Spring MVC Framework

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.

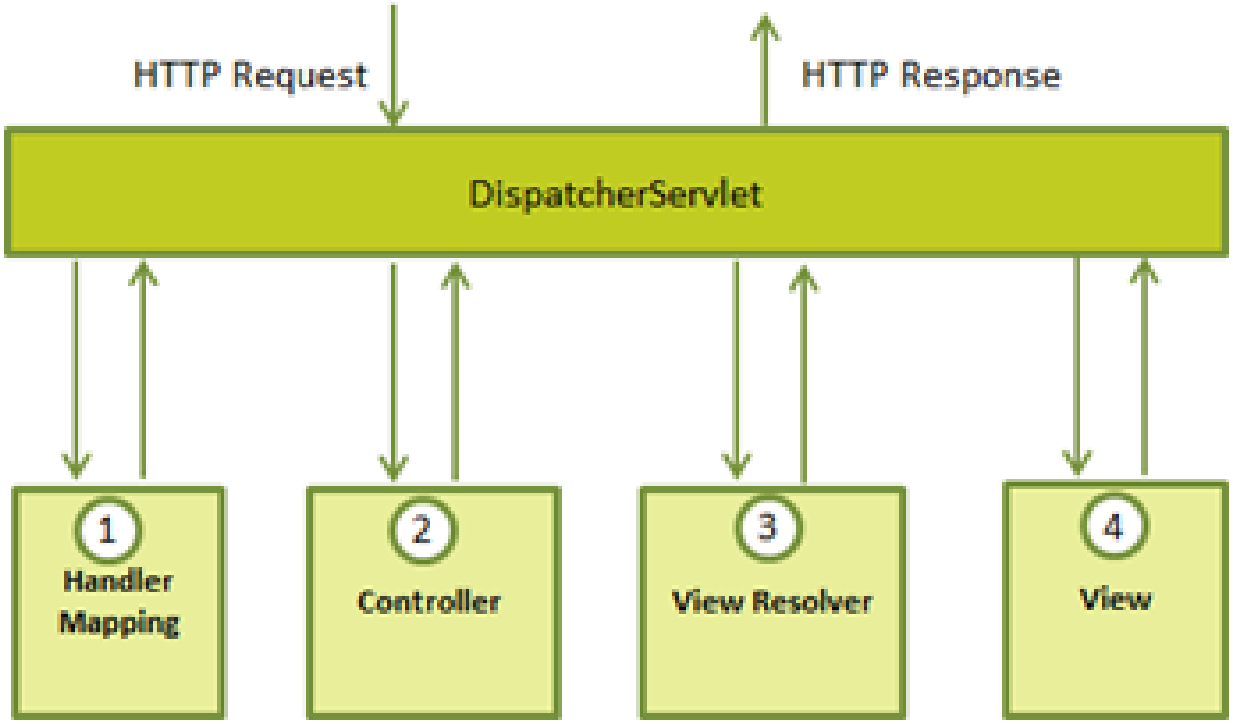
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.

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

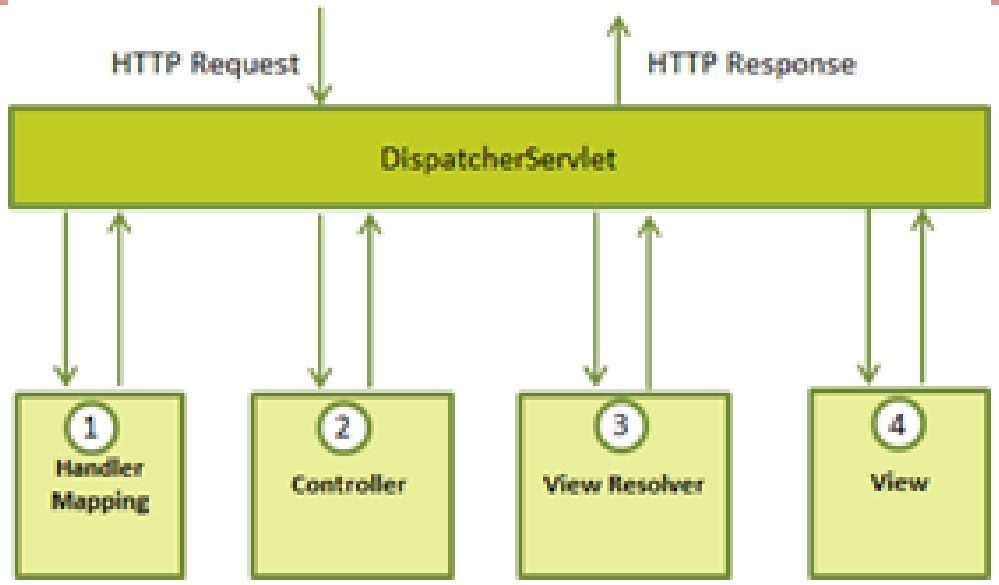
The DispatcherServlet

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:



The DispatcherServlet



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.

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

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- 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*.
- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

Deployment Descriptor file (web.xml)

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID" version="3.1">
 <display-name>SpringMVCDemo</display-name>
<servlet>
<servlet-name>mvc-dispatcher</servlet-name>
      <servlet-class> org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
       <servlet-name>mvc-dispatcher</servlet-name>
      <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Convention over configuration

For a lot of projects, sticking to established conventions and having reasonable defaults is just what they (the projects) need... this theme of convention-over-configuration now has explicit support in Spring Web MVC.

What this means is that if you establish a set of naming conventions and suchlike, you can *substantially* cut down on the amount of configuration that is required to set up handler mappings, view resolvers, ModelAndView instances, etc.

This is a great boon with regards to rapid prototyping, and can also lend a degree of (always good-to-have) consistency across a codebase should you choose to move forward with it into production.

This convention over configuration support address the three core areas of MVC - namely, the models, views, and controllers.

Convention

Deployment Descriptor file (web.xml)

Upon initialization of **mvc-dispatcher** *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 **mvc-dispatcher-servlet.xml** and should be placed under /WEB-INF folder.

When the DispatcherServlet is initialized, the framework will try to load the application context from a file named *[servlet-name]-servlet.xml* located in /WEB-INF/ directory.

The <servlet-mapping> element of web.xml file specifies what URLs will be handled by the DispatcherServlet.

Configuration: Method 1

Changing the default dispatcher servlet name

The servlet name we used in the web.xml was **mvc-dispatcher.**

Due to this the framework will look for a file, **mvc-dispatcher-servlet.xml** (servletname-servlet.xml) to load the Spring MVC configurations.

If we have used a different name for the servlet, say **frontcontroller,** then the framework will look for a file with name **frontcontroller.xml** to load MVC configurations.

We can override this behaviour by explicitly specifying the c configuration file using the parameter **contextConfigLocation in <context-param> tag. Also weneed to add an listener.**

web.xml

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/frontController.xml</param-value>
</context-param>

<listener>
      <listener-class>
            org.springframework.web.context.ContextLoaderListener
      </listener-class>
</listener>
```

Configuration: Method 2

…….

```
<servlet>
   <servlet-name>SpringController</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring-mvc.xml</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>SpringController</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

……..

mvc-dispatcher-servlet.xml

```
<?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:context="http://www.springframework.org/schema/context"
	xmlns:mvc="http://www.springframework.org/schema/mvc"
	xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
		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">

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

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

</beans>
```

Note: The spring configuration file has to be placed under WEB-INF folder

mvc-dispatcher-servlet.xml

Following are the important points about **mvc-dispatcher-servlet** *file:*

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

- Spring container will search for all annotated classes under **org.asr.spring** package.

- The **org.springframework.web.servlet.view.InternalResourceViewResolver** is defined as a bean, and is used as internal resource views resolver, meaning that it will find the jsp and html files in the **WebContent/WEB-INF/ jsp** folder. We can set properties such as **prefix** or **suffix** to the view name to generate the final view page URL

Defining a Controller

The **Controller** is where the **DispatcherServlet** will delegate requests.

The **@Controller** annotation indicates that the 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. Here, it is used for both cases.

The **HelloWorldController.java** class consists of a method, **printHello(ModelMap model)** that will handle a **GET** request from the Dispatcher.

The **org.springframework.ui.ModelMap** is used as a generic model holder.

Controller class : HelloWorldController

HelloWorldController.java

```
@Controller
@RequestMapping("/hello")
public class HelloWorldController {

   @RequestMapping(method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", " Hello World! Welcome to Spring MVC ");
      return "hello";
       }
   }
```

View name

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.

