Chapter 13. Web MVC framework

13.1. Introduction

Spring's Web MVC framework is designed around a `DispatcherServlet` that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for upload files. The default handler is a very simple `Controller` interface, just offering a `ModelAndView handleRequest(request,response)` method. This can already be used for application controllers, but you will prefer the included implementation hierarchy, consisting of, for example `AbstractController`, `AbstractCommandController` and `SimpleFormController`. Application controllers will typically be subclasses of those. Note that you can choose an appropriate base class: if you don't have a form, you don't need a form controller. This is a major difference to Struts.

“**Open for extension...”**

One of the overarching design principles in Spring Web MVC (and in Spring in general) is the “*Open for extension, closed for modification*” principle.

The reason that this principle is being mentioned here is because a number of methods in the core classes in Spring Web MVC are marked `final`. This means of course that you as a developer cannot override these methods to supply your own behavior... this is *by design* and has not been done arbitrarily to annoy.

The book 'Expert Spring Web MVC and Web Flow' by Seth Ladd and others explains this principle and the reasons for adhering to it in some depth on page 117 (first edition) in the section entitled 'A Look At Design'.

If you don't have access to the aforementioned book, then the following article may be of interest the next time you find yourself going “Gah! Why can't I override this method?” (if indeed you ever do).

1. Bob Martin, The Open-Closed Principle (PDF)

Note that you cannot add advice to final methods using Spring MVC. This means it won't be possible to add advice to for example the `AbstractController.handleRequest()` method. Refer to Section 6.6.1, “Understanding AOP proxies” for more information on AOP proxies and why you cannot add advice to final methods.

Spring Web MVC allows you to use any object as a command or form object - there is no need to implement a framework-specific interface or base class. Spring's data binding is highly flexible: for example, it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. All this means that you don't need to duplicate your business objects' properties as simple, untyped strings in your form objects just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it is often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes such as `Action` and `ActionForm`.

Compared to WebWork, Spring has more differentiated object roles. It supports the notion of a `Controller`, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data; instead, a WebWork `Action` combines all those roles into one single object. WebWork does allow you to use existing business objects as part of your form, but only by making them bean properties of the

respective `Action` class. Finally, the same `Action` instance that handles the request is used for evaluation and form population in the view. Thus, reference data needs to be modeled as bean properties of the `Action` too. These are (arguably) too many roles for one object.

Spring's view resolution is extremely flexible. A `Controller` implementation can even write a view directly to the response (by returning `null` for the `ModelAndView`). In the normal case, a `ModelAndView` instance consists of a view name and a model `Map`, which contains bean names and corresponding objects (like a command or form, containing reference data). View name resolution is highly configurable, either via bean names, via a properties file, or via your own `ViewResolver` implementation. The fact that the model (the M in MVC) is based on the `Map` interface allows for the complete abstraction of the view technology. Any renderer can be integrated directly, whether JSP, Velocity, or any other rendering technology. The model `Map` is simply transformed into an appropriate format, such as JSP request attributes or a Velocity template model.

13.1.1. Pluggability of other MVC implementations

There are several reasons why some projects will prefer to use other MVC implementations. Many teams expect to leverage their existing investment in skills and tools. In addition, there is a large body of knowledge and experience avalailable for the Struts framework. Thus, if you can live with Struts' architectural flaws, it can still be a viable choice for the web layer; the same applies to WebWork and other web MVC frameworks.

If you don't want to use Spring's web MVC, but intend to leverage other solutions that Spring offers, you can integrate the web MVC framework of your choice with Spring easily. Simply start up a Spring root application context via its `ContextLoaderListener`, and access it via its `ServletContext` attribute (or Spring's respective helper method) from within a Struts or WebWork action. Note that there aren't any "plugins" involved, so no dedicated integration is necessary. From the web layer's point of view, you'll simply use Spring as a library, with the root application context instance as the entry point.

All your registered beans and all of Spring's services can be at your fingertips even without Spring's web MVC. Spring doesn't compete with Struts or WebWork in this scenario, it just addresses the many areas that the pure web MVC frameworks don't, from bean configuration to data access and transaction handling. So you are able to enrich your application with a Spring middle tier and/or data access tier, even if you just want to use, for example, the transaction abstraction with JDBC or Hibernate.

13.1.2. Features of Spring Web MVC

Spring WebFlow

Spring Web Flow (SWF) aims to be the best solution for the management of web application page flow.

SWF integrates with existing frameworks like Spring MVC, Struts, and JSF, in both servlet and portlet environments. If you have a business process (or processes) that would benefit from a conversational model as opposed to a purely request model, then SWF may be the solution.

SWF allows you to capture logical page flows as self-contained modules that are reusable in different situations, and as such is ideal for building web application modules that guide the user through controlled navigations that drive business processes.

For more information about SWF, consult the Spring WebFlow site.

Spring's web module provides a wealth of unique web support features, including:

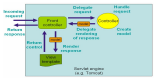
- Clear separation of roles - controller, validator, command object, form object, model object, `DispatcherServlet`, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy referencing across contexts, such as from web controllers to business objects and validators.
- Adaptability, non-intrusiveness. Use whatever controller subclass you need (plain, command, form, wizard, multi-action, or a custom one) for a given scenario instead of deriving from a single controller for everything.
- Reusable business code - no need for duplication. You can use existing business objects as command or form objects instead of mirroring them in order to extend a particular framework base class.
- Customizable binding and validation - type mismatches as application-level validation errors that keep the offending value, localized date and number binding, etc instead of String-only form objects with manual parsing and conversion to business objects.
- Customizable handler mapping and view resolution - handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. This is more flexible than some web MVC frameworks which mandate a particular technique.
- Flexible model transfer - model transfer via a name/value `Map` supports easy integration with any view technology.
- Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, etc.
- A simple yet powerful JSP tag library known as the Spring tag library that provides support for features such as data binding and themes. The custom tags allow for maximum flexibility in terms of markup code. For information on the tag library descriptor, see the appendix entitled Appendix D, *spring.tld*
- A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier. For information on the tag library descriptor, see the appendix entitled Appendix E, *spring-form.tld*
- Beans whose lifecycle is scoped to the current HTTP request or HTTP `Session`. This is not a specific feature of Spring MVC itself, but rather of the `WebApplicationContext` container(s) that Spring MVC uses. These bean scopes are described in detail in the section entitled Section 3.4.4, “The other scopes”

13.2. The `DispatcherServlet`

Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality facilitating the development of web applications. Spring's `DispatcherServlet` however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

The request processing workflow of the Spring Web MVC `DispatcherServlet` is illustrated in the

following diagram. The pattern-savvy reader will recognize that the `DispatcherServlet` is an expression of the "Front Controller" design pattern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).



