesn’t represent a view’s implementation but rather a logical view i.e. without any file extension. You can map these logical views to right implementation into applicationContext file so that you can easily change your view layer code without even touching request handler class code.

To resolve the correct file for a logical name is the responsibility of view resolvers. Once the controller class has resolved a view name into a view implementation, per the view implementation’s design, it renders the objects.



Spring MVC Hello World Example

In this application, I am creating most simple employee management application demo having only one feature i.e. list all available employees in system. Let’s note down the directory structure of this application.

pom.xml

Below pom.xml file contains dependencies for spring mvc and taglibs support for writing jsp files.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.howtodoinjava.demo</groupId>

<artifactId>springmvcexample</artifactId>

<packaging>war</packaging>

<version>1.0-SNAPSHOT</version>

<name>springmvcexample Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!-- Spring MVC support -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<!-- Tag libs support for view layer -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId> 🡪 Note: Here is relate to Java Stand Tag Library

<version>1.2</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>taglibs</groupId>

<artifactId>standard</artifactId>

<version>1.1.2</version>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<finalName>springmvcexample</finalName>

</build>

</project>

web.xml

This minimum web.xml file declares one servlet (i.e. dispatcher servlet) to receive all kind of requests. Dispatcher servlet here acts as front controller.

<web-app id="WebApp\_ID" version="2.4"

xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring Web MVC Hello World Application</display-name>

Note: Here in web.xml we define servlet as DispatchServlet and mapping for url handle

<servlet>

<servlet-name>spring</servlet-name> 🡪 Note: Servlet name is “spring” will match for servletname-servlet.xml below which for loading application context

<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>/</url-pattern>

</servlet-mapping>

</web-app

spring-servlet.xml (You can have applicationContext.xml as well)

We are using annotated classes at request handler, service and dao layer so I have enabled annotation processing for all class files in base package “com.howtodoinjava.demo“. The servletname-servlet.xml used for loading application context when initialize DispatchServlet.

See <http://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="com.howtodoinjava.demo" />

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

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

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

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

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

</bean>

</beans>

Note: What is InternalResourceViewResolver ?

<http://www.mkyong.com/spring-mvc/spring-mvc-internalresourceviewresolver-example/>

In Spring MVC, InternalResourceViewResolver is used to resolve “internal resource view” (in simple, it’s final output, jsp or htmp page) based on a predefined URL pattern. In additional, it allow you to add some predefined prefix or suffix to the view name (prefix + view name + suffix), and generate the final view page URL.

What’s internal resource views?

In Spring MVC or any web application, for good practice, it’s always recommended to put the entire views or JSP files under “WEB-INF” folder, to protect it from direct access via manual entered URL. Those views under “WEB-INF” folder are named as internal resource views, as it’s only accessible by the servlet or Spring’s controllers class.

EmployeeController.java

Annotation @RequestMapping at class level and method level determine the URL at which method will be invoked.

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

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

import com.howtodoinjava.demo.service.EmployeeManager;

@Controller

@RequestMapping("/employee-module")

public class EmployeeController

{

**@Autowired**

**EmployeeManager manager; 🡪 Note: Controller contain Service bean**

@RequestMapping(value = "/getAllEmployees", method = RequestMethod.GET) 🡪

Note: Controller work with requestmapping

public String getAllEmployees(Model model) 🡪 Note: Controller operates with model

{

model.addAttribute("employees", manager.getAllEmployees());

🡪 Note: This “employee” attribute here will mapping to the employeesListDisplay.jsp’s definition as <c:forEach items="${employees}" var="employee">, the usage can be check on

<http://stackoverflow.com/questions/18975077/how-to-add-object-in-using-model-addattributes-in-spring-mvc>

<http://stackoverflow.com/questions/20584411/passing-a-new-object-in-spring-mvc-model-addattribute-method>

return "employeesListDisplay"; 🡪 Note: return with string pattern as logical view without file extension.

