

## Table of Contents

<b>CHAPTER 1 .....</b>	<b>1-1</b>
<b>INTRODUCING SPRING WEB MVC.....</b>	<b>1-1</b>
CHAPTER OVERVIEW .....	1-2
JAVA WEB APPLICATIONS .....	1-3
SPRING WEB MVC ARCHITECTURE .....	1-7
SPRING WEB MVC HELLO WORLD.....	1-16
CHAPTER SUMMARY.....	1-32
<b>CHAPTER 2 .....</b>	<b>2-1</b>
<b>SPRING WEB MVC CONFIGURATION .....</b>	<b>2-1</b>
CHAPTER OVERVIEW .....	2-2
CONTEXT CONFIGURATION LOCATION .....	2-3
CONTEXT LOADER.....	2-6
MORE VIEW RESOLVERS .....	2-10
BEANNAMEVIEWRESOLVER.....	2-14
XMLVIEWRESOLVER.....	2-16
RESOURCEBUNDLEVIEWRESOLVER .....	2-17
MULTIPLE VIEW RESOLVERS.....	2-21
CHAPTER SUMMARY.....	2-22
<b>CHAPTER 3 .....</b>	<b>3-1</b>
<b>MVC CONTROLLERS.....</b>	<b>3-1</b>
CHAPTER OVERVIEW .....	3-2
STEREOTYPE CONTROLLERS .....	3-3
REQUEST MAPPING BY ANNOTATION .....	3-4
PATH VARIABLES.....	3-7
@REQUESTPARAM .....	3-9
REQUEST/RESPONSE ANNOTATIONS .....	3-10
HANDLER METHOD PARAMETERS.....	3-12
HANDLER METHOD RETURN TYPES .....	3-14
REDIRECT/FORWARDS .....	3-15
COMMAND BEANS AND WORKING WITH HTML FORMS.....	3-17
BINDINGRESULT AND ERRORS .....	3-22
@INITBINDER .....	3-24
@MODELATTRIBUTE .....	3-26
@SESSIONATTRIBUTES.....	3-29
EXCEPTION HANDLING.....	3-31
ASYNCHRONOUS REQUEST PROCESSING .....	3-34
CHAPTER SUMMARY.....	3-37
<b>CHAPTER 4 .....</b>	<b>4-1</b>
<b>VALIDATION .....</b>	<b>4-1</b>
CHAPTER OVERVIEW .....	4-2
VALIDATION .....	4-3

VALIDATOR .....	4-4
JSR-303 VALIDATION .....	4-10
CHAPTER SUMMARY .....	4-14
<b>CHAPTER 5 .....</b>	<b>5-1</b>
<b>SPRING WEB MVC VIEWS .....</b>	<b>5-1</b>
CHAPTER OVERVIEW .....	5-2
SPRING WEB MVC VIEW .....	5-3
SPRING FORM TAG LIBRARY .....	5-5
BIND FORM DATA .....	5-6
EXTERNALIZED MESSAGES .....	5-14
ERROR MESSAGES .....	5-21
NON-TEMPLATE VIEWS .....	5-24
EXCEL VIEW .....	5-27
PDF VIEW .....	5-32
CHAPTER SUMMARY .....	5-35
<b>CHAPTER 6 .....</b>	<b>6-1</b>
<b>FORMATTING .....</b>	<b>6-1</b>
CHAPTER OVERVIEW .....	6-2
FORMATTING ANNOTATIONS .....	6-3
CUSTOM FORMATTING .....	6-9
CHAPTER SUMMARY .....	6-14
<b>APPENDIX A .....</b>	<b>A-1</b>
<b>SPRING WEB FLOW .....</b>	<b>A-1</b>
CHAPTER OVERVIEW .....	A-2
WEB FLOW .....	A-3
FLOWS .....	A-5
WEB FLOW SETUP .....	A-7
DEFINING FLOWS .....	A-12
VIEW EVENTS .....	A-18
VARIABLES .....	A-20
ACTIONS .....	A-23
ACTION AND DECISION STATES .....	A-28
VALIDATION .....	A-31
MORE WITH WEB FLOW .....	A-34
CHAPTER SUMMARY .....	A-35
<b>APPENDIX B .....</b>	<b>B-1</b>
<b>SPRING SECURITY .....</b>	<b>B-1</b>
CHAPTER OVERVIEW .....	B-2
SPRING SECURITY .....	B-3
GETTING AND CONFIGURING SS .....	B-5
WEB/HTTP SECURITY – SECURITY FILTER CHAIN .....	B-8
NAMESPACE WEB SECURITY CONFIGURATION .....	B-9
JAVA CONFIGURATION WEB SECURITY CONFIGURATION .....	B-13

ALTERNATE AUTHENTICATION PROVIDERS .....	B-16
METHOD SECURITY .....	B-19
CHAPTER SUMMARY.....	B-21
<b>APPENDIX C.....</b>	<b>C-1</b>
<b>SPRING WEB MVC TESTING.....</b>	<b>C-1</b>
CHAPTER OVERVIEW .....	C-2
SPRING MVC TEST FRAMEWORK.....	C-3
TEST SETUP .....	C-5
CREATING CONTROLLER TESTS .....	C-6
CHAPTER SUMMARY.....	C-8

Do Not Print

## Chapter 1

### Introducing Spring Web MVC

#### *Objectives:*

- Understand the complexity of Web application development and the need for a framework.
- Learn what a Web MVC framework is and how it helps in the development of applications.
- Learn about the many Java EE Web MVC frameworks.
- Explore Spring's Web MVC Framework.
- See how to set up and configure an application to use the Spring Web MVC Framework.
- Examine the components of a Spring Web MVC application.
- See a simple Spring Web MVC example application to understand the purpose of the Spring Web MVC components.

## Chapter Overview

The Spring Web MVC module provides a Web MVC application framework. Like other Java MVC frameworks, it provides state management, workflow, validation, etc. In addition, like other Web MVC frameworks, it is based primarily on existing servlet, JSP and JSP tag technology.

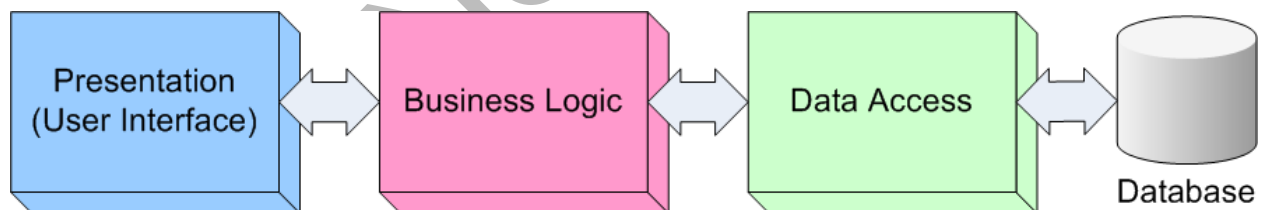
Unlike other frameworks, Spring uses JavaBeans and DI to create and assemble the Web MVC environment. The Spring Web MVC Framework is modular, so various components within the framework can be easily replaced.

Do Not Print

## Java Web Applications

---

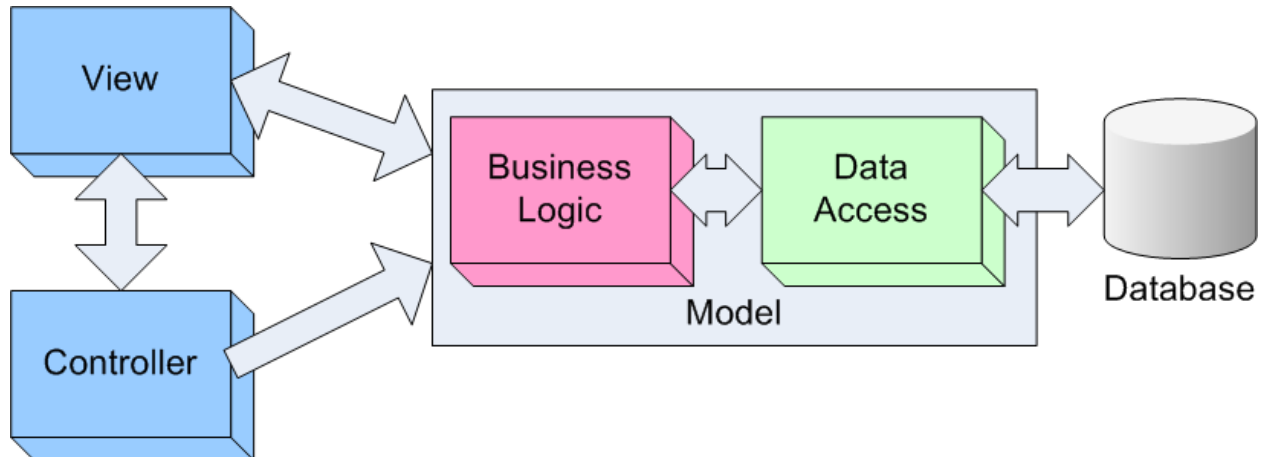
- **Building Web applications can be a difficult task.**
  - All information exchanged in HTML over HTTP is in the form of Strings.
  - This form of data exchange requires a lot of data conversion and is subject to many data validation issues.
  - The server side is usually required to have extensive knowledge of the client side UI in order to extract data from the key/value pairs in the request object.
  - HTTP is generally stateless, so tracking a user's actions and information over many requests and responses requires state management activities – usually via session.
  - Working with HTML can be tedious and error-prone, and it can be difficult to create graphically pleasing user interfaces.
  - Web site navigation and workflow can be difficult to orchestrate and even harder to change/adapt to new business needs.
- **Most software engineers soon learn that it is important to divide any application (Web or otherwise) into separate layers.**
  - Specifically, most significant applications are divided into presentation (or user interface), business or domain logic, and data access layers.



- When implemented correctly, applications divided in this manner allow each layer to be significantly modified, or even replaced, without affecting the others.
- For example, the data access layer can be changed to get data from a different source without causing the domain logic or UI to change.
- Loose coupling between the layers provides for flexibility and ease of maintenance.

- **The Model View Controller (MVC) pattern carries this idea further.**

- In MVC applications, the presentation layer is further divided into view and controller layers.
- In an MVC application, the business or domain logic and data access layers are seen collectively as the model.



- The model encapsulates the raw data and business logic that operate on that data. The model should also notify observers when it has changed.
- The controller responds to events, typically user actions, and instructs the model and the view to perform actions based on the events.
- The controller may invoke changes on the model.
- The view renders information supplied by the model in a form suitable for user interaction. Multiple views may exist for different displays of the same model.
- The view may also serve to capture events to send to the controller.

- **In the Java EE arena, several frameworks provide assistance to developers in implementing Web MVC applications.**
  - While servlets, JSPs, JSP tag libraries, and JavaBeans provide the raw ingredients for developing Web MVC applications, there is still a lot of work to do.
  - A framework provides a lot of the application infrastructure, or “plumbing,” allowing developers to concentrate on business code, not infrastructure code.
  - For example, a framework often provides infrastructure code for handling data validation.
  - A framework also helps to formalize the use of the Java Web technologies, making the application easier to develop and maintain.
  - For instance, frameworks often dictate how servlets, acting as controllers, react to events and call on the next view to be displayed.
  - Most frameworks also provide common out-of-the-box services and functionality that Web developers often need.
  - These include, but are not limited to: data conversion, internationalization/localization, populating form fields, etc.
- **The table below lists some of the more popular Web MVC frameworks in Java.**

Web MVC Frameworks
Spring Web MVC
Struts
JavaServer Faces
Cocoon
Tapestry



- **In this class, you will explore the Spring Web MVC Framework.**
- **However, as you have probably come to expect, Spring can integrate with “best of breed” solutions in all aspects of application development.**
  - In the Web MVC world, Spring holds to this premise.
  - While not covered in this class, the Spring Framework integrates to several popular MVC frameworks like Struts 2 and JSF.
  - In addition, Spring Web MVC can be used with several popular templating and ancillary technologies like Velocity, FreeMarker, and Tiles.
  - A Spring Web MVC Portlet Framework is also provided for those who need to build Web portals.
  - The Spring Web MVC Portlet Framework mimics the architecture and style of the Spring Web MVC Framework.
- **In addition to the modules and dependencies used by other parts of your Spring application, you need the Spring Web and Servlet modules.**
  - Specifically, you need to add the spring-webmvc dependency to your Maven pom.xml file.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>
```

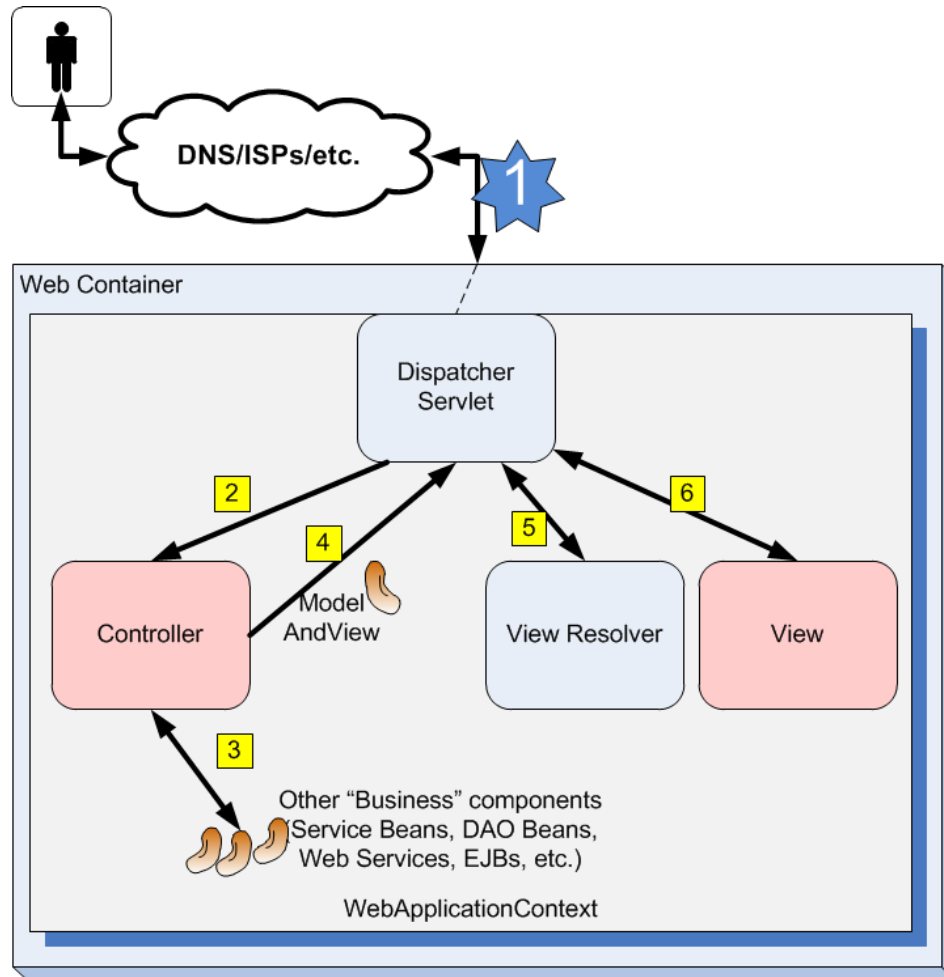
- The spring-webmvc is dependent on the spring-web and spring-context (including spring-beans and spring-core) modules.
- These are automatically included in your Spring Web MVC project by including spring-webmvc in your pom.xml.

## Spring Web MVC Architecture

---

- **In the Spring Web MVC world, there are many components to orchestrate each request/response to the user.**
  - Each component has a specific task.
  - While this might seem to add complexity, it allows for maximum flexibility and customization.
  - Each component can be configured or re-engineered for specific application needs.
- **In a Spring Web MVC world, just as in all Web environments, Web applications are about processing a client's (typically a browser's) Web request.**
  - Web requests carry information about what the user wants in the form of a request URL.
  - The URL of the request is used by the ISPs to eventually get the request to the Web container containing your Spring application.

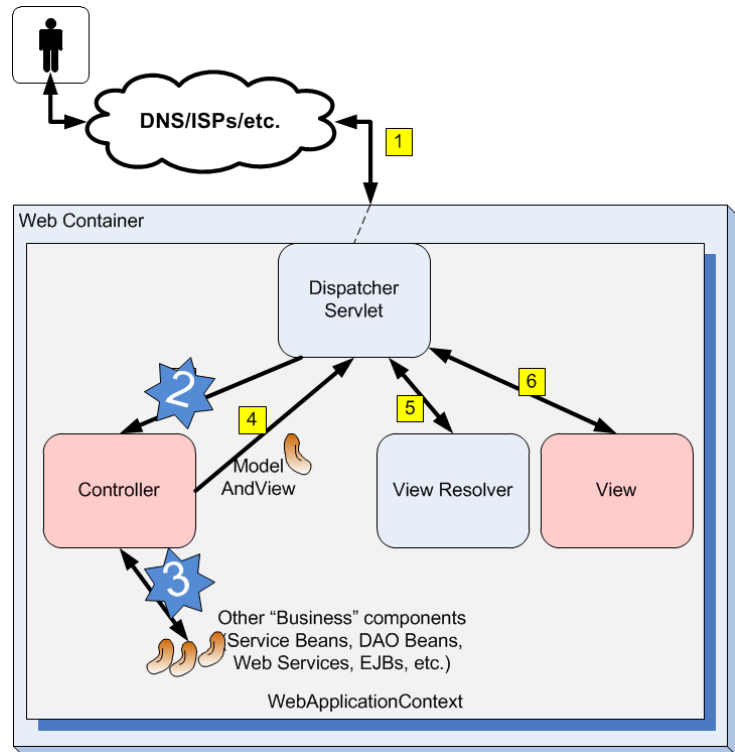
- **Once the container has the request, it channels the request to the Spring Web MVC Framework-provided DispatcherServlet. (See #1 in the diagram below.)**



- There is at least one DispatcherServlet in a Spring Web MVC application to handle requests.
- The DispatcherServlet's job is to marshal requests to other components and to orchestrate a response back to the client.
- Spring's DispatcherServlet is the "front controller" of the MVC environment. A servlet serves as the front controller in many Web MVC frameworks.
- The DispatcherServlet is synonymous with the ActionServlet or FacesServlet in Struts and JSF environments.

- **The task of actually processing the request and calling on various business logic components is the job of a *controller* component in Spring Web MVC.**
  - The controller is the real “C” in this MVC implementation.
  - In most Web applications, there are going to be several controller components. Each is dedicated to specific requests and actions by the user.
  - Controllers are designated as such with @Controller stereotype component annotations.
  - The DispatcherServlet’s first job is to get the right requests to the right controller.
- **The DispatcherServlet does not innately have information about which controller to call for a request.**
  - Annotations on the controller inform the Spring Web MVC environment about which requests are mapped to the controller.
  - Specifically, the @RequestMapping annotation informs Spring how to map URL requests to a controller or even a specific method on the controller.

- Based on the mapping annotations, the DispatcherServlet calls on the appropriate controller component. (See #2 in the diagram.)

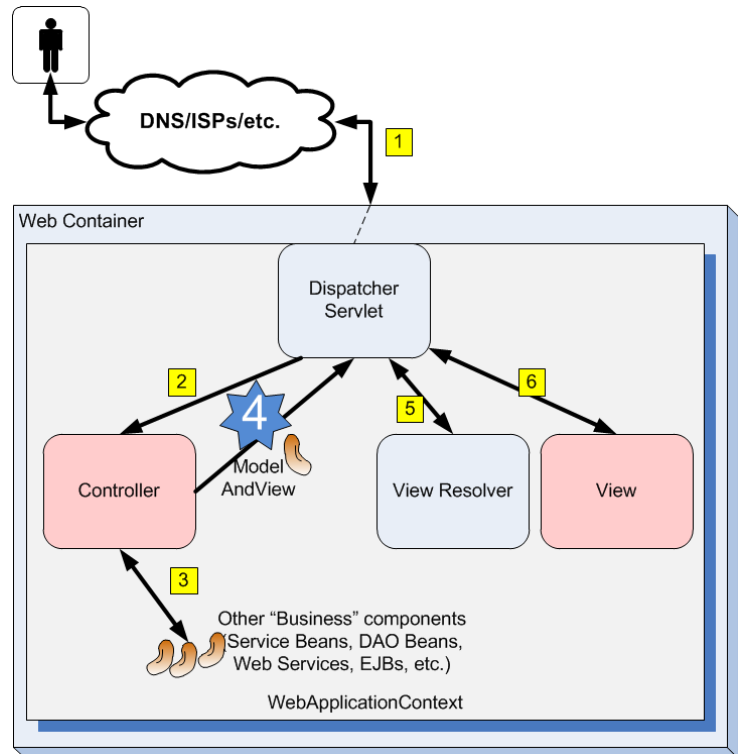


- The controller is responsible for taking the user's request data, processing it, and providing the DispatcherServlet with data and the **view** to show the user next.
- In most cases, the controller merely calls to other business and/or backend components. (See #3 in the diagram above.)
- As a Web developer, you must write the controller.
- Controllers are akin to Strut's Action classes or JSF's managed beans.
- Controller components are not thread-safe and may react to multiple requests coming from the DispatcherServlet.
- Therefore, good controllers quickly call to other service components and get ready for the next request.

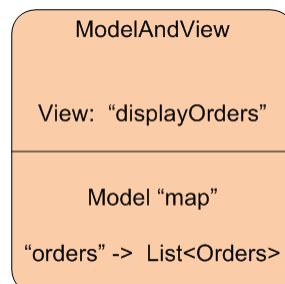
- **When its job is complete, the controller informs the `DispatcherServlet` which view to show next and what data should go on that display.**
  - Typically, the next view is comprised of HTML created from a JSP.
  - However, the next view may be a PDF document, Excel spreadsheet or other format of response.
  - The data to be displayed is ***model*** information obtained via the backend services.
  - Model information can be many things, depending on the view.
  - For example, it might be a collection of order objects to be shown in the display of open orders or the customer object of the customer record to be edited.
  - It may simply be a piece of text to display to the user.

Do Not Print

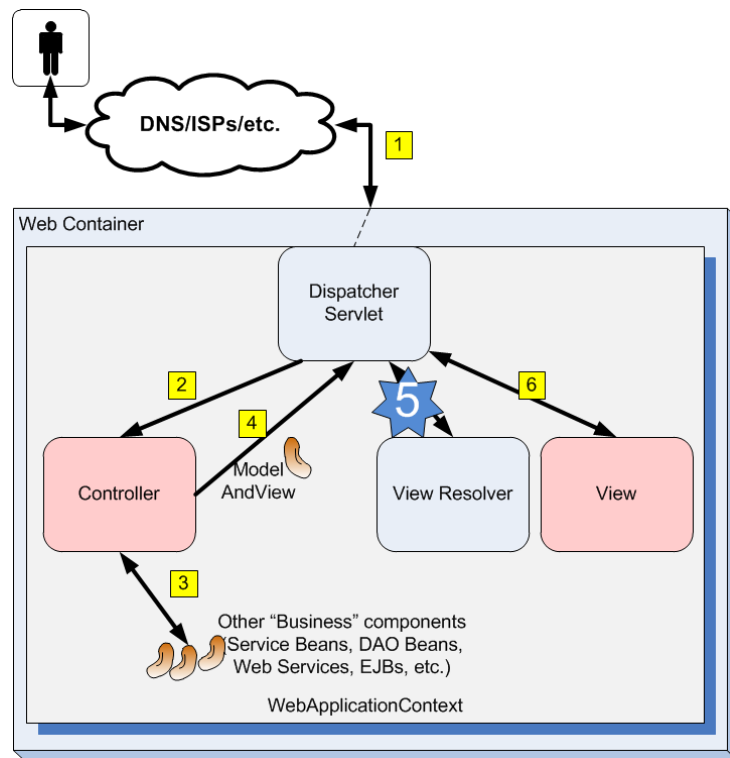
- **Spring Web MVC provides a special object to communicate both the next view and model data to the DispatcherServlet.**
- It is called, appropriately, the ModelAndView object. (See #4 in the diagram.)



- The “view” half of the ModelAndView object is a String that is the logical name of the next view to be displayed.
- The view name is not the actual or physical name of the next view. (For example, it is not the name of a JSP.)
- The view name as a logical name (not physical name) allows the logical name to be resolved dynamically.
- The “model” half of the ModelAndView is a key-value pair. The model information is given a name or key by which it can be referred to in the view.
- The actual data or model information to be used in the next view is the value part of the key-value pair in the “model” half.



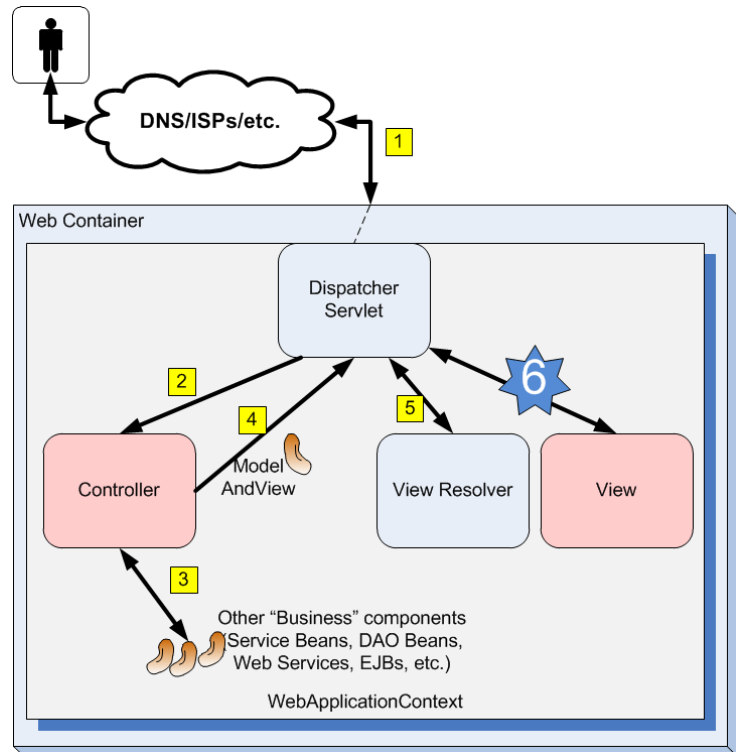
- **The ModelAndView object that the DispatcherServlet receives from the controller contains the logical name of the next view.**
  - The DispatcherServlet does not know how to resolve this information into a physical view to be obtained and displayed back to the user.
  - It turns to a ViewResolver component to resolve the logical view name in the ModelAndView into a physical view name. (See #5 in the diagram.)



- **The ViewResolver is a component that is again provided by Spring.**
  - In fact, there are several types of view resolvers from which to choose.
  - Developers do not have to write the ViewResolver, but they often have to configure the ViewResolver in the Spring configuration file.
  - The configuration information helps the ViewResolver determine the physical view name.
  - The ViewResolver (and its configuration) allows the physical view to be resolved dynamically.
  - This allows the controller code and logical name it returns to remain static while allowing the display workflow to change more dynamically.
  - For example, the "displayOrders" logical name may today resolve to the displayorders.jsp, but tomorrow resolve to the advertisement.jsp.



- **Lastly, with the physical view name and model information in hand, the DispatcherServlet can call to render the view and respond to the user (#6).**



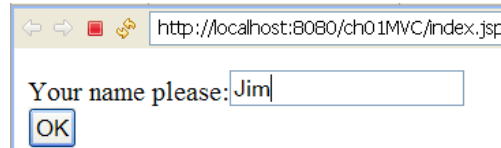
- In the view layer, Spring calls on view beans. View beans know how to create a response. Sometimes the view beans need the assistance of a view template.
- Templates are documents. These documents use some type of markup language to provide instructions on how/what to display.
- JSPs are considered templated views. However, there are a number of template technologies.
- Velocity and FreeMarker are two other template view technologies.
- In some cases, view beans may render without the aid of a template. These view beans help render "non-templated" views.
- The view bean may create and respond with a MIME type like plain text, an Excel spreadsheet, PDF document, etc.

- **When JSP templates are used to provide the view, a special set of JSP tags can help create the view.**
  - Spring Web MVC has a couple of JSP tag libraries to assist developers create template views.
  - In particular, these tags can be used to bind model data to HTML form views and vice versa (not unlike what Struts and JSF offer).
- **Of course, the term “lastly” means the last step in any single HTTP request/response cycle of a Web application.**
  - Once the view is shipped back and displayed on the browser, the whole process starts again with the next user action and request of the Web site.
  - To accomplish something as simple as online shopping and checkout might require tens or hundreds of such requests/responses.

Do Not Print

## Spring Web MVC Hello World

- **To begin your study of Spring Web MVC, a simple and ubiquitous Hello World application is used to demonstrate the necessary components.**
  - It should give you an appreciation of the Spring Web MVC components and how they work together to manage Web application requests/responses.
  - In this example, a simple JSP (index.jsp) requests a user's name.



- After entering his or her name and selecting “OK,” (causing a submit request of the HTML form), the Web application replies to the user with an appropriate greeting.



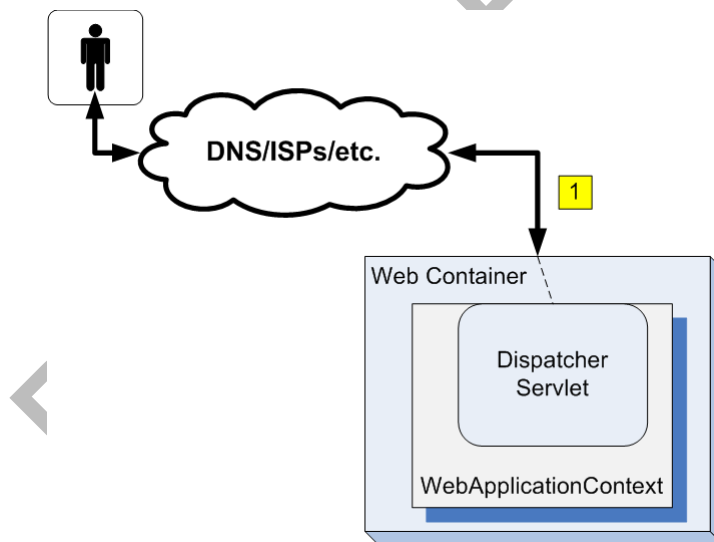
- **The index.jsp page that captures the user's input and sends the initial request into the Spring Web MVC application is shown below.**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!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>Greeting Request</title>
</head>
<body>
    <form action="greeting.request" method="post">
        Your name please:<input type="text" name="name" /><br />
        <input type="submit" value="OK" /><br />
    </form>
</body>
</html>
```

- There isn't anything particularly Spring Web MVC-special here. In fact, it is just a good old-fashioned HTML form.
- Take particular note, however, that the form's action is “greeting.request”.
- This means that when the user fills out the form and selects submit, the request URL sent into the server is “http://server:port/app context/greeting.request”.

- **Who/what receives the request? The Spring Web MVC DispatcherServlet must be configured to get this request.**
  - The DispatcherServlet is provided by the Spring Web MVC Framework. It must be configured as any servlet is.
  - That is, the DispatcherServlet must be configured and its request mappings defined in the web.xml file.
  - At a minimum, <servlet> and <servlet-mapping> elements must be added to the web.xml file (typically located in the WebContent/WEB-INF folder).

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.request</url-pattern>
</servlet-mapping>
```



- **In Servlet 3.0 (or better environments), the web.xml file is no longer required.**
  - Under the Servlet 3.0 API, servlet registration and mappings are accomplished by annotations.
  - Therefore, as of Spring 3.1, developers can configure the DispatcherServlet programmatically and avoid having to add any elements to web.xml.
  - Developers need to implement the `WebApplicationInitializer` interface provided by Spring Web MVC.
  - The implementation of this interface registers the DispatcherServlet with the Web container and specifies its mapping.
  - The implementation below is equivalent to the web.xml additions in the section above.

```
public class WebMVCApplicationInitializer implements
    WebApplicationInitializer {

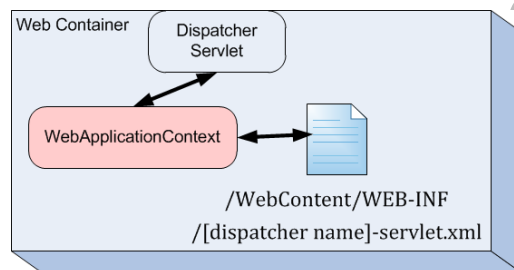
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration =
            container.addServlet("dispatcher", new DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("*.request");
    }
}
```

- When a custom `WebApplicationContext` is required, it is recommended that the Spring container be created in the initializer and attached to the Web container.