The requesting processing workflow in Spring Web MVC (high level)

The `DispatcherServlet` *is* an actual `Servlet` (it inherits from the `HttpServlet` base class), and as such is declared in the `web.xml` of your web application. Requests that you want the `DispatcherServlet` to handle will have to be mapped using a URL mapping in the same `web.xml` file. This is standard J2EE servlet configuration; an example of such a `DispatcherServlet` declaration and mapping can be found below.

```
<web-app>

    <servlet>
        <servlet-name>example</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>

</web-app>
```

In the example above, all requests ending with `.form` will be handled by the `'example'` `DispatcherServlet`. This is only the first step in setting up Spring Web MVC... the various beans used by the Spring Web MVC framework (over and above the `DispatcherServlet` itself) now need to be configured.

As detailed in the section entitled Section 3.8, "The ApplicationContext", `ApplicationContext` instances in Spring can be scoped. In the web MVC framework, each `DispatcherServlet` has its own `WebApplicationContext`, which inherits all the beans already defined in the root `WebApplicationContext`. These inherited beans defined can be overridden in the servlet-specific scope, and new scope-specific beans can be defined local to a given servlet instance.

Context hierarchy in Spring Web MVC

The framework will, on initialization of a `DispatcherServlet`, *look for a file named* `[servlet-name]-servlet.xml` in the `WEB-INF` directory of your web application and create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

Consider the following `DispatcherServlet` servlet configuration (in the `'web.xml'` file.)

```
<web-app>
    ...
    <servlet>
```

```
        <servlet-name>golfing</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>golfing</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

With the above servlet configuration in place, you will need to have a file called `'/WEB-INF/golfing-servlet.xml'` in your application; this file will contain all of your *Spring Web MVC-specific* components (beans). The exact location of this configuration file can be changed via a servlet initialization parameter (see below for details).

The `WebApplicationContext` is an extension of the plain `ApplicationContext` that has some extra features necessary for web applications. It differs from a normal `ApplicationContext` in that it is capable of resolving themes (see Section 13.7, “Using themes”), and that it knows which servlet it is associated with (by having a link to the `ServletContext`). The `WebApplicationContext` is bound in the `ServletContext`, and by using static methods on the `RequestContextUtils` class you can always lookup the `WebApplicationContext` in case you need access to it.

The Spring `DispatcherServlet` has a couple of special beans it uses in order to be able to process requests and render the appropriate views. These beans are included in the Spring framework and can be configured in the `WebApplicationContext`, just as any other bean would be configured. Each of those beans is described in more detail below. Right now, we'll just mention them, just to let you know they exist and to enable us to go on talking about the `DispatcherServlet`. For most of the beans, sensible defaults are provided so you don't (initially) have to worry about configuring them.

Table 13.1. Special beans in the `WebApplicationContext`

Bean type	Explanation
Controllers	Controllers are the components that form the `'C'` part of the MVC.
Handler mappings	Handler mappings handle the execution of a list of pre- and post-processors and controllers that will be executed if they match certain criteria (for instance a matching URL specified with the controller)
View resolvers	View resolvers are components capable of resolving view names to views
Locale resolver	A locale resolver is a component capable of resolving the locale a client is using, in order to be able to offer internationalized views
Theme resolver	A theme resolver is capable of resolving themes your web application can use, for example, to offer personalized layouts
multipart file resolver	A multipart file resolver offers the functionality to process file uploads from HTML forms
Handler exception resolver(s)	Handler exception resolvers offer functionality to map exceptions to views or implement other more complex exception handling code

When a `DispatcherServlet` is set up for use and a request comes in for that specific `DispatcherServlet`, said `DispatcherServlet` starts processing the request. The list below describes the complete process a request goes through when handled by a `DispatcherServlet`:

1. The `WebApplicationContext` is searched for and bound in the request as an attribute in order for the controller and other elements in the process to use. It is bound by default under the key `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`.
2. The locale resolver is bound to the request to let elements in the process resolve the locale to use when processing the request (rendering the view, preparing data, etc.) If you don't use the resolver, it won't affect anything, so if you don't need locale resolving, you don't have to use it.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use. The theme resolver does not affect anything if you don't use it, so if you don't need themes you can just ignore it.
4. If a multipart resolver is specified, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a `MultipartHttpServletRequest` for further processing by other elements in the process. (See the section entitled Section 13.8.2, “Using the MultipartResolver” for further information about multipart handling).
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) will be executed in order to prepare a model (for rendering).
6. If a model is returned, the view is rendered. If no model is returned (which could be due to a pre- or postprocessor intercepting the request, for example, for security reasons), no view is rendered, since the request could already have been fulfilled.

Exceptions that are thrown during processing of the request get picked up by any of the handler exception resolvers that are declared in the `WebApplicationContext`. Using these exception resolvers allows you to define custom behaviors in case such exceptions get thrown.

The Spring `DispatcherServlet` also has support for returning the *last-modification-date*, as specified by the Servlet API. The process of determining the last modification date for a specific request is straightforward: the `DispatcherServlet` will first lookup an appropriate handler mapping and test if the handler that is found *implements the interface* `LastModified` interface. If so, the value of the `long getLastModified(request)` method of the `LastModified` interface is returned to the client.

You can customize Spring's `DispatcherServlet` by adding context parameters in the `web.xml` file or servlet initialization parameters. The possibilities are listed below.

Table 13.2. `DispatcherServlet` initialization parameters

Parameter	Explanation
`contextClass`	Class that implements `WebApplicationContext`, which will be used to instantiate the context used by this servlet. If this parameter isn't specified, the `XmlWebApplicationContext` will be used.
`contextConfigLocation`	String which is passed to the context instance (specified by `contextClass`) to indicate where context(s) can be found. The string is potentially split up into multiple strings (using a comma as a delimiter) to support multiple contexts (in case of multiple context locations, of beans that are defined twice, the latest takes precedence).
`namespace`	the namespace of the `WebApplicationContext`. Defaults to `[servlet-name]-servlet`.

13.3. Controllers

The notion of a controller is part of the MVC design pattern (more specifically it is the *'C'* in MVC. Controllers provide access to the application behavior which is typically defined by a service interface. Controllers interpret user input and transform such input into a sensible model which will be represented to the user by the view. Spring has implemented the notion of a controller in a very abstract way enabling a wide variety of different kinds of controllers to be created. Spring contains form-specific controllers, command-based controllers, and controllers that execute wizard-style logic, to name but a few.

Spring's basis for the controller architecture is the `org.springframework.web.servlet.mvc.Controller` interface, the source code for which is listed below.