}

}

EmployeeVO.java

This class act as model for MVC pattern.-->

Note: Usually, models are responsible for encapsulating application data for views to present.

package com.howtodoinjava.demo.model;

import java.io.Serializable;

public class EmployeeVO implements Serializable

{

private static final long serialVersionUID = 1L;

private Integer id;

private String firstName;

private String lastName;

//Setters and Getters

@Override

public String toString() {

return "EmployeeVO [id=" + id + ", firstName=" + firstName

+ ", lastName=" + lastName + "]";

}

}

EmployeeDAO.java

The classes at third tier in 3-tier architecture. Responsible for interacting with underlying DB storage. 🡪 Data layer

import java.util.List;

import com.howtodoinjava.demo.model.EmployeeVO;

public interface EmployeeDAO

{

public List<EmployeeVO> getAllEmployees();

}

EmployeeDAOImpl.java

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Repository;

import com.howtodoinjava.demo.model.EmployeeVO;

@Repository 🡪 Note: The data layer class should note with @Repository

public class EmployeeDAOImpl implements EmployeeDAO {

public List<EmployeeVO> getAllEmployees()

{

List<EmployeeVO> employees = new ArrayList<EmployeeVO>();

EmployeeVO vo1 = new EmployeeVO();

vo1.setId(1);

vo1.setFirstName("Lokesh");

vo1.setLastName("Gupta");

employees.add(vo1);

EmployeeVO vo2 = new EmployeeVO();

vo2.setId(2);

vo2.setFirstName("Raj");

vo2.setLastName("Kishore");

employees.add(vo2);

return employees;

}

}

EmployeeManager.java

The classes at second tier in 3-tier architecture. Responsible for interacting with DAO Layer. 🡪 Service layer

import java.util.List;

import com.howtodoinjava.demo.model.EmployeeVO;

public interface EmployeeManager

{

public List<EmployeeVO> getAllEmployees();

}

EmployeeManagerImpl.java

import java.util.List;

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

import org.springframework.stereotype.Service;

import com.howtodoinjava.demo.dao.EmployeeDAO;

import com.howtodoinjava.demo.model.EmployeeVO;

@Service

public class EmployeeManagerImpl implements EmployeeManager {

**@Autowired**

**EmployeeDAO dao; 🡪 Note: Service layer class contain Data layer bean**

public List<EmployeeVO> getAllEmployees()

{

return dao.getAllEmployees();

}

}

employeesListDisplay.jsp

This jsp is used to display all the employees in system. It iterates the collection of employees in loop, and print their details in a table. This fits into view layer of MVC pattern.

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<html>

<head>

<title>Spring MVC Hello World</title>

</head>

<body>

<h2>All Employees in System</h2>

<table border="1">

<tr>

<th>Employee Id</th>

<th>First Name</th>

<th>Last Name</th>

</tr>

<c:forEach items="${employees}" var="employee"> 🡪 Note: The ${employee} name match the “attributeName” in model.addAttribute(attributeName, serviceImplClass.method())

<tr>

<td>${employee.id}</td>

<td>${employee.firstName}</td>

<td>${employee.lastName}</td>

</tr>

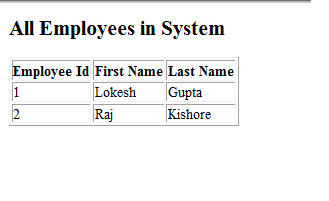
</c:forEach>

</table>

</body>

</html>

Now deploy the application in your application server (i am using tomcat 7). And hit the URL “http://localhost:8080/springmvcexample/employee-module/getAllEmployees“. You will see below screen if you have configured everything correctly.



Conclusion

Web.xml 🡪 Servlet/Servlet-mapping 🡪 DispatchServlet 🡪 ServletName-servlet.xml 🡪 InternalResourceViewResolver

Controller (@Controller) 🡪 RequestMapping (@RequestMapping) 🡪 value/method 🡪 model/ModelAndView/modelAttribute 🡪 JSP

Presentation layer (Spring MVC) 🡪 Businesss layer (Service/@Service) 🡪 Data Access layer (DAO impl/@Repository) 🡪 @Autowired

Combine data/service/presentation 3-tier together with how jsp used with Spring and how hibernate integrate with Spring (Not through JDBCtemplate but SessionFactory initialized in DAO(data layer interface) implement class), this is a full picture of CYLC project Spring design pattern.

(Note: If using Java based annotation, we should use @Configuration + @Bean + @Import)