```
public class WebMVCApplicationInitializer implements
    WebApplicationInitializer {

    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext =
        new XmlWebApplicationContext();
        ServletRegistration.Dynamic registration =
            container.addServlet("dispatcher",
                new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("*.request");
    }
}
```

- **The name chosen for the servlet is arbitrary. However, it can impact the name of the Spring configuration file that is used by default.**
  - By default, when the DispatcherServlet is loaded, a Spring application context container is started.
  - More precisely, each DispatcherServlet has its own XmlWebApplicationContext container.
  - XmlWebApplicationContext is an implementation of the WebApplicationContext interface.
  - The WebApplicationContext interface is an extension of ApplicationContext.
  - A WebApplicationContext has some extra features necessary for web applications (such as resolving themes).



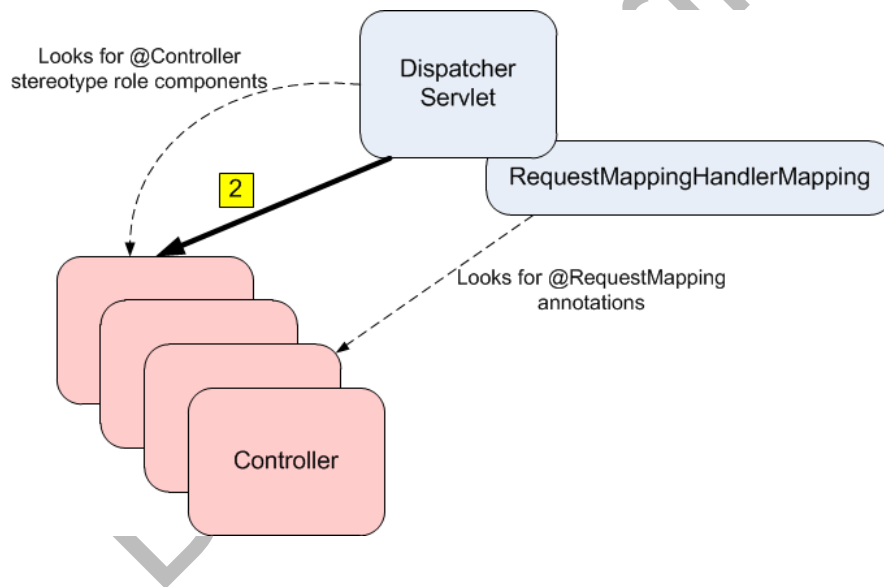
- **By default, the WebApplicationContext is loaded with beans defined in an XML file called [servlet-name]-servlet.xml found in the WebContent/WEB-INF folder.**
  - In this example, that would mean all Spring beans are defined in a /WebContent/WEB-INF/dispatcher-servlet.xml file.
  - Later, you will learn how to modify this default configuration.
- **It should be noted that the WebApplicationContext container is bound in the ServletContext.**
  - This allows the WebApplicationContext (the bean factory) to be accessed anywhere in the Web application.
  - While available through the ServletContext, a special Spring utility class (RequestContextUtils) allows the bean factory to be easily retrieved from anywhere.

```
//use any HttpServletRequest object to retrieve the bean factory.
WebApplicationContext factory =
RequestContextUtils.getWebApplicationContext(request);
Object o = factory.getBean("greetingService");
```

- **The url-pattern is also arbitrary, however, \*.request is somewhat common in Spring Web MVC development circles.**
  - The url-pattern is used by the Web container to route all matching request URL traffic to the Spring Web MVC DispatcherServlet.
  - Developers who want to really obfuscate the MVC engine that runs the application have chosen \*.htm as the url-pattern.
- **With requests now routed to the DispatcherServlet, the DispatcherServlet looks to route traffic to controllers.**
  - Controllers are designated with @Controller stereotype annotations.
  - Recall that you must include a component-scan element (shown below) in the Spring configuration file for Spring to automatically detect the stereotype components.

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

- **The DispatcherServlet scans known Controller stereotypes for @RequestMapping annotations.**
  - The @RequestMapping annotations map request URLs to the controller and/or controller method.
  - You will learn more about controllers and mappings in the next chapter.
- **Before Spring 2.5, an MVC component called the Handler Mapping was used in place of @RequestMapping annotations to map requests to controllers.**
  - It can still be used today, but annotations are preferred.
  - In fact, as of Spring 3.1, the DispatcherServlet works under the covers with a RequestMappingHandlerMapping object.
  - It is actually the default-annotation-based handler mapping object that looks for @RequestMapping annotations and determines which controller to call.





- **Here is a very simple controller for the Hello World MVC application.**

```
import javax.servlet.http.HttpServletRequest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import com.intertech.service.HelloService;

@Controller
public class HelloWorldController {

    @Autowired
    private HelloService service;

    public void setService(HelloService service) {
        this.service = service;
    }

    @RequestMapping("/greeting.request")
    protected ModelAndView sayHello(HttpServletRequest request) {
        return new ModelAndView("greet", "message",
            service.getGreetingMessage(request.getParameter("name")));
    }
}
```

- Again, the @Controller annotation designates this class as a controller, which the DispatcherServlet detects.
- The @RequestMapping tells the DispatcherServlet what requests to route to the sayHello method on the controller.
- **Methods like sayHello are called handler methods in Spring Web MVC. They “handle” incoming requests.**
  - In this case, when a URL of “greeting.request” comes in, it is routed to the sayHello( ) method of an instance of the HelloWorldController.
  - Notice the “/greeting.request” parameter matches the request URL coming from the submit request on the index.jsp page!
  - You may also note that the sayHello( ) method is passed the standard HttpServletRequest object.
  - The request object is used here to retrieve the user’s name (supplied via the HTML form) through a call to request.getParameter( ).
  - Spring detects the parameter in the handler method and automatically provides the needed object – in this case the request object.

- **Because the controller starts the business model work, it is often dependency-injected with service, DAO, and other beans that assist in request processing.**
  - In this example, the controller is dependency-injected with a HelloService bean.
  - Below is the HelloService bean implementation.

```
package com.intertech.service;

import org.springframework.stereotype.Service;

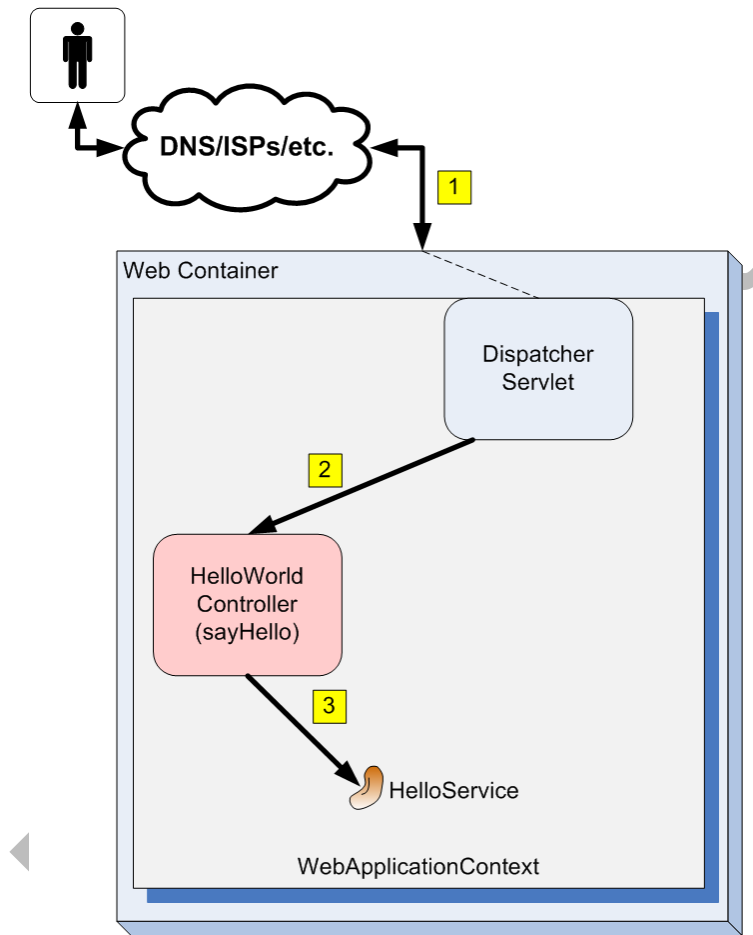
@Service
public class HelloService {

    private String greetingMessage = "Greetings and Salutations";

    public String getGreetingMessage(String name) {
        return greetingMessage + " " + name;
    }
}
```

- Note the @Service stereotype annotation in the code above.
- As is typical, controller objects often communicate with service role objects to get business logic performed.

- **Now you have seen how traffic is routed into business logic with the help of Spring Web MVC components (steps #1-3).**
  - What's next?
  - How is the response formulated and relayed back to the user?
  - How is model data shown on that response?



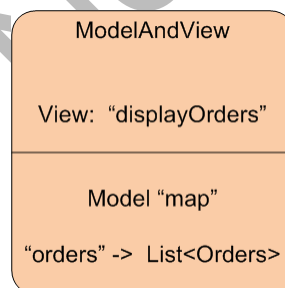
- In this example, the controller's request handling method (**sayHello**) returns a **ModelAndView** object.

```
@RequestMapping("/greeting.request")
protected ModelAndView sayHello(HttpServletRequest request) {
    return new ModelAndView("greet", "message",
        service.getGreetingMessage(request.getParameter("name")));
}
```

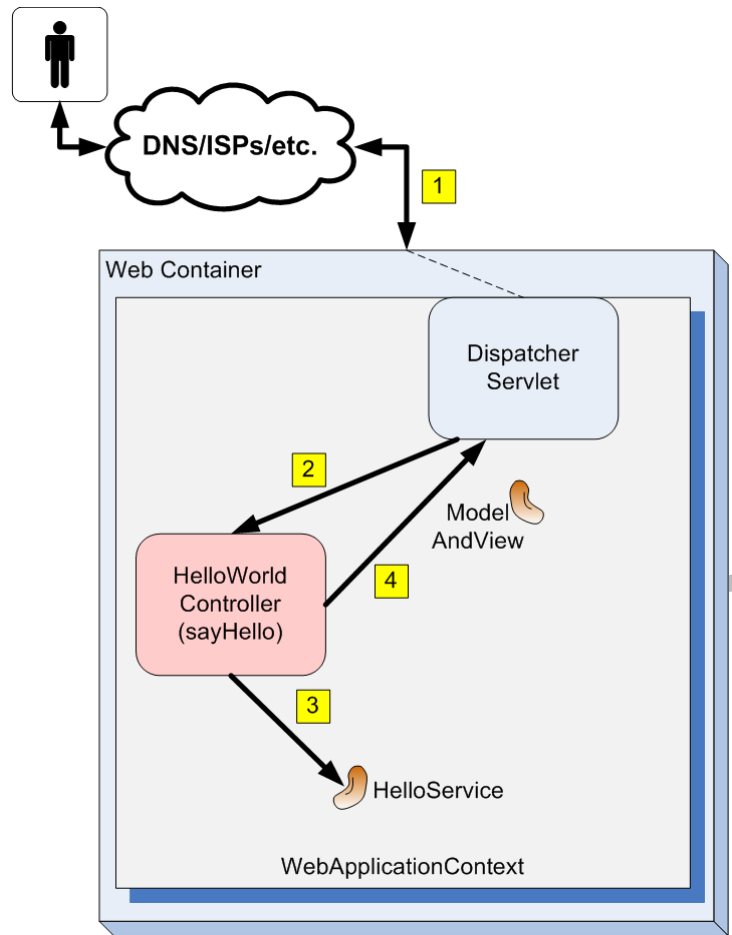
- There are several ModelAndView constructors. However, a common constructor that is used in this example takes as its parameters the following:

#	ModelAndView constructor parameters
1	Logical view name
2	Model name or key
3	Model value

- The “view” half of the ModelAndView object is a String that is the logical name of the next view to be displayed.
- In this case, the logical name of the next view is “greet.”
- The “model” half of the ModelAndView is a map of key-value pairs of the actual data or model information to be used in the next view.



- In this example case, the data to be displayed includes only one key-value pair.
- The key to the key/value pair is "message."
- The value to the “message” key is whatever is returned from the call to `service.getGreetingMessage(request.getParameter("name"))`.

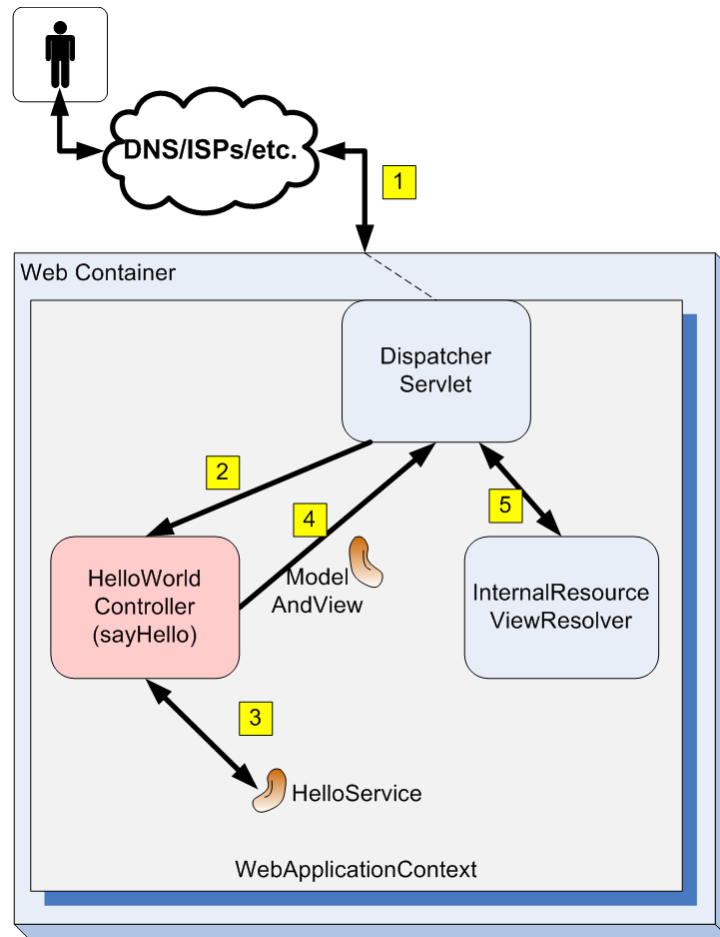


- **So, the Controller returns a ModelAndView object to the DispatcherServlet. Again, the servlet must turn to an assisting component for direction.**
  - A ViewResolver assists the DispatcherServlet in figuring out which view to show to the user next based on the “view” information in the ModelAndView object.
  - View resolvers are also beans. These beans must implement the `org.springframework.web.servlet.ViewResolver` interface.
  - Spring comes with several ViewResolver implementations. This allows you to simply use and configure a view resolver rather than build your own.
  - The `InternalResourceViewResolver` is used in this example.
  - The `InternalResourceViewResolver` takes the logical view name and attaches a “prefix” and “suffix” to create the physical view name.
  - Prefix and suffix are provided in Spring configuration of the `InternalResourceViewResolver` bean.

```
<bean class=
  "org.springframework.web.servlet.view.InternalResourceViewResolver">
  <!-- prefix not really required here, but shown for example sake-->
  <property name="prefix" value="" />
  <property name="suffix" value=".jsp" />
</bean>
```

- In this example, it takes the logical view name and adds “” to the front of the logical name and appends “.jsp” to the end of the logical name.
- Recall in the controller that the logical name returned in the ModelAndView object is “greet.”
- Therefore, the physical name that this view resolver creates based on that logical name would be “greet.jsp.”
- The prefix here is blank (and therefore could be left out).
- If the JSP pages were located in a sub folder of WebContent, the prefix could be used to specify the sub folder.
- For example, if all pages were stored in `/WebContent/hello/jsps`, then the prefix might be configured as shown below.

```
<property name="prefix" value="/hello/jsps/" />
```



- **The DispatcherServlet, with the assistance of the view resolver, now knows the name of the actual view to display next.**
  - So, the last step in handling this single user request is to create the view using the JSP template and use the model data to populate the view appropriately.
  - Spring uses an InternalResourceView bean to create views from JSP templates – in this case the greet.jsp template.
  - Below is greet.jsp.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!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>Greeting</title>
</head>
<body>
    <H1>${message}</H1>
</body>
</html>
```

- JSP expression language can and often is used to extract and display the model data from the ModelAndView object supplied by the controller.
- In this case, recall that the controller added the greeting message string to the ModelAndView object under the key name of "message."

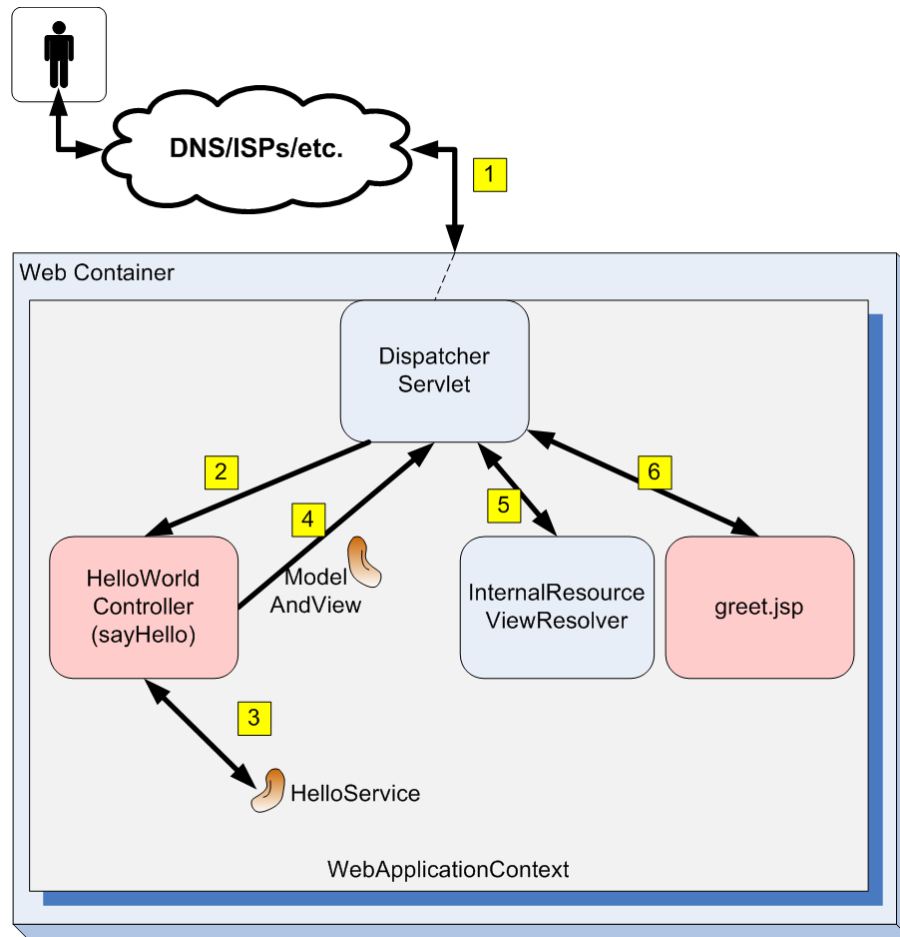
```
return new ModelAndView("greet", "message",
    service.getGreetingMessage(request.getParameter("name")));
```

- In the JSP, expression language uses the model key to extract this data and display it back to the user.



- In more complex applications, the model may be much more complex and require more sophisticated processing (sometimes involving JSTL or other tags) to display.





## Lab Exercise – MVC Intro Lab

## Chapter Summary

---

- **Building Web applications can be a difficult task.**
  - Most software engineers soon learn that it is important to divide any application (Web or otherwise) into separate layers.
  - The Model View Controller (MVC) pattern carries this idea further.
  - In MVC applications, the presentation layer is further divided into view and controller layers.
  - In the Java EE arena, several frameworks assist developers in implementing MVC Web applications.
  - In the Spring Web MVC world, there are many components to orchestrate each request/response to the user.
  - Each component has a specific task. While this might seem to add complexity, it allows for maximum flexibility and customization.
- **The DispatcherServlet's job is to marshal requests to other components and to orchestrate a response back to the client.**
  - The task of actually processing the request and calling on various business logic components is the job of a controller component in Spring Web MVC.
  - The DispatcherServlet looks to route traffic to controllers.
  - Controllers are designated with @Controller stereotype annotations.
  - The DispatcherServlet scans known Controller stereotypes for @RequestMapping annotations.
  - The @RequestMapping annotations map request URLs to the controller and/or controller method.
  - The controller is responsible for taking the user's request data, processing it, and providing the DispatcherServlet with data and the *view* to show the user next.
  - Spring Web MVC provides a special object called the ModelAndView object to communicate the next view and model data to the DispatcherServlet.
  - The DispatcherServlet turns to a ViewResolver component to resolve the logical view name in the ModelAndView into a physical view name.
  - Lastly, with the physical view name and model information in hand, the DispatcherServlet can respond to the user with the next view (and data).

## Chapter 2

### Spring Web MVC Configuration

#### *Objectives:*

- Explore Spring configuration loading.
- Examine how to load multiple configuration files.
- Learn how to create and load additional DispatcherServlets.
- Achieve a better understanding of the view resolver role.
- See additional Spring Web MVC-provided view resolvers.
- Learn how to use multiple view resolvers.

## Chapter Overview

As an application gets large or needs to scale, it may be necessary to break the Spring bean configuration file up and use multiple `DispatcherServlet`s. In this chapter, you will learn how to configure Spring Web MVC.

You will also learn about the many view resolver types Spring Web MVC provides. While you can build your own, many application needs can be satisfied with the Spring Web MVC-provided view resolver classes.

Do Not Print

## Context Configuration Location

---

- **By default, the application context (the Spring bean factory and container in a Spring Web MVC application) is loaded from a single XML file.**
  - The name of the default XML configuration file is in the form of [DispatcherServlet name]-servlet.xml.
  - In most cases, however, developers want to A - name the XML configuration file(s) differently, and B – split the XML Spring configuration file into multiple files.
  - A single XML configuration file becomes too large and unmanageable in a real world application.
- **To address these issues, the XML configuration file names can be explicitly specified in the setup of the DispatcherServlet in the web.xml.**
  - Add a <init-param> element to <servlet> to override the default name and location of the XML Spring configuration file.
  - In fact, you can use the parameter to specify multiple configuration files to be loaded.
  - In the example below, the default /WEB-INF/dispatcher-servlet.xml file name is overridden with /WEB-INF/mvc-beans.xml and /WEB-INF/biz-beans.xml.

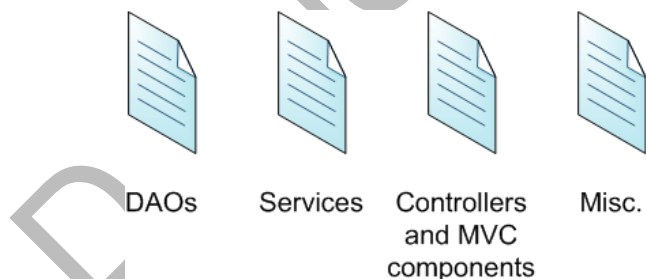
```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/mvc-beans.xml
      /WEB-INF/biz-beans.xml</param-value>
    </init-param>
</servlet>
```

- **If you are using a `WebApplicationInitializer` to configure the `DispatcherServlet` without `web.xml`, you can implement the same affect with the code below.**

```
public class WebMVCAApplicationInitializer implements
WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext =
            new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/mvc-beans.xml,
            /WEB-INF/biz-beans.xml");
        ServletRegistration.Dynamic registration = container.addServlet(
            "dispatcher", new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}
```

- **While there are no rules about how the Spring beans have to be split among multiple XML files, there are a couple of common patterns you can follow.**
  - The most popular means of breaking up the configuration file is to split the application context across application layers.
  - That is, put all DAO beans in one XML file, put all service beans in another file, etc.



- This division allows any layer of the application to be more easily refactored or even replaced with little or no impact to the others.

- **An alternative is to divide the XML configuration by domain. Place all beans related to the processing of a particular type of business object in a single file.**
  - For example, place all DAOs, service beans, and UI components having to do anything with customer domain objects in a single XML file.



Orders   Customers   Shipments   Invoicing

- Applications are rarely developed in one big collective release. Applications are more often built one layer of business functionality at a time.
- This organization helps to support an application that is really just several smaller applications stitched together over time.

Do Not Print

## Context Loader

---

- **It is possible and often preferable to have multiple DispatcherServlets in a single Web application.**
  - Multiple DispatcherServlets allow the request load of a large application to be shared among several request processing “front controller” servlets.
  - For example, all order requests might be handled by one DispatcherServlet while all shipment status requests are handled by a second DispatcherServlet.
- **Each DispatcherServlet has its own associated WebApplicationContext (Spring container).**
  - It is likely that the MVC components for each need to be configured separately.
  - However, the WebApplicationContexts likely share middle-tier beans such as services, DAOs, etc.
- **Spring provides a context loader to assist in this endeavor.**
  - A context loader, as its name implies, loads additional application context configuration files.
  - Spring provides a context loader called the ContextLoaderListener that implements the Java EE ServletContextListener interface.
  - A ServletContextListener object is meant to receive notifications that the initialization process (and destruction) of the Web application is starting.
  - In the case of a ContextLoadListener, it loads application context files on the initialization of the Web application.
  - To configure and use the Spring provided context loader, you must register it in the web.xml file.

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



- **By default, the context loader looks for a Spring configuration file called `applicationContext.xml` in the `WEB-INF` folder.**
  - As in the configuration of the `DispatcherServlet`, the default configuration file can be overridden.
  - To override the default configuration file the context loader uses, specify the `contextConfigLocation` parameter when configuring the context loader.
  - In the `web.xml` file, specify a `<context-param>` element and list the configuration files to be loaded via the `contextConfigLocation` parameter.

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/dao-beans.xml
    /WEB-INF/service-beans.xml
  </param-value>
</context-param>
```

- The beans in these files are loaded into each of the `WebApplicationContext` containers.
- That is, these beans are loaded in addition to the beans specified by the Spring configuration of the `DispatcherServlet`.
- This allows common beans, such as DAOs, services, etc. to be loaded by a context loader and then shared by multiple `WebApplicationContext` containers.

- **To accomplish the same thing without XML, again use a `WebApplicationInitializer` implementation and programmatically attach a `ContextLoaderListener`.**

```
import javax.servlet.ServletContext;
import javax.servlet.ServletRegistration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.
    XmlWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class WebMVCApplicationInitializer implements
WebApplicationInitializer {

    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext =
            new XmlWebApplicationContext();
        appContext
            .setConfigLocation("/WEB-INF/mvc-beans.xml",
                "/WEB-INF/biz-beans.xml");
        ContextLoaderListener listener = new ContextLoaderListener();
        container.addListener(listener);
        container.setInitParameter("contextConfigLocation",
            "/WEB-INF/dao-beans.xml, /WEB-INF/service-beans.xml");
        ServletRegistration.Dynamic registration = container.addServlet(
            "dispatcher", new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}
```

- Typically, the MVC components are in a separate configuration file and loaded by the `DispatcherServlet`.
  - A single context loader can then load the rest of the beans for the multiple `DispatcherServlet`'s `WebApplicationContainers`.
  - Examine the `web.xml` file below to see how MVC and common beans are loaded by the `DispatcherServlet` and context loader respectively.

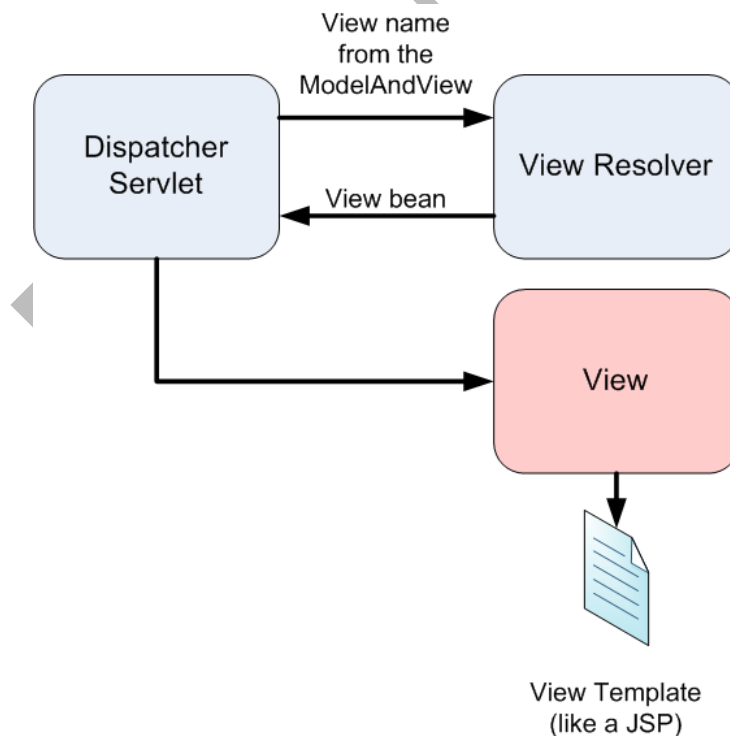
```
<servlet>
  <servlet-name>orderDispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/orderMVC-beans.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>orderDispatcher</servlet-name>
  <url-pattern>*.orderRequest</url-pattern>
</servlet-mapping>
<servlet>
  <servlet-name>shipmentDispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/shipmentMVC-beans.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>shipmentDispatcher</servlet-name>
  <url-pattern>*.shipmentRequest</url-pattern>
</servlet-mapping>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/dao-beans.xml
    /WEB-INF/service-beans.xml
  </param-value>
</context-param>
```

## More View Resolvers

---

- **Controllers in Spring Web MVC typically return a logical view name to indicate the next view to display to the DispatcherServlet.**
  - In the last chapter, the HelloWorldController returned a ModelAndView object containing a logical name to indicate the next view.
  - You also saw how an InternalResourceViewResolver helped the DispatcherServlet resolve the view name in the ModelAndView into a physical view.
- **The ViewResolver actually resolves the logical view name to a View bean (an object implementing the org.springframework.web.servlet.View interface).**
  - The View bean is what actually renders the results to the user.
  - There are all sorts of View beans.
  - However, the most pre-dominate type of View beans for the Web delegate rendering work to a template in the Web application context.
  - JSPs are the most common form of templates in Java Web applications. However, there are also Velocity, Tiles, FreeMarker and other view templates.



- Also, the View bean may not delegate the rendering work at all.
- For example, there are view beans that produce PDF documents or Excel spreadsheets.

- **Spring Web MVC offers a variety of view resolvers.**

- Below are some other view resolvers.

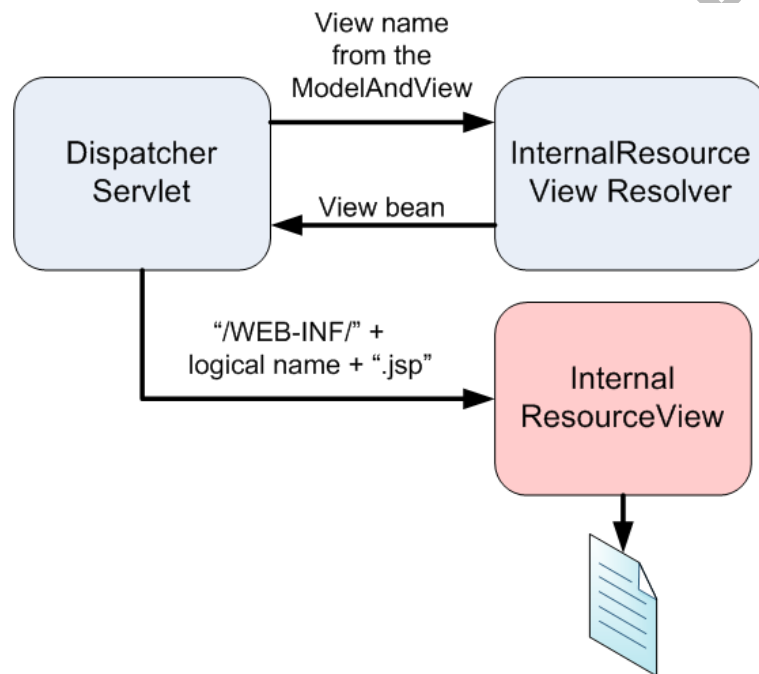
View Resolver Class	Description
InternalResourceViewResolver	Maps logical view names to view beans that delegate to templates identified by the concatenation of a prefix, suffix and the logical view name strings.
BeanNameViewResolver	Maps a logical view name to a bean with the same name in the Spring configuration file (application context).
XmlViewResolver	Maps logical view names to beans defined in a separate Spring XML configuration file.
ResourceBundleViewResolver	Maps logical view names to views defined in a resource bundle.
ContentNegotiatingViewResolver	Delegates to other view resolvers to resolve a view based on the request file name or Accept header. Often used in RESTful Web services applications. See Intertech Blog for example. <sup>1</sup>
VelocityViewResolver	Supports Velocity templates.
FreeMarkerViewResolver	Supports FreeMarker templates.

- View resolvers implement the `org.springframework.web.servlet.ViewResolver` interface.
- **Again, recall the `InternalResourceViewResolver` takes the logical view name and uses a “prefix” and “suffix” property to create the physical view name.**

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

<sup>1</sup> <http://www.intertech.com/Blog/springs-contentnegotiatingviewresolver/>

- **InternalResourceViewResolvers, by default, resolve to a Spring Web MVC provided InternalResourceView object (an object that implements the View interface).**
  - This InternalResourceView object delegates to a JSP template located in the Web application context.
  - The JSP is what actually renders the view.
  - The name of the JSP template is derived from the logical view name and the view resolver's properties.
  - In this example, it takes the logical view name and adds “/WEB-INF” to the front of the logical name and appends “.jsp” to the end of the logical name.



/WEB-INF/[logical view name].jsp

- A side note about the example above: It's actually a good idea to put JSP files under the WEB-INF folder in some applications.
- This hides the JSPs from direct access (manually entered URLs) and allows only framework components to access them.

- **The InternalResourceViewResolver can be configured to use an alternate View bean.**
  - To use an alternate View bean, specify an additional viewClass property for the InternalResourceViewResolver. (See example below.)
  - For example, if the template (or more precisely a JSP template) uses JSTL tags, you would want to use the JstlView bean.
  - A JstlView bean delegates to JSP views as InternalResourceView objects do, but they also expose JSTL specific attributes.

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

## BeanNameViewResolver

---

- **The BeanNameViewResolver looks for a view bean with the name that matches the logical view name it has been given by the DispatcherServlet.**
  - The bean that the BeanNameViewResolver calls must implement the View interface, and it must render the next view.
  - The view that the bean renders, however, does not have to come from a template like a JSP.
  - In fact, you often use the BeanNameViewResolver to call on view beans that create Excel, PDF, or other such output.
  - More is covered on non-template views later.
- **The BeanNameViewResolver is easy to configure in the Spring configuration file.**
  - It does not need any properties. This is because it automatically matches logical view names to view beans of the same name.
  - The example below shows all that is required to configure a BeanNameViewResolver.