```
public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the
DispatcherServlet
     * will render.
     */
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception;

}
```

As you can see, the `Controller` interface defines a single method that is responsible for handling a request and returning an appropriate model and view. These three concepts are the basis for the Spring MVC implementation - `ModelAndView` and `Controller`. While the `Controller` interface is quite abstract, Spring offers a lot of `Controller` implementations out of the box that already contain a lot of the functionality you might need. The `Controller` interface just defines the most basic responsibility required of every controller; namely handling a request and returning a model and a view.

13.3.1. `AbstractController` and `WebContentGenerator`

To provide a basic infrastructure, all of Spring's various `Controller` inherit from `AbstractController`, a class offering caching support and, for example, the setting of the mimetype.

Table 13.3. Features offered by the `AbstractController`

Feature	Explanation
`supportedMethods`	indicates what methods this controller should accept. Usually this is set to both `GET` and `POST`, but you can modify this to reflect the method you want to support. If a request is received with a method that is not supported by the controller, the client will be informed of this (expedited by the throwing of a `ServletException`).
`requiresSession`	indicates whether or not this controller requires a HTTP session to do its work. If a session is not present when such a controller receives a request, the user is informed of this by a `ServletException` being thrown.

Feature	Explanation
`synchronizeSession`	use this if you want handling by this controller to be synchronized on the user's HTTP session.
`cacheSeconds`	when you want a controller to generate a caching directive in the HTTP response, specify a positive integer here. By default the value of this property is set to *-1* so no caching directives will be included in the generated response.
`useExpiresHeader`	tweaks your controllers to specify the HTTP 1.0 compatible *"Expires"* header in the generated response. By default the value of this property is `true`.
`useCacheHeader`	tweaks your controllers to specify the HTTP 1.1 compatible *"Cache-Control"* header in the generated response. By default the value of this property is `true`.

When using the `AbstractController` as the baseclass for your controllers you only have to override the `handleRequestInternal(HttpServletRequest, HttpServletResponse)` method, implement your logic, and return a `ModelAndView` object. Here is short example consisting of a class and a declaration in the web application context.

```
package samples;

public class SampleController extends AbstractController {

    public ModelAndView handleRequestInternal(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        ModelAndView mav = new ModelAndView("hello");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}

<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
</bean>
```

The above class and the declaration in the web application context is all you need besides setting up a handler mapping (see the section entitled Section 13.4, “Handler mappings”) to get this very simple controller working. This controller will generate caching directives telling the client to cache things for 2 minutes before rechecking. This controller also returns a hard-coded view (which is typically considered bad practice).

13.3.2. Other simple controllers

Although you can extend `AbstractController`, Spring provides a number of concrete implementations which offer functionality that is commonly used in simple MVC applications. The `ParameterizableViewController` is basically the same as the example above, except for the fact that you can specify the view name that it will return in the web application context (and thus remove the need to hard-code the viewname in the Java class).

The `UrlFilenameViewController` inspects the URL and retrieves the filename of the file

request and uses that as a viewname. For example, the filename of `http://www.springframework.org/index.html` request is `index`.

13.3.3. The `MultiActionController`

Spring offers a multi-action controller with which you aggregate multiple actions into one controller, thus grouping functionality together. The multi-action controller lives in a separate package - `org.springframework.web.servlet.mvc.multiaction` - and is capable of mapping requests to method names and then invoking the right method name. Using the multi-action controller is especially handy when you have a lot of common functionality in one controller, but want to have multiple entry points to the controller, for example, to tweak behavior.

Table 13.4. Features offered by the `MultiActionController`

Feature	Explanation
`delegate`	there are two usage-scenarios for the `MultiActionController`. Either you subclass the `MultiActionController` and specify the methods that will be resolved by the `MethodNameResolver` on the subclass (in which case you don't need to set the delegate), or you define a delegate object, on which methods resolved by the `MethodNameResolver` will be invoked. If you choose this scenario, you will have to define the delegate using this configuration parameter as a collaborator.
`methodNameResolver`	the `MultiActionController` needs a strategy to resolve the method it has to invoke, based on the incoming request. This strategy is defined by the `MethodNameResolver` interface; the `MultiActionController` exposes a property sp that you can supply a resolver that is capable of doing that.

Methods defined for a multi-action controller need to conform to the following signature:

```
// anyMeaningfulName can be replaced by any methodname
public [ModelAndView | Map | void] anyMeaningfulName(HttpServletRequest,
HttpServletResponse [, Exception | AnyObject]);
```

Please note that method overloading is *not* allowed since it would confuse the `MultiActionController`. Furthermore, you can define *exception handlers* capable of handling exceptions that are thrown by the methods you specify.

The (optional) `Exception` argument can be *any* exception, as long as it's a subclass of `java.lang.Exception` or `java.lang.RuntimeException`. The (optional) `AnyObject` argument can be *any* class. Request parameters will be bound onto this object for convenient consumption.

Find below some examples of valid `MultiActionController` method signatures.

The standard signature (mirrors the `Controller` interface method).

```
public ModelAndView doRequest(HttpServletRequest, HttpServletResponse)
```

This signature accepts a `Login` argument that will be populated (bound) with parameters stripped from the request

```
public ModelAndView doLogin(HttpServletRequest, HttpServletResponse, Login)
```

The signature for an `Exception` handling method.

```
public ModelAndView processException(HttpServletRequest, HttpServletResponse,
IllegalArgumentException)
```

This signature has a void return type (see the section entitled Section 13.11, “Convention over configuration” below).

```
public void goHome(HttpServletRequest, HttpServletResponse)
```

This signature has a `Map` return type (see the section entitled Section 13.11, “Convention over configuration” below).

```
public Map doRequest(HttpServletRequest, HttpServletResponse)
```

The `MethodNameResolver` is responsible for resolving method names based on the request coming in. Find below details about the three `MethodNameResolver` implementations that Spring provides out of the box.

- `ParameterMethodNameResolver` - capable of resolving a request parameter and using that as the method name (`http://www.sf.net/index.view?testParam=testIt` will result in a method `testIt(HttpServletRequest, HttpServletResponse)` being called). The `paramName` property specifies the request parameter that is to be inspected).
- `InternalPathMethodNameResolver` - retrieves the filename from the request path and uses that as the method name (`http://www.sf.net/testing.view` will result in a method `testing(HttpServletRequest, HttpServletResponse)` being called).
- `PropertiesMethodNameResolver` - uses a user-defined properties object with request URLs mapped to method names. When the properties contain `/index/welcome.html=doIt` and a request to `/index/welcome.html` comes in, the `doIt(HttpServletRequest, HttpServletResponse)` method is called. This method name resolver works with the `PathMatcher`, so if the properties contained `/**/welcom?.html`, it would also have worked!

Here are a couple of examples. First, an example showing the `ParameterMethodNameResolver` and the delegate property, which will accept requests to URLs with the parameter method included and set to `retrieveIndex`:

```
<bean id="paramResolver"
class="org....mvc.multiaction.ParameterMethodNameResolver">
  <property name="paramName" value="method"/>
</bean>

<bean id="paramMultiController"
class="org....mvc.multiaction.MultiActionController">
  <property name="methodNameResolver" ref="paramResolver"/>
  <property name="delegate" ref="sampleDelegate"/>
</bean>

<bean id="sampleDelegate" class="samples.SampleDelegate"/>

## together with

public class SampleDelegate {
```

```
    public ModelAndView retrieveIndex(HttpServletRequest req, HttpServletResponse
resp) {

        return new ModelAndView("index", "date", new
Long(System.currentTimeMillis()));
    }
}
```

When using the delegates shown above, we could also use the `PropertiesMethodNameResolver` to match a couple of URLs to the method we defined:

```
<bean id="propsResolver"
class="org....mvc.multiaction.PropertiesMethodNameResolver">
  <property name="mappings">
    <value>
        /index/welcome.html=retrieveIndex
        /**/notwelcome.html=retrieveIndex
        /*/user?.html=retrieveIndex
    </value>
  </property>
</bean>