Additional attributes in @RequestMapping

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

HelloWorldController.java

```
@Controller
public class HelloWorldController{
@RequestMapping(value = "/hello", method = RequestMethod.GET)
	public String printHello(ModelMap model) {
		model.addAttribute("message", "Hello World! Welcome to Spring MVC");
		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.

The **model** presents a placeholder to hold the information you want to display on the view

About controller class

Following are important points to be noted about the controller defined above:

- Business logic has to be defined inside a service method. You can call another methods inside this method as per requirement.
- Based on the business logic defined, create a **model** within this method.
- Add attributes the **model** object and these attributes will be accessed by the view to present the final result.
- The defined service method (*ex. printHello()*) can return a String which contains the name of the **view** to be used to render the model.

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.

Let us write a simple **hello.jsp** file in /WEB-INF/jsp/ folder

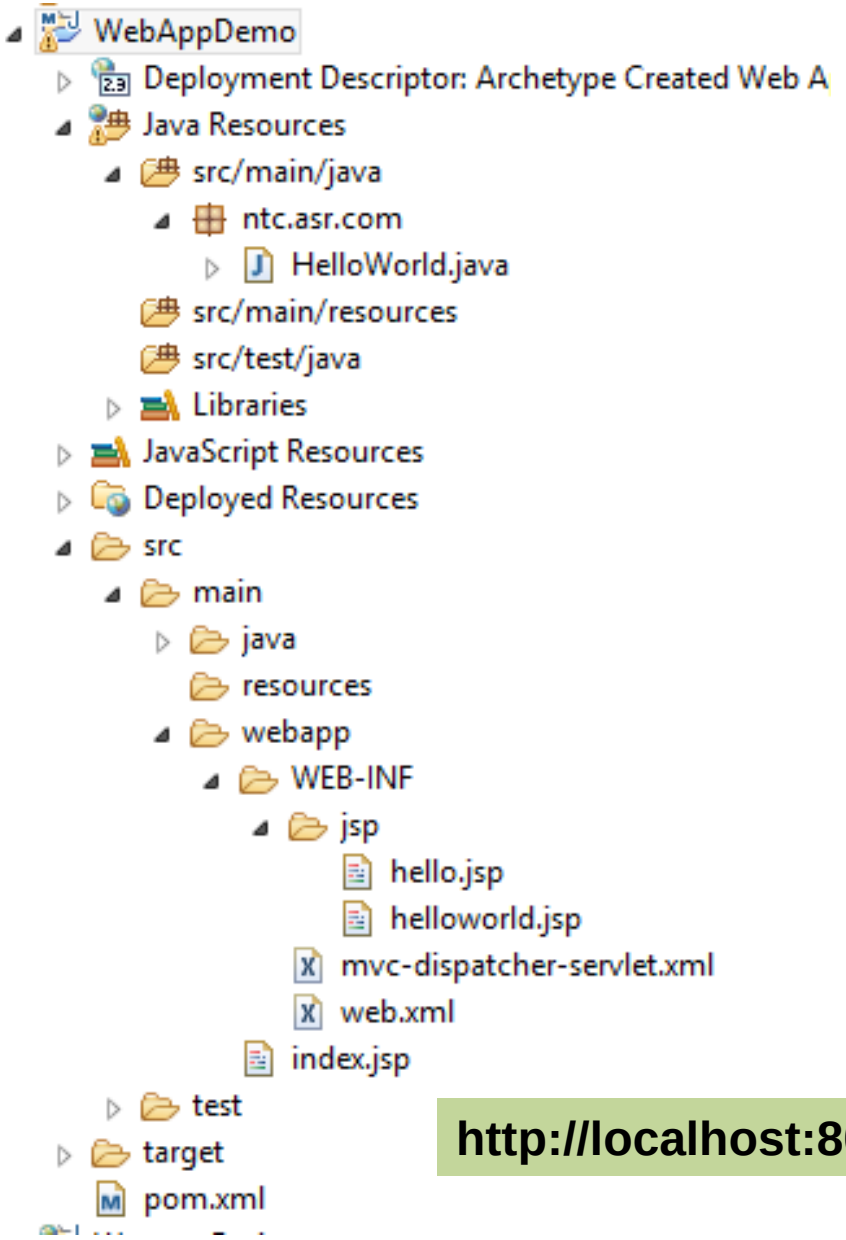
hello.jsp

```
<html>
<head>
<title>Hello Spring MVC</title> </head>
 <body>
        <h2> ${message}</h2>
</body>
</html>
```

If you are using the **old JSP 1.2 descriptor, defined by DTD** ,for example **web.xml ,** The EL is disabled or ignored by default, you have to enable it manually, so that it will outputs the value store in the "msg" model.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" isELIgnored="false"%>
```

WebAppDemo



Additional jar files required:

spring-web-4.0.6.RELEASE.jar

spring-webmvc-4.0.6.RELEASE.jar

Note :

The user library has to placed WEB-INF/lib folder (through Deployment Assembly of configure build path)

Note :

mvc-dispatcher-servlet.xml file should be placed under /WEB-INF folder

http://localhost:8080/WebAppDemo/hello2

web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <servlet>
        <servlet-name>mvc-dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
        <servlet-name>mvc-dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

web.xml

JSP 1.2

If you are using the **old JSP 1.2 descriptor, defined by DTD** ,for example **web.xml**

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app> //...
</web-app>

The EL is disabled or ignored by default, you have to enable it manually, so that it will outputs the value store in the “message” model.

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %**>**
<html>
<head> <%@ page isELIgnored="false" %> </head>
<body> ${message} **</body> <**
/html>

web.xml

JSP 2.0

If you are using the **standard JSP 2.0 descriptor, defined by w3c schema** ,for example

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"> /
/...
</web-app>
```

The EL is enabled by default in standard JSP 2.0

HelloWorld.java : Controller class

```
@RequestMapping(value="/hello1", method=RequestMethod.GET)
public String sayHello(Model model){
    String message="Hello World! Welcome to Spring Framework";
    model.addAttribute("message",message);
    return "helloworld";
}
.....
```

@RequestParam annotation which indicates that a method parameter should be bound to a web request parameter.

Optional Elements	
Modifier and Type	Optional Element and Description
String	**defaultValue** The default value to use as a fallback when the request parameter value is not provided or empty.
boolean	**Required** Whether the parameter is required.
String	**Value** The name of the request parameter to bind to.

HelloWorld.java : Controller

```
@RequestMapping( value="/hello2", method=RequestMethod.GET)
public ModelAndView sayHello(@RequestParam(value = "name", required = false,
    defaultValue = "World") String name){
String message="Hello World! Welcome to Spring Framework";
ModelAndView modelAndView= new ModelAndView();
modelAndView.setViewName("hello");
modelAndView.addObject("message", message);
modelAndView.addObject("name", name);
return modelAndView;
}
```

This method returns ModelAndView object

http://localhost:8080/WebAppDemo/hello2?name=Srini

This class is a holder for both Model and View in the web MVC framework.

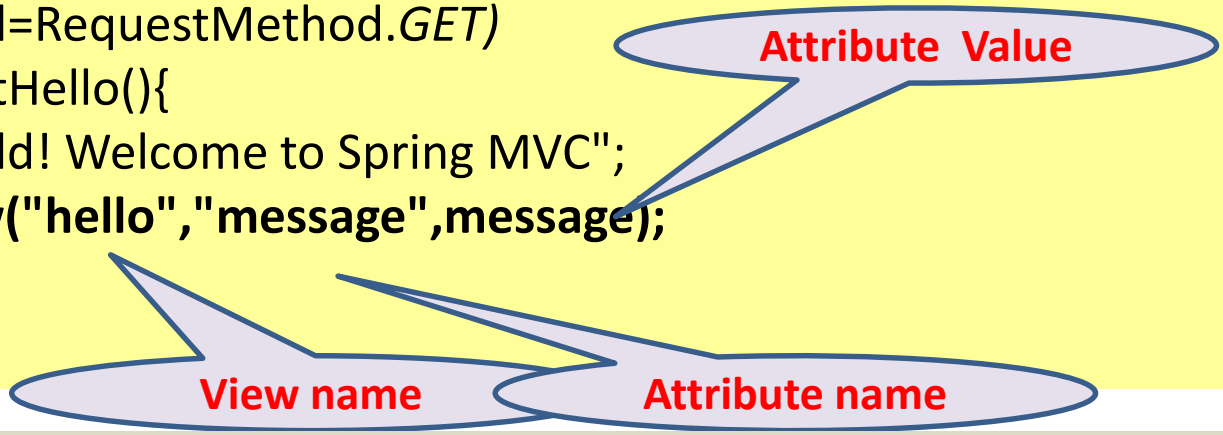
Note that Model and View are entirely distinct. This class merely holds both to make it possible for a controller to return both model and view in a single return value.

```
@RequestMapping(value="/hello1", method=RequestMethod.GET)
public String sayHello(Model model){
    String message="Hello World! Welcome to Spring Framework";
    model.addAttribute("message",message);
    return "helloworld";
}
```

This method returns String object which is name of the jsp file

ModelAndView class

```
@RequestMapping(method=RequestMethod.GET)
public ModelAndView printHello(){
String message="Hello World! Welcome to Spring MVC";
return new ModelAndView("hello","message",message);
}
}
```



mvc-dispatcher-servlet.xml

```
<?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: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">

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

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

helloworld.jsp and hello.jsp

```
<%@ page language="java"
contentType="text/html; charset=ISO-
8859-1" pageEncoding="ISO-8859-1"
isELIgnored="false"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD
HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h2> ${message}</h2>
</body>
</html>
```

```
<%@ page language="java"
contentType="text/html; charset=ISO-
8859-1"
    pageEncoding="ISO-8859-1"
isELIgnored="false"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD
HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h2>
${message}<br>
${name}
</h2>
</body>
</html>
```

Thank You!