```
<bean class=
    "org.springframework.web.servlet.view.BeanNameViewResolver"/>
```

- **Now assume that a controller returns a ModelAndView object with a logical view name of “boring” as shown below.**

```
return new ModelAndView("boring", "message", "Hello World");
```

- **The BeanNameViewResolver uses the logical view name, in this case “boring,” and looks for a view bean with that name.**
  - So, a view bean must be registered in the Spring configuration file with an id of “boring.”

```
<bean id="boring" class="com.intertech.mvc.BoringResponse"/>
```

- Again, a BoringResponse view bean that the BeanNameViewResolver finds must implement the View interface.



- **Below is an example of a very simple View bean. In this case, the View bean extends `AbstractView`.**
  - `AbstractView` is a Spring provided implementation of the View interface. `AbstractView` provides basic View support and can be easily extended.

```
public class BoringResponse extends AbstractView{  
    protected void renderMergedOutputModel(Map arg0, HttpServletRequest  
    arg1, HttpServletResponse arg2) throws Exception {  
        PrintWriter out = arg2.getWriter();  
        out.println("Hi");  
    }  
}
```

- As shown here, a View object does not have to delegate to a template.
- It can simply create and render its own output.

## XmlViewResolver

---

- **The XmlViewResolver works similarly to the BeanNameViewResolver in that it simply matches logical view names to view beans.**
  - However, the view beans are defined in a separate XML file and not in the Spring configuration file.
  - This allows the view resolution to be isolated from the rest of the application.
- **Like the BeanNameViewResolver, little is required to configure the XmlViewResolver.**
  - Simply register it as a bean in the Spring beans configuration file.

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver" />
```

- By default, the XmlViewResolver looks for the view bean definitions in a /WEB-INF/views.xml file.
- If you wish to override this default and specify the location and/or name of the file, add a location property to the XmlViewResolver.

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="location">
    <value>/WEB-INF/boring-beans.xml</value>
  </property>
</bean>
```

- The configuration file that the XmlViewResolver uses is formatted just as any Spring configuration file.
- Below is an example of what the boring-beans.xml file might look like holding the definition of the BoringResponse view bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  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-2.5.xsd">
  <bean id="boring" class="com.intertech.mvc.BoringResponse" />
</beans>
```

## ResourceBundleViewResolver

---

- **The ResourceBundleViewResolver maps logical view names to view beans and view templates through definitions stored in properties files.**
  - Properties files are also known as resource bundles.
  - Like the BeanNameViewResolver and XmlViewResolver, simply register the ResourceBundleViewResolver in the Spring configuration file.

```
<bean class=
  "org.springframework.web.servlet.view.ResourceBundleViewResolver"/>
```

- By default, the ResourceBundleViewResolver looks for a views.properties file that provides view mapping located on the classpath.
- However, you can provide a basename property to override the default file name.
- In this example, the ResourceBundleViewResolver now expects the view mappings to be defined in a greeting-views.properties file.

```
<bean class=
  "org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename">
    <value>greeting-views</value>
  </property>
</bean>
```

- **Now assume that a controller returns a ModelAndView object with a logical view name of "greet" as shown below.**

```
return new ModelAndView("greet", "message", "Hello World");
```

- **What is in the properties file? Properties files contain key-value pairs where the key and value are separated by “=”.**
  - The properties file must specify the view bean class and optionally the URL for the template if the view bean needs a template.
  - For example, in the case of the logical view name “greet” as shown above, the greeting-views.properties file might contain the following properties.

```
greet.(class)=
    org.springframework.web.servlet.view.InternalResourceView
greet.url=/greet.jsp
```

- This informs Spring to create and use an instance of the InternalResourceView bean to delegate to the greet.jsp template.
- Note that prior to Spring 3, the view bean class was designated without parentheses around class in the properties file.

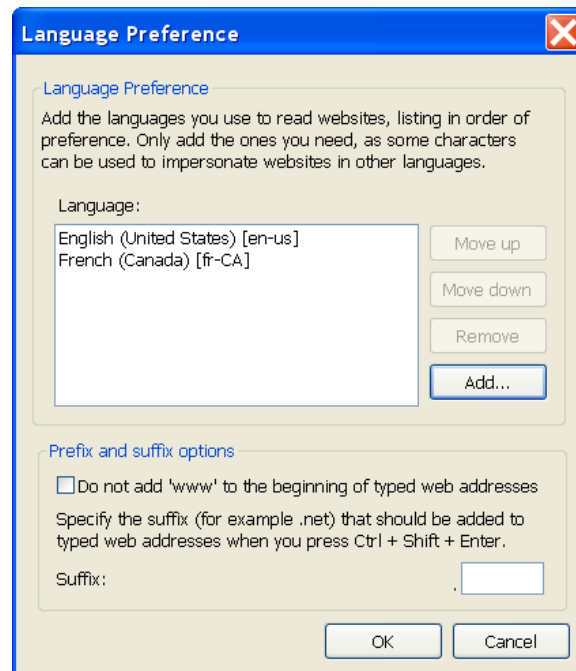
```
greet.class=org.springframework.web.servlet.view.InternalResourceView
```

- **The “logical view name”.class property tells the ResourceBundleViewResolver what View object to create and use.**
  - The View object may or may not need a URL to specify the location of the template to render.
  - In the previous example, the “logical view name”.url property specifies the JSP template that the InternalResourceView resolver used.
  - These two properties basically made the ResourceBundleViewResolver a fancy InternalResourceViewResolver.
- **However, look at the example greeting-views.properties file below. In this case, only a class property is provided.**

```
greet.(class)=com.intertech.mvc.BoringResponse
```

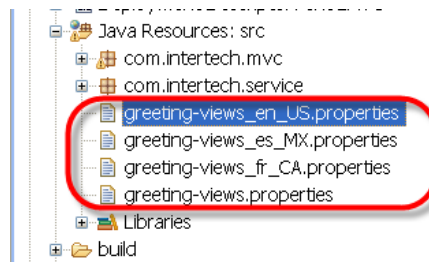
- Since the view bean (an instance of BoringResponse from above) takes care of rendering its own response without a template, no url property is needed.
- In effect, this made the ResourceBundleViewResolver act like a fancy BeanNameViewResolver.

- **Given many key-value pairs in its properties file, the ResourceBundleViewResolver can resolve all types of views.**
  - Sometimes it resolves to view beans that delegate to templates.
  - Sometimes it resolves to view beans that take care of their own rendering.
- **The ResourceBundleViewResolver can be used to help map logical view names to different view beans and templates based on the locale.**
  - The browser can be configured for language/locale preferences.



- This information is transmitted by the browser with each request.
- The ResourceBundleViewResolver can use the locale information supplied in the HTTP request to resolve logical view names from an appropriate property file.

- The “**basename**” property to the **ResourceBundleViewResolver** specifies, as its name implies, the **basename of the properties file**.
  - Locales can be added to the basename to give the **ResourceBundleViewResolver** several properties files from which to choose.
  - The **ResourceBundleViewResolver** uses the properties file that matches the locale passed by the browser.
  - For example, if the basename is **greeting-views**, you might have several **greeting-viewsXX\_XX.properties** files as shown below.



- If the browser passes in en-US (English, US) as the locale, then the first mapping file is used.
- If the browser passes in es-MX (Spanish, Mexico), then the second file is used.
- A properties file without a locale (the default file, such as **greeting-views.properties** here) is used when there is no matching locale file.
- This allows your application to address internationalization and locale issues.
- The **ResourceBundleViewResolver** provides an opportunity for different view beans and templates to be used for different languages and locales.

## Multiple View Resolvers

---

- You can register and use several view resolvers simultaneously.
- Use the **Ordered** interface to set the priority of each of the view resolvers.
  - All of the built-in Spring view resolvers already implement this interface.
  - Dependency inject all view resolvers with an order property.

```
<bean class=
  "org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename">
    <value>greeting-views</value>
  </property>
  <property name="order" value="10" />
</bean>
<bean class=
  "org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="order" value="20" />
  <property name="prefix" value="" />
  <property name="suffix" value=".jsp" />
</bean>
```

- This establishes the order in which view resolvers should be consulted when mapping the logical view names to view beans and views.
- The `InternalResourceViewResolver` should always be given the lowest or last priority.
- This is because the `InternalResourceViewResolver` always attempts to add its prefix and suffix to the logical view name to produce the next view template.



### Lab Exercise – View Resolver Lab

## Chapter Summary

---

- **By default, the application context (the Spring bean factory and container in a Spring Web MVC application) is loaded from a single XML file.**
  - The name of the default XML configuration file is in the form of [DispatcherServlet name]-servlet.xml.
  - In most cases, however, developers want to 1 - name them XML configuration file(s) differently, and 2 – split the XML Spring configuration file in to multiple files.
  - To address these issues, add <init-param> elements to override the default name and location of the XML Spring configuration file.
- **It is possible and often preferable to have multiple DispatcherServlets in a single Web application.**
  - Spring provides a context loader to assist in this endeavor.
  - By default, the context loader looks for a Spring configuration file in the WEB-INF folder called applicationContext.xml.
  - To override the default configuration file the context loader uses, specify the contextConfigLocation parameter when configuring the context loader.
  - The beans in these files are loaded into each of the WebApplicationContext containers.
- **The ViewResolver actually resolves the logical view name to a View bean (an object implementing the org.springframework.web.servlet.View interface).**
  - The View bean is what actually renders the results to the user.
  - The most often-used type of View beans for the Web delegate rendering work to a template in the Web application context.
  - JSPs are the most common form of templates in Java Web applications. However, there are also Velocity, Tiles, FreeMarker and other view templates.
  - Spring provides several types of view resolvers.



## Chapter 3

### MVC Controllers

#### *Objectives:*

- Explore the function and configuration of the Spring Web MVC controller.
- Understand how to designate a bean as controller.
- Learn how to route requests to controller handler methods.
- See how to pass parameters to handler methods.
- Explore handler method return types.
- Learn the role of command beans in form processing.
- Examine exception handling.
- See how Spring supports asynchronous request processing.

## Chapter Overview

While the `DispatcherServlet` is at the heart of Spring Web MVC applications, controllers make up the brains. Controller components are the “C” in this MVC environment. The `@Controller` annotated stereotype beans serve as those “C” components in Spring Web MVC applications.

In this chapter, you will explore how to route request traffic to the controllers and how the controllers return responses through the `DispatcherServlet` and ultimately to the user. You will also look at how model information and other data can be passed into the controller to allow it to do its job – to orchestrate business logic calls and return responses. There are many additional annotations and supporting components in Spring Web MVC that help to configure the controller and help move data between the view and model in the MVC world.

## Stereotype Controllers

---

- **Like other stereotype components, the `@Controller` annotation lets you designate any Spring Web MVC controller as such without configuration.**

```
@Controller
public class HelloWorldController {

    ...

}
```

- `@Controller` was an annotation added in Spring 2.5.
- `@Controller` is defined in the `org.springframework.stereotype` package.
- Unlike the other stereotype annotations, the Spring Web MVC library provides many additional annotations to work with the `@Controller` annotation.
- While not covered here, it should be mentioned that the MVC annotations are available for Portlet MVC environments too.
- Also, controllers are at the heart of Spring RESTful applications.
- RESTful applications are not covered in class, but the Spring Web MVC framework serves as the backbone for Spring RESTful applications.
- **Like other stereotype components, the controller can be explicitly named and scoped.**

```
@Controller("greetingController")
@Scope("prototype")
public class HelloWorldController {

    ...

}
```

## Request Mapping by Annotation

---

- The **@RequestMapping** annotation maps request URLs to a controller.
  - Use this annotation on the controller type itself and on the controller methods.
  - When **@RequestMapping** is used on methods of the controller (method-level annotation), it directs traffic to a specific method within the class.

```
@RequestMapping("/greeting.request")
protected ModelAndView sayHello (HttpServletRequest request)
{
    String name = request.getParameter("name");
    return new ModelAndView("greet", "message", name);
}
```

- When used on a method, a **@RequestMapping** on the class level is not required.
- Without a **@RequestMapping** annotation on the class, all paths are absolute, and not relative.
- In other words, the paths specified on methods are relative to the path specified first at the class level **@RequestMapping**.

```
@RequestMapping("/greeting.request") //absolute
public class HelloWorldController {

    @RequestMapping("/morepath") //relative to /greeting.request
    protected ModelAndView sayHello (HttpServletRequest request)
    {
        ...
    }
}
```

- Absolute paths are relative to the root of the file system – in this case the root directory of the Web application (and its context).

- **When the `@RequestMapping` annotation is used on the controller class (type-level annotation), it directs a pattern of requests to the controller.**
  - As you saw, a parameter to a `@RequestMapping` annotation used on the methods specifies a relative path to the mapping provided at the class level.
  - On the method below, a path parameter to the annotation is used to designate the appropriate method to call for `http://.../greeting.request/mom`.

```
@RequestMapping("/greeting.request")
public class HelloWorldController {

    @RequestMapping("/mom")
    protected ModelAndView sayHiToMom() {
        return new ModelAndView("greet", "message", "Hi Mom!");
    }
}
```

- Specific method-level annotations can also narrow the mapping for a specific HTTP request method ("GET"/"POST").

```
@RequestMapping("/greeting.request")
public class HelloWorldController {

    @Autowired
    private HelloService service;

    public void setService(HelloService service) {
        this.service = service;
    }

    @RequestMapping(method = RequestMethod.GET)
    protected ModelAndView sayHello(HttpServletRequest request) {
        return new ModelAndView("greet", "message",
            service.getGreetingMessage(request.getParameter("name")));
    }

    @RequestMapping(method = RequestMethod.POST)
    protected ModelAndView sayHi() {
        return new ModelAndView("greet", "message", "Hi everyone");
    }
}
```

- Above, the `@RequestMapping` annotation directs all requests for `http://.../greeting.request` to this `HelloWorldController`.
- However, HTTP request method type determines which method to call in the controller.

- Both method path and type can be used simultaneously as shown in this last example.

```
@RequestMapping(value="/dad", method = RequestMethod.GET)
protected ModelAndView sayHiToDad() {
    return new ModelAndView("greet", "message", "Hi Dad!");
}

@RequestMapping(value="/dad", method = RequestMethod.POST)
protected ModelAndView sayHiToDadWithUrgency() {
    return new ModelAndView("greet", "message", "Hi Dad. Send money!");
}
```

- Use path patterns (wildcards) to route multiple URL via the `@RequestMapping` annotation.

```
@RequestMapping(value="/*.request", method=RequestMethod.POST)
protected String doGenericHello (){
    return "greet";
}
```

## Path Variables

---

- **Many applications put variables in the URLs. These are called URI path parameters.**
  - Path parameters are segments of the URL designated by a URI template.
  - For example, a URI template may suggest that path parameters are added to a request URL by name, a slash, and then a parameter value.

```
http://www.example.com/parametername/parametervalue
```

- For example, in the URL below, the path parameter is `userid` and the value of the path parameter is `jwhite`.

```
http://www.intertech.com/userid/jwhite
```

- **The URI template can be specified in the `@RequestMapping` annotation.**

```
@RequestMapping("/greeting.request/welcomemsg/{welcomemsg}")
protected ModelAndView sayHello(@PathVariable("welcomemsg") String message) {
    return new ModelAndView("greet", "message", message);
}
```

- `@PathVariable` annotation binds the path variable to a handler method parameter.
- If the variable name matches the method argument name, you can omit the variable name.

```
@RequestMapping("/greeting.request/welcomemsg/{welcomemsg}")
protected ModelAndView sayHello(@PathVariable String welcomemsg) {
    return new ModelAndView("greet", "message", message);
}
```

- `@PathVariables` are limited to simple types (`int`, `long`, `Date`, `String`, etc.).
- Spring throws a `TypeMismatchException` if the variable cannot be converted to the correct type (as specified by the handler method parameter type).

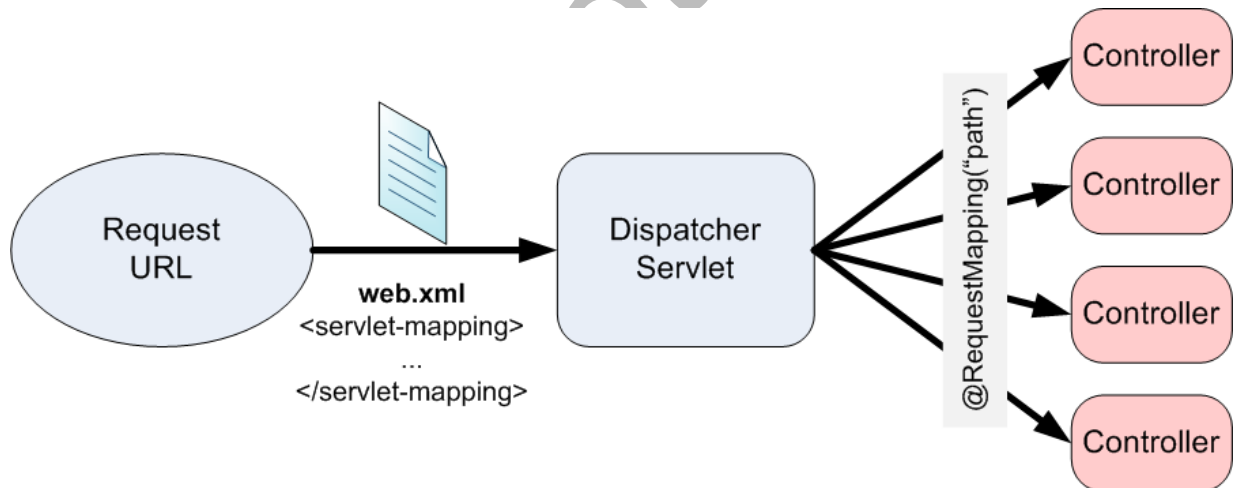
- **Remember the Web container must first be able to route any request to the DispatcherServlet.**
  - Path variables add extra information to the URL that affects routing.
  - Expand the servlet mappings in the web.xml file (or programmatically in the WebApplicationInitializer ) to allow for the additional path variables.
  - In the example below, the mapping allows for any URL.

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

- Had the web.xml only mapped \*.request (as below) to the DispatcherServlet, <servlet-mapping>, request and parameter mapping become immaterial.

```
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.request</url-pattern>
</servlet-mapping>
```

- A request with path variables would not have reached the DispatcherServlet, let alone the handler method under this mapping.



- **The @PathVariable was added to Spring 3.0. Therefore, it is not available in older versions of Spring Web MVC.**



## @RequestParam

---

- The **@RequestParam** annotation can be used on a handler method to bind a request parameter directly to a method parameter.
  - You could code handler methods to get the `HttpServletRequest` object and programmatically fetch the parameter as shown below.

```
@RequestMapping("/greeting.request")
protected ModelAndView sayHello (HttpServletRequest request) throws Exception
{
    String name = request.getParameter("name");
    return new ModelAndView("greet", "message", name);
}
```

- However, the **@RequestParam** annotation accomplishes the same task more directly.

```
@RequestMapping("/greeting.request")
protected ModelAndView sayHello (@RequestParam("name") String name){
    return new ModelAndView("greet", "message", name);
}
```

- By default, parameters annotated with **@RequestParam** are required.
- You can make a parameter optional by setting the required attribute of **@RequestParam** to false.
- A **defaultValue** attribute can also specify a value to use when the parameter is not provided.

```
@RequestMapping("/greeting.request")
protected ModelAndView sayHello (@RequestParam(value="name", required=false,  
    defaultValue="Mysterious") String name){
    return new ModelAndView("greet", "message", "Hello " + name);
}
```

## Request/Response Annotations

---

- With Spring 3.0, a collection of new annotations (listed below) was added to bind request and response elements to handler method parameters.

Spring 3.0 Method Parameter Annotations
@RequestHeader
@RequestBody
@ResponseBody
@CookieValue

- The **@RequestHeader** binds an HTTP request header to a method parameter.
  - Below is a partial list of headers you might find in an HTTP request.

```
Accept-Language: en-us,en;q=0.7,fr;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

- Here is the code to get a couple of the parameters from the header.

```
@RequestMapping("/greeting.request")
protected ModelAndView sayHello (@RequestHeader("Accept-Language") String
lang, @RequestHeader("Accept-Encoding") String encoding){
    if (lang.contains("en-us")) {
        ...
    } else {
        ...
    }
    ...
}
```

- The **@RequestBody** and **@ResponseBody** annotations work in a very similar fashion.
  - Both bind the contents of the HTTP message body (request and response respectively) to a method parameter.

```
@RequestMapping(value = "/greeting.request", method = RequestMethod.PUT)
protected ModelAndView receiveGreetings(@RequestBody String messageBody) {
    return new ModelAndView("greet", "message", messageBody);
}
```

- Note the PUT method type in the request mapping used in this example. Only HTTP PUT and POST messages contain a body.

- **The `@ResponseBody` on a method causes the return type to be written to the HTTP response body.**
  - Handler method return types (like `ModelAndView`, `Model`, and a `String` interpreted as the view name) are covered later in this chapter.
  - Using `@ResponseBody` bypasses these normal return types.
  - Whatever the handler method returns becomes the body of the response message.

```
@RequestMapping("/greeting.request")
@ResponseBody
protected String sayHello (@RequestParam(value="name",
    required=false, defaultValue="Mysterious") String name){
    return "<HTML><B>Hello " + name + "</B></HTML>";
}
```

- **Spring provides a set of message converters that can be used with `@RequestBody` or `@ResponseBody` to convert the body to an appropriate type.**
  - For example, the contents of a message body could be converted to an object and vice versa.
- **Finally, the `@CookieValue` annotation allows a method parameter to be bound to the value of an HTTP cookie.**

```
@RequestMapping("/greeting.request")
protected ModelAndView sayHello(@CookieValue("JSESSIONID") String sessionID)
{
    ...
}
```

## Handler Method Parameters

---

- **Handler methods (methods with the `@RequestMapping` annotation) can be passed arguments.**
  - Pass any argument from the table below to a handler method without further annotation in the controller.

Parameter Type	Description
BindingResult	Spring Web MVC BindingResult object (org.springframework.validation.BindingResult)
Command object	Spring Web MVC defined Command objects
Errors	Spring Web MVC Errors object (org.springframework.validation.Errors)
HttpEntity<?>	Spring-provided object containing the entire HTTP request or response in object/entity form with methods to access headers and body elements
InputStream/Reader	Servlet API-provided raw InputStream/Reader (java.io.InputStream or Reader)
Locale	Request locale (java.util.Locale)
Map	Spring model as java.util.Map
Model	Spring model as org.springframework.ui.Model
ModelMap	Spring model as org.springframework.ui.ModelMap
OutputStream/Writer	Servlet API-provided raw OutputStream/Writer (java.io.OutputStream or Writer)
Principal	Java security principal containing the currently authenticated user
Request	Servlet API Request object (use either HttpServletRequest or ServletRequest)
Response	Servlet API Response object (use either HttpServletResponse or ServletResponse)
RedirectAttributes	A Spring Web MVC class to specify attributes of a redirect (see below)
Session	Servlet API Session object
SessionStatus	Spring-provided handle for marking when form processing is finished (org.springframework.web.bind.support.SessionStatus)
UriComponentsBuilder	A Spring Web MVC builder for preparing a URL object relative to the current request's host, port, scheme, and context path
WebRequest	Spring-provided objects for parameter and attribute access without ties to the Servlet API (use org.springframework.web.context.request.WebRequest or NativeWebRequest)

- **As seen in previous examples, the Spring container automatically detects the needed parameter and passes the appropriate object to the handler method.**

```
@Controller
public class HelloWorldController {
    @RequestMapping("/greeting.request")
    protected String doGenericHello (HttpServletRequest req){
        String name = req.getParameter("name");
        return "greet";
    }
}
```

- **The order of some parameters is important.**
  - Most parameters to handler methods can be used in any order.
  - However, Errors and BindingResult parameters must follow the command or model object if used. (More on this point in a bit.)
- **As a cautionary note: When creating the handler methods, consider some of the ramifications in using some parameter types.**
  - Using parameters like Model or HttpServletRequest couples the controller to Spring, the Servlet API, or both.
  - Spring always espouses loose coupling – even to Spring itself.
  - As you use components from Spring Web MVC or the Servlet API, you couple that component to a particular technology/API.
  - This renders the component obsolete if technology changes. For example, if you decide to swap Spring Web MVC for JSF.
  - It also reduces the ability for the component to be used in non MVC/Web environments.

## Handler Method Return Types

---

- You have already seen handler methods return **ModelAndView** objects.
- However, handler methods in stereotype controllers can return many types of data.
  - Data returned can be used as the model, view or other such Spring Web MVC component, depending on its type.
  - The table below describes the types of data that can be returned by a handler method and how that data is used.

Return Data Type	How the return data is used
Callable<?>	The application produces the return value asynchronously in a thread managed by Spring Web MVC (see later in this chapter).
DeferredResult<?>	The application produces a return value from a thread managed on its own. (See later in this chapter.)
HttpEntity<?>	Spring-provided object containing HTTP request or response entity, consisting of headers and body.
ModelAndView	Spring-provided component that designates the next view and the data to display on the next view (org.springframework.web.servlet.ModelAndView).
Model	The Model object (of the ModelAndView) containing key-value pairs of model data (org.springframework.ui.Model).
Map	Key-value pair model data (java.util.Map).
ResponseEntity<?>	Spring-provided object containing HTTP response entity, consisting of headers and body.
View	The View object (of the ModelAndView) containing information on the logical view name (org.springframework.web.servlet.View).
String	Interpreted as the logical view name as would be represented in a View object.
void	Indicates that the handler method handles the response rendering itself.
other	Any other return type gets treated as a single model attribute value accessed by the name specified in @ModelAttribute (if provided) or based on the return type class name (when @ModelAttribute is not used).

- Again, remember that using a Spring type (like ModelAndView) as the return type of a handler method does couple the controller to Spring.

## Redirect/Forwards

---

- **In many cases, the controller returns a logical view name that the view resolver resolves to a particular view technology.**
  - There are times when the view resolver/view capability needs to be bypassed and have the controller issue an HTTP redirect or forward back to the client.
  - The redirect or forward is provided to the client before any view is rendered.
- **There are several reasons for redirecting.**
  - A POST request could have been issued, and the response needs to be delegated to another controller.
  - An internal forward would result in a POST to the second controller when it would ordinarily expect a GET request.
  - Redirects can also help to prevent users from submitting form data multiple times.
- **You can redirect from a controller by returning an instance of Spring's `RedirectView` from the handler method.**

```
@RequestMapping(value = "editcustomer.request", method = RequestMethod.POST)
protected RedirectView updateCustomer(Customer customer, Errors errors) {
    ...
    return new RedirectView("customerdisplay");
}
```

- `RedirectView` (`org.springframework.web.servlet.view`) issues an `HttpServletResponse.sendRedirect( )`.

- **As a better alternative to creating and responding with a RedirectView, use a “redirect:” prefix in the view string returned.**
  - Using RedirectView requires the controller to have awareness that a redirection is happening, which is typically an unwanted tight coupling.
  - The redirect prefix allows the ViewResolver to recognize and handle the redirect need.

```
@RequestMapping(value = "editcustomer.request", method = RequestMethod.POST)
protected String updateCustomer(Customer customer, Errors errors) {
    ...
    return "redirect:/customerdisplay";
}
```

- A redirect string like “redirect:/displayinfo.jsp” redirects the client relative to the current context.
- A redirect string like “redirect:http://www.cnn.com” redirects to an absolute URL.
- **You can also forward using the forward: prefix in view names.**
  - This causes a RequestDispatcher.forward( ) on the rest of the URL after “forward:” to happen under the covers.

```
@RequestMapping(value = "editcustomer.request", method = RequestMethod.POST)
protected String updateCustomer(Customer customer, Errors errors) {
    ...
    return "forward:/customerdisplay";
}
```

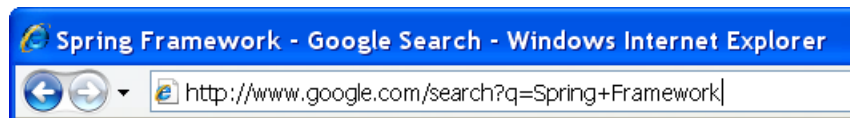
- **A RedirectAttributes can also be used to specify attributes in the redirection or forward.**
  - RedirectAttributes is from the org.springframework.web.servlet.mvc.support package.

```
@RequestMapping(value = "/accounts", method = RequestMethod.POST)
public String handle(RedirectAttributes redirectAttrs) {
    // handle new account request data and save account ...
    redirectAttrs.addAttribute("id", newAcctId).
        addFlashAttribute("message", "New account created!");
    return "redirect:/accounts/{id}";
}
```

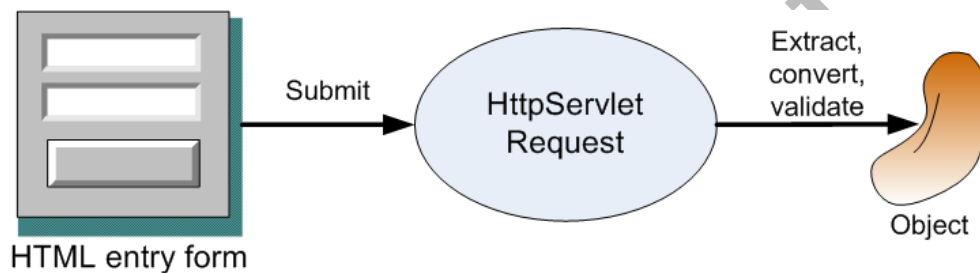


## Command Beans and Working with HTML Forms

- A common activity with regard to Web applications is to collect data from a user and use that data in business processing.
  - The data is often collected via HTML form, but it may just be request parameter data (like the parameters you pass in a Google search).



- On the server, the data must be extracted out of the HttpServletRequest object, converted to a proper data format, loaded into an object, and validated.



- The data, once in an object, can be passed to the business components for appropriate processing.
- Spring can bind HTTP form or request parameter values to JavaBean (or Spring bean) properties.
  - JavaBeans bound to form/request parameters are called **command objects**. They are also known as form-backing objects.
  - Spring does the work of extracting data from the request and populating the command object.
  - Spring can also orchestrate some basic data conversion and validation data as it moves the data from the request parameters to the JavaBean properties.
  - If you provide Spring some rules, validation that is even more complex can be accomplished. You will learn this in a later chapter.

- To understand command beans, assume you had to collect customer information from a user.
  - A simple HTML form page like the one below could be used to collect the data.

```
<!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>Customer</title>
</head>
<body>
<form action="addcustomer.request" method="post">
First name:<input type="text" name="firstName"/>
Last name:<input type="text" name="lastName"/>
Marital status: <input type="text" name="status"/>
Age:<input type="text" name="age"/>
<input type="submit" value="Add"/>
</form>
</body>
</html>
```

First name:  Last name:  Marital status:  Age:

- **To collect the data from this page, create a plain old Java object/JavaBean with properties that match the request parameter names.**
  - In this case, the properties match the HTML form field names.

```
public class Customer {
    private String firstName;
    private String lastName;
    private String status;
    private int age;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

- **When building a controller, map the request to a handler method that takes a Customer parameter.**
  - Spring and the controller automatically bind request parameters to the properties in a Customer object and pass it to the handler method.
  - Again, the request parameter names must match property names of the command object.

```
@Controller
public class CustomerController {

    @RequestMapping("addcustomer.request")
    protected String addCustomer(Customer customer) {
        // save customer, do work with customer
        ...
        return "customerdisplay";
    }
}
```

- Where possible, Spring converts the HTTP request data (always in String form) to the appropriate type of data expected by the command object properties.
- For example, a value of 33 entered into the customer age field in the HTML form is converted to an int for the age property on the command object.
- **Many controllers are set up to fetch and display command objects on GET requests and then update on POST requests.**

```
@Controller
public class CustomerController {

    @Autowired
    private CustomerDao dao;

    @RequestMapping(value="editcustomer.request", method=RequestMethod.GET)
    protected String fetchCustomer(int id, Model model) {
        // use the DAO to fetch the customer with specified id
        ...
        model.addAttribute("customer", customer);
        return "editcustomer";
    }

    @RequestMapping(value="editcustomer.request", method=RequestMethod.POST)
    protected String updateCustomer(Customer customer, Errors errors) {
        // use the DAO to save/update the customer data
        ...
        return "customerdisplay";
    }
}
```

- **This requires the HTML form to know about the command bean and how to fetch and use the command bean's properties to populate form fields.**
  - Below, Unified Expression Language is used in the JSP to fetch and display command bean properties.
  - In a later chapter, you will learn how to connect the form to the bean more directly using Spring Web MVC provided JSP tags.

```
<form action="editcustomer.request" method="post">
First name:<input type="text" name="firstName" value=${customer.firstName}>
Last name:<input type="text" name="lastName" value=${customer.lastName}>
Marital status: <input type="text" name="status" value=${customer.status}>
Age:<input type="text" name="age" value=${customer.age}>
DOB:<input type="text" name="dob" value=${customer.dob}>
<input type="submit" value="Save" />
</form>
```

First name:  Last name:  Marital status:  Age:  DOB:

## BindingResult and Errors

---

- **What if someone enters invalid data into your HTML form?**
  - Binding errors can be created as Spring tries to populate the command object with request data.
  - For example, if a user enters “old” in the age field of the HTML form, Spring Web MVC cannot successfully convert “old” to an int for the Customer command object.

First name:  Last name:  Marital status:  Age:

```
Field error in object 'command' on field 'age': rejected value [old];
codes [typeMismatch.command.age,typeMismatch.age,typeMismatch.int,
typeMismatch]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable:
codes [command.age,age]; arguments []; default message [age]]; default
message [Failed to convert property value of type [java.lang.String]
to required type [int] for property 'age'; nested exception is
java.lang.NumberFormatException: For input string: "old"]
```

- **Spring Web MVC creates a BindingException (org.springframework.validation) as it tries and fails to bind data to command object properties.**
  - A BindingException implements BindingResult, which in turn is a sub-interface of Errors (all from org.springframework.validation).
  - Errors define data-binding and validation errors that occur when binding data to a specific object.
  - The BindingResult interface, as the name would imply, deal only with data-binding errors.

- **Adding a `BindingResult` or `Errors` parameter to the handler method allows you to capture and react to data binding issues.**
  - The `Errors` parameter allows you to capture and react to both data binding and validation issues (more on this later).
  - A `BindingResult` or `Errors` parameter serves as a collector of Spring Web MVC exceptions that occurred at bind time.
  - You need a `BindingResult` or `Errors` parameter for each command object in the parameter list.
  - The `BindingResult` or `Errors` parameter must immediately follow the command object in the parameter list.

```
@RequestMapping("addcustomer.request")
protected String addCustomer(Customer customer, Errors errors) {
    ...
}
```

- **The `Errors` (or `BindingResult`) object allows you to write code in the controller to continue with business operations or take alternate actions.**
  - In fact, the `Errors` (or `BindingResult`) object is potentially a collection of Exceptions.
  - When the `Errors` (or `BindingResult`) collection parameter is empty, binding and validation occurred successfully.
  - When the `Errors` (or `BindingResult`) collection is not empty, then at least one exception was raised during binding.

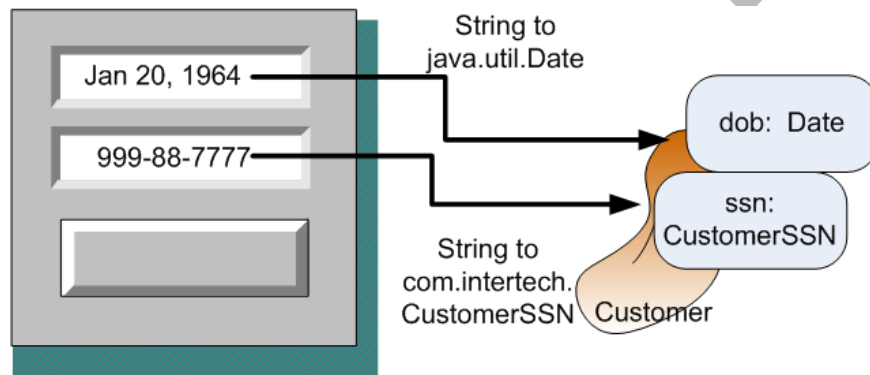
```
@RequestMapping("addcustomer.request")
protected String addCustomer(Customer customer, Errors errors) {
    if (errors.hasErrors()) {
        return "editcustomer";
    }
    else {
        // save customer, do work with customer
        ...
        return "customerdisplay";
    }
}
```

- Alternate action often includes returning the user to the same input form to correct mistakes and retry submitting data.
- Subsequent chapters provide you with some tools to display information about the errors discovered back to users on the HTML form.
- You also learn about how to perform validation as part of the binding process.

## @InitBinder

---

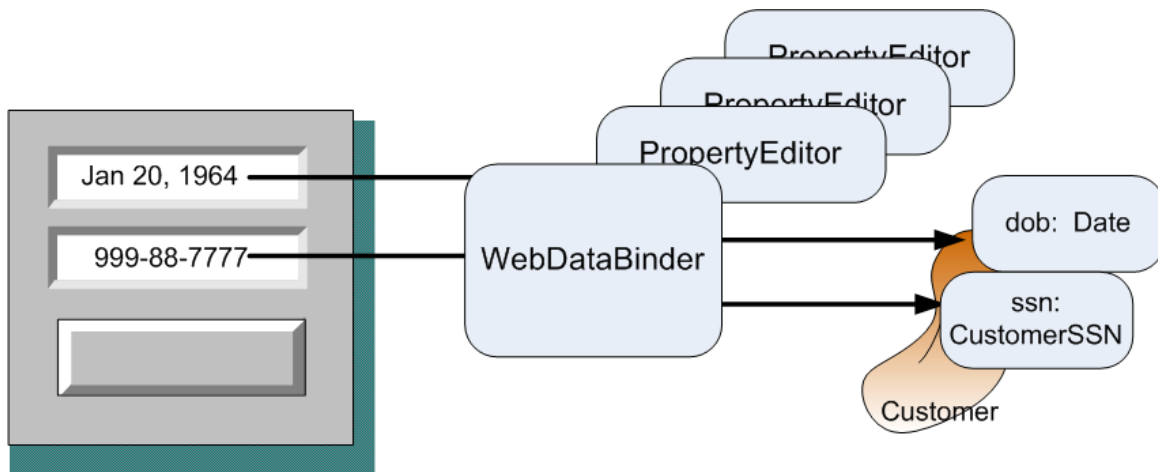
- **Spring Web MVC binds form and parameter data into command beans.**
  - As you have learned, where possible, Spring converts the request data to the appropriate type as expected by the command object properties.
  - In particular, Spring can automatically convert simple types such as primitives (int, long, etc.) and dates.
  - What happens when the command property type is not so simple? For example, what if a command bean property held a CustomerSSN object?
  - What if you wanted dates entered in a different format?



- **You may recall that Spring comes with several predefined property editors.**
  - In the configuration of beans, Spring sends data through the editor before being set in the bean.
  - For example, a `URLEditor` would convert a String containing a URL to a `java.net.URL` object.
  - In a similar fashion, editors can be used to transform request parameter data into a command property.
  - As you learn in another chapter, the same editors can also transform command property data back to Strings.



- Use the **@InitBinder** annotation on a controller method that establishes the editors you want to use in data binding operations.
  - In particular, the @InitBinder method allows you to initialize and register custom editors with a WebDataBinder (org.springframework.web.bind).
  - A WebDataBinder object is used by Spring Web MVC under the covers to populate command objects.



- In order to put the @InitBinder annotation on a controller method, the method must not have a return value.
- The method can take any argument that a handler method takes except for a command object and Errors/BindingResult object.
- Typically, these methods take a WebDataBinder parameter (as shown below).
- This allows the method to add needed editors to the WebDataBinder.

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    DateFormat dateFormat = new SimpleDateFormat("d-MM-yyyy");
    CustomDateEditor dateEditor = new CustomDateEditor(dateFormat, true);
    binder.registerCustomEditor(Date.class, dateEditor);
    SSNEditor ssnEditor = new SSNEditor();
    binder.registerCustomEditor(CustomerSSN.class, ssnEditor);
}
```

- Here, the init-binder method is used to register a couple of editors.
- The custom date editor converts strings in a custom format to Date objects and back.
- The SSN editor (from a previous chapter), converts strings in 999-99-9999 format to CustomerSSN objects and back.

## @ModelAttribute

---

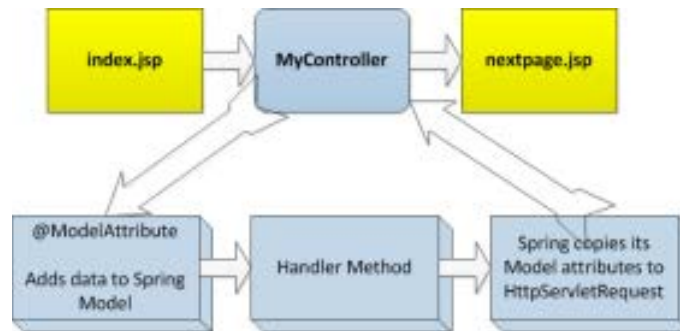
- The **@ModelAttribute** annotation can be used on a handler method to bind a model object's attribute to a method parameter.

```
@Controller
public class DisplayCustomer {

    @RequestMapping("/displaycustomer.request")
    public String show(@ModelAttribute("customer") Customer cust) {
        ...
        return "successfuladd";
    }
}
```

- In this example, the Model map must contain a "customer" key containing a Customer object.
- The **@ModelAttribute** annotation has a second use. It can also be used to put information, like reference data, into the model.
  - When the @ModelAttribute annotation is placed on a method, the method it annotates is executed before the appropriate handler method.
  - These methods add data to a java.util.Map that is added to the Spring model before the execution of the handler method.
  - This allows for pre-populating the model.

- Under the covers, data from the Spring model is stored in the standard Java request (`HttpServletRequest`) scope - eventually.
- That is, data from the model is added to the request object after the execution of the handler and before presentation of the next view.



- Command beans that are created in the controller are also put in the model (and therefore part of the request object as well).
- Consider a simple form entry page such as the HTML code provided below (in this case under the filename `getname.jsp`).

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!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>Greeting Request</title>
</head>
<body>
    <form action="greeting.request" method="post">
        Your name please:<select name="title">
            <option>${titles[0]}</option>
            <option>${titles[1]}</option>
            <option>${titles[2]}</option>
            <option>${titles[3]}</option>
            <option>${titles[4]}</option></select>
            <input type="text" name="name" /><br />
            <input type="submit" value="OK" /><br />
        </form>
    </body>
</html>
```

- The **@ModelAttribute** annotated method in the controller below populates the “titles” options used in the getname.jsp before the page displays.

```
@Controller
public class HelloWorldController {

    @ModelAttribute("titles")
    public List<String> populateTitles() {
        String[] titles = { "Mr.", "Mrs.", "Miss", "Ms.", "Dr." };
        return Arrays.asList(titles);
    }

    @RequestMapping(value="/greeting.request",method=RequestMethod.GET)
    protected String showForm (){
        return "getname";
    }

    @RequestMapping(value="/greeting.request",method=RequestMethod.POST)
    protected String doGenericHello (@RequestParam(value="name")
        String name, @RequestParam(value="title") String title){
        System.out.println("Your name is: " + title + " " + name);
        return "greet";
    }
}
```

- That is because the populateTitles method executes before the showForm method that causes the getname.jsp page to display.
- Again, the information is placed into the model (and then HttpServletRequest object) prior to the display of the getname.jsp above).
- For a deeper explanation, see <http://www.intertech.com/Blog/understanding-spring-mvc-model-and-session-attributes/>.

## @SessionAttributes

---

- The **@SessionAttributes** annotation determines which model attributes should be stored in the session (**HttpSession**).
- The **@SessionAttributes** allows conversational state to be stored between requests (**GET** and **POST** in this case) to a controller.
  - An example can help demonstrate what **@SessionAttributes** does.
  - Assume you have two methods on a controller. One method reacts to a **GET** request. The other method reacts to the **POST** request.

```
@Controller
@RequestMapping("/customer.request")
@SessionAttributes("username")
public class CustomerController {

    ...

    @RequestMapping(method=RequestMethod.GET)
    public String displayForm(Model model){
        String username = getUsername();
        model.addAttribute("username",username);
        ...
        return "greet";
    }

    @RequestMapping(method=RequestMethod.POST)
    public String formSubmitted(@ModelAttribute("username") String username){
        ...
        // do work with username
        ...
        return "somepage";
    }
}
```

- Again, you might see these types of controller methods used with an **HTML** form.
- In this case, the **GET** request puts a **String** object under the name “username” into the controller’s “session.”
- On the form submittal (the **POST** request), the controller’s session attribute allows the **String** to be retrieved from the model.
- Under the covers, **@SessionAttributes** informs **Spring** which of your model attributes is to be copied to **HttpSession** before rendering the view.

- **When the controller session is complete and data in the session is no longer needed, remember to clean up the session.**
  - First, add a `SessionStatus` (`org.springframework.web.bind.support`) parameter to the handler method.
  - Then, call `setComplete()` on the `SessionStatus` object to initiate appropriate cleanup (such as releasing stored object references).

```
@RequestMapping(method=RequestMethod.POST)
public String formSubmitted(@ModelAttribute("username") String username,
    SessionStatus status) {
    ...
    // do work with username
    ...
    status.setComplete();
    return "somepage";
}
```

- This causes the session attribute(s) added by Spring to be removed from `HttpSession` (without having to kill the entire `HttpSession`).

## Exception Handling

---

- **When an exception is raised in a Spring Web MVC application, Spring tries to handle typical exceptions by returning HTTP status code responses by default.**
  - Implementations of the `HandlerExceptionResolvers` interface work behind the scenes to resolve exceptions thrown during handler mapping or execution.
  - Find `HandlerExceptionResolver` in the `org.springframework.web.servlet` package.
  - Instances of `HandlerExceptionResolver` must be registered with Spring in order to handle exceptions.
  - By default, the `DispatcherServlet` registers the `DefaultHandlerExceptionResolver` (`org.springframework.web.servlet.mvc.support`).
  - The table below lists the exceptions this `HandlerExceptionResolver` maps to common HTTP status code responses.

Exception	HTTP Status Code
<code>BindException</code>	400 (Bad Request)
<code>ConversionNotSupportedException</code>	500 (Internal Server Error)
<code>HttpMediaTypeNotAcceptableException</code>	406 (Not Acceptable)
<code>HttpMediaTypeNotSupportedException</code>	415 (Unsupported Media Type)
<code>HttpMessageNotReadableException</code>	400 (Bad Request)
<code>HttpMessageNotWritableException</code>	500 (Internal Server Error)
<code>HttpRequestMethodNotSupportedException</code>	405 (Method Not Allowed)
<code>MethodArgumentsNotValidException</code>	400 (Bad Request)
<code>MissingServletRequestParameterException</code>	400 (Bad Request)
<code>MissingServletRequestPartException</code>	400 (Bad Request)
<code>NoHandlerFoundException</code>	404 (Not Found)
<code>NoSuchRequestHandlingMethodException</code>	404 (Not Found)
<code>TypeMismatchException</code>	400 (Bad Request)

- **You can implement your own `HandlerExceptionResolver`.**
  - The `HandlerExceptionResolver` interface requires you to implement a `resolveException(Request, Response, Handler, Exception)` method.
  - The method returns a `ModelAndView` object.

- For example, suppose your `CustomerController` received a request to edit a customer, but the id provided is not for any known customer.

```
@RequestMapping(value="editcustomer.request", method=RequestMethod.GET)
protected String fetchCustomer(int id, Model model)
    throws BadCustomerIdException{
    // use the DAO to fetch the customer with specified id

    if ((id == null) || (customer == null)) {
        throw new BadCustomerIdException(id);
    }

    model.addAttribute("customer", customer);
    return "editcustomer";
}
```

- Spring looks for a `HandlerExceptionResolver` to call on to help it determine how to resolve the exception request.
- In the example below, a custom `HandlerExceptionResolver` displays a more appropriate display page with information about the incorrect customer id.

```
@Component
public class CustomeHandlerExceptionResolver implements
    HandlerExceptionResolver {

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response, Object arg2, Exception exception) {

        if (exception instanceof BadCustomerIdException) {
            return new ModelAndView("customeridnotfound", "id",
                ((BadCustomerIdException) exception).getId());
        }
        ...
    }
}
```

- You might note the use of the `@Component` annotation.
- As with any component picked up in the component scan, this registers the `HandlerExceptionResolver` with Spring.



- **Spring 3.0 added an annotation as an alternate to the `HandlerExceptionResolver` interface.**
  - The `@ExceptionHandler` designates any method in a controller as the method to call when an exception is thrown from the controller.
  - Returning to the `CustomerController` and the bad customer id example above, the same exception can be handled using the `@ExceptionHandler` annotation below.

```
@ExceptionHandler(BadCustomerIdException.class)
protected ModelAndView badCustomerId(Exception exception) {
    return new ModelAndView("customeridnotfound ", "id",
        ((BadCustomerIdException) exception).getId());
}
```

- Unlike the `resolveException` method of a `HandlerExceptionResolver`, however, the `@ExceptionHandler` annotated methods do not have to return a `ModelAndView`.
- The return type can be a `String` (view name) or `ModelAndView` object.
- In addition, the `@ExceptionHandler` can be set to an array of Exception types.
- This allows the `@ExceptionHandler` annotated method to serve as a handling method for multiple types of exceptions.

## Asynchronous Request Processing

---

- **As of Spring 3.2, as part of its support of the Servlet 3.0 API, the framework supports asynchronous request processing.**
  - Ordinarily, the servlet container (like Tomcat) creates a thread for each request.
  - That thread is occupied while the response is being generated and until the response is sent back to the client.
- **Spring allows the controller handler method to return a Callable that will produce the return value from a separate thread.**
  - This allows the thread that dealt with the incoming request to be released for processing other requests.
  - Meanwhile, Spring automatically invokes the Callable on a separate thread and when the Callable returns, the request is dispatched back to the servlet container.
  - The servlet container resumes processing with the value returned by the Callable.

```
@RequestMapping("/greeting.request")
public Callable<String> sayHello(final Model model) {

    return new Callable<String>() {
        @Override
        public String call() throws Exception {
            Thread.sleep(5000); // doing work here
            return "hello";
        }
    };
}
```

- A Callable is similar to a Runnable object. Like a Runnable, a Callable instance is typically executed on another thread.
- Unlike a Runnable, a Callable can return results and throw checked exceptions.

- In addition to the handler method returning a Callable<?>, you must add an `<mvc:annotation-driven>` element to the container configuration.
- Note that the configuration also specifies the default timeout for any asynchronous processing.

```
<?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/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
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd">

  <mvc:annotation-driven>
    <mvc:async-support default-timeout="10000" />
  </mvc:annotation-driven>
</beans>
```

- The asynchronous request processing feature is only supported in Servlet 3.0+ containers.
- The web.xml configuration and container requires support for asynchronous processing to be turned on.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <async-supported>true</async-supported>
  </servlet>
  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

- **Alternatively, if web.xml is not used for web configuration, set the `asyncSupported` flag to true on the `WebApplicationInitializer`.**

```
@Override
public void onStartup(ServletContext container) {
    XmlWebApplicationContext appContext =
        new XmlWebApplicationContext();
    ServletRegistration.Dynamic registration =
        container.addServlet("dispatcher",
            new DispatcherServlet(appContext));
    registration.setLoadOnStartup(1);
    registration.addMapping("*.request");
    registration.setAsyncSupported(true);
}
```

- **A second option to having the handler method return a `Callable<?>` is to have the controller return a `DeferredResult` instance.**
  - Again, the return value will be produced from a separate thread, but one not known to the Spring Web MVC framework.
  - A `DeferredResult` can be passed off to a separate thread that could be executed from anywhere (and unknown to Spring).
  - This is helpful in situations where the thread needs to be produced by an external event such as when a JMS message is received.

```
@RequestMapping("/hello")
@ResponseBody
public DeferredResult<String> sayHello() {
    DeferredResult<String> deferredResult =
        new DeferredResult<String>();
    // Save the deferredResult in in-memory queue ...
    return deferredResult;
}

// In some other thread...
deferredResult.setResult(data);
```



### Lab Exercise – Controller Lab

## Chapter Summary

---

- **Like other stereotype components, the `@Controller` annotation lets you designate any Spring Web MVC controller as such without configuration.**
- **The `@RequestMapping` annotation maps request URLs to a controller.**
  - Use this annotation on the controller type itself or on the controller methods.
  - When `@RequestMapping` is used on methods of the controller (method-level annotation), it directs traffic to a specific method within the class.
  - When the `@RequestMapping` annotation is used on the controller class (type-level annotation), it directs a pattern of requests to the controller.
  - However, specific method-level annotations must narrow the mapping for a specific HTTP request method ("GET"/"POST").
- **`@PathVariable` annotation binds the path variable to a handler method parameter.**
  - `@PathVariables` are limited to simple types (int, long, Date, String, etc.).
  - The `@RequestParam` annotation can be used on a handler method to bind a request parameter directly to a method parameter.
- **Handler methods can be passed arguments.**
  - The Spring container automatically detects the needed parameter and passes the appropriate object to the handler method.
- **Handler methods in stereotype controllers can return almost any type of data.**
  - Data returned can be used as the model, view or other such Spring Web MVC component depending on its type.
- **Use the `@InitBinder` annotation on a controller method that establishes the editors you want to use in data binding operations.**
- **In a similar fashion, the `@ModelAttribute` annotation can be used on a handler method to bind a model object's attribute to a method parameter.**
  - The `@ModelAttribute` annotation has a second use. It can also be used to put information, like reference data, into the model.
  - When the `@ModelAttribute` annotation is placed on a method, the method it annotates is executed before the appropriate handler method.

## Chapter 4

### Validation

#### *Objectives:*

- **Learn about validation options in Spring.**
- **Explore the Validator interface and how to apply it.**
- **See how to tie validation into Spring Web MVC controllers.**
- **Examine the JSR-303 bean validation API.**
- **Look at how Spring has adopted and integrates with JSR-303.**

## Chapter Overview

Bad data can create havoc in applications. Bad data can come from a number of sources, such as user data entry or poor configuration. How can you prevent or at least limit bad data from entering your application? As you saw in the last chapter, data binding operations help detect and prevent data that is the wrong type from entering your application. In this chapter, you will learn about Spring's support for data validation. Specifically, you will learn how to validate data as it is applied to the properties of your beans. Validation helps to prevent data that might be the right type but not have the right characteristics from entering your application.

Do Not Print

## Validation

---

- **How do you prevent bad data from entering your beans?**
  - Bad data could come from poor dependency injection configuration (XML or annotation).
  - Bad data could come from data entered via HTML form entry or other user interface.
  - Regardless of how it arrives, you want to protect your beans from bad data through validation.
  - You could add validation logic into bean setter methods. Do you really want business-styled validation logic in your POJOs/beans?
  - Furthermore, what happens if non-setter injection is used to set data into beans?
- **Spring provides a couple of validation capabilities to assist keeping bad data out of good beans.**
- **In the last chapter, you learned about data binding.**
  - Specifically, you learned how Spring Web MVC binds data into command beans.
  - Data binding and validation are different operations that can occur at the same time.
  - Data binding gets String data converted into appropriate types and pushed into beans.
  - Inappropriate data types (a String “old” when expecting an integer) leads to BindingExceptions.
- **Data can be of the right type and legally available for data binding, but still be invalid.**
  - For example, a value of 250 may be an integer, but it is probably not valid data for an age property (unless you are talking about the age of a turtle).
  - Data binding is about checking data for the right type.
  - The validation process checks to insure data fits within certain acceptable parameters.



## Validator

---

- **Spring provides a Validator interface that allows you to write custom validation logic and associate it to beans.**
  - This allows the actual validation logic to be externalized from the beans.
  - It also allows the validation logic to be reused across several bean types.
- **Validator objects are used to validate data in other objects; specifically command objects.**
  - Validator objects are created from classes that implement the `org.springframework.validation.Validator` interface.
  - This interface requires the implementing class to provide `support(Class)` and `validate(Object, org.springframework.validation.Errors)` methods.
  - The `supports` method indicates the types of objects (beans) the Validator checks.
  - The `validate` method performs the actual validation.
- **To explore Validators by way of example, assume you wanted to validate Customer objects.**
  - You want to make sure a first name and last name are provided for any customer.
  - You also want to make sure the age is an integer between 0 and 120.

- **The CustomerValidator class below implements the Validator interface.**

```
public class CustomerValidator implements Validator {

    public boolean supports(Class c) {
        return c.equals(Customer.class);
    }

    public void validate(Object object, Errors errors) {
        Customer customer = (Customer) object;
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
            "firstNameRequired");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName",
            "lastNameRequired", "Customers must have a last name");
        validateAge(customer.getAge(), errors);
    }

    private void validateAge(int age, Errors errors){
        //if age has a binding error, no sense in checking the value
        if(errors.getFieldError("age")==null) {
            if ((age < 0) || (age > 120)) {
                errors.rejectValue("age", "ageValue");
            }
        }
    }
}
```

- Again, the supports( ) method indicates the classes that this Validator can validate. Some Validators might be used across several classes.
- In this example, the CustomerValidator validates just one type of object: Customer objects.
- **The validate( ) method is where the validation actually occurs.**
  - Notice that it is passed an Errors object. This same Errors object collects any binding errors when used with a controller.
  - The validate( ) method also is passed the object to validate.
  - In the case of Spring Web MVC applications, this object reference would be a command bean used to collect data from the HTTP request.

- **As part of the validation package, Spring provides a `ValidationUtils` (`org.springframework.validation`) class.**
  - This class comes with several built-in convenience methods for checking and rejecting any empty fields (with options for checking with or without whitespace).
  - In this example, `ValidationUtils` is used to check that both `firstName` and `lastName` fields are provided.
- **The parameters to the reject methods specify the `Errors` object to add an error to -- the field name to check and the error code if the field is found empty.**
  - As you learned, the `Errors` object is a collector of data-binding and validation errors for a specific object.
  - The error code refers to an externalized resource message code.
  - Externalized message code and message code resolution is covered in the view chapter.
  - In the case of the `lastName`, a default message String is also provided ("Customers must have a last name") in case an error code cannot be resolved.
- **While rejecting empty fields is important, you may have many more complex validation needs.**
  - In this example, age is checked to insure the integer falls within a certain range.
  - For other validation needs such as the age check, write a method to check the value in the appropriate field in the command object.
  - Use `reject()` or `rejectValue()` methods on the `Errors` object when the data in the field is not valid.
  - Notice the private `validateAge` method in this example. This method checks the value in the age property of the customer to ensure that it is between 0 and 120.
  - If it is not, the `rejectValue` method registers a field error for the specified field (in this case "age") and specifies the error code of the error to display to the user.
  - You might notice that the `validateAge` method first checks that an integer has been successfully bound to the object by checking the `Errors` collection.
  - There is no sense in checking the age value if it is not even a valid integer. Often you can use `bind errors` to avoid costly and lengthy validation checks.

- **With the Validator in place, how does the Spring Web MVC environment know when/how to use it?**
  - Through dependency injection - of course!
  - You can simply register your Validators as Spring beans and have them dependency injected into other objects that need to perform validation.

```
<bean id="customervalidator" class="com.intertech.mvc.CustomerValidator"/>
```

- Controllers are big users of Validators. For example, controllers can use Validators to validate data in command beans.

```
@Controller
public class CustomerController {

    @Autowired
    private CustomerValidator validator;

    ...

    @RequestMapping(value = "editcustomer.request", method=RequestMethod.POST)
    protected String updateCustomer(Customer customer, Errors errors) {
        validator.validate(customer, errors);
        if (errors.hasErrors()) {
            // if data binding or validation failed
            return "editcustomer";
        } else {
            // use the DAO to save/update the customer data
            // no binding or validation errors
            return "customerdisplay";
        }
    }
}
```

- **As an alternative, as of Spring 3.0, you can configure a `DataBinder` with a `Validator`.**
  - Recall that the `@InitBinder` annotation allows you to initialize and register custom editors with a `WebDataBinder` (`org.springframework.web.bind`).
  - A `WebDataBinder` is a more specific type of `DataBinder`.
  - So, to add validation to your controller, simply add your `Validator` to the `WebDataBinder`.

```
@Autowired
private CustomerValidator validator;

@InitBinder
public void initBinder(WebDataBinder binder) {
    DateFormat dateFormat = new SimpleDateFormat("MMM d, yyyy");
    CustomDateEditor dateEditor = new CustomDateEditor(dateFormat, true);
    binder.registerCustomEditor(Date.class, dateEditor);
    binder.setValidator(validator);
}
```

- **Now, to invoke the validation logic (`validate` method) in validators, add the `@Valid` annotation as an argument to handler methods.**

```
@RequestMapping(value = "editcustomer.request", method=RequestMethod.POST)
protected String updateCustomer(@Valid Customer customer, Errors errors) {
    // validator.validate(customer, errors);
    if (errors.hasErrors()) {
        // if data binding or validation failed
        return "editcustomer";
    } else {
        // use the DAO to save/update the customer data
        // no binding or validation errors
        return "customerdisplay";
    }
}
```

- With this annotation in place, you do not need to call the `validate` method programmatically.
- `@Valid` is part of the JSR-303 Bean Validation API. Therefore, this annotation is not tied to Spring.
- JSR-303 standardizes validation constraint declaration and metadata for the Java platform.
- Spring 3 supports this API. You will learn more about JSR-303 later in this chapter.