<bean id="paramMultiController"
class="org....mvc.multiaction.MultiActionController">
    <property name="methodNameResolver" ref="propsResolver"/>
    <property name="delegate" ref="sampleDelegate"/>
</bean>
```

13.3.4. Command controllers

Spring's *command controllers* are a fundamental part of the Spring Web MVC package. Command controllers provide a way to interact with data objects and dynamically bind parameters from the `HttpServletRequest` to the data object specified. They perform a somewhat similar role to the Struts `ActionForm`, but in Spring, your data objects don't have to implement a framework-specific interface. First, lets examine what command controllers are available straight out of the box.

- `AbstractCommandController` - a command controller you can use to create your own command controller, capable of binding request parameters to a data object you specify. This class does not offer form functionality; it does however offer validation features and lets you specify in the controller itself what to do with the command object that has been populated with request parameter values.
- `AbstractFormController` - an abstract controller offering form submission support. Using this controller you can model forms and populate them using a command object you retrieve in the controller. After a user has filled the form, the `AbstractFormController` binds the fields, validates the command object, and hands the object back to the controller to take the appropriate action. Supported features are: invalid form submission (resubmission), validation, and normal form workflow. You implement methods to determine which views are used for form presentation and success. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the application context.
- `SimpleFormController` - a form controller that provides even more support when creating a form with a corresponding command object. The `SimpleFormController` let's

you specify a command object, a viewname for the form, a viewname for page you want to show the user when form submission has succeeded, and more.

- `AbstractWizardFormController` - as the class name suggests, this is an abstract class - your wizard controller should extend it. This means you have to implement the `validatePage()`, `processFinish()` and `processCancel()` methods.

  You probably also want to write a contractor, which should at the very least call `setPages()` and `setCommandName()`. The former takes as its argument an array of type String. This array is the list of views which comprise your wizard. The latter takes as its argument a String, which will be used to refer to your command object from within your views.

  As with any instance of `AbstractFormController`, you are required to use a command object - a JavaBean which will be populated with the data from your forms. You can do this in one of two ways: either call `setCommandClass()` from the constructor with the class of your command object, or implement the `formBackingObject()` method.

  `AbstractWizardFormController` has a number of concrete methods that you may wish to override. Of these, the ones you are likely to find most useful are: `referenceData(..)` which you can use to pass model data to your view in the form of a `Map`; `getTargetPage()` if your wizard needs to change page order or omit pages dynamically; and `onBindAndValidate()` if you want to override the built-in binding and validation workflow.

  Finally, it is worth pointing out the `setAllowDirtyBack()` and `setAllowDirtyForward()`, which you can call from `getTargetPage()` to allow users to move backwards and forwards in the wizard even if validation fails for the current page.

  For a full list of methods, see the Javadoc for `AbstractWizardFormController`. There is an implemented example of this wizard in the jPetStore included in the Spring distribution: `org.springframework.samples.jpetstore.web.spring.OrderFormController`.

13.4. Handler mappings

Using a handler mapping you can map incoming web requests to appropriate handlers. There are some handler mappings you can use out of the box, for example, the `SimpleUrlHandlerMapping` or the `BeanNameUrlHandlerMapping`, but let's first examine the general concept of a `HandlerMapping`.

The functionality a basic `HandlerMapping` provides is the delivering of a `HandlerExecutionChain`, which must contain the handler that matches the incoming request, and may also contain a list of handler interceptors that are applied to the request. When a request comes in, the `DispatcherServlet` will hand it over to the handler mapping to let it inspect the request and come up with an appropriate HandlerExecutionChain. Then the `DispatcherServlet` will execute the handler and interceptors in the chain (if any).

The concept of configurable handler mappings that can optionally contain interceptors (executed before or after the actual handler was executed, or both) is extremely powerful. A lot of supporting functionality can be built into custom `HandlerMapping`s. Think of a custom handler mapping that chooses a handler not only based on the URL of the request coming in, but also on a specific state of the session associated with the request.

This section describes two of Spring's most commonly used handler mappings. They both extend the `AbstractHandlerMapping` and share the following properties:

- `interceptors`: the list of interceptors to use. `HandlerInterceptor`s are discussed in Section 13.4.3, "Intercepting requests - the HandlerInterceptor interface".
- `defaultHandler`: the default handler to use, when this handler mapping does not result in a matching handler.
- `order`: based on the value of the order property (see the `org.springframework.core.Ordered` interface), Spring will sort all handler mappings available in the context and apply the first matching handler.
- `alwaysUseFullPath`: if this property is set to `true`, Spring will use the full path within the current servlet context to find an appropriate handler. If this property is set to `false` (the default), the path within the current servlet mapping will be used. For example, if a servlet is mapped using `/testing/*` and the `alwaysUseFullPath` property is set to true, `/testing/viewPage.html` would be used, whereas if the property is set to false, `/viewPage.html` would be used.
- `urlPathHelper`: using this property, you can tweak the UrlPathHelper used when inspecting URLs. Normally, you shouldn't have to change the default value.
- `urlDecode`: the default value for this property is `false`. The `HttpServletRequest` returns request URLs and URIs that are *not* decoded. If you do want them to be decoded before a `HandlerMapping` uses them to find an appropriate handler, you have to set this to `true` (note that this requires JDK 1.4). The decoding method uses either the encoding specified by the request or the default ISO-8859-1 encoding scheme.
- `lazyInitHandlers`: allows for lazy initialization of *singleton* handlers (prototype handlers are always lazily initialized). Default value is `false`.

(*Note: the last four properties are only available to subclasses of* *`org.springframework.web.servlet.handler.AbstractUrlHandlerMapping`*).

13.4.1. `BeanNameUrlHandlerMapping`

A very simple, but very powerful handler mapping is the `BeanNameUrlHandlerMapping`, which maps incoming HTTP requests to names of beans, defined in the web application context. Let's say we want to enable a user to insert an account and we've already provided an appropriate form controller (see Section 13.3.4, "Command controllers" for more information on command- and form controllers) and a JSP view (or Velocity template) that renders the form. When using the `BeanNameUrlHandlerMapping`, we could map the HTTP request with the URL `http://samples.com/editaccount.form` to the appropriate form `Controller` as follows:

```
<beans>
  <bean id="handlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

  <bean name="/editaccount.form"
class="org.springframework.web.servlet.mvc.SimpleFormController">
    <property name="formView" value="account"/>
    <property name="successView" value="account-created"/>
    <property name="commandName" value="account"/>
```

```
    <property name="commandClass" value="samples.Account"/>
  </bean>
<beans>
```

All incoming requests for the URL `/editaccount.form` will now be handled by the form `Controller` in the source listing above. Of course we have to define a servlet-mapping in `web.xml` as well, to let through all the requests ending with `.form`.

```
<web-app>
    ...
    <servlet>
        <servlet-name>sample</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

   <!-- maps the sample dispatcher to *.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Note

If you want to use the `BeanNameUrlHandlerMapping`, you don't necessarily have to define it in the web application context (as indicated above). By default, if no handler mapping can be found in the context, the `DispatcherServlet` creates a `BeanNameUrlHandlerMapping` for you!

13.4.2. `SimpleUrlHandlerMapping`

A further - and much more powerful handler mapping - is the `SimpleUrlHandlerMapping`. This mapping is configurable in the application context and has Ant-style path matching capabilities (see the Javadoc for the `org.springframework.util.PathMatcher` class). Here is an example:

```
<web-app>
    ...
    <servlet>
        <servlet-name>sample</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- maps the sample dispatcher to *.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>

    <!-- maps the sample dispatcher to *.html -->
    <servlet-mapping>
```

```
        <servlet-name>sample</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

The above `web.xml` configuration snippet enables all requests ending with .html and `.form` to be handled by the sample dispatcher servlet.

```
<beans>

    <!-- no 'id' required, HandlerMapping beans are automatically detected by the
DispatcherServlet -->
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /*/account.form=editAccountFormController
                /*/editaccount.form=editAccountFormController
                /ex/view*.html=helpController
                /**/help.html=helpController
            </value>
        </property>
    </bean>

    <bean id="helpController"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

    <bean id="editAccountFormController"
          class="org.springframework.web.servlet.mvc.SimpleFormController">
        <property name="formView" value="account"/>
        <property name="successView" value="account-created"/>
        <property name="commandName" value="Account"/>
        <property name="commandClass" value="samples.Account"/>
    </bean>
<beans>
```

This handler mapping routes requests for `'help.html'` in any directory to the `'helpController'`, which is a `UrlFilenameViewController` (more about controllers can be found in the section entitled Section 13.3, “Controllers”). Requests for a resource beginning with `'view'`, and ending with `'.html'` in the directory `'ex'` will be routed to the `'helpController'`. Two further mappings are also defined for `'editAccountFormController'`.

13.4.3. Intercepting requests - the `HandlerInterceptor` interface

Spring's handler mapping mechanism has the notion of handler interceptors, that can be extremely useful when you want to apply specific functionality to certain requests, for example, checking for a principal.

Interceptors located in the handler mapping must implement `HandlerInterceptor` from the `org.springframework.web.servlet` package. This interface defines three methods, one that will be called *before* the actual handler will be executed, one that will be called *after* the handler is executed, and one that is called *after the complete request has finished*. These three methods should provide enough flexibility to do all kinds of pre- and post-processing.

The `preHandle(..)` method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns `true`, the handler execution chain will continue, when it returns false, the `DispatcherServlet` assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

The following example provides an interceptor that intercepts all requests and reroutes the user to a specific page if the time is not between 9 a.m. and 6 p.m.

```
<beans>
    <bean id="handlerMapping"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="officeHoursInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <value>
                /*.form=editAccountFormController
                /*.view=editAccountFormController
            </value>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
          class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime" value="9"/>
        <property name="closingTime" value="18"/>
    </bean>
<beans>

package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;

    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }

    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }

    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler) throws Exception {

        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour < closingTime) {
            return true;
        } else {
```

```
            response.sendRedirect("http://host.com/outsideOfficeHours.html");
            return false;
        }
    }
}
```

Any request coming in, will be intercepted by the `TimeBasedAccessInterceptor`, and if the current time is outside office hours, the user will be redirected to a static html file, saying, for example, he can only access the website during office hours.

As you can see, Spring has an adapter class (the cunningly named `HandlerInterceptorAdapter`) to make it easier to extend the `HandlerInterceptor` interface.

13.5. Views and resolving them

All MVC frameworks for web applications provide a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. Out of the box, Spring enables you to use JSPs, Velocity templates and XSLT views, for example. The section entitled Chapter 14, *Integrating view technologies* has details of how to integrate and use a number of disparate view technologies.

The two interfaces which are important to the way Spring handles views are `ViewResolver` and `View`. The `ViewResolver` provides a mapping between view names and actual views. The `View` interface addresses the preparation of the request and hands the request over to one of the view technologies.

13.5.1. Resolving views - the `ViewResolver` interface

As discussed in the section entitled Section 13.3, “Controllers”, all controllers in the Spring Web MVC framework return a `ModelAndView` instance. Views in Spring are addressed by a view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. We'll list most of them and then provide a couple of examples.

Table 13.5. View resolvers

`ViewResolver`	Description
`AbstractCachingViewResolver`	An abstract view resolver which takes care of caching views. Often views need preparation before they can be used, extending this view resolver provides caching of views.
`XmlViewResolver`	An implementation of `ViewResolver` that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is `/WEB-INF/views.xml`.
`ResourceBundleViewResolver`	An implementation of `ViewResolver` that uses bean definitions in a `ResourceBundle`, specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath. The default file name is `views.properties`.

ViewResolver	Description
UrlBasedViewResolver	A simple implementation of the ViewResolver interface that effects the direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
InternalResourceViewResolver	A convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses such as JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass(..). See the Javadocs for the UrlBasedViewResolver class for details.
VelocityViewResolver / FreeMarkerViewResolver	A convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them.

As an example, when using JSP for a view technology you can use the UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over the RequestDispatcher to render the view.

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

When returning test as a viewname, this view resolver will hand the request over to the RequestDispatcher that will send the request to /WEB-INF/jsp/test.jsp.

When mixing different view technologies in a web application, you can use the ResourceBundleViewResolver:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].class as the view class and the value of the property [viewname].url as the view url. As you can see, you can identify a parent view, from which all views in the properties file sort of extend. This way you can specify a default view class, for example.

A note on caching - subclasses of AbstractCachingViewResolver cache view instances they have resolved. This greatly improves performance when using certain view technologies. It's possible to turn off the cache, by setting the cache property to false. Furthermore, if you have the requirement to be able to refresh a certain view at runtime (for example when a Velocity template has

been modified), you can use the `removeFromCache(String viewName, Locale loc)` method.

13.5.2. Chaining ViewResolvers

Spring supports more than just one view resolver. This allows you to chain resolvers and, for example, override specific views in certain circumstances. Chaining view resolvers is pretty straightforward - just add more than one resolver to your application context and, if necessary, set the `order` property to specify an order. Remember, the higher the order property, the later the view resolver will be positioned in the chain.

In the following example, the chain of view resolvers consists of two resolvers, a `InternalResourceViewResolver` (which is always automatically positioned as the last resolver in the chain) and an `XmlViewResolver` for specifying Excel views (which are not supported by the `InternalResourceViewResolver`):

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

<bean id="excelViewResolver"
class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

If a specific view resolver does not result in a view, Spring will inspect the context to see if other view resolvers are configured. If there are additional view resolvers, it will continue to inspect them. If not, it will throw an `Exception`.

You have to keep something else in mind - the contract of a view resolver mentions that a view resolver *can* return null to indicate the view could not be found. Not all view resolvers do this however! This is because in some cases, the resolver simply cannot detect whether or not the view exists. For example, the `InternalResourceViewResolver` uses the `RequestDispatcher` internally, and dispatching is the only way to figure out if a JSP exists - this can only be done once. The same holds for the `VelocityViewResolver` and some others. Check the Javadoc for the view resolver to see if you're dealing with a view resolver that does not report non-existing views. As a result of this, putting an `InternalResourceViewResolver` in the chain in a place other than the last, will result in the chain not being fully inspected, since the `InternalResourceViewResolver` will *always* return a view!

13.5.3. Redirecting to views

As has been mentioned, a controller normally returns a logical view name, which a view resolver resolves to a particular view technology. For view technologies such as JSPs that are actually processed via the Servlet/JSP engine, this is normally handled via `InternalResourceViewResolver` / `InternalResourceView` which will ultimately end up issuing an internal forward or include, via the Servlet API's `RequestDispatcher.forward(..)` or `RequestDispatcher.include()`. For other view technologies, such as Velocity, XSLT, etc., the view itself produces the content on the response stream.

It is sometimes desirable to issue an HTTP redirect back to the client, before the view is rendered. This is desirable for example when one controller has been called with `POST`ed data, and the response is actually a delegation to another controller (for example on a successful form submission). In this case, a normal internal forward will mean the other controller will also see the same `POST` data, which is potentially problematic if it can confuse it with other expected data. Another reason to do a redirect before displaying the result is that this will eliminate the possibility of the user doing a double submission of form data. The browser will have sent the initial `POST`, will have seen a redirect back and done a subsequent `GET` because of that, and thus as far as it is concerned, the current page does not reflect the result of a `POST`, but rather of a `GET`, so there is no way the user can accidentally re-`POST` the same data by doing a refresh. The refresh would just force a `GET` of the result page, not a resend of the initial `POST` data.

13.5.3.1. `RedirectView`

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring's `RedirectView`. In this case, `DispatcherServlet` will not use the normal view resolution mechanism, but rather as it has been given the (redirect) view already, will just ask it to do its work.

The `RedirectView` simply ends up issuing an `HttpServletResponse.sendRedirect()` call, which will come back to the client browser as an HTTP redirect. All model attributes are simply exposed as HTTP query parameters. This does mean that the model must contain only objects (generally Strings or convertible to Strings) which can be readily converted to a string-form HTTP query parameter.

If using `RedirectView` and the view is created by the controller itself, it is preferable for the redirect URL to be injected into the controller so that it is not baked into the controller but configured in the context along with the view names.

13.5.3.2. The `redirect:` prefix

While the use of `RedirectView` works fine, if the controller itself is creating the `RedirectView`, there is no getting around the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really care about how the response gets handled... it should generally think only in terms of view names that have been injected into it.

The special `redirect:` prefix allows this to be achieved. If a view name is returned which has the prefix redirect:, then `UrlBasedViewResolver` (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a `RedirectView`, but now the controller

itself can deal just in terms of logical view names. A logical view name such as `redirect:/my/response/controller.html` will redirect relative to the current servlet context, while a name such as `redirect:http://myhost.com/some/arbitrary/path.html` will redirect to an absolute URL. The important thing is that as long is this redirect view name is injected into the controller like any other logical view name, the controller is not even aware that redirection is happening.

13.5.3.3. The `forward:` prefix

It is also possible to use a special `forward:` prefix for view names that will ultimately be resolved by `UrlBasedViewResolver` and subclasses. All this does is create an `InternalResourceView` (which ultimately does a `RequestDispatcher.forward()`) around the rest of the view name, which is considered a URL. Therefore, there is never any use in using this prefix when using `InternalResourceViewResolver` / `InternalResourceView` anyway (for JSPs for example), but it's of potential use when you are primarily using another view technology, but still want to force a forward to happen to a resource to be handled by the Servlet/JSP engine. (Note that you may also chain multiple view resolvers, instead.)

As with the `redirect:` prefix, if the view name with the prefix is just injected into the controller, the controller does not have to be aware that anything special is happening in terms of handling the response.

13.6. Using locales

Most parts of Spring's architecture support internationalization, just as the Spring web MVC framework does. `DispatcherServlet` enables you to automatically resolve messages using the client's locale. This is done with `LocaleResolver` objects.

When a request comes in, the `DispatcherServlet` looks for a locale resolver and if it finds one it tries to use it to set the locale. Using the `RequestContext.getLocale()` method, you can always retrieve the locale that was resolved by the locale resolver.

Besides the automatic locale resolution, you can also attach an interceptor to the handler mapping (see Section 13.4.3, “Intercepting requests - the HandlerInterceptor interface” for more information on handler mapping interceptors), to change the locale under specific circumstances, based on a parameter in the request, for example.

Locale resolvers and interceptors are all defined in the `org.springframework.web.servlet.i18n` package, and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

13.6.1. `AcceptHeaderLocaleResolver`

This locale resolver inspects the `accept-language` header in the request that was sent by the browser of the client. Usually this header field contains the locale of the client's operating system.

13.6.2. `CookieLocaleResolver`

This locale resolver inspects a `Cookie` that might exist on the client, to see if a locale is specified. If so, it uses that specific locale. Using the properties of this locale resolver, you can specify the name of

the cookie, as well as the maximum age. Find below an example of defining a `CookieLocaleResolver`.

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when
browser shuts down) -->
    <property name="cookieMaxAge" value="100000">

</bean>
```

Table 13.6. `CookieLocaleResolver` properties

Property	Default	Description
cookieName	classname + LOCALE	The name of the cookie
cookieMaxAge	Integer.MAX_INT	The maximum time a cookie will stay persistent on the client. If -1 is specified, the cookie will not be persisted. It will only be available until the client shuts down his or her browser.
cookiePath	/	Using this parameter, you can limit the visibility of the cookie to a certain part of your site. When cookiePath is specified, the cookie will only be visible to that path, and the paths below it.

13.6.3. `SessionLocaleResolver`

The `SessionLocaleResolver` allows you to retrieve locales from the session that might be associated with the user's request.

13.6.4. `LocaleChangeInterceptor`

You can build in changing of locales using the `LocaleChangeInterceptor`. This interceptor needs to be added to one of the handler mappings (see Section 13.4, “Handler mappings”). It will detect a parameter in the request and change the locale (it calls `setLocale()` on the `LocaleResolver` that also exists in the context).

```
<bean id="localeChangeInterceptor"
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
```

```
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

All calls to all `*.view` resources containing a parameter named `siteLanguage` will now change the locale. So a request for the following URL, `http://www.sf.net/home.view?siteLanguage=nl` will change the site language to Dutch.

13.7. Using themes

13.7.1. Introduction

The *theme* support provided by the Spring web MVC framework enables you to further enhance the user experience by allowing the look and feel of your application to be *themed*. A theme is basically a collection of static resources affecting the visual style of the application, typically style sheets and images.

13.7.2. Defining themes

When you want to use themes in your web application you'll have to set up a `org.springframework.ui.context.ThemeSource`. The `WebApplicationContext` interface extends `ThemeSource` but delegates its responsibilities to a dedicated implementation. By default the delegate will be a `org.springframework.ui.context.support.ResourceBundleThemeSource` that loads properties files from the root of the classpath. If you want to use a custom `ThemeSource` implementation or if you need to configure the basename prefix of the `ResourceBundleThemeSource`, you can register a bean in the application context with the reserved name "themeSource". The web application context will automatically detect that bean and start using it.

When using the `ResourceBundleThemeSource`, a theme is defined in a simple properties file. The properties file lists the resources that make up the theme. Here is an example:

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

The keys of the properties are the names used to refer to the themed elements from view code. For a JSP this would typically be done using the `spring:theme` custom tag, which is very similar to the `spring:message` tag. The following JSP fragment uses the theme defined above to customize the look and feel:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
   <head>
      <link rel="stylesheet" href="<spring:theme code="styleSheet"/>"
type="text/css"/>
   </head>
   <body background="<spring:theme code="background"/>">
      ...
   </body>
</html>
```

By default, the `ResourceBundleThemeSource` uses an empty basename prefix. As a result the properties files will be loaded from the root of the classpath, so we'll have to put our `cool.properties` theme definition in a directory at the root of the classpath, e.g. in `/WEB-INF/classes`. Note that the `ResourceBundleThemeSource` uses the standard Java resource bundle loading mechanism, allowing for full internationalization of themes. For instance, we could have a `/WEB-INF/classes/cool_nl.properties` that references a special background image, e.g. with Dutch text on it.

13.7.3. Theme resolvers

Now that we have our themes defined, the only thing left to do is decide which theme to use. The `DispatcherServlet` will look for a bean named "themeResolver" to find out which `ThemeResolver` implementation to use. A theme resolver works in much the same way as a `LocaleResolver`. It can detect the theme that should be used for a particular request and can also alter the request's theme. The following theme resolvers are provided by Spring:

Table 13.7. `ThemeResolver` implementations

Class	Description
`FixedThemeResolver`	Selects a fixed theme, set using the "defaultThemeName" property.
`SessionThemeResolver`	The theme is maintained in the users HTTP session. It only needs to be set once for each session, but is not persisted between sessions.
`CookieThemeResolver`	The selected theme is stored in a cookie on the user-agent's machine.

Spring also provides a `ThemeChangeInterceptor`, which allows changing the theme on every request by including a simple request parameter.

13.8. Spring's multipart (fileupload) support

13.8.1. Introduction

Spring has built-in multipart support to handle fileuploads in web applications. The design for the multipart support is done with pluggable `MultipartResolver` objects, defined in the `org.springframework.web.multipart` package. Out of the box, Spring provides `MultipartResolver`s for use with *Commons FileUpload* (http://jakarta.apache.org/commons/fileupload) and *COS FileUpload* (http://www.servlets.com/cos). How uploading files is supported will be described in the rest of this chapter.

By default, no multipart handling will be done by Spring, as some developers will want to handle multiparts themselves. You will have to enable it yourself by adding a multipart resolver to the web application's context. After you have done that, each request will be inspected to see if it contains a multipart. If no multipart is found, the request will continue as expected. However, if a multipart is found in the request, the `MultipartResolver` that has been declared in your context will be used. After that, the multipart attribute in your request will be treated like any other attribute.

13.8.2. Using the `MultipartResolver`

The following example shows how to use the `CommonsMultipartResolver`:

```
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
```

```
    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

This is an example using the `CosMultipartResolver`:

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.cos.CosMultipartResolver">

    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

Of course you also need to put the appropriate jars in your classpath for the multipart resolver to work. In the case of the `CommonsMultipartResolver`, you need to use `commons-fileupload.jar`; in the case of the `CosMultipartResolver`, use `cos.jar`.

Now that you have seen how to set Spring up to handle multipart requests, let's talk about how to actually use it. When the Spring `DispatcherServlet` detects a multi-part request, it activates the resolver that has been declared in your context and hands over the request. What the resolver then does is wrap the current `HttpServletRequest` into a `MultipartHttpServletRequest` that has support for multipart file uploads. Using the `MultipartHttpServletRequest` you can get information about the multiparts contained by this request and actually get access to the multipart files themselves in your controllers.

13.8.3. Handling a file upload in a form

After the `MultipartResolver` has finished doing its job, the request will be processed like any other. To use it, you create a form with an upload field (see immediately below), then let Spring bind the file onto your form (backing object). To actually let the user upload a file, we have to create a (HTML) form:

```
<html>
    <head>
        <title>Upload a file please</title>
    </head>
    <body>
        <h1>Please upload a file</h1>
        <form method="post" action="upload.form" enctype="multipart/form-data">
            <input type="file" name="file"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

As you can see, we've created a field named after the property of the bean that holds the `byte[]`. Furthermore we've added the encoding attribute (`enctype="multipart/form-data"`) which is necessary to let the browser know how to encode the multipart fields (do not forget this!).

Just as with any other property that's not automagically convertible to a string or primitive type, to be able to put binary data in your objects you have to register a custom editor with the `ServletRequestDatabinder`. There are a couple of editors available for handling files and setting the results on an object. There's a `StringMultipartEditor` capