The Spring web module provides some basic utility classes for web applications and also provides integration points with Struts and JSF. But, by itself, the web module doesn't provide an MVC implementation.

The Spring MVC module is a subframework built upon Spring and the Spring Web Module providing a complete alternative to other MVC implementations.

So, if you're using Struts in your presentation layer and Spring in the service layer, you'll be using the Spring Web Module, but not the Spring MVC module. But if you're using Spring MVC in your presentation layer, you're using both modules.

1. Spring AOP:

One of the key components of Spring is the AOP framework. AOP is used in Spring:

• To provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative transaction management, which builds on Spring's transaction abstraction.

• To allow users to implement custom aspects, complementing their use of OOP with AOP

2. Spring ORM:

The ORM package is related to the database access. It provides integration layers for popular object-relational mapping APIs, including JDO, Hibernate and iBatis.

3. Spring Web:

The Spring Web module is part of Spring?s web application development stack, which includes Spring MVC.

4. Spring DAO:

The DAO (Data Access Object) support in Spring is primarily for standardizing the data access work using the technologies like JDBC, Hibernate or JDO.

5. Spring Context:

This package builds on the beans package to add support for message sources and for the Observer design pattern, and the ability for application objects to obtain resources using a consistent API.

6. Spring Web MVC:

This is the Module which provides the MVC implementations for the web applications.

7. Spring Core:

The Core package is the most import component of the Spring Framework.

This component provides the Dependency Injection features. The BeanFactory provides a factory pattern which separates the dependencies like initialization, creation and access of the objects from your actual program logic.

**Circular Dependencies**

If you use predominantly constructor injection, it may create an unresolvable circular dependency scenario (something similar to deadlock scenario).

For example:

* ClassA requires an instance of ClassB through constructor injection, and ClassB requires an instance of ClassA through constructor injection.
* If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a **BeanCurrentlyInCreationException**.

**ClassA.java**

**public** **class** ClassA

{

**private** ClassB bRef;

**public** ClassA(ClassB bRef)

{

**this**.bRef = bRef;

}

**public** String printMSG()

{

**return** "I am a Class A Method";

}

}//End of Class

**ClassB.java**

**public** **class** ClassB

{

**private** ClassA aRef;

**public** ClassB(ClassA aRef)

{

**this**.aRef = aRef;

}

**public** String printMSG()

{

**return** "I am a Class B Method";

}

}//End of Class

**Application-Context.xml**

<bean id=*"aBean"* class=*"ClassA"*>

<constructor-arg name=*"bRef"* ref=*"bBean"*/>

</bean>

<bean id=*"bBean"* class=*"ClassB"*>

<constructor-arg name=*"aRef"* ref=*"aBean"*/>

</bean>

**SpringTest.java**

ApplicationContext appContext

= **new** ClassPathXmlApplicationContext("applicationcontext.xml");

ClassA aObj = (ClassA) appContext.getBean("aBean");

System.*out*.println(aObj.printMSG());

System.*out*.println(aObj.getbRef().printMSG());

ClassB bObj = (ClassB) appContext.getBean("bBean");

System.*out*.println(bObj.printMSG());

System.*out*.println(bObj.getaRef().printMSG());

**Exception:**

Exception in thread "main" org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'aBean' defined in class path resource [applicationcontext.xml]: Instantiation of bean failed; nested exception is org.springframework.beans.BeanInstantiationException: Could not instantiate bean class [ClassA]: No default constructor found; nested exception is java.lang.NoSuchMethodException: ClassA.<init>()

**Solution:**

Although it is not recommended, you can configure circular dependencies with setter injection. Unlike the typical case (with no circular dependencies), a circular dependency between ClassA and ClassB forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

**ClassA.java**

**public** **class** ClassA

{

**private** ClassB bRef;

**public** ClassA() { }

/\*

\* Generate Getter & Setter

\*/

}//End of Class

**ClassB.java**

**public** **class** ClassB

{

**private** ClassA aRef;

**public** ClassB() { }

/\*

\* Generate Getter & Setter

\*/

}//End of Class

**Application-Context.xml**

<bean id=*"aBean"* class=*"ClassA"*>

<property name=*"bRef"* ref=*"bBean"* />

</bean>

<bean id=*"bBean"* class=*"ClassB"*>

<property name=*"aRef"* ref=*"aBean"* />

</bean>

**Polymorphism in Springs [Interface Injection]:**

**public** **interface** CompanyNameInterface

{

**public** String getCompanyNM();

}//End of Interface

**public** **class** CompanyNameImpl1 **implements** CompanyNameInterface

{

**private** String companyNM;

@Override

**public** String getCompanyNM()

{

**return** companyNM;

}

**public** **void** setCompanyNM(String companyNM) {

**this**.companyNM = companyNM;

}

}//End of CompanyNameImpl

**public** **class** CompanyNameImpl2 **implements** CompanyNameInterface

{

**private** String companyNM;

@Override

**public** String getCompanyNM()

{

**return** companyNM;

}

**public** **void** setCompanyNM(String companyNM) {

**this**.companyNM = companyNM;

}

}//End of CompanyNameImpl2

**public** **class** CompanyProfile

{

**private** String address;

**private** CompanyNameInterface name;

/\*

\* Generate Getter & Setter

\*/

**public** String getCompanyInfo()

{

**return** "Company Name : "

+ name.getCompanyNM()

+ " & Company Address : "

+ address;

}

}//End of Class

**Application-Context.xml**

<bean id=*"compNM1"* class=*"com.jspiders.springs.beans.CompanyNameImpl1"*>

<property name=*"companyNM"* value=*"Hewlet-Packard"* />

</bean>

<bean id=*"compNM2"* class=*"com.jspiders.springs.beans.CompanyNameImpl2"*>

<property name=*"companyNM"* value=*"NetApp"* />

</bean>

<bean id=*"compProfile"* class=*"com.jspiders.springs.beans.CompanyProfile"*>

<property name=*"address"*

value=*"#123, 4th Cross, 5th Main, ABC Nagar, Bangalore"* />

<!-- <property name="name" ref="compNM1" /> -->

<property name=*"name"* ref=*"compNM2"* />

</bean>

**Polymorphism.java**

ApplicationContext context

= **new** ClassPathXmlApplicationContext("ApplicationContext.xml");

CompanyProfile comp = (CompanyProfile) context.getBean(CompanyProfile.**class**);

System.*out*.println(comp.getCompanyInfo());

**Inheritance in Springs:**

* In spring, inheritance is supported for reusing already written bean, so that beans can share common attributes and methods among them.
* Child bean will have all attributes and methods of parent bean, also child bean can override parent bean's attributes or methods.

**Scenario 1: When Super Class is a Normal Class**

**public** **class** SuperClass

{

**private** String empNM;

**public** SuperClass() {}

/\*

\* Generate Getter & Setter

\*/

}//End of SuperClass

**public** **class** SubClass **extends** SuperClass

{

**private** **int** empID;

**public** SubClass() { }

/\*

\* Generate Getter & Setter

\*/

}//End of Class

**ApplicationContext.java**

<bean id=*"superBean"* class=*"com.jspiders.springs.beans.SuperClass"*>

<property name=*"empNM"* value=*"Praveen"* />

</bean>

<bean id=*"subBean"*

class=*"com.jspiders.springs.beans.SubClass"*

parent=*"superBean"* >

<property name=*"empID"* value=*"1234"* />

</bean>

**Inheritence.java**

ApplicationContext context

= **new** ClassPathXmlApplicationContext("ApplicationContext.xml");

SubClass subRef = (SubClass) context.getBean(SubClass.**class**);

System.*out*.println(subRef.getEmpNM());

System.*out*.println(subRef.getEmpID());

**Scenario 2: When Super Class is an Abstract Class**

**public** **abstract** **class** AbstractSuperClass

{

**private** **int** age;

**public** **abstract** String getInfo();

/\*

\* Generate Getter & Setter

\*/

}//End of Class

**public** **class** SubClass2 **extends** AbstractSuperClass

{

**private** String name;

@Override

**public** String getInfo() {

**return** name+ " age is :" +getAge() ;

}

/\*

\* Generate Getter & Setter

\*/

}//End of Class

**ApplicationContext.java**

<bean id=*"abstractSuperClass"*

class=*"com.jspiders.springs.beans.AbstractSuperClass"* abstract=*"true"* >

<property name=*"age"* value=*"99"* />

</bean>

<bean id=*"subBean2"*

class=*"com.jspiders.springs.beans.SubClass2"* parent=*"abstractSuperClass"* >

<property name=*"name"* value=*"Praveen"* />

</bean>

**Inheritence2.java**

ApplicationContext context

= **new** ClassPathXmlApplicationContext("ApplicationContext.xml");