- **As a final alternative, you can also use a new Spring 3.0 mvc namespace to register a global WebBindingInitializer.**
  - Spring 3 provides an mvc XML configuration namespace to simplify the setup of some Spring Web MVC components for your web application.

```
<?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/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
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
  ...
</beans>
```

- Add a `<mvc:annotation-driven />` element to your configuration.

```
<bean id="customervalidator" class="com.intertech.mvc.CustomerValidator"/>
<b>mvc:annotation-driven validator="customervalidator"</b>
```

- This tag performs several operations to assist in MVC development.
- Notably, in this case, the validator parameter allows you to specify a Validator instance as the “global validator.”
- That is, it allows you to configure a Validator instance that is used across all controllers (components annotated with `@Controller`).
- No dependency injection or setting the validator on a `DataBinder` is required in the controller class to receive the benefits of this Validator.
- Simply add the `@Valid` argument to all handler methods to have the global validator (`CustomerValidator`) check and validate all model data.

```
protected String updateCustomer(@Valid Customer customer, Errors errors) {
  ...
}
```

- However, this configuration does require you to establish a single Validator. This Validator must be capable of validating all types of model objects.
- This can cause the Validator class to become quite large in some applications.

## JSR-303 Validation

---

- **As mentioned, as of Spring 3.0, Spring supports JSR-303.**
  - JSR-303 simplifies the integration of a Validator into Spring applications.
  - However, the API also provides several annotations to simplify validation (often without the need for a visible Validator).
  - JSR-303 also standardizes validation across all Java platforms.
  - This allows your domain model POJOs to specify validation requirements without ties to Spring or other technology.
- **The table below lists the JSR-303 validation annotations you can apply to your bean properties.**

JSR-303 Annotation	Property Validation Description
@AssertFalse	Must be false.
@AssertTrue	Must be true.
@DecimalMin	Must be a numeric value greater than or equal to an indicated value.
@DecimalMax	Must be a numeric value less than or equal to an indicated value.
@Digits	Must have a numeric value that can't have more integer digits and fraction digits than indicated by integer and fraction attributes.
@Past	Date must be in the past.
@Future	Date must be in the future.
@Min	Must be greater than or equal to a given value.
@Max	Must be less than or equal to a given value.
@NotNull	Must not be null.
@Null	Must be null.
@Pattern	Must match the regular expression that is given in the attribute.
@Size	The size of the String or collection is between a given min and max values, min and max included.

- **These constraining annotations can be applied on bean fields or methods.**

- **As an example, some of the JSR-303 validation annotations are applied to the Customer class below.**

```
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;

public class Customer {
    @NotNull
    private String firstName;
    @NotNull
    private String lastName;
    private String status;
    @Min(0)
    @Max(120)
    private int age;
    @Past
    private java.util.Date dob;
    ...
}
```

- **JSR-303 is just an API specification. In order to use it, you must have an implementation (a JAR file that implements the API).**
  - The Hibernate Validator library is the reference implementation of JSR-303 and the one used with Spring.
  - To add the necessary libraries to use the JSR-303 annotations, add the hibernate-validator dependency to your Maven pom.xml file.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.2.0.Final</version>
</dependency>
```

- **You must also make the Spring container aware that you are accomplishing validation via JSR-303 annotations.**
  - Spring provides a JSR-303 Validator to signal you are using JSR-303 annotated beans for validation.
  - When using JSR-303 annotations, you must add this bean to your configuration.

```
<bean id="jsrvalidator" class=
  "org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```



- **From your application perspective, this bean is like any Validator.**
  - You must now choose a mechanism for configuring other beans, such as controllers, to use the Validator.
  - You can choose to have the LocalValidatorFactoryBean dependency injected into your controllers.

```
@Controller
public class CustomerController {

    @Autowired
    private LocalValidatorFactoryBean validator;

    ...
    @RequestMapping(value = "editcustomer.request", method=RequestMethod.POST)
    protected String updateCustomer(Customer customer, Errors errors) {
        validator.validate(customer, errors);
        if (errors.hasErrors()) {
            // if data binding or validation failed
            return "editcustomer";
        } else {
            // use the DAO to save/update the customer data
            // no binding or validation errors
            return "customerdisplay";
        }
    }
}
```

- You can choose to add the Validator into the DataBinder.

```
@Autowired
private LocalValidatorFactoryBean validator;

@InitBinder
public void initBinder(WebDataBinder binder) {
    DateFormat dateFormat = new SimpleDateFormat("MMM d, yyyy");
    CustomDateEditor dateEditor = new CustomDateEditor(dateFormat, true);
    binder.registerCustomEditor(Date.class, dateEditor);
    binder.setValidator(validator);
}

@RequestMapping(value = "editcustomer.request", method=RequestMethod.POST)
protected String updateCustomer(@Valid Customer customer, Errors errors) {
    if (errors.hasErrors()) {
        // if data binding or validation failed
        return "editcustomer";
    } else {
        // use the DAO to save/update the customer data
        // no binding or validation errors
        return "customerdisplay";
    }
}
```

- You can choose to use the Spring 3.0 mvc namespace and register the Validator as a global validator.

```
<bean id="jsrvalidator" class=
    "org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
<mvc:annotation-driven validator="jsrvalidator"/>
```

- Remember that you still need to use the @Valid argument to all handler methods for the validation (in this case the JSR-303 annotations) to apply.

```
protected String updateCustomer(@Valid Customer customer, Errors errors) {
    ...
}
```

- **Lastly, when using Spring Web MVC with JSR-303, there is an even easier way to configure validation.**
  - A single javax.validation.Validator instance can validate all model objects with JSR-303 annotations in Spring Web MVC.
  - To use this Validator, simply register the <mvc:annotation-driven /> element (with no validator attribute).

```
<mvc:annotation-driven />
```

- This causes the Spring Web MVC container to locate any javax.validation.Validator implementer (such as the Hibernate Validator) in your classpath.
- With the Validator detected, Spring automatically enables JSR-303 support across all controllers.
- Simply add the @Valid to handler methods to invoke the validation.



### Lab Exercise – Validation Lab

## Chapter Summary

---

- **Spring provides a Validator interface that allows you to write custom validation logic and associate it to beans.**
  - Validator objects are used to validate data in other objects -- specifically command objects.
  - Validator objects are created from classes that implement the `org.springframework.validation.Validator` interface.
  - This interface requires the implementing class to provide `support(Class)` and `validate(Object, org.springframework.validation.Errors)` methods.
  - The `supports` method indicates the types of objects (beans) the Validator checks.
  - The `validate` method performs the actual validation.
  - As part of the validation package, Spring provides a `ValidationUtils` (`org.springframework.validation`) class.
  - This class comes with several built-in convenience methods for checking and rejecting any empty fields (with options for checking with or without whitespace).
- **As of Spring 3.0, Spring supports JSR-303.**
  - JSR-303 simplifies the integration of a Validator into Spring applications.
  - The API also provides several annotations to simplify validation (often without the need for a visible Validator).
  - JSR-303 also standardizes validation across all Java platforms.
  - This allows your domain model POJOs to specify validation requirements without ties to Spring or other technology.
  - JSR-303 is just an API specification. In order to use it, you must have an implementation (a JAR file that implements the API).
  - Spring provides the Hibernate implementation of JSR-303.

## Chapter 5

### Spring Web MVC Views

#### *Objectives:*

- Examine the “V” in Spring Web MVC development.
- Explore Spring’s custom tag libraries for support of JSP template development.
- Look at the form tag library for binding form data to command beans.
- Explore view text externalization through the message tag.
- Understand how text externalization can be used to support internationalization and localization needs.
- See how to use the errors tag to display validation and binding errors in the view.
- Learn how to produce non-HTML views in Spring.
- See how Spring can produce PDF and Excel views.

## Chapter Overview

To assist Java/Spring Web developers in producing views that have awareness of Spring command beans and message bundles, Spring Web MVC comes with a couple of custom tag libraries. In this chapter, you will study these custom tag libraries. These tag libraries help bind form data to command beans and vice versa. They also help assist in externalizing any text displayed in a Spring JSP template, which helps support an application's internationalization needs.

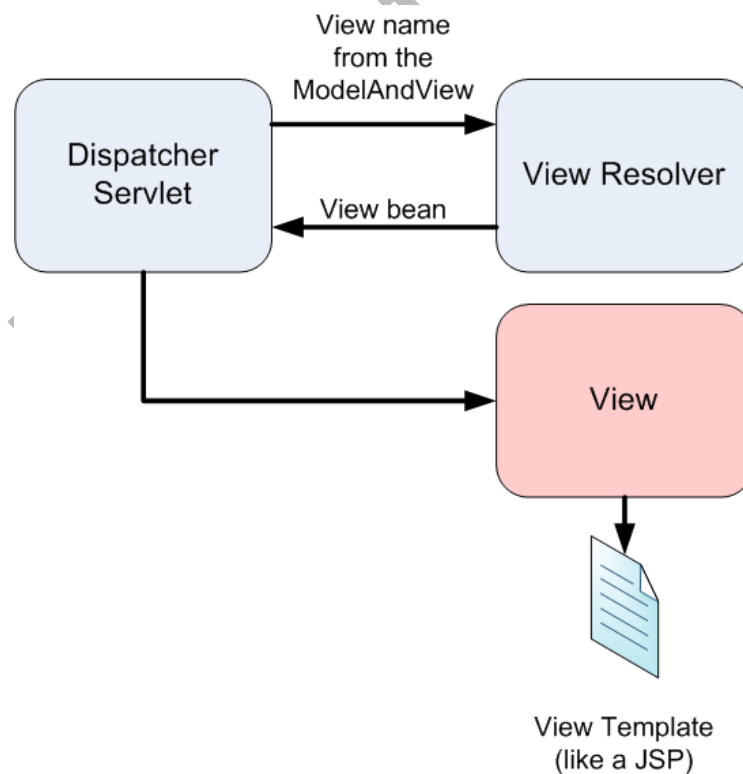
While most Web applications render HTML to the user, there are parts of Web apps that are not in HTML but other MIME types such as images, PDF documents, and Microsoft documents like an Excel spreadsheet. So, in addition to examining Spring's custom tag library, you will also look at how Spring provides view classes for rendering non-templated output.

Do Not Print

## Spring Web MVC View

---

- **In a previous chapter, you learned that view resolvers have the job of mapping a logical view name to a View bean.**
  - The logical view name is often returned by a controller in a ModelAndView object or simple String.
  - View beans are objects that implement the `org.springframework.web.servlet.View` interface. There are all sorts of view beans.
  - Most view beans, like the `InternalResourceView` used by default with the `InternalResourceViewResolver`, simply delegate work to a template.
  - JSPs are the most common form of templates in Java Web applications.
  - However, not all views are rendered by templates.
  - In some cases, the view bean may produce its own output. These components are often used to produce a PDF, Excel or other type of MIME output.
  - The “V” in the MVC environment is typically comprised of several components and possibly a template.



- **In this chapter, you will explore Spring-provided tags that assist in JSP template development.**
  - Some of these tags were alluded to in earlier chapters.
  - In particular, there are tags to help bind form data to command beans and vice versa.
  - There are also tags to assist in externalizing labels and other messages displayed in the template.
  - These tags help support an application's internationalization and localization needs.
  - Finally, there are a set of tags to display error messages produced by data binding and validation operations.
- **You will also look at view beans that do not delegate work to a template.**
  - That is, you will explore how to build view beans that render non-HTML output (in what are called non-templated views).
  - In particular, you will explore the output of Excel and PDF (two popular non-HTML outputs).
  - However, through an examination of these types of output, you will also learn how views could be created to produce any type of output.

## Spring Form Tag Library

---

- **Spring 2.0 introduced a set of custom tags to provide form binding and message/label externalization.**
  - You have not seen these tags in example code shown in this text so far.
  - So, you can see that it is not absolutely necessary to use these tags when creating a Web application.
  - JSP scriptlets (Java code), Unified Expression Language, and other tag libraries like JSTL can be used to create the templates and access/display model information.
  - However, the Spring Form Tag Library custom tags make the development of views and view templates much easier.

Do Not Print



## Bind Form Data

- **In the Form Tag Library, a set of Spring Web MVC command bean-aware tags are provided to bind form elements to command model data in JSPs.**
  - These custom tags (to include the spring-form.tld file) are shipped in the spring-webmvc JAR file.
  - Therefore, they are easy to use. The spring-webmvc dependency, added to use the Spring Web MVC framework, also brings these tags into your application.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>
```

- Also, add the following JSP directive to the top of JSPs that use the tags.

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

- Notice “form” is used as the prefix for the form tags. While using “form” is not required, it is considered common practice and convention when using these tags.
- **In JSPs, the Spring form tags replace the HTML form elements. The table below outlines the form tag replacements for the HTML form elements.**

HTML form element	Spring Form Tag replacement
<form>	<form:form>
<input type="text">	<form:input>
<input type="checkbox">	<form:checkbox> or <form:checkboxes>
<input type="radio">	<form:radiobutton> or <form:radiobuttons>
<input type="password">	<form:password>
<select>	<form:select>
<option>	<form:option> or <form:options>
<textarea>	<form:textarea>
<input type="hidden">	<form:hidden>

<label>	<form:label>
---------	--------------

Do Not Print

- **The <form:form> tag renders an HTML form.**
  - Like an HTML form element, the form tag offers optional attributes to set the action and method for the form.
  - In addition, the form tag allows you to bind the form (and its fields) to a command bean (a.k.a. form backing object) by name, per the commandName attribute.

```
<form:form action="addCustomer.request" method="post"
commandName="customer">
First name:<form:input path="firstName"/>
Last name:<form:input path="lastName"/>
Marital status:<form:select path="status" items="${statusList }"/>
Age:<form:input path="age"/>
<input type="submit" value="Add"/>
</form:form>
```

First name:  Last name:  Marital status:  Age:

- While the commandName attribute is optional, the command bean name is assumed to be “command” by default. Therefore, it is usually provided.
- All other form tag elements must be nested within a <form:form> tag.
- **When a form is bound to a command bean by name, the form data and command bean properties are inexorably tied to each other.**
  - As a command bean (the bean named customer in the example) is passed to the JSP template, its properties are bound to the field values.
  - For example, the firstName property from the customer is bound to the value of the first <form:input> field via the path attribute (path=“firstName”).
  - Likewise, when the form is submitted, the data in the form fields is bound to the customer command bean.
  - So, a value in the firstName field is returned in the firstName property of the customer command bean.

- Each of the nested form tags has a path attribute that specifies the “path” to the command bean property for data binding.
- Most of the form tags like `<form:input>` and `<form:select>` shown in the example resemble their HTML element counterparts in attribute offerings.
  - Therefore, complete coverage of each tag and all of its setup is not warranted. A few, however, offer some different features that are covered here.
  - Section 36 of the Spring framework documentation carries a complete reference of each of the form tags.
  - Appendix E is available at <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#spring-form.tld>.
- The form select and option tags offer some unique capability that requires some mention.
  - The `<form:select>` tag renders an HTML select tag. In the example above, a list of marital status choices were made available via the `<form:select>` tag.

```
Marital status:<form:select path="status" items="${statusList }"/>
```

- The items attribute uses any collection, map, or array of objects used to generate the inner option tags.
- Nested `<form:option>` tags can be used explicitly to provide the options as well.

```
Marital status:<form:select path="status" >
<form:option value="single"/>
<form:option value="married"/>
<form:option value="separated"/>
<form:option value="divorced"/>
<form:option value="widow(er)"/>
</form:select>
```

- A `<form:options>` tag can be used to provide a collection of options.
  - The `<form:option>` and `<form:options>` elements can be used together as shown below.

```
Marital status:<form:select path="status" >
<form:option value="-" label="-- Please choose -- " />
<form:options items="${statusList }"/>
</form:select>
```

First name:  Last name:  Marital status:  Age:

-- Please choose --  
 single  
 married  
 separated  
 divorced  
 widow(er)

- The `<form:options>` tag comes with optional `itemLabel` and `itemValue` attributes.
- These can be used to set the label/value used for options when the collection of items is a collection of more complex objects.
- For example, if a `<form:options>` tag provided a list of customers to choose from, the `itemLabel` might be used to show the customer's name in the select field.
- The `itemValue` might then be used to bind the id for the selected customer to the command bean's id property.

```
Owner:<form:select path="customer" >
<form:option value="-" label="-- Select Customer -- " />
<form:options items="${customerList }" itemValue="id"
itemLabel="lastname"/>
</form:select>
```

- **Checkbox and radiobutton tags also offer some interesting capabilities.**

- The `<form:checkbox>` tag renders an HTML input element with the type set to “checkbox.”

```
<form:checkbox path="citizen" label="Citizen"/>
```

- The `<form:checkbox>` element can be bound, however, to three different types of properties in the command bean.
- **When bound to a boolean property (boolean or `java.lang.Boolean`), the checkbox is checked when the property is true.**

```
private boolean citizen;
```

- Likewise, the property is set to true when the form is submitted and the checkbox has been checked.
- **When bound to an array or `java.util.Collection` property in the command bean, the checkbox is checked if the checkbox value is in the array or collection.**

```
<form:checkbox path="citizenships" value="American" label="American"/>
```

```
private String[] citizenships = {"American"};
```

- **Lastly, when bound to any other type of property, the checkbox is checked if the value of checkbox matches the property value.**

```
<form:checkbox path="numberOfCitizenships" value="1" label="Single  
Citizenship"/>
```

```
private int numberOfCitizenships=1;
```

- For any other bound value type, the input(checkbox) is marked as “checked” if the configured `setValue(Object)` is equal to the bound value.

- The `<form:checkboxes>` tag renders multiple HTML input checkboxes.
  - The `items` attribute is set with an Array, a List or a Map containing the available checkbox options.
  - The `<form:checkboxes>` tag also binds to an Array, List or Map property in the command bean.

```
<form:checkboxes path="citizenships" items="${citizenList}"/>
```

```
private java.util.List citizenships;
```

- The `@ModelAttribute` annotated method of a controller is often where the item options are created and made available for use in the JSP template.

```
@ModelAttribute("statusList")
protected List<String> status(HttpServletRequest request) {
    String[] statusStrings = {"single", "married", "separated",
        "divorced", "widow(er)"};
    return Arrays.asList(statusStrings);
}
```

- **Radiobutton form tags have a similar configuration if not a slightly different behavior from the checkbox tags.**
  - Representing a single select choice, radio buttons are often used in a group of two or more.
  - The `<form:radiobutton>` tag renders a single HTML input of type “radio.”
  - The path attribute of the radiobutton tags in a group are all set to the same property in a command bean.

```
<form:radiobutton path="gender" value="Male" label="Male" />
<form:radiobutton path="gender" value="Female" label="Female" />
```



- The command bean property type should match the type of value returned. In this example, the gender property should be a String.

```
private String gender;
```

- If the value attributes on the `<form:radiobuttons>` was something like “1” and “2,” then the gender property could be of type int.
- **The `<form:radiobuttons>` tag, like `<form:checkboxes>` allows multiple radiobuttons to be rendered based on an Array, List or Map of options.**
  - The items property is used to specify the options for the radiobuttons.
  - Unlike `<form:checkboxes>`, a single choice is made from the radiobutton options so the bound property is not an Array, List or Map.

```
<form:radiobuttons path="gender" items="${genderList}" />
```

- The command bean property type bound to `<form:radiobuttons>` should match the type of value returned by the group of radiobuttons.



- **As mentioned, the form tags resemble their HTML element counterparts in attribute offerings.**
  - Of particular note, all of the form tags come with HTML event attributes (onclick, onfocus, onmousedown, etc.).

```
<form:input path="firstName" onchange="ChangeAlert()" />
```

- These are important when tying the JSP templates into rich Internet application (RIA) capability such as that offered by AJAX.
- Tag attributes also allow the CSS style and class to be set.

```
<head>
<title>Customer info</title>
<style type="text/css">
input.important {color:red;}
</style>
</head>
<body>
<form:form action="addcustomer.request" method="post"
commandName="customer">
First name:<form:input path="firstName" cssClass="important"/>
...
```

First name:  Last na

## Externalized Messages

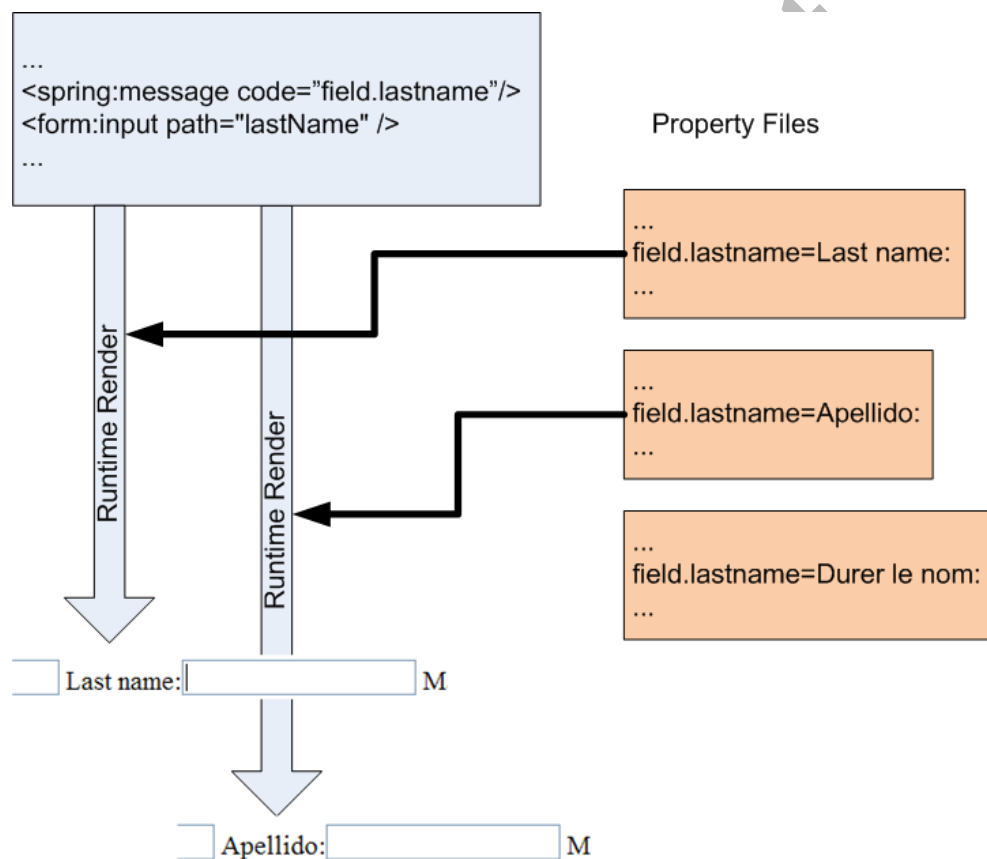
---

- **When building Web applications, using hardcoded labels and other text messages (like error messages) in the templates is usually unacceptable.**
  - For example, the label “Last name:” is hardcoded into the JSP template for the last name field below.

```
Last name:<form:input path="lastName" />
```

- What if the Web application is to be used by people who speak a different language? Are they going to know what “last name” means?
  - Even if the application is expected to be used in one language, having labels and other text hardcoded into the templates can create maintenance issues.
  - What if someone decides that “last name” is confusing?
  - Management decides that all “last name” labels must be changed to “surname” or “family name” throughout the application.
  - Now, someone has to go through and change all the places where “last name” appears in the application.
- **Therefore, it is preferable to externalize all text from view templates.**

- **To address this issue, Spring provides a set of custom tags to externalize all template text.**
  - The tag is a placeholder for a piece of text. At runtime, when the template is rendered to the user, the tag is replaced by text held in an external properties file.
  - The external properties file holding the text is also known as a message file, message bundle, or resource bundle.
  - The external properties file allows the text to be changed more easily and universally when the text is shared across templates.
  - Also, multiple property files can exist, allowing the application to support internationalization (i18n) or localization (l10n).



- **The Spring “placeholder” tag for text is `<spring:message>`.**
  - To use this Spring custom tag for message externalization, add the following JSP directive to the top of JSPs that use `<spring:message>`.

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

- **Now, replace each piece of text that should be externalized with the `<spring:message code=" " />` tag.**
  - The code attribute serves as a lookup identifier when Spring replaces the tag with the actual text from the message file.
  - Each text, label, or message should have its own unique identifier.
  - Typically, the codes that identify text should help to identify the purpose of the text.
  - For example, labels for fields are coded as “field...”, error messages are coded as “error...”.
- **Here is the customer JSP template with all hardcoded text replaced by `<spring:message>` tags.**

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title><spring:message code="title.customerinfo"/></title>
</head>
<body>
<form:form action="addcustomer.request" method="post"
commandName="customer">
<spring:message code="field.firstname"/><form:input path="firstName"/>
<spring:message code="field.lastname"/><form:input path="lastName"/>
<spring:message code="field.maritalstatus" />
<form:select path="status" items="{statusList}"/>
<spring:message code="field.age"/><form:input path="age"/><form:errors
path="age"/>
<input type="submit" value="<spring:message code="button.add"/>" />
</form:form>
</body>
</html>
```

- Note that even the page title and submit button label are replaced by `<spring:message>` tags.
- All static text should be replaced by these placeholders. It wouldn't do much good to have the page displayed mainly in Spanish but have an “Add” button.

- **Message or properties files are developed that translate the message codes to the actual text to be displayed on the screen.**
  - The message file contains a set of key-value pairs. The key in the file is the code as seen in the <spring:message> tags.
  - The value in the file is the text that Spring should use to replace the tag at runtime.
  - An example message file containing the key-values for the customer JSP template page might look like the example below.

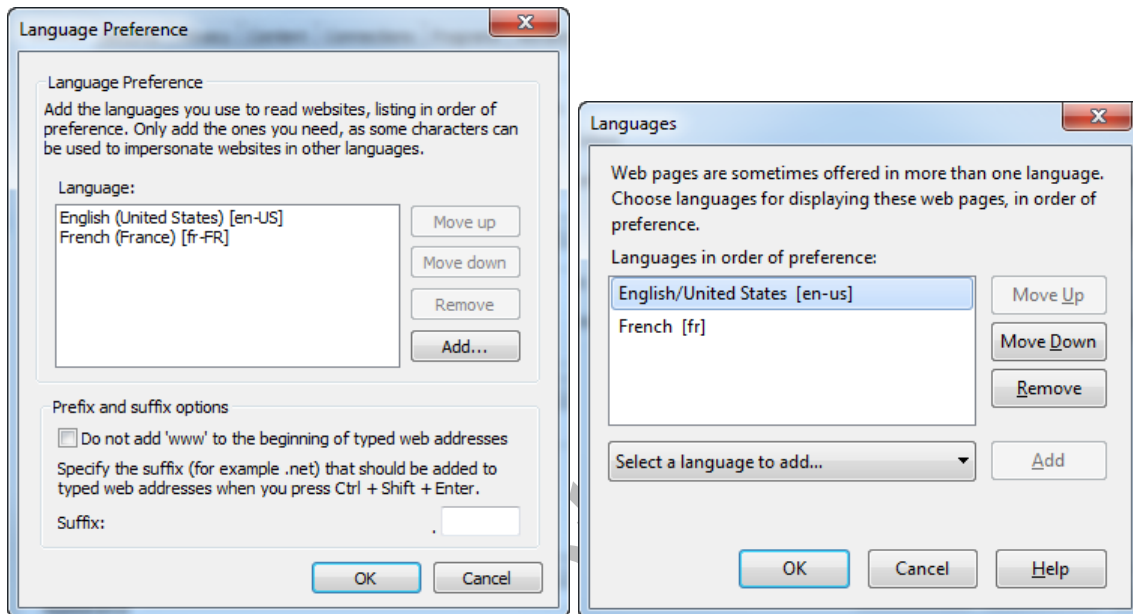
```
field.firstname=First name:
field.lastname=Last name:
field.age=Age:
field.maritalstatus=Marital status:
button.add=Add
title.customerinfo=Customer Data
```

- **The default message file is typically stored in a messages.properties file.**
  - The “messages” portion of the message file name is called the basename.
  - For each language and/or locale that is to be supported by the application, another message file is created.
  - In these other message files are the key-value pairs for the alternate language and/or locale. The keys are the same, but the values are different.
  - For example, the key-value pairs for French are provided below.

```
field.firstname=Premier nom:
field.lastname=Durer le nom:
field.age=Age:
field.maritalstatus=Statut marital:
button.add=Ajouter
title.customerinfo=Données clientèles
```

- This file would be stored in a messages\_fr.properties file.
- That is, alternate language/locale message files should be stored in a file with the file basename concatenated with the language/locale code of the file.

- **The user's language and locale is specified in the HTTP request message sent by the user's browser.**
  - All Web browsers provide for the ability to set the language and locale of choice for the user.
  - Language selection windows for Microsoft's Internet Explorer and Mozilla's Firefox are shown below.



- The HTTP header carries the user's preference in the Accept-Language value.
- The language/locale codes sent by the browser are the same codes used to complete the name of the message files.
- For example, the message file to support French readers is going to be stored in messages\_fr.properties.
- Spring reads the header and uses the best match among the message files of your application to replace the <spring:message> tags.
- If a match is not found, it uses the default (messages.properties) file to replace the <spring:message> tags with the appropriate text.

- **Lastly, to inform Spring to use the message files when creating views, register a `ResourceBundleMessageSource` bean.**
  - This bean is provided by Spring so you don't have to create it, but it must be registered under the id of "messageSource" in the Spring configuration file.
  - It works with the `<spring:message>` tag to resolve the message codes to text or message values.
  - When configuring this bean in the Spring configuration file, you can set the `basename` property. This property identifies the basename of the message files.
  - Again, by convention, this is "messages."

```
<bean id="messageSource" class=
  "org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>messages</value>
    </list>
  </property>
</bean>
```

- **With these pieces in place, the JSP template is now devoid of language-specific and hard to maintain text.**
  - Also, based on a user's browser language/locale setup, the page may be displayed in a fashion more suitable to the user.

First name:  Last name:  Marital status:  Age:

---

Premier nom:  Durer le nom:  Statut marital:  Age:

- **A few more tags were added to the Spring's JSP tag library (the same tag library where `<spring:message>` is found) in Spring 3.0.**
  - The `<spring:eval>` tag allows for evaluating a SpEL expression and embedding its result into a JSP page.

```
<spring:eval expression="customer.lastName"/>
```

- The `<spring:url>` tag displays URLs with support for URI template variables, HTML/XML escaping, and JavaScript escaping.
- It was modeled after the JSTL `c:url` tag.
- The `<spring:url>` tag is often used with `<spring:param>` to provide value for the template variables.

```
<spring:url value="http://www.intertech.com/customer/{customer}" var="xyz">  
  <spring:param name="customer" value="${customer.lastName}" />  
</spring:url>
```



## Error Messages

---

- **In a previous chapter, you saw how binding and validation errors are collected in an Errors object.**
  - When a field's value is rejected during binding or validation, an error is added to the Errors object.
  - Each error is associated with a message code.
  - Below are a couple of the methods to add an error to the Errors object showing that the error code is always part of adding an error to the Errors object.

```
//Register a global error for the entire target object, using the
//given error description.
reject(String errorCode, Object[] errorArgs, String defaultMessage)
//Register a field error for the specified field of the current object
//(respecting the current nested path, if any), using the
//given error description.
rejectValue(String field, String errorCode, Object[] errorArgs, String
defaultMessage)
```

- **Another one of the tags from the form tag library allows the binding and validation issues in the Errors object to be rendered on a view.**
  - The tag to display errors on a view is <forms:errors>.
  - After rejecting field values, the form view is redisplayed to the user.
  - The <forms:errors> tag allows the errors collected to be displayed.

- **However, since error messages are going to be seen by the user, these error messages should also be externalized - just as all labels and text.**
- The error code associated to each error in the Errors collection allows each error message to be displayed from the externalized properties file.
- Error message codes and appropriate text are additional key-value pairs stored in the message properties file.

```
field.firstname=First name:
field.lastname=Last name:
field.age=Age:
field.maritalstatus=Marital status:
button.add=Add
title.customerinfo=Customer Data
firstNameRequired=First name is a required field
lastNameRequired=Last name is a required field
typeMismatch.customer.age=Age must be an integer
ageValue=Age must be between 0 and 120
```

- Remember, the error messages must be added to all of the language/locale message property files.
- **The <forms:errors> tag takes an optional path attribute.**
- The error text of any error code associated to the field specified by the path attribute is displayed in place of the <forms:errors> tag.

```
<spring:message code="field.firstname"/><form:input
path="firstName"/><form:errors path="firstName" />
<spring:message code="field.lastname"/><form:input
path="lastName"/><form:errors path="lastName" />
<spring:message code="field.maritalstatus" />
<form:select path="status" items="${statusList}"/>
<form:errors path="status" />
<spring:message code="field.age"/><form:input path="age"/>
<form:errors path="age" />
```

- Typically, as shown in this example, a <forms:errors> tag with matching path attribute is written next to its associated form field.
- That way, if a validation or binding error occurs for that field, the message is displayed next to it when the form is redisplayed.

First name:  First name is a required field.

- If more than one validation or binding error occurs with any field, use an asterisk if you want all the errors associated to that field displayed.

```
<spring:message code="field.age"/><form:input path="age"/>
<form:errors path="age*" />
```

- A cascading style and cssClass attribute on the <form:errors> tag is often used to highlight errors in the display.

```
...
<html>
<head>
<title><spring:message code="title.customerinfo" /></title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>
<body>
<form:form action="addcustomer.request" method="post"
commandName="customer">
    <spring:message code="field.firstname" />
    <form:input path="firstName" size="20" />
    <form:errors path="firstName" cssClass="error"/>
...

```

First name:  First name is a required field

- To display the entire list of errors, no matter the field they are associated with, use an asterisk in the path attribute.

```
<form:errors path="*" />
```



## Lab Exercise – Template Views Lab

## Non-template Views

---

- **Recall that the `BeanNameViewResolver` looks for a view bean with the name that matches the logical view name it has been given.**
  - The bean that the `BeanNameViewResolver` calls must implement the `View` interface, and it must render the next view.
  - However, the view bean does not have to render the view via template like a JSP.
  - In fact, the `BeanNameViewResolver` is often used to call on view beans that create non-HTML output.
- **As a refresher, assume that a controller returns a `ModelAndView` object with a logical view name of “boring” as shown below.**

```
return new ModelAndView("boring", "message", "Hello World");
```

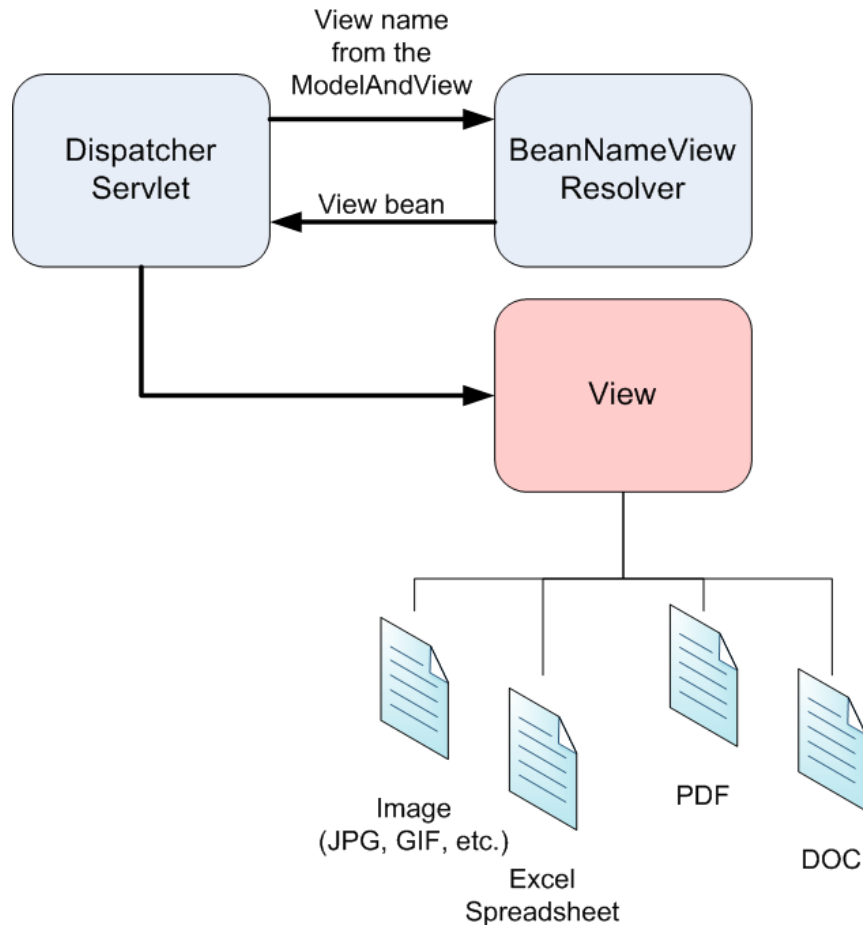
- The `BeanNameViewResolver` uses the logical view name, in this case “boring,” and looks for a bean (implementing the `View` interface) with that name.
- So, in this example, a bean must be registered in the Spring configuration file with an id of “boring,” as shown below.

```
<bean id="boring" class="com.intertech.mvc.BoringResponse"/>
```

- Again, `BoringResponse` must implement the `View` interface, or extend a class that implements the `View` interface, as is the case in this example (`AbstractView`).

```
public class BoringResponse extends AbstractView{  
  
    protected void renderMergedOutputModel(Map model, HttpServletRequest  
req, HttpServletResponse resp) throws Exception {  
        //do view rendering  
    }  
}
```

- **Spring provides several convenience classes that implement the View interface.**
  - In particular, there are convenience classes that implement View for creating Excel spreadsheets and PDF documents (explored in a bit).
  - However, by implementing View (or extending the AbstractView class), any type of output is possible.



- **Consider the following implementation of AbstractView.**

```
public class ImageView extends AbstractView {
    protected void renderMergedOutputModel(Map model, HttpServletRequest
req, HttpServletResponse resp) throws Exception {
        resp.setContentType("image/jpeg");
        ServletOutputStream out = resp.getOutputStream();
        String jpegFilename = (String) model.get("jpegImage");
        File file = new File(jpegFilename);
        BufferedImage img = ImageIO.read(file);
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(os);
        encoder.encode(img);
        out.write(os.toByteArray());
        out.flush();
    }
}
```

- This View class renders any JPEG stored in a file whose name is passed in the model.
- The Map object passed as a parameter to the renderMergedOutputModel is the key-value pairs of the Model from the ModelAndView object.
- Below is the controller that requests the logical view name of “imageView” and puts the name of the JPEG file in the model.

```
public class ImageController {
    @RequestMapping("image.jpg")
    protected ModelAndView generateJPEG(HttpServletRequest req)
    {
        String filename = req.getParameter("imagename");
        return new ModelAndView("imageView", "jpegImage", filename);
    }
}
```

- **Again, the BeanNameViewResolver resolves any logical view name for imageView to a bean by the same name.**
- So, to use this view resolver, register the ImageView class as a bean under the id “imageView.”

```
<bean class=
    "org.springframework.web.servlet.view.BeanNameViewResolver"/>
<bean id="imageView" class="com.intertech.mvc.ImageView"/>
```

- Using the View interface or extending the AbstractView class, just about any MIME type object can be rendered to the browser.

## Excel View

---

- Thankfully, Spring provides a couple of View implementation classes that make developing some popular non-HTML output a snap.
- To produce and render an Excel spreadsheet, create a view class that extends the Spring provided `AbstractExcelView` or `AbstractJExcelView`.
- The `AbstractExcelView` class uses the Jakarta POI project to work with and produce an Excel document.
  - Jakarta POI is a collection of APIs for manipulating various file formats based upon Microsoft's OLE 2 Compound Document format.
  - The Jakarta POI library needs to be added to your project to use the `AbstractExcelView`.
  - Add the poi dependency to your Maven pom.xml.

```
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>3.9</version>
</dependency>
```

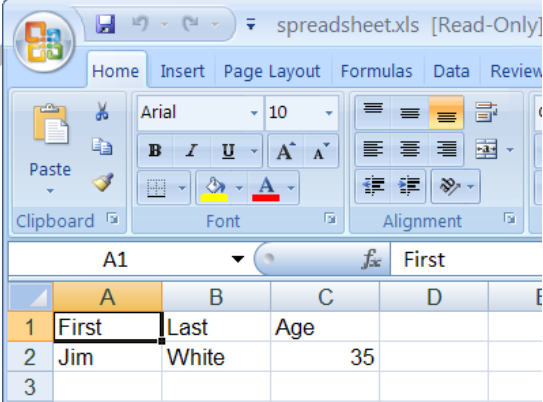
- To successfully extend `AbstractExcelView`, you must implement the `buildExcelDocument` method.
  - This method is passed the Model map (from the `ModelAndView`), request and response objects, as well as an `HSSFWorkbook` object.
  - The `HSSFWorkbook` object is from Jakarta POI and it represents an Excel workbook.
  - The development members of the POI project think a lot of Microsoft's format. HSSF stands for the "Horrible Spreadsheet Format."
  - The framework creates the object and calls on the view's method, but you must use the POI API and the `HSSFWorkbook` to create the contents of the spreadsheet.

- The simple example below demonstrates how to create an Excel-producing view while also demonstrating the use of the POI API.

```
public class CustomerSpreadsheetView extends AbstractExcelView{

    protected void buildExcelDocument(Map model, HSSFWorkbook workbook,
        HttpServletRequest req, HttpServletResponse resp) throws
    Exception {
        Customer cust = (Customer) model.get("customer");
        HSSFSheet sheet = workbook.createSheet();
        HSSFRow header = sheet.createRow(0);
        HSSFCell c = header.createCell(0);
        super.setText(c, "First");
        c = header.createCell(1);
        super.setText(c, "Last");
        c = header.createCell(2);
        super.setText(c, "Age");
        HSSFRow custRow = sheet.createRow(1);
        c = custRow.createCell(0);
        super.setText(c, cust.getFirstName());
        c = custRow.createCell(1);
        super.setText(c, cust.getLastName());
        c = custRow.createCell(2);
        c.setCellValue(cust.getAge());
    }
}
```

- Note that the view rendering method (buildExcelDocument) returns void.
- Simply add content to the already provided HSSFWorkbook. This Excel workbook is what is rendered by the framework.
- Here are the sample output results produced by the view above.



	A	B	C	D	E
1	First	Last	Age		
2	Jim	White	35		
3					



- **The BeanNameViewResolver is again responsible for calling on the view bean above.**

A controller, like the one below, merely creates a ModelAndView object.

```
public class CustomerDataController {

    @RequestMapping("customer.xls")
    protected ModelAndView customerSpreadsheet(HttpServletRequest req) {
        Customer c = new Customer();
        //fetch the appropriate customer and set its properties
        return new ModelAndView("spreadSheetView", "customer", c);
    }
}
```

- The BeanNameViewResolver is responsible for matching the logical view name to the Excel view bean.

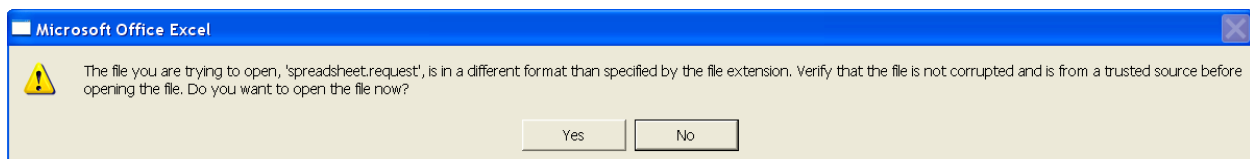
```
<bean class=
    "org.springframework.web.servlet.view.BeanNameViewResolver"/>
<bean id="spreadSheetView" class=
    "com.intertech.mvc.CustomerSpreadsheetView"/>
```

- **It should be noted that in theory any request URL could be routed to the CustomerDataController.**
- However, since an Excel spreadsheet is returned, you might want to map request URLs that end with an .xls suffix.

```
public class CustomerDataController {

    @RequestMapping("customer.xls")
    protected ModelAndView customerSpreadsheet(HttpServletRequest req) {
        ...
    }
}
```

- Both the user and the browser are likely to expect the request URL to match what is rendered.
- In fact, if a different (non-.xls ending) request URL suffix is used, the browser actually complains and seeks guidance.



- **As an alternative, the AbstractJExcelView class uses the JExcelApi to work with and produce an Excel document.**
  - JExcelApi is an open source API focused on working with Excel spreadsheets.
  - You can find out more about the JExcelApi at: <http://jexcelapi.sourceforge.net/>.
  - The JExcelApi library needs to be added to your project to use the AbstractJExcelView.
  - Add the JExcel dependency to your Maven pom.xml file.

```
<dependency>
  <groupId>net.sourceforge.jexcelapi</groupId>
  <artifactId>jxl</artifactId>
  <version>2.6.12</version>
</dependency>
```

- **You might be asking, “Which Excel Spreadsheet package is best?” Which should I use?**
  - Both offer the same general capability.
  - Here is a quote from the Spring documentation that might help you decide.<sup>1</sup>

We've found that the JExcelApi is somewhat more intuitive, and furthermore, JExcelApi has slightly better image-handling capabilities. There have been memory problems with large Excel files when using JExcelApi however.

- **To successfully extend AbstractJExcelView, you must also implement the buildExcelDocument method.**
  - This method is passed the Model map (from the ModelAndView), request and response objects as well as a JExcelApi WritableWorkbook object.
  - The WritableWorkbook object is from JExcelApi library, and it represents an Excel workbook.

<sup>1</sup> <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#view-document-configsubclasses>

- **The example below, which mimics the one from POI, demonstrates how to create an Excel-producing view while also demonstrating the use of JExcelApi.**

```
public class CustomerSpreadsheetView extends AbstractJExcelView {  
  
    protected void buildExcelDocument(Map model, WritableWorkbook wb,  
        HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
        Customer cust = (Customer) model.get("customer");  
        WritableSheet sheet = wb.createSheet("Customer data", 0);  
        sheet.addCell(new Label(0, 0, "First"));  
        sheet.addCell(new Label(1, 0, "Last"));  
        sheet.addCell(new Label(2, 0, "Age"));  
        sheet.addCell(new Label(0, 1, cust.getFirstName()));  
        sheet.addCell(new Label(1, 1, cust.getLastName()));  
        sheet.addCell(new Label(2, 1, "" + cust.getAge()));  
    }  
}
```

- **Again, you must configure the BeanNameViewResolver to call on this view bean, just as you did for the AbstractExcelView version.**

## PDF View

---

- **So Spring helps produce Excel documents. Can it help with other MIME type rendering?**
- **Yes! The AbstractPdfView can be extended to render PDF documents.**
  - The AbstractPdfView uses the iText PDF library to produce PDF documents. It can also be used to generate documents in RTF and HTML formats.
  - iText (see [www.lowagie.com](http://www.lowagie.com)) is an open source Java API for creating PDF documents.
  - Therefore, the iText PDF library needs to be part of your project to use AbstractPdfView.
  - Add the iText dependency to your Maven pom.xml.

```
<dependency>  
  <groupId>com.lowagie</groupId>  
  <artifactId>itext</artifactId>  
  <version>4.2.1</version>  
</dependency>
```

- **To successfully extend AbstractPdfView, you must implement the buildPdfDocument method.**
  - Again, this method is passed the Model map (from the ModelAndView), request and response objects.
  - It is also passed instances of a Document and PdfWriter.
  - The Document and PdfWriter classes are from the iText API.
  - The Document object represents the PDF document to be rendered.
  - While it is created by the Spring framework and passed to this method, you must provide the PDF Document content.
  - The PdfWriter object translates all the text and data added to the Document object into PDF format.
  - The Document object is your interface for adding data to the PDF.
  - However, the PdfWriter is listening to data added to the Document and is what really produces the PDF-formatted content that is rendered.
- **The simple example below demonstrates how to create a PDF-producing view while also demonstrating the use of the iText API.**

```

public class CustomerLetterPdfView extends AbstractPdfView {

    protected void buildPdfDocument(Map model, Document doc,
        PdfWriter writer, HttpServletRequest req,
        HttpServletResponse resp) throws Exception {
        Customer c = (Customer) model.get("customer");
        doc.add(new Paragraph("Dear " + c.getFirstName() + " "
            + c.getLastName() + " :"));
        doc.add(new Paragraph(" "));
        doc.add(new Paragraph(
            "We would like to welcome you to Spring development."));
        doc.add(new Paragraph(" "));
        doc.add(new Paragraph("Sincerely,\nIntertech, Inc.));
    }
}

```

- Again, note that the view rendering method (`buildPdfDocument`) returns void.
- Simply add content to the already provided Document object.
- **Here is the sample output produced by the view above.**



- **As with the other non-HTML views, all that is required is to register this PDF-producing view bean in the Spring configuration file.**
  - The BeanNameViewResolver is again responsible for calling on the PDF-producing view bean.
  - As with the Excel document requests, you may want to use request URLs that end with .pdf when working with the AbstractPdfView beans.
  - This helps the user (and some browsers) better understand what is returned.



### Lab Exercise – Non-HTML Views Lab

Do Not Print

## Chapter Summary

---

- **In the Form Tag Library, a set of Spring Web MVC command bean aware tags are provided to bind JSP form elements to command model properties.**
  - In JSP pages, the Spring form tags replace the HTML form elements. The `<form:form>` tag renders an HTML form.
  - All other form tag elements must be nested within a `<form:form>` tag.
  - When a form is bound to a command bean by name, the form data and command bean properties are inexorably tied to each other.
  - Each of the nested form tags has a path attribute that specifies the “path” to command bean property for data binding.
  - Most of the form tags resemble their HTML element counterparts in attribute offerings.
- **When building Web applications, using hardcoded labels and other text messages (like error messages) in the templates is usually unacceptable.**
  - To address this issue, Spring provides a custom tag to externalize all template text. The Spring “placeholder” tag for text is `<spring:message>`.
  - The message file contains a set of key-value pairs. The key in the file is the code as seen in the `<spring:message>` tags.
  - The value in the file is the text that Spring should use to replace the tag at runtime.
  - The language/locale code sent by the browser is used to select the right message file and replace the message tag with the appropriate text.
- **The `BeanNameViewResolver` looks for a view bean with the name that matches the logical view name it has been given (in the `ModelAndView`).**
  - The bean that the `BeanNameViewResolver` calls must implement the `View` interface, and it must render the next view.
  - However, the view bean does not have to render the view via template like a JSP.
  - In fact, the `BeanNameViewResolver` is often used to call on view beans that create non-HTML output.
  - Spring provides several convenience classes that implement the `View` interface to render non-HTML output like PDF documents and Excel spreadsheets.

## Chapter 6

### Formatting

#### *Objectives:*

- Learn about Spring's data formatting capability.
- Explore Spring's formatting annotations.
- Look at how to build custom formatting.

Do Not Print



## Chapter Overview

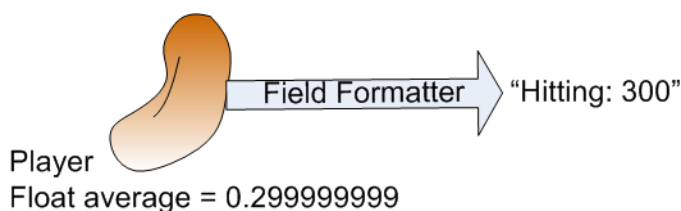
Displaying data from beans (POJOs) often requires a lot of custom code. Something as simple as displaying a double that represents money requires a lot of work. You need to make sure the number displays with the correct currency symbol and shows the correct number of decimal places.

In this chapter, you will learn about Spring's field formatting capability. You will learn how it can assist with burdensome formatting activity. Out of the box, Spring provides annotations that can assist in simple number and date/time display. Additionally, you can add your own "formatters" to consistently display any data type.

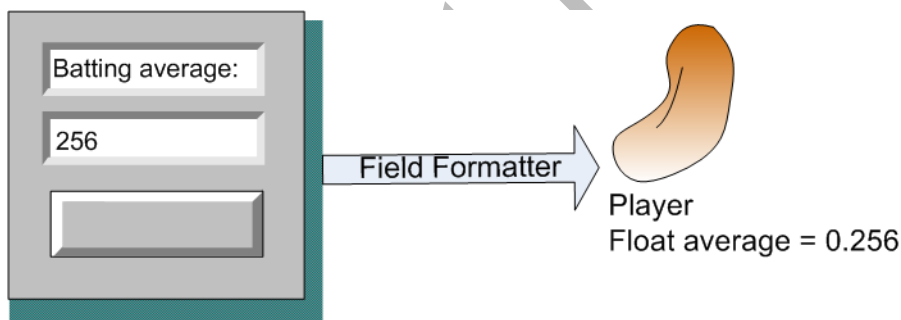
Do Not Print

## Formatting Annotations

- **Spring 3.0 provided a field formatting system.**
  - Use field formatting to render bean property data.
  - In particular, the rendering capability is used in user interfaces and report generation. Sometimes, the field data must be localized before being displayed.



- Also, use field formatting to help parse and convert String data into bean fields of a variety of types.
- This data may be coming from user interface entry.
- In this way, the field formatting offers an alternative to the type conversion system and PropertyEditors.



- **So, when should you use the Converter and when should you use the formatting system?**
  - The Spring documentation suggests the following:

In general, use the Converter SPI when you need to implement general-purpose type conversion logic; for example, for converting between a `java.util.Date` and `java.lang.Long`. Use the Formatter SPI when you're working in a client environment, such as a web application, and need to parse and print localized field values.<sup>1</sup>

<sup>1</sup> <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#format>

- **Out of the box, Spring 3.0 offers two annotations that you can use to format numbers, date, and time bean properties and method parameters.**
  - `@NumberFormat` formats any numeric primitive or `java.lang.Number` instance.
  - This includes `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long` and `Short`.
  - `@DateTimeFormat` formats any `java.util.Date`, `java.util.Calendar`, `java.util.Long` (representing a date/time in milliseconds) or `JodaTime`.
  - `JodaTime` is an open source package that provides a “quality replacement for the Java date and time classes.”<sup>2</sup>
  - Both are found in the `org.springframework.format.annotation` package as part of the `org.springframework.context-X.RELEASE.jar`.
  - Which means the Formatting SPI can be used in your application as long as the `spring-context` dependency is part of your `pom.xml`.

---

<sup>2</sup> <http://joda-time.sourceforge.net/>

- To specify the formatting of any numeric or date/time property in your Spring bean class, add `@NumberFormat` or `@DateTimeFormat` to the definition.
- For example, here is the Customer class with formatting annotations on date of birth (dob) and salary properties.

```
public class Customer {
    private String firstName;
    private String lastName;
    private String status;
    private int age;
    @DateTimeFormat(style = "M-")
    private java.util.Date dob;
    @NumberFormat(style = Style.CURRENCY)
    private double salary;
    ...
}
```

- Below are display screens of a Customer before and after adding these formatting annotations.

First name yes:

Last name:

Marital status:

Age:

DOB:

Salary:

First name yes:

Last name:

Marital status:

Age:

DOB:

Salary:

- On the left is the unformatted Customer. On the right is the annotation formatted Customer.
- Annotation attributes affect how both the `@NumberFormat` and `@DateTimeFormat` annotations work.

- The **@NumberFormat** annotation has two optional attributes: **style** and **pattern**.
  - The style attribute allows you to select from one of three `org.springframework.format.annotation.NumberFormat.Style` enums.
  - The table below outlines the three choices.

NumberFormat.Style Enums	Default
NUMBER	Yes
CURRENCY	No
PERCENT	No

- Use the pattern attribute to specify a custom pattern for numbers.
- The pattern follows Java's standard numeric formatting patterns.<sup>3</sup>
- By default, an empty string defines the pattern (suggesting no pattern is to be applied).
- As an example, the pattern below formats a test number as "0012.380".
- This is because the 0 character specifies showing leading and trailing zeros.

```
@NumberFormat(pattern = "0000.000")
private BigDecimal test = 12.38;
```

- Using this next format, a test number would display as "12.4". The # character specifies to round up.

```
@NumberFormat(pattern = "####.#")
private BigDecimal test = 12.38;
```

<sup>3</sup> <http://download.oracle.com/javase/tutorial/i18n/format/numberintro.html>

- **The @DateTimeFormat annotation also has optional attributes: style, pattern and iso.**
  - The style attribute allows you to provide a two-character string that dictates how the date and time should be formatted.
  - The first character dictates the date formatting, and the second dictates time formatting.
  - The table below provides a listing of the options and shows you example output.

DateTimeFormat Style	Two-character String	Example Output
Short form	SS	8/30/64 11:24 AM
Medium form	MM	Aug 30, 1964 11:24:41 AM
Long form	LL	August 30, 1964 11:24:41 AM CDT
Full form	FF	Sunday, August 30, 1964 11:24:41 AM CDT
Use a dash to omit the date or time. (Omitting time in this example, with medium format for the date.)	M-	Aug 30, 1964

- **The pattern attribute allows you to use a custom date/time pattern string to format the date and/or time.**
  - The pattern string follows the Java standard date/time formatting.<sup>4</sup>
  - Again, by default, the pattern string is empty. No formatting occurs when the pattern is an empty string.

<sup>4</sup> <http://download.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html>

- **The ISO attribute allows you to specify one of four ISO standard date/time format patterns.**
  - The ISO attribute patterns are represented by the `org.springframework.format.annotation.DateTimeFormat.ISO` enums.
  - The table below shows the ISO enum options and example output.

DateTimeFormat.ISO Enums	Example output
DATE	2010-04-22
DATE_TIME	2010-04-22T11:27:12.940-05:00
TIME	11:27:12.940-05:00
NONE (the default)	Ignore the attribute

- **To use the formatting annotations, you must also inform Spring to be on the lookout for your formatting annotations.**
  - Add the annotation-driven element to your Spring bean XML configuration file.

```
<mvc:annotation-driven/>
```

- If you use Joda to type date/time properties, you must also place the Joda JAR file on the classpath.
- Add the Joda time dependency to your Maven pom.xml.

```
<dependency>
  <groupId>joda-time</groupId>
  <artifactId>joda-time</artifactId>
  <version>2.3</version>
</dependency>
```

## Custom Formatting

---

- **If you need additional formatting or do not like the formatting Spring provides, you can add your own custom formatting.**
- **At the heart of the formatting system is the `Formatter` interface.**
  - The `Formatter` interface is typed (`T` below) to allow you to display or parse Strings from your designated target type.

```
public interface Formatter<T> extends Printer<T>, Parser<T> { }
```

- Even the formatting annotations shown above use this interface.
- **This interface requires all implementers to provide `parse()` and `print()` methods.**
  - As the names of these methods suggest, the `parse` method takes a String and produces an instance of the target type.
  - The `print` method, on the other hand, takes an instance of the target type and produces a String representation of the target type object.
  - Both methods also take a `Locale` parameter so that the String (whether input or output) is localized to the user's preference.




- A **CustomerSSN** property is added to the **Customer** bean class above in order to demonstrate custom formatting.

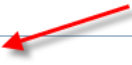
```
public class Customer {
    ...
    private CustomerSSN ssn = new CustomerSSN();
    ...
}
```

- You saw the **CustomerSSN** type earlier in class. Recall this object holds the three parts to a SSN number.

```
public class CustomerSSN {
    private int area; //first 3
    private int group; //middle 2
    private int serial; // last 4
    ...
}
```

- Trying to display **CustomerSSN** in an HTML form at this time (with no formatter or custom displaying code) would result in a less than acceptable display.

First name yes:	<input type="text" value="Jim"/>
Last name:	<input type="text" value="White"/>
Marital status:	<input type="text" value="Single"/> 
Age:	<input type="text" value="22"/>
DOB:	<input type="text" value="Mar 20, 1970"/>
SSN:	<input type="text" value="com.intertech.mvc.CustomerSSN@60029d"/>
Salary:	<input type="text" value="\$40,000.00"/>
<input type="button" value="Add"/>	



- A custom **Formatter** can allow the **CustomerSSN** to be displayed in the expected fashion (###-##-####) when a **Customer** object is displayed in a form.
- It can also help convert a **String** of the same format entered into an HTML form to a **CustomerSSN** object when submitted.

- **Here is an example custom Formatter for CustomerSSN (CustomerSSN serving as the example formatter's target type).**

```
import java.text.ParseException;
import java.util.Locale;
import org.springframework.format.Formatter;

public class SSNFormatter implements Formatter<CustomerSSN> {

    public String print(CustomerSSN ssn, Locale locale) {
        return ssn.getArea() + "-" +
            ssn.getGroup() + "-" + ssn.getSerial();
    }

    public CustomerSSN parse(String source, Locale locale)
        throws ParseException {
        int area = Integer.parseInt(source.substring(0, 3));
        int group = Integer.parseInt(source.substring(4, 6));
        int serial = Integer.parseInt(source.substring(7, 11));
        return new CustomerSSN(area, group, serial);
    }
}
```

- **In order to use Formatters, they must be registered at runtime with a FormatterRegistry (org.springframework.format).**
  - Extend the FormattingConversionServiceFactoryBean to register your Formatters with a FormatterRegistry.
  - The FormattingConversionServiceFactoryBean is in org.springframework.format.support.
- **Below, a new class extending FormattingConversionServiceFactoryBean registers the SSNFormatter with the FormatterRegistry.**

```
public class MyFormattingFactory extends
    FormattingConversionServiceFactoryBean {

    public void installFormatters(FormatterRegistry registry) {
        super.installFormatters(registry);
        registry.addFormatterForFieldType(CustomerSSN.class,
            new SSNFormatter());
    }
}
```

- **By default, the `<mvc:annotation-driven />` configuration element automatically registers Spring formatters and converters for common types.**

```
<mvc:annotation-driven />
```

- This is what allows the `@NumberFormat` and `@DateTimeFormat` to work.
- You need to alert Spring to use your `FormattingConversionServiceFactoryBean` to register your custom formatters.
- To do this, add a `conversion-service` attribute on the `<mvc:annotation-driven />` element that refers to your `FormattingConversionServiceFactoryBean`.

```
<bean id="myFormattingFactory"
class="com.intertech.mvc.MyFormattingFactory" />
<mvc:annotation-driven conversion-service="myFormattingFactory" />
```

- **The terminology and configuration here may seem a little goofy.**
  - In fact, `FormatterRegistry` is an interface, but `FormattingConversionService` is an implementation of `FormatterRegistry`.
  - Spring's `FormattingConversionService` installs default formatters (like `@NumberFormat` and `@DateTimeFormat`).
  - The `<mvc:annotation-driven />` element implicitly builds Spring's `FormattingConversionService` under the covers.
  - A `FormattingConversionServiceFactoryBean` is a factory for `FormattingConversionService` beans.
  - A bean that extends `FormattingConversionServiceFactoryBean` builds a custom `FormattingConversionService` instance that internally registers custom formatters and converters.
  - In essence, you are replacing Spring's default `FormattingConversionService` bean with your own when you build the `MyFormattingFactory`.
  - The call in to `super.installFormatters(registry)` in the `installFormatters()` method of the `MyFormattingFactory` makes sure existing formatters and converters still work.

- **Spring's formatting API also allows you to build your own formatting annotations (like @NumberFormat and @DateTimeFormat).**
  - In addition to creating your own annotation, you need only implement an AnnotationFormatterFactory rather than implement a Formatter.
  - You also must register your AnnotationFormatterFactory with the FormatterRegistry using the FormattingConversionServiceFactoryBean.
  - The FormattingConversionServiceFactoryBean has an addFormatterForAnnotation( ) method for this purpose.



### Lab Exercise – Formatting Lab

Do Not Print

## Chapter Summary

---

- **Spring 3 also provided a field formatting system.**
  - Use field formatting to render bean property data.
  - In particular, the rendering capability is used in user interfaces and report generation.
  - Also, use field formatting to help parse and convert String data into bean fields of a variety of types.
- **Out of the box, Spring offers two annotations that you can use to format numbers, date, and time bean properties and method parameters.**
  - `@NumberFormat` formats any numeric primitive or `java.lang.Number` instance.
  - `@DateTimeFormat` formats any `java.util.Date`, `java.util.Calendar`, `java.util.Long` (representing a date/time in milliseconds) or Joda Time.
- **The `@NumberFormat` annotation has two optional attributes: style and pattern.**
  - The style attribute allows you to select from one of three `org.springframework.format.annotation.NumberFormat.Style` enums.
  - Use the pattern attribute to specify a custom pattern for numbers.
- **The `@DateTimeFormat` annotation also has optional attributes: style, pattern and iso.**
  - The style attribute allows you to provide a two-character string that dictates how the date and time should be formatted.
  - The pattern attribute allows you to use a custom date/time pattern string to format the date and/or time.
  - The ISO attribute allows you to specify one of four ISO standard date/time format patterns.
- **If you need additional formatting or do not like the formatting Spring provides, you can add your own custom formatting.**
  - At the heart of the formatting system is the `Formatter` interface.
  - This interface requires all implementers to provide `parse()` and `print()` methods.
  - Formatters must be registered at runtime with a `FormatterRegistry`.

## Appendix A

### Spring Web Flow

#### *Objectives:*

- **Explore Spring Web Flow – a Spring Web MVC extension to manage conversational Web interactions.**
- **See how to setup and configure Web Flow.**
- **Learn how to define a flow.**
- **Understand the different kinds of states in flow.**
- **See how events trigger transitions among flow states.**
- **Learn how variables can be defined and accessed in a flow.**
- **Examine how actions trigger process work in a flow.**

## Chapter Overview

Web applications are usually built to allow users to freely navigate a pile of information offered by a site or they are built to carefully walk a user through a series of data collecting steps in order to transact some business like the purchase of a book or completion of a loan application. It is this latter type of Web interaction that Web Flow is built for. Web Flow is an extension to Spring Web MVC to provide a sequence of steps as part of a conversational application

In this chapter, you learn what “flows” are and how they are constructed and managed using Web Flow. While the components of Spring Web MVC can certainly be used to build such interactive Web applications, you also learn why Web Flow might be a better choice.

Do Not Print

## Web Flow

---

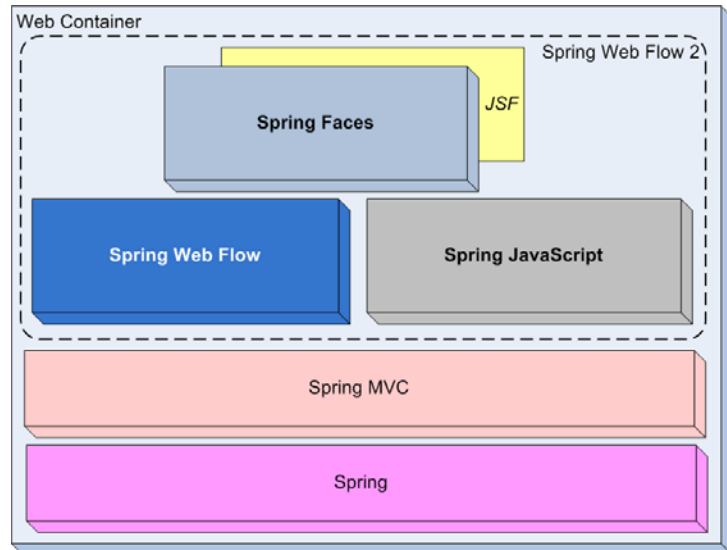
- **As Craig Wall author of Spring in Action puts it, there are essentially two types of interactions users have with Web applications.<sup>1</sup>**
  - One is very user driven. The user freely navigates around the Web application clicking on links, buttons, etc. in full control of the information they are presented.
  - While it is tough to completely label all applications, these applications tend to be more information providing types of applications.
  - Think WebMD.com or us.gov.
  - The second type of user interaction is one in which the application is more in control.
  - In this second type, the application holds a conversation with the user; asking questions and walking the user through a series of information gathering screens.
  - The information gathering triggers some functionality, which is usually toward some business result like purchasing a product or requesting a loan.
  - Think of your interactions to purchase a book on Amazon.com or requesting an insurance quote from an insurance company.
- **It is this second type of Web application interaction that Web Flow is meant to address.**

---

<sup>1</sup> Spring in Action, 2<sup>nd</sup> Edition (Manning) by Craig Walls, pg. 582.



- **Spring Web Flow is a framework for building conversational Web interactions that fit into a predefined flow.**
- Spring Web Flow (often referenced by the acronym SWF) is an extension to Spring and Spring Web MVC.
- In fact, SWF 2 (covered here) includes several modules.



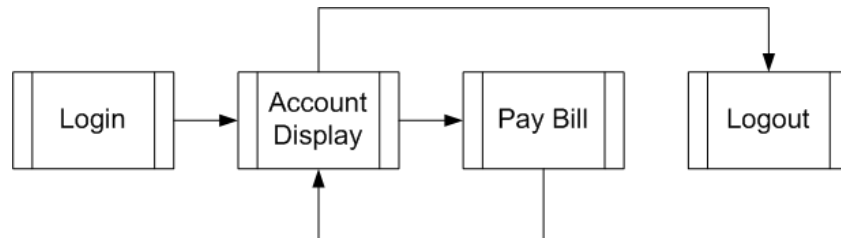
- The SWF module is an extension to Spring Web MVC and is the focus of this chapter.
- Spring JavaScript is a JavaScript abstraction framework. Its purpose is to make it easier to integrate JavaScript into Spring Web applications.
- This, in turn, allows for the integration of AJAX integration.
- Lastly, Spring Faces provides support for JavaServer Faces (JSF). JSF is an MVC framework defined by Java specification.

## Flows

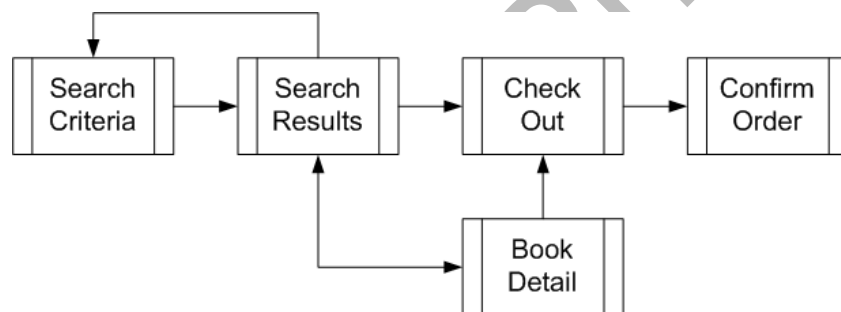
---

- **So what's a Web flow or flow for short?**

- Chances are good that you have encountered dozens of such flows in your software development career or at least in your use of the Internet.
- Do you use on-line banking to pay your bills? If so, does the general set of pages you go through look something like the flow shown below?



- Have you ever purchased a book on-line? If so, did the set of Web pages you went through to find the book and purchase it look like this flow?



- A flow is a sequence of steps as part of a conversational application.
- More precisely given the Web application context, a flow is the management of a user interaction requiring several requests/responses in order to complete.
- A flow is both navigation as well as control logic.
- Another way to put it is that a flow is the controlled navigation among a collection of Web pages to accomplish a user desired task.
- Web Flow should make creating and changing the flow easier.

- **Throughout this class, you have learned how to create Web applications using Spring Web MVC components.**
  - Controlling navigation was part of Spring Web MVC. So why the need for Web Flow?
  - In Spring Web MVC components, an application's navigational flow is scattered across controllers, controller configuration and JSPs (or views).
  - This makes it hard to see and modify the entire application's flow logic.
  - To get a complete picture of how a Web site works, a developer has to follow and study all the MVC components and how they link to one another.
- **SWF emphasizes loose coupling – again the term that is at the heart of all things Spring!**
  - Web Flow allows the entire navigation to be defined in a single and separate place.
  - The views (JSP pages), controllers, and other components do not contain navigation or flow information nor do they refer to each other.
  - Therefore, the MVC components are more loosely coupled and navigation control is more easily examined/modified in the larger context of the whole application.

## Web Flow Setup

---

- **The current version of SWF is 2.3. As of this writing, a version 2.4 is being constructed and is in milestone release form.**
  - SWF is a separate project in the Spring IO Platform (see <http://projects.spring.io/spring-webflow/>).
  - The Quick Start guide for the SWF recommends using a dependency management system to add SWF to your project.
  - Specifically, the guide provides Maven and Gradle build script snippets that can be added to your build for the required release of SWF.<sup>2</sup>
  - Given the number of modules and open source dependencies that SWF requires, this is often the best approach to getting SWF and its needed dependencies.
- **In this class, you study SWF 2.3. You will use Maven as your dependency management system in your labs.**
  - Therefore, the following XML will be added to the Maven pom.xml of your lab projects.

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-webflow</artifactId>
  <version>2.3.2.RELEASE</version>
</dependency>
```

- Note that SWF is identified by the spring-webflow artifact id – similar to all the other Spring modules/artifacts.
  - The spring-webflow module requires many Spring and Spring Web MVC modules as well, which are automatically brought in by the Maven build system.
  - In addition, the spring-webflow module requires 3<sup>rd</sup> party libraries (like commons logging), which are also brought in with the spring-webflow module.
- **As of SWF 2.3, a Java 5 or greater is required.**
  - SWF extends Spring and Spring Web MVC.
  - As such, Web Flow 2.3 requires Spring 3.0 or better (this includes Spring Web MVC).

---

<sup>2</sup> <http://projects.spring.io/spring-webflow/#quick-start>

- **While not covered in this appendix, SWF can be integrated with JSF (through Spring Faces).**
  - In order to integrate SWF with JSF, you need to declare spring-faces in your Maven dependencies.

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-faces</artifactId>
  <version>x.y.z.RELEASE</version>
</dependency>
```

- The spring-faces dependency will add spring-webflow, spring-js and all the other Spring and Spring Web MVC dependencies to the project.
- In this class, you explore SWF integration with Spring Web MVC.
- **A special Web Flow schema is provided to configure the Web Flow components in the Spring bean configuration file.**
  - Below is the schema namespace declaration needed to register and use the Web Flow bean definitions.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:webflow="http://www.springframework.org/schema/webflow-config"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/webflow-config
    http://www.springframework.org/schema/webflow-config/spring-
webflow-config-2.3.xsd">
  ...
</beans>
```

- Unfortunately, SWF has not yet adopted the version agnostic XSD convention as the rest of the Spring Framework has.
- Therefore, note the version of SWF in the namespace definition.
- More on flow definition is forthcoming.

- **Web traffic is directed to the Spring Web MVC DispatcherServlet just as it always is in a Spring Web MVC application – through servlet mapping in web.xml.**

```
<servlet>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/web-application-config.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
  <url-pattern>/*.request</url-pattern>
</servlet-mapping>
```

- Of course, you can also skip the creation of a web.xml file in a Spring 3.1 or better environment and running in a Servlet 3.0 or better Web container.
- In this case, implement the `WebApplicationInitializer` interface provided by Spring Web MVC.

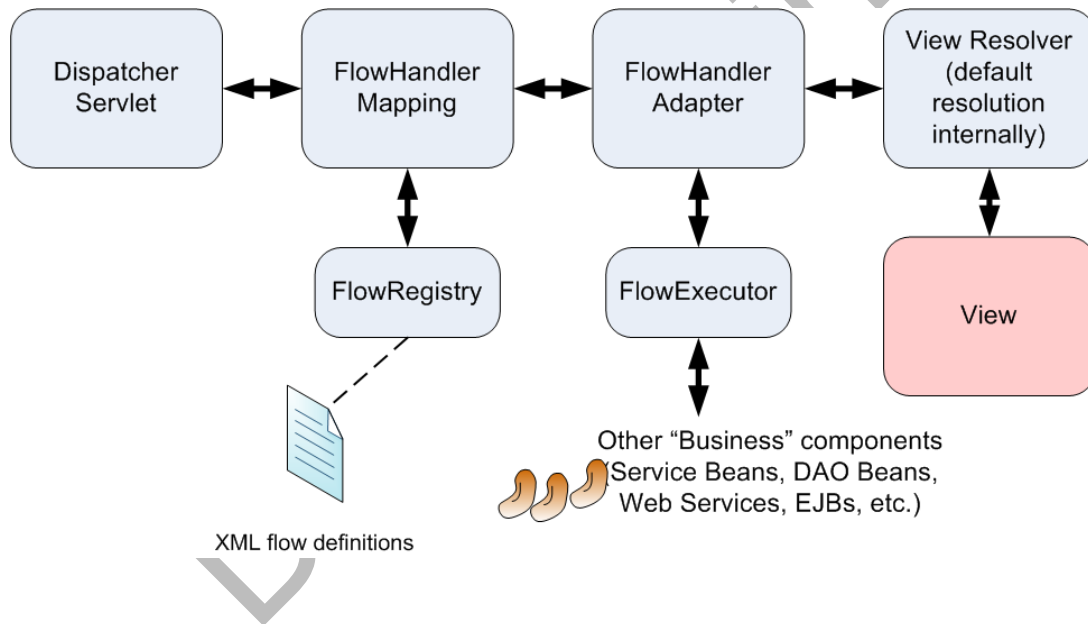
```
public class WebMVCApplicationInitializer implements
WebApplicationInitializer {
  public void onStartup(ServletContext container) {
    XmlWebApplicationContext appContext =
      new XmlWebApplicationContext();
    ServletRegistration.Dynamic registration =
      container.addServlet("dispatcher",
        new DispatcherServlet(appContext));
    registration.setLoadOnStartup(1);
    registration.addMapping("*.request");
  }
}
```

- **Next, add the FlowHandlerMapping and FlowRegistry beans to the Spring configuration.**
  - You learned to use @RequestMapping annotations to route traffic to controller handler methods in Spring Web MVC.
  - In SWF, you don't have to route traffic in the same way. Traffic routing is what Web Flow is all about.
  - Instead, you must register the FlowHandlerMapping bean.
  - It directs the application specific requests to the flows. For this, it needs to know where the flows are which is provided by a FlowRegistry.
  - The FlowRegistry is the container for flow definitions.
  - The flow definitions are often provided to the FlowRegistry via XML. In the example below, the flows are defined in /WEB-INF/flow/customer-flow.xml.

```
<bean class=
    "org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
    <property name="flowRegistry" ref="flowRegistry" />
</bean>
<webflow:flow-registry id="flowRegistry">
    <webflow:flow-location path="/WEB-INF/flows/customer-flow.xml" />
</webflow:flow-registry>
```

- **You also need to add the FlowHandlerAdapter and FlowExecutor beans to the Spring configuration.**
  - The FlowHandlerAdapter acts as the controller in the Web Flow application.
  - It manages the work around URL requests for the flows. However, the FlowHandlerAdapter doesn't really execute the flows. It serves as an orchestrator.
  - The FlowExecutor bean is the service that drives the execution of flow definitions and carries out the steps described in a flow (more on flow definitions in a bit).

```
<webflow:flow-executor id="flowExecutor" />
<bean class=
    "org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

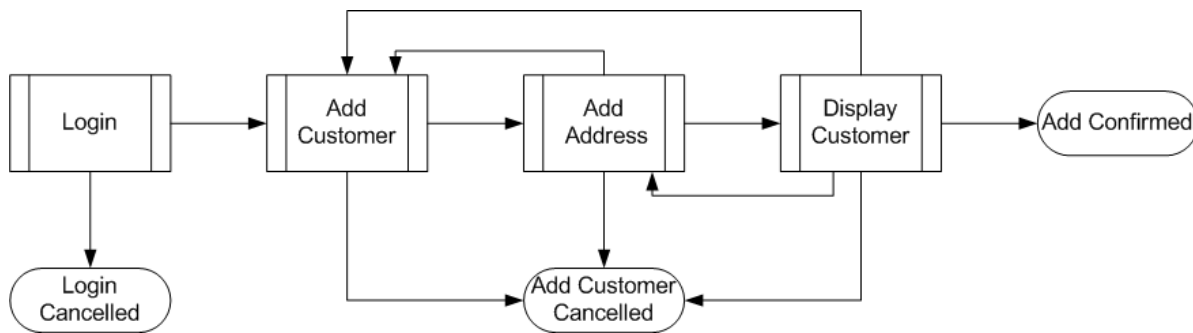




## Defining Flows

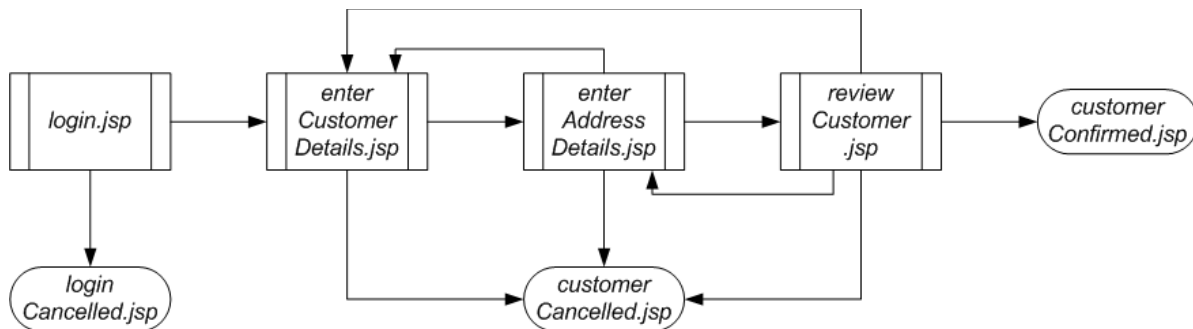
---

- **Suppose you were required to build an application that allowed users to add new customers to the system.**
  - In this hypothetical application, you might envision a set of form entry pages and steps that are outlined by the diagram below.

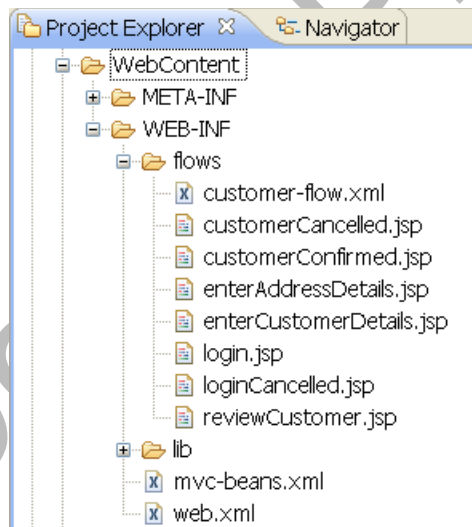


- After logging in, the user is presented with a form to enter the basic customer information; namely first and last name, age, etc.
- After submitting that form, the user is presented with the opportunity to provide the customer's address information; namely street, city, state, etc.
- Since address information is collected on lots of things (customers, orders, etc.) make this a separate and isolated step that can be reused in other flows.
- A customer display page allows the user to confirm the customer and customer address information has been correctly entered.
- Note that from any of the data gathering pages, the user should be able to get "back" to any of the form pages to enter or re-enter data.
- Of course, the user can at any time decide to cancel the request to add the customer or even decide not to login.
- These requests lead to pages indicating the cancellation of the process.
- This flow provides enough complexity to be able to see how flows are defined in XML and used in a SWF application.

- By convention, flow definitions are defined in XML files ending with "-flow.xml" in any subdirectory of /WEB-INF/flows.
- Also by convention, the view templates (views) that the flow renders and any other resources are stored in the same directory as the flow definition XML file.
- In this example, JSP templates are used for the views. The JSPs and their relationship to the states of the flow are shown below.



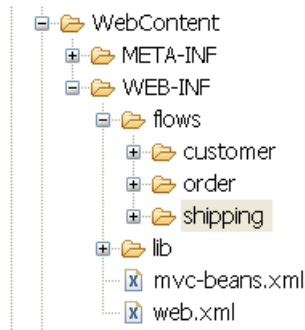
- Given convention, the flow definition file (customer-flow.xml) and all these templates are stored together under /WEB-INF/flows.



- Under this organization, and given the example DispatcherServlet mapping above, here is the HTTP request URL to start the flow.

```
http://[host][:port]/[Web app context]/customer-flow.request
```

- In more complex applications where several flows exist, subdirectories off /WEB-INF/flows (one for each flow) might be in order.



- XML flow definition files are defined by schema. The root element of a flow definition file is `<flow>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

</flow>
```

- **Inside of the flow definition are defined the states of the flow.**
  - A state represents a step or some activity in the flow. There are a few different kinds of states.
  - Perhaps the most important type of state in all of Web Flow is the view-state.
  - View-states represent a step in the flow that renders a view.
  - When a view-state is encountered, the system renders the view and waits for a user event to resume the flow.
  - View-states are defined in <view-state> elements in the flow.
  - Below, the view-states of the example application are defined.

```
<flow ...>
  <view-state id="login">
    ...
  </view-state>
  <view-state id="enterCustomerDetails">
    ...
  </view-state>
  <view-state id="enterAddressDetails">
    ...
  </view-state>
  <view-state id="reviewCustomer">
    ...
  </view-state>
  ...
</flow>
```

- **The first state in a flow is the start state where a flow begins. In this example, login is the start state.**

- **As you can probably see through the example, the id uniquely identifies the view-state element.**
  - By convention, the view-state id correlates to a view template in the directory where the flow is located.
  - For example, id="login" points the Web Flow engine to the view template login.jsp in the same folder as this flow definition file.
  - However, the view can be explicitly identified with the view attribute (shown below). This allows the id and view names to be different.

```
<view-state id="login" view="mylogin.jsp">
...
</view-state>
```

- The view can be identified by relative path (shown above) or absolute path in the Web application root folder (shown below).

```
<view-state id="login" view="/WEB-INF/views/auth/login.jsp">
...
</view-state>
```

- The view can also be identified by logical view name that is resolved by the Spring Web MVC framework (for example by a view resolver in the framework).
- **The <end-state> element signals the end of a flow.**
  - When a flow encounters an end-state, it terminates and returns the outcome of the flow.
  - There can be more than one end-state as defined for the example below.

```
<end-state id="loginCancelled" view="loginCancelled.jsp"/>
<end-state id="customerConfirmed" view="customerConfirmed.jsp" />
<end-state id="customerCancelled" view="customerCancelled.jsp"/>
```

- The view attribute may not always be provided in defining an end-state.
- The caller (perhaps, for example, some Spring Web MVC controller that kicks off the flow) may choose to handle what happens on return from the flow.
- When a view attribute is not provided, the “result view” is the responsibility of the caller based on what it sees in the flow results.
- When the end-state view is provided, it must be a flow-relative path (in the flow’s directory) to a view template which gets rendered to the user.

- **Getting between the view-states or to an end-state is accomplished through transitions.**
  - At the completion of any state, the state launches an event. Events are strings in Web Flow.
  - A transition handles the events thrown by a state.
  - Transitions are defined by <transition> elements in the flow definition.
  - Specifically, with regard to a view-state, they can be used to indicate the next view-state to “go to” or render.
  - However, not all states that transitions “go to” are view-states (more in a bit).
  - Put another way, transition elements drive the navigation between states.

```
<view-state id="login" view="mylogin.jsp">
  <transition on="login" to="enterCustomerDetails" />
  <transition on="cancel" to="loginCancelled" />
</view-state>
<view-state id="enterCustomerDetails">
  <transition on="submit" to="enterAddressDetails" />
  <transition on="cancel" to="customerCancelled" />
</view-state>
<view-state id="enterAddressDetails">
  <transition on="submit" to="reviewCustomer" />
  <transition on="cancel" to="customerCancelled" />
</view-state>
<view-state id="reviewCustomer">
  <transition on="confirm" to="customerConfirmed" />
  <transition on="revise" to="enterCustomerDetails" />
  <transition on="cancel" to="customerCancelled" />
</view-state>
<end-state id="loginCancelled" view="loginCancelled.jsp"/>
<end-state id="customerConfirmed" view="customerConfirmed.jsp" />
<end-state id="customerCancelled" view="customerCancelled.jsp"/>
```

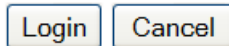
- The on attribute on a transition element specifies the event that triggers the transition.
- The to attribute on a transition indicates the next state to execute. In the case of a view-state, it is the next view to render when the event occurs.

## View Events

---

- **Events are triggered by pushing buttons or clicking on links in the view.**
- **When using Web Flow with Spring Web MVC, request parameters from the view carry the information that a button has been pressed.**
  - From an HTML form (or other template view), this means the name attribute of the submit buttons (<input type="submit"> must conform to certain standards.
  - The name attribute of buttons must begin with \_eventId\_.
  - The rest of the name attribute serves as the event id.
  - Below, the Login button sends an event id of "login" while the Cancel button throws an event id of "cancel".

```
<input type="submit" name="_eventId_login" value="Login"/>  
<input type="submit" name="_eventId_cancel" value="Cancel"/>
```



- **A hidden field can also be used to signal the event on submit.**
  - Use a submit button to submit the form, but use a hidden field to submit the event as a parameter.
  - The same standards must be applied to the hidden field's name attribute.
  - In the example below, the event id sent is "continue".

```
<input type="submit" value="Continue" />  
<input type="hidden" name="_eventId" value="continue" />
```

- Of course, using a hidden field for event submission can only be used when there is just one event that can be sent by the form.
- This is because only one field can have a name of "\_eventId".

- **An event can also be sent via HTML link.**

```
<a href="{flowExecutionUrl}&_eventId=revise">Revise Customer Info</a>
```

- Clicking on a link also results in issuing an HTTP request.
- The Web Flow engine actually looks for a parameter that begins with `_eventId`.
- However, when no `_eventId` parameter is found, the system looks for a parameter that starts with `_eventId_` and uses the remaining substring as the event id.
- If neither `_eventId` nor `_eventId_` are present, no flow event is triggered.
- So when clicking on this link, an `_eventId` parameter is sent. In this example, the event id is “revise”.

Do Not Print



## Variables

---

- **A flow can almost be thought of as a little program: a set of instructions to be executed by the SWF engine.**
  - Many of the instructions of the “program” are instructions to render a view.
  - However, as a kind of program, in addition to displaying views, it can execute work (coming up) and it can declare variables to be used throughout the flow.
  - Flow variables, or what is more precisely called flow instance variables, are allocated when a flow starts.
  - Flow variable must implement `java.io.Serializable` since the variable’s state is persisted by the Web Flow engine between states.
- **The first view of the example flow is a login view. A special flow variable class is created to capture the view’s data.**

```
public class User implements Serializable {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

- **Use the `<var>` element to create an instance of `User` to use throughout the flow.**

```
<var name="user" class="com.intertech.domain.User" />
```

- Variables are declared at the top of the flow.
- Typically, `@Autowired` is used to wire any dependent objects/values that the instance variables need.
- **The model attribute on a view-state can be used to declare a model object for the view.**
  - Data from the model object can be bound to the view allowing model data to be shown in the display.
  - As the view is submitted, the view data is placed back in the model.
  - The model object can be from any scope (more on scopes in a bit).
  - Instance variables are part of what is called flow scope.

```
<var name="user" class="com.intertech.domain.User" />
...
<view-state id="login" model="user">
  <transition on="login" to="authorizeUser" />
  <transition on="cancel" to="loginCancelled" />
</view-state>
```

- In this example, the `User` object is bound to the login view not unlike how command beans are bound to the views in Spring Web MVC.
- In fact, the `commandName` attribute can be used with the model when the view is provided via templates with the Spring Web MVC form tags.

- To provide some context to this binding, say a default value was supplied in the User's username property as shown below.

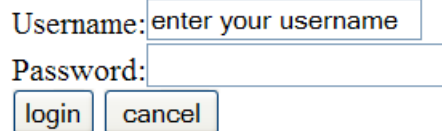
```
public class User implements Serializable {  
    private String username = "enter your username";  
    private String password;  
    ...  
}
```

- Now assume the code below provides the relevant parts of login.jsp view.

```
<body>  
Login  
<form:form method="post" commandName="user">  
Username:<form:input path="username" /><br />  
Password:<form:password path="password" /><br />  
<input type="submit" name="_eventId" value="login" />  
<input type="submit" name="_eventId" value="cancel" />  
</form:form>  
</body>
```

- The image here shows what the login view with bound User data looks like when the flow is requested.

Login



Username: enter your username

Password:

login cancel

- When the login button is pushed, whatever data was entered into the username form field is populated back into the username property of User model object.

## Actions

---

- **Throughout a flow, some work also needs to get done by the application.**
  - That is, as the user hits the “submit” button, it triggers activity on the server as much as it may trigger the display of the next view.
  - So while view navigation is nice, it is only part of what Web Flow has to offer.
- **In Web Flow, actions represent the invocation of business services.**
  - Actions are defined in expression language (more precisely Unified EL by default) as you would see in a JSP.
  - Actions, or action expressions, are most often seen in <evaluate> elements.

```
<evaluate expression="authenticationService.logout()" />
```

- **Actions can be executed at several points in a flow.**
  - Actions can be executed on flow start up and completion (end), state entry and exit, view rendering and on view transition.
  - The evaluate elements are typically child elements of an element indicating the state and context of when the action should be executed.
  - The evaluate element can be used to invoke methods on Spring beans (defined in Spring XML configuration or by annotations) or flow instance variables.
- **For example, the <on-start> element allows an action to be executed as the flow begins.**

```
<on-start>
  <evaluate expression="logService.startLogging()" />
</on-start>
```

- **The <on-end> element allows an action to be executed as the flow exits.**

```
<on-end>
  <evaluate expression="logService.stopLogging()" />
</on-end>
```

- The `<on-entry>` and `<on-exit>` elements can be used to trigger the evaluation of an action expression upon entering or exiting a state.
  - The example below shows a logout call as the `loggedOut` view-state is entered (executing before the view is actually rendered).

```
<view-state id="loggedOut">
  <on-entry>
    <evaluate expression="authenticationService.logout(user)" />
  </on-entry>
  <transition on="login" to="login" />
</view-state>
```

- In the example above, the expression uses a Spring service bean (`authenticationService` reference) and a flow variable (`user`).
- An action can also be called on transition between views as shown by this call to `cancelCustomerAdd` on the `customerService` when the `cancel` event occurs.

```
<view-state id="enterCustomerDetails">
  <transition on="submit" to="enterAddressDetails" />
  <transition on="cancel" to="customerCancelled">
    <evaluate
      expression="cusomterService.cancelCustomerAdd(customer)" />
  </transition>
</view-state>
```

- **The return value from any action evaluation can be captured.**
  - Use the result attribute on the <evaluate> element to capture the result from an action.

```
<var name="customer" class="com.intertech.domain.Customer" />
...
<view-state id="enterCustomerDetails">
  <on-entry>
    <evaluate expression="customerService.createDefaultCustomer()"
      result="flowScope.customer" />
  </on-entry>
  <transition on="submit" to="enterAddressDetails" />
  <transition on="cancel" to="customerCancelled"/>
</view-state>
```

- In this example, on entering the enterCustomerDetails view, a call is made to the CustomerService to create a default customer object (see service below).

```
@Service
public class CustomerService {

  ...

  public Customer createDefaultCustomer() {
    return new Customer("enter first name", "enter last name", 35);
  }
}
```

- The resulting Customer object is returned and stored in the flow's customer instance variable.

- Notice the “**flowScope**” reference in the return result attribute above.
  - All variables (and their data) in a flow belong to a scope and there are several scopes.
  - Flow instance variables are part of the “flow scope”.
  - The expression language provides implicit variables to access the scopes and all of the scope’s variables.
  - The table below lists some of the important scopes and implicit variables to access the scopes.

Implicit Variable	Scope data and lifecycle
flowScope	Scope for all flow instance variables. Allocated when a flow starts and is destroyed when the flow ends.
viewScope	Allocated when a view-state is entered and destroyed when the state exits.
requestScope	Access to request variables. Allocated when a flow is called and destroyed when the flow responds (returns).
flashScope	Allocated when a flow starts and cleared after every view is rendered and destroyed when the flow ends.
conversationScope	Used with nested flows. Allocated when a top-level flow starts and destroyed when the top-level flow ends.
requestParameters	Access to request parameters.
currentEvent	Access to the attributes of the current flow event.
currentUser	Access to the current authenticated Principal (see Java EE security).

- Take particular note of when the scope is created/destroyed and some of the information that can be accessed through the implicit variable.

- So in the example, the result data is stored to the flowScope, but it could just as easily have been stored to any one of these other scopes.

```
<on-entry>
  <evaluate expression="customerService.createDefaultCustomer()"
    result="viewScope.customer" />
</on-entry>
```

- When assigning a variable to one of the flow scopes, the scope must be explicitly referenced.
- When referencing a variable to retrieve data from a flow, use of explicit scope is optional.
- The scopes are searched in order of request, flash, view, flow, and conversational.
- Scopes are searched in that order when no explicit scope is used when referencing a variable.



## Action and Decision States

---

- **View-states are just one type of state in a flow.**
- **Action-states are kind of combination of action and transitions.**
  - Actions-states include an action (<evaluate> element) which returns a result.
  - Based on the return result of the action, a transition is made to another state.
- **In this example, once the login view to collect the username and password has been submitted, the action-state calls on an action.**

```
<view-state id="login" model="user">
  <transition on="login" to="authorizeUser" />
  <transition on="cancel" to="loginCancelled" />
</view-state>
<action-state id="authorizeUser">
  <evaluate expression="authenticationService.login(user)" />
  <transition on="yes" to="enterCustomerDetails" />
  <transition on="no" to="loginDenied" />
</action-state>
```

- The action calls the login method of the authenticationService to check the user's credentials.

- Below is a simple example **AuthenticationService** to show how the action-state works.

```
@Service("authenticationService")
public class AuthenticationService {

    public boolean login(User user) {
        if (user.getUsername().equals("Jim")){
            return false;
        } else {
            return true;
        }
    }
    ...
}
```

- The action returns a true or false boolean. These values are interpreted as yes (true) or no (false) by the action-state.
- When the login method returns true (yes – the user is authorized) then the “yes” transition element sends the flow to the “enterCustomerDetail” state.
- If the login method returns false (no – the user is not authorized) then the “no” transition element sends the flow to the “loginDenied” state.
- The return result from an action in an action-state does not have to be a boolean.**
  - For example, the action method (login) could return “pass” and “fail” Strings.

```
<action-state id="authorizeUser">
    <evaluate expression="authenticationService.login(user)" />
    <transition on="pass" to="enterCustomerDetails" />
    <transition on="fail" to="loginDenied" />
</action-state>
```

- This table provides a mapping between possible action results and how they are interpreted by the action-state.

Action return type	Action-state interpretation
boolean	true = yes, false = no
String	String value
Enum	Enum name
Any other result	success (yes)

- Decision-states are similar to action-states.**

- A decision-state allows the flow to branch based on a conditional.
- The decision-state uses an if/then/else conditional to determine the next state.
- **Assuming the `authenticationService.login( )` method returns a boolean, the same authorization check in the flow can be made with a decision-state.**

```
<decision-state id="authorizeUser">
  <if test="authenticationService.login(user)"
    then="enterCustomerDetails" else="loginDenied"/>
</decision-state>
```

Do Not Print

## Validation

---

- **SWF supports validation of model data – both programmatic and declarative validation with JSR-303 Bean Validation Annotations.**
  - When provided with validation logic, model data is validated automatically on transition between states.
  - Validation logic for the model can be provided in a couple of ways.
- **Programmatic validation logic can be embedded in model object.**
  - Name the validation method “validate” + the view-state id.
  - The validateLogin method in User below gets called ...

```
public class User implements Serializable {
    private String username = "enter your username";
    private String password;

    ...

    public void validateLogin(ValidationContext context) {
        MessageContext messages = context.getMessageContext();
        if ((username == null) || (username.length() < 1)) {
            messages.addMessage(new
                MessageBuilder().error().source("username")
                    .defaultText("Username must be provided").build());
        }
        if ((password == null) || (password.length() < 1)) {
            messages.addMessage(new
                MessageBuilder().error().source("password")
                    .defaultText("Password must be provided").build());
        }
    }
}
```

- ... on the either transition from the “login” view-state depicted in the flow element below.

```
<view-state id="login" model="user">
    <transition on="login" to="authorizeUser" />
    <transition on="cancel" to="loginCancelled" />
</view-state>
```

- **The MessageContext (org.springframework.binding.message) from the Web Flow Binding library allows you to record messages.**
  - The messages are used by the <form:errors> tag to display validation issues to the user.
- **The second option for providing programmatic validation logic is to build a “Validator” component.**
  - When building a Validator class, name the class the [upper case model expression name] + “Validator”.
  - In the case of the User, the name of the validator must be “UserValidator”.
  - Use the @Component stereotype annotation to mark it as a component to be picked up in the component scan.
  - The validator must have a public method with the name “validate” + [view-state id]. This is just like the method name of the model embedded validation logic.

```
@Component
public class UserValidator {
    public void validateLogin(User user, ValidationContext context) {
        MessageContext messages = context.getMessageContext();
        if ((user.getUsername() == null) ||
            (user.getUsername().length() < 1)) {
            messages.addMessage(new
                MessageBuilder().error().source("username")
                    .defaultText("Username must be provided").build());
        }
        if ((user.getPassword() == null) ||
            (user.getPassword().length() < 1)) {
            messages.addMessage(new
                MessageBuilder().error().source("password")
                    .defaultText("Password must be provided").build());
        }
    }
}
```

- **Using a Validator, the validation method can also accept a Spring Web MVC Errors object.**
  - The Errors object may be more familiar and friendly to use (usually requiring less code), especially to Spring Web MVC developers.
  - The Errors object also allows ValidationUtils to be used to add to the Errors object.

```
@Component
public class UserValidator {
    public void validateLogin(User user, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "username",
            "usernameRequired");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password",
            "passwordRequired");
    }
}
```

- **As indicated, the validation logic triggers on transition - automatically.**
- **Use the validate="false" attribute to suppress validation on any transition.**

```
<view-state id="login" model="user">
    <transition on="login" to="authorizeUser" />
    <transition on="cancel" to="loginCancelled" validate="false"/>
</view-state>
```

- **JSR-303 Bean Validation is supported with SWF 2.3 or later.**
  - To enable JSR-303 validation, configure the flow-builder-services with a Spring Web MVC LocalValidatorFactoryBean.

```
<webflow:flow-registry id="flowRegistry"
    flow-builder-services="flowBuilderServices">
    <webflow:flow-location path="/WEB-INF/flows/customer-flow.xml" />
</webflow:flow-registry>
<webflow:flow-builder-services id="flowBuilderServices"
    validator="validator" />
<bean id="validator" class=
    "org.springframework.validation.beanvalidation.
    LocalValidatorFactoryBean" />
```

- Validation will be applied to all model attributes after data binding.

## More with Web Flow

---

- **SWF is a large framework.**
- **In addition to the basics covered here, there is additional flow functionality.**
  - Flows can inherit from another flow.
  - A SWF provided Action interface and MultiAction class can be extended to provide better type safety, exception handling, etc.
  - Flows can have input and output attributes.
  - Flows can have subflows.
  - Views can be rendered in a popup.
- **SWF integrates well with other technologies to assist in several other aspects of Web development.**
  - Flows can be tied to persistent management.
  - Flows can create, commit and close persistence context.
  - Web Flow integrates with both Hibernate and JPA persistence technologies.
  - Flows can be secured using Spring Security.
- **Of course, the SWF architecture also provides ...**
  - JSF integration
  - JavaScript and Ajax integration
  - A version of SWF for portlets.



### Lab Exercise – Web Flow Lab

## Chapter Summary

---

- **SWF is a framework for building conversational Web interactions that fit into a predefined flow.**
  - Web Flow allows the entire navigation to be defined in a single and separate place.
  - The views (JSP pages), controllers, and other components do not contain navigation or flow information nor do they refer to each other.
  - Therefore, the MVC components are more loosely coupled and navigation control is more easily examined/modified in the larger context of the whole application.
- **A flow is a sequence of steps as part of a conversational application.**
  - More precisely given the Web application context, a flow is the management of a user interaction requiring several requests/responses in order to complete.
  - A flow is both navigation as well as control logic.
- **By convention, flow definitions are defined in XML files ending with "-flow.xml" in any subdirectory of /WEB-INF/flows.**
  - Inside of the flow definition are defined the states of the flow.
  - A state represents a step or some activity in the flow. There are a few different kinds of states.
  - Perhaps the most important type of state in all of Web Flow is the view-state.
  - View-states represent a step in the flow that renders a view.
  - Getting between the view-states or to an end-state is accomplished through transitions.
  - Use the <var> element to create flow instance variables.
  - Throughout a flow, some work also needs to get done by the application. Actions represent the invocation of business services.
  - Actions are defined in expression language (more precisely Unified EL by default) as you would see in a JSP.
  - Actions, or action expressions, are most often seen in <evaluate> elements.
  - Action-states and decision-states are a kind of combination of action and transitions.



## Appendix B

### Spring Security

#### *Objectives:*

- Understand what the Spring Security framework provides.
- See how to obtain and configure Spring Security.
- Learn how to secure Web applications using Spring Security.
- Learn how to secure any service (any bean) using method security.

## Chapter Overview

Spring Security is Spring's framework for providing authentication and authorization services. While it is a separate Spring project, it is built on top of the Spring container and often utilizes several Spring features such as AOP and data access.

Spring Security provides a rich API and it can be used to secure all types of applications and components. In particular it can be used to secure Web applications or method-level access to any Spring bean.

Do Not Print

## Spring Security

---

- **Spring Security (SS) is a separate, but very popular, Spring project in the Spring IO Platform.**
  - Meaning, SS is not part of the core framework.
  - It requires additional configuration and additional modules/JAR files be added to your project.
  - It is also one of the more “mature” Spring projects. It was started in 2003.
  - It was originally called Acegi Security.
  - It is open source (available through Apache license) and maintained by Pivotal today (as part of the Spring IO Platform).
  - As of this writing, version 3.2 is the latest version of the SS.
- **What does SS provide?**
  - SS provides the two main security services required by most applications: authentication and authorization.
  - Authentication is the ability to identify a user or more generically a principal.
  - A principal might be a human user, a device, or some other system.
  - Authentication systems try to answer two questions for the application: “Who is the principal?” and “Is the principal really who he/she/it represents themselves to be?”
  - Authorization is also known as access control.
  - Authorization determines what resources a principal can access and what actions or operations a principal can perform on a system.
  - In other words, authorization systems try to answer a single question: “What can the principal do?”
  - For example, can a principal see customers in the database (access)? Can a principal add or remove customers from that same database (operations)?
  - Authentication and authorization are linked in that authorization systems depend on identifying a principal in order to prevent unauthorized access or actions.

- **Most people think “user login” when they think about authentication.**
  - However, more generally, authentication is about an exchange of a shared secret between the system and principal.
  - That is, the exchange of information that is only supposed to be known and available to the system and the principal trying to identify themselves to the system.
  - The shared secret is often thought of as a password in a user login situation.
  - However, authentication systems may work off other information mechanisms.
  - Physical property of the individual (fingerprint, retinal vascularization pattern, etc.), or derived property (smartcard) might otherwise provide the shared secret.
  - In order to verify the identity of a principal, the authenticating system typically challenges the user to provide their unique information.
  - “Provide your username and password” is the most often seen authentication challenge.
  - If the authenticating system can verify that the shared secret was presented correctly, the principal is considered authenticated.

## Getting and Configuring SS

---

- **Spring IO recommends the use of a dependency management system to get all the required JAR files and modules for any Spring IO project.**
  - In this appendix, you study Spring Security 3.2 and you use Maven as your dependency management system in your labs.
  - Therefore, the following XML will be added to the Maven pom.xml of your lab projects.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>3.2.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>3.2.0.RELEASE</version>
</dependency>
```

- Note the artifactId follows names the Spring module naming convention (spring-\*).
- The spring-security-web and spring-security-config modules require many Spring modules as well, which are automatically brought in by the Maven build system.
- **As of SS 3.2, two options exist for configuring your application to leverage SS.**
- **The first approach to leverage SS is to use namespace configuration.**
  - Namespace configuration is provided through the spring-security-config.jar file.
  - Spring Security Namespace configuration has been available since Spring Security 2.0.
  - It allows developers to add special SS elements to the Spring configuration to define security needs for your application.
  - These elements often conceal the fact that multiple beans or processing steps are added to the application context under the covers.
  - The alternative is to define and configure all the underlying SS beans.

- To use the security namespace in your application, add the SS schema to the top of your Spring bean configuration file(s).

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:security="http://www.springframework.org/schema/security"
  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.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-
security.xsd">
  ...
</beans>
```

- The second approach is to use Java Configuration to configure SS.
  - Create a class that extends `WebSecurityConfigurerAdapter`.
  - Annotate the class with `@EnableWebSecurity` (in addition to `@Configuration`).
  - A basic example is shown below.

```
@Configuration
@EnableWebSecurity
public class MySecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
    throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("password").roles("USER");
    }
}
```

- **The SS project is a large, powerful and very extendible framework.**
  - A rich set of interfaces and classes exist to provide all sorts of authentication and authorization services.
  - Most applications, however, simply need to authenticate and authorize user's access to a Web application/site.
  - Web/HTTP Security is provided via SS provided servlet filters and associated beans.
  - Additionally, especially for non-Web applications, applications need to authenticate and authorize access to the service layer components.
  - This later use case is accomplished by providing authentication and authorization around backend/service component method calls.
  - SS secures the service layer (what is called Method Security) via Aspect Oriented Programming.

Do Not Print

## Web/HTTP Security – Security Filter Chain

---

- **SS uses filters to intercept incoming Web/HTTP request traffic.**
  - Therefore, the first step in using SS for Web applications is to register SS's DelegatingFilterProxy in your web.xml file.

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- The DelegatingFilterProxy is a SS framework object that delegates to a SS framework bean.
- Behind the scenes, the registration of this element causes a FilterChainProxy bean named "springSecurityFilterChain" to be created in the container.
- **In Servlet 3.0 or better environments, the filter can be registered programmatically.**
  - The code below does the equivalent work of the elements in web.xml.

```
public class SecurityWebApplicationInitializer
  extends AbstractSecurityWebApplicationInitializer {
}
```

- This initializer automatically registers the springSecurityFilterChain for every URL in your application.
- It should be noted that this initializer class configuration is used for applications using Spring MVC.
- A slightly different initialization is needed if you are not already using Spring or Spring MVC in your application. See the documentation for further details.<sup>1</sup>

---

<sup>1</sup> <http://docs.spring.io/spring-security/site/docs/3.2.0.RELEASE/reference/htmlsingle/#abstractsecuritywebapplicationinitializer-without-existing-spring>



## Namespace Web Security Configuration

---

- **Using namespace configuration, you configure your Web security needs via the <security:http> element in your Spring beans configuration file.**
  - Here is a simple starting point for securing your Web application.

```
<security:http auto-config="true">
  <security:intercept-url pattern="/**" access="ROLE_USER" />
</security:http>
```

- The <http> element encapsulates the security configuration for your Web application.
- **The <intercept-url> element defines a pattern which is matched against the incoming request URLs.**
  - Ant path or regular-expressions can be used to define the matching URLs.
  - The access attribute on defines the access requirements.
  - In this example (which is also the default configuration), the access attribute is a comma separated list of roles.
  - Users that authenticate must have one of the role authorities listed in the access attribute to be given access.
  - SS is not limited to use of simple roles. More complex configuration allows for different types of access.

- **In the <http> element, there can be multiple <intercept-url> elements.**
  - Each defines different access requirements for different resources (as defined by the set of URLs).
  - The list of <intercept-url> elements will be evaluated in the order listed. The first match will be used.
  - Therefore, put the most specific matches at the top.
  - You can also add a method attribute to limit the match to a particular HTTP method (GET, POST, PUT etc.).

```
<security:http pattern="/index.html" security="none" />
<security:http pattern="/*.jsp" security="none" />
<security:http auto-config="true">
  <security:intercept-url pattern="/*.request" access="ROLE_USER" />
</security:http>
```

- See the link below for more details on the request pattern matching.

```
http://docs.spring.io/spring-
security/site/docs/3.2.0.RELEASE/reference/htmlsingle/#request-
matching
```

- **With your access rules in place, you now need to give SS details on the users authentication and the access (roles) they are assigned.**
  - This is defined in an <authentication-manager> element.

```
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service>
      <security:user name="jim" password="password"
        authorities="ROLE_USER, ROLE_ADMIN" />
      <security:user name="sally" password="password"
        authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

- The <authentication-manager> is given one or more <authentication-providers> (only one is shown in this example).

- **As is obvious in this example, the authentication provider defines the authorized users, passwords and access levels right in the configuration.**
  - However, you can define authentication providers that get authentication and access control information from a variety of sources.
  - For example, you may want to get user information from a database, LDAP server, JAAS (Java Authentication and Authorization Service) or other source.
  - In fact, SS comes with a number of authentication provider implementations that you can use without having to code anything.
  - Of course, you can also create your own custom authentication provider (implementing Spring Security's `AuthenticationProvider` interface).
- **With just the `<http>` and `<authentication-manager>` elements in place, SS is turned on and ready to use with your Web application.**
  - An attempt to access a page within the Web application would result in the following page display.

### Login with Username and Password

User:

Password:

- Holy cow! Where did this login page come from?
- Spring Security generates one automatically.
- You can, however, customize the login. To provide your own login, change the `<http>` configuration to the following (and define your own `login.jsp` page).

```
<security:http pattern="/login.jsp" security="none" />
<security:http pattern="/badlogin.jsp" security="none" />
<security:http auto-config="true">
  <security:form-login login-page="/login.jsp"
    authentication-failure-url="/login.jsp" />
  <security:intercept-url pattern="/**" access="ROLE_USER" />
</security:http>
```

- **Above, note the extra <security:http> elements that allow any users access to the login and badlogin pages.**
  - Without these, the request to login would be matched by the pattern `/**`. Therefore, it wouldn't be possible to access the login page itself without being logged in.
  - This is a common configuration error and results in an infinite loop.
- **Conveniently, SS provides logout behavior via URL.**
  - By default, logout (and end the user's session) by directing to `/j_spring_security_logout`.
  - This action drops the session, logs out the user and redirects the application to the root Web application URL.
  - You can override the default logout behavior by providing a <logout> element to your configuration.

```
<security:http pattern="/login.jsp" security="none" />
<security:http pattern="/badlogin.jsp" security="none" />
<security:http pattern="/goodbye.jsp" security="none" />
<security:http auto-config="true">
  <security:form-login login-page="/login.jsp"
    authentication-failure-url="/badlogin.jsp" />
  <security:intercept-url pattern="/**" access="ROLE_USER" />
  <security:logout logout-url="/goodbye.jsp"
    logout-success-url="/goodbye.jsp" />
</security:http>
```

- **Note the auto-config="true" attribute on some examples above.**
  - The auto-config attribute, when set to true, is shorthand for the following configuration.

```
<security:http>
  <security:form-login />
  <security:http-basic />
  <security:logout />
</http>
```

- When other elements are provided, they override this default behavior.

## Java Configuration Web Security Configuration

---

- As an alternate means to namespace configuration above, you can use the **Java Config** class to configure your Web applications HTTP security.
- By default, the `WebSecurityConfigurerAdapter` (shown again below), provides default configuration for HTTP security.

```
@Configuration
@EnableWebSecurity
public class MySecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
    throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("password").roles("USER");
    }
}
```

- In `WebSecurityConfigurerAdapter` class is a `configure()` method that looks like the code below that defines the default HTTP security.

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated().and()
        .formLogin().and().httpBasic();
}
```

- This default configuration provides HTTP security equivalent to the following XML namespace configuration.

```
<security:http>
    <security:intercept-url pattern="/**" access="authenticated"/>
    <security:form-login />
    <security:http-basic />
</security:http>
```

- Use additional method calls to programmatically modify or add configuration to satisfy your application security needs.

- **Given the following XML namespace configuration assembled from the previous section ...**

```
<security:http pattern="/login.jsp" security="none" />
<security:http pattern="/badlogin.jsp" security="none" />
<security:http pattern="/goodbye.jsp" security="none" />
<security:http auto-config="true">
  <security:form-login login-page="/login.jsp"
    authentication-failure-url="/badlogin.jsp" />
  <security:intercept-url pattern="/**" access="ROLE_USER" />
  <security:logout logout-url="/goodbye"
    logout-success-url="/goodbye.jsp" />
</security:http auto-config="true">
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service>
      <security:user name="jim" password="password"
        authorities="ROLE_USER, ROLE_ADMIN" />
      <security:user name="sally" password="password"
        authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

- ... here is the equivalent in the Java Configuration form.

```

@EnableWebSecurity
@Configuration
public class CustomWebSecurityConfigurerAdapter extends
    WebSecurityConfigurerAdapter {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) {
        auth.inMemoryAuthentication()
            .withUser("jim")
                .password("password")
                .roles("ROLE_USER", "ROLE_ADMIN")
                .and()
            .withUser("sally")
                .password("password")
                .roles("ROLE_USER");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeUrls()
                .antMatchers("/login.jsp*", "/badlogin.jsp*", "/goodbye.jsp*")
                .permitAll()
                .antMatchers("/**").hasRole("ROLE_USER")
                .anyRequest().authenticated()
                .and()
            .formLogin()
                .loginPage("/login.jsp")
                .failureUrl("/badlogin.jsp")
                .permitAll()
                .and()
            .logout()
                .logout()
                .deleteCookies("remove")
                .invalidateHttpSession(false)
                .logoutUrl("/goodbye")
                .logoutSuccessUrl("/goodbye.jsp")
                .permitAll();
    }
}

```

## Alternate Authentication Providers

---

- **Except in prototype and development situation, you will need a more scalable means of providing user authentication information.**
  - As mentioned previously, SS allows for many alternate authentication providers.
  - In fact, you can list multiple authentication providers to allow your application to get its authentication information from many resources.
- **As an example, when you need to get your user information from a database, use the `<jdbc-user-service>` as shown below.**

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="myDataSource" />
  </authentication-provider>
</authentication-manager>
```

- In Java Configuration form, autowire a data source and a JDBC authentication method call.

```
@Autowired
@Qualifier("myDataSource")
private DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
    auth
        .jdbcAuthentication()
            .dataSource(dataSource)
            .withDefaultSchema();
}
```



- **The referenced data source must contain standard SS defined user data tables (see DDL below).**

```
create table users(
    username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(50) not null,
    enabled boolean not null);

create table authorities (
    username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fk_authorities_users foreign key(username)
        references users(username));
create unique index ix_auth_username on authorities
    (username,authority);
```

- The DDL statements are provided in the documentation appendix (Security Database Schema).<sup>2</sup>
- **As another alternate, if you have an LDAP server, register your server using SS's <ldap-server> element.**
  - Then use an <ldap-authentication-provider> in the <authentication-manager> provider list.
  - Configure the provider properties for user, group, etc. searches and filters.

```
<ldap-server id="ldapServer"
    url="ldap://yourdirectoryserver:338899/" />

<authentication-manager>
    <ldap-authentication-provider server-ref="ldapServer"
        user-search-base="ou=people,dc=sourceallies,dc=com"
        user-search-filter="(uid={0})"
        group-role-attribute="cn"
        group-search-base="ou=groups,dc=sourceallies,dc=com"
        group-search-filter="(memberUid={1})"
        role-prefix="ROLE_" />
</authentication-manager>
```

<sup>2</sup> <http://docs.spring.io/spring-security/site/docs/3.2.0.RELEASE/reference/htmlsingle/#appendix-schema>

- **Again, using Java Configuration you can similarly add support for LDAP based authentication.**

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
    auth
        .ldapAuthentication()
        .userSearchBase("ou=people,dc=sourceallies,dc=com")
        .userSearchFilter("(uid={0})")
        .groupRoleAttribute("cn")
        .groupSearchBase("ou=people,dc=sourceallies,dc=com")
        .groupSearchFilter("(memberUid={1})")
        .rolePrefix("ROLE_");
}
```

- **Spring Security provides many other features to secure Web/HTTP requests to include:**
  - Password encoding
  - Remember-me or persistent-login authentication that allows the identity of a principal between sessions.
  - Adding HTTP/HTTPS Channel Security (requiring certain URLs can only be accessed over HTTPS).
  - Session management
  - Support for OpenID
- **Consult the Spring Security documentation for more details.**

<http://docs.spring.io/spring-security/site/docs/3.2.0.RELEASE/reference/htmlsingle/>

## Method Security

---

- **SS also provides the means to secure components behind a Web front end.**
  - Known as method security, SS allows you to secure calls to a bean's methods.
  - It also supports JSR-250's (Common Annotations for the Java) security annotations.
  - Method security is provided through Spring AOP.
- **Enable SS's annotation-based method security in your application by adding the `<global-method-security>` element to your configuration.**

```
<security:global-method-security secured-annotations="enabled" />
```

- **To enable SS's annotation-based method security via Java Config, add a Java Configuration class with a `@EnableGlobalMethodSecurity` annotation on it.**

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class MyMethodSecurityConfig {
    // ...
}
```

- **With method security turned on, simply add an annotation to any method to limit the access to that method.**

```
import org.springframework.security.access.annotation.Secured;
public class MyService {

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public String someMethod(){
        ...
    }

    @Secured("ROLE_ADMIN")
    public void someOtherMethod(String param);
    ...
}
```

- You can also add the annotation to the class or interface to limit the entire type.

- To enable JSR-250 annotations, change the <global-method-security> attribute.

```
<global-method-security jsr250-annotations="enabled" />
```

- To enable JSR-250 with Java Config, use the jsr250Enabled attribute to the @EnableGlobalMethodSecurity annotation.

```
@Configuration
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class MyMethodSecurityConfig {
    // ...
}
```

- Use JSR-250 annotations to again limit the access to the method.

```
import javax.annotation.security.RolesAllowed;
public class MyService {

    @RolesAllowed({ "IS_AUTHENTICATED_ANONYMOUSLY " })
    public String someMethod(){
        ...
    }

    @RolesAllowed({ "ROLE_ADMIN" })
    public void someOtherMethod(String param);
    ...
}
```

- Additional JSR-250 annotations are shown in the table below.

JSR-250 Security Annotations
javax.annotation.security.PermitAll
javax.annotation.security.DenyAll
javax.annotation.security.RolesAllowed
javax.annotation.security.DeclareRoles
javax.annotation.security.RunAs



## Lab Exercise – Spring Security Lab

## Chapter Summary

---

- **Spring Security (SS) is a separate, but very popular, Spring project.**
- **SS provides the two main security services required by most applications: authentication and authorization.**
  - Authentication is the ability to identify a user or more generically a principal.
  - Authentication systems try to answer two questions for the application: “Who is the principal?” and “Is the principal really who he/she/it represents themselves to be?”
  - Authorization determines what resources a principal can access and what actions or operations a principal can perform on a system.
  - In other words, authorization systems try to answer a single question: “What can the principal do?”
- **As of SS 3.2, two options exist for configuring your application to leverage SS.**
- **The first approach to leverage SS is to use namespace configuration.**
  - It allows developers to add special SS elements to the Spring configuration to define security needs for your application.
  - These elements often conceal the fact that multiple beans and processing steps are added to the application context under the covers.
- **The second approach is to use Java Configuration to configure SS.**
  - Create a class that extends `WebSecurityConfigurerAdapter`.
- **SS uses filters to intercept incoming Web/HTTP request traffic.**
  - Web applications need to register SS’s `DelegatingFilterProxy` in your `web.xml` file.
  - In Servlet 3.0 or better environments, the filter can be registered programmatically with a class that extends `AbstractSecurityWebApplicationInitializer`
  - This initializer automatically registers the `springSecurityFilterChain` for every URL in your application.
- **Configure Web security needs via the `<security:http>` element in your beans configuration file.**
  - Alternately, override the `configure()` method in the `WebSecurityConfigurerAdapter` extending class.

- **SS also provides the means to secure methods behind an application.**
  - Method level security is provided by annotations.
  - It also supports JSR-250's (Common Annotations for the Java) security annotations.
  - Method security is provided through Spring AOP.

Do Not Print

## Appendix C

### Spring Web MVC Testing

#### *Objectives:*

- Explore the Spring MVC Test framework.
- Learn how to setup tests of controllers.
- See the API for examining the controller return results.

Do Not Print

## Chapter Overview

While Spring has always espoused good unit testing, it has not always provided adequate tools in the framework to unit test your components – especially in a way to test them in isolation.

Spring MVC components, in particular controllers, have always been difficult to test, and even more difficult to test in isolation. They rely on the Servlet container, and Spring MVC infrastructure (like `DispatcherServlet`) to be exercised. The Spring MVC Test framework was added to Spring to correct this deficiency.

Do Not Print



## Spring MVC Test Framework

---

- **Spring Framework 2.5 added a unit test framework to better support the unit test principal to which Spring espouses.**
  - The unit test framework is called the Test Context Framework.
  - Unfortunately, that framework did not provide much help to Web MVC application developers.
- **The Spring Web MVC application runs in a Web container and relies on the Servlet API.**
  - Requests are processed by the DispatcherServlet running in the Web container and the Servlet API provided by the Web container.
  - Components, such as controllers, are not easily tested without the full Web container and full Spring Web MVC environment in place.
  - Prior to Spring Framework 3.2, developers could test controllers by creating instances of controllers and dependency injecting them with mock or stub objects.
  - Even then, many of the annotations (like @RequestMapping) went untested.
- **Spring 3.2 introduced the Spring MVC Test framework.**
  - This framework allows Spring MVC controllers to be more easily and thoroughly tested.
  - Spring MVC Test is still built on mock implementations, but these are provided without much developer work.

- **The Spring MVC Test framework is provided with the spring-test module as of Spring 4.0.**
  - So in order to use the Spring MVC Test framework, just make sure the spring-test dependency is in your Maven pom.xml file.
  - You will also need to add JUnit to provide for the unit testing framework.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
</dependency>
```

## Test Setup

---

- **There are two options for setting up a test class capable of testing a controller.**
  - One option allows you to dependency inject a `WebApplicationContext` into the test case and use the context to drive tests.
  - The controller is exercised through calls sent through the `WebApplicationContext`.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration("dispatcher-servlet.xml")
public class MyMVCTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
    }
    ...
}
```

- The second option has you dependency inject a particular controller into the test case directly and without the need for a `WebApplicationContext`.

```
public class MyControllerTests {
    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = MockMvcBuilders.
            standaloneSetup(new HelloWorldController()).build();
    }
    ...
}
```

- As you can see, the first option is more of an integration test as it does rely on the availability of the `WebApplicationContext` and the configuration of that container.
- The second option does not even rely on any other components or container (or even the Test Context framework – notice no `@RunWith` annotation).

## Creating Controller Tests

---

- **Regardless of which setup is used, the MockMvc object (from `org.springframework.test.web.servlet`) provides the ability to test controllers.**
  - MockMvc can perform a request to the controller and examine the results.

```
@Test
public void getHelloWorld() throws Exception {
    String view = mockMvc.perform(get("/greeting.request")).andReturn()
        .getModelAndView().getViewName();
    Assert.assertTrue("view does not match", "hello".equals(view));
}
```

- What comes back from the `perform()` method is a `org.springframework.test.web.servlet.ResultAction` object.
- Through the `ResultAction` object and results it can provide access to, you can examine all aspects of the controllers work results.
- The example above explores the logical view name (String) in the returned `ModelAndView`.
- **In addition to the get request (shown above), the MockMvc can be used to perform other types of requests.**

```
mockMvc.perform(post("/greeting.request/welcomemsg/hello"));
```

- The `get`, `post`, etc. methods are overloaded to provide path variables.

```
mockMvc.perform(post("/greeting.request/welcomemsg/{msg}", "hello"));
```

- Request parameters can be added using the `param()` method.

```
mockMvc.perform(get("/greeting.request").param("name", "Jim"));
```

- Learn about more options at <http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#spring-mvc-test-framework>.

- **Test return results can be explored for all sorts of expectations.**

- Check the status (HTTP status) of a result.

```
mockMvc.perform(get("/greeting.request")).andExpect(status().isOk());
mockMvc.perform(get("/greeting.request")).andExpect(status().
    isMovedTemporarily());
mockMvc.perform(get("/greeting.request")).andExpect(status().
    isNotFound());
mockMvc.perform(get("/greeting.request")).andExpect(status().
    isForbidden());
```

- The `andReturn()` called in place of `andExpect()` returns an `MvcResult` object.
- This object can provide a `MockHttpServletResponse` with a call to `getResponse()` on the `MvcResult`.

```
MockHttpServletResponse resp =
    mockMvc.perform(get("/greeting.request")).andReturn().getResponse();
```

- From the `MockHttpServletResponse`, check on cookies, header information, contents, etc.

```
resp.getCookie("cookieName");
resp.getHeader("headerName");
resp.getContentAsString();
```



## Lab Exercise – Spring MVC Test Lab

## Chapter Summary

---

- **Spring 3.2 introduced the Spring MVC Test framework.**
  - This framework allows Spring MVC controllers to be more easily and thoroughly tested.
  - The Spring MVC Test framework is provided with the spring-test module as of Spring 4.0.
  - Spring MVC Test is built on mock implementations.
- **There are two options for setting up a test class capable of testing a controller.**
  - One option allows you to dependency inject a `WebApplicationContext` into the test case and use the context to drive tests.
  - The second option has you dependency inject a particular controller into the test case directly.
- **Regardless of which setup is used, the `MockMvc` object (from `org.springframework.test.web.servlet`) provides the ability to test controllers.**
  - `MockMvc` can perform a request to the controller and examine the results.
  - What comes back from the `perform()` method is a `org.springframework.test.web.servlet.ResultAction` object.
  - Through the `ResultAction` object and results it can provide access to, you can examine all aspects of the controllers work results.