SubClass2 subRef = (SubClass2) context.getBean(SubClass2.**class**);

System.*out*.println(subRef.getInfo());

<bean id=*"eaiLog"* class=*"com.hmh.util.dao.EAI\_Log"*>

<constructor-arg name=*"transId"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"event\_id"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"ebs\_Name"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"source"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"severity"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"cmpnt\_Name"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"evnt\_payLoad"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"err\_StackTrace"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"event\_Actvty\_TS"* type=*"java.sql.Timestamp"* value=*"#{ null }"*/>

<constructor-arg name=*"err\_Family"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"err\_Type"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"msg\_Text"* type=*"java.lang.String"* value=*""* />

<constructor-arg name=*"load\_Actvty\_TS"* type=*"java.util.Date"* value=*"#{ null }"* />

<constructor-arg name=*"destination"* type=*"java.lang.String"* value=*""* />

</bean>

TODO :

<http://www.mkyong.com/spring/spring-how-to-pass-a-date-into-bean-property-customdateeditor/>

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

<http://www.mkyong.com/spring/spring-bean-configuration-inheritance/>

<http://www.mkyong.com/spring/spring-properties-dependency-checking/>

<http://www.mkyong.com/spring/spring-dependency-checking-with-required-annotation/>

Spring Expression Language (Spring 3.0)

<http://www.mkyong.com/spring3/spring-el-hello-world-example/>

<http://www.mkyong.com/spring3/spring-el-bean-reference-example/>

<http://www.mkyong.com/spring3/spring-el-method-invocation-example/>

<http://www.mkyong.com/spring3/spring-el-operators-example/>

<http://www.mkyong.com/spring3/spring-objectxml-mapping-example/>

# **Spring Web MVC Framework**

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

<http://thecafetechno.com/tutorials/spring/hibernate-web-jsp-servlet-integration-example/3/> ===> Vey Good

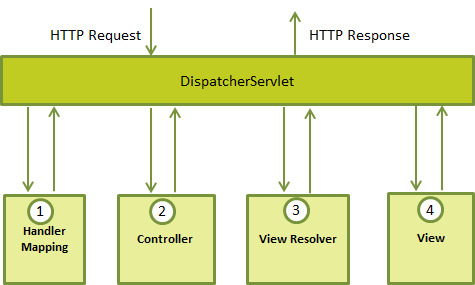
<http://howtodoinjava.com/2013/03/21/spring-3-and-hibernate-integration-tutorial-with-example/>===> Vey Good

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:



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.

## Required Configuration

You 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.

If you do not want to go with default filename as *[servlet-name]-servlet.xml* and default location as *WebContent/WEB-INF*, you can customize this file name and location by adding the servlet listener *ContextLoaderListener* in your web.xml file as follows:

<web-app...>

<!-------- *DispatcherServlet* definition goes here----->

....

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

</web-app>

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* .

Next section will show you how to create your actual components ie. Controller, Model and View.

## Defining 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>

</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.

public class **ModelAndView** extends [Object](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)

* It’s a concrete class
* Holder for both Model and View in the web MVC framework. Note that these are entirely distinct. This class merely holds both to make it possible for a controller to return both model and view in a single return value.
* Represents a model and view returned by a handler, to be resolved by a DispatcherServlet. The view can take the form of a String view name which will need to be resolved by a ViewResolver object; alternatively a View object can be specified directly. The model is a Map, allowing the use of multiple objects keyed by name.

**Constructor and Description**

[**ModelAndView**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html#ModelAndView-java.lang.String-)([**String**](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true) viewName)

Convenient constructor when there is no model data to expose.

[**ModelAndView**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html#ModelAndView-java.lang.String-java.util.Map-)([**String**](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true) viewName, [**Map**](http://docs.oracle.com/javase/8/docs/api/java/util/Map.html?is-external=true)<[**String**](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true),?> model)

Creates new ModelAndView given a view name and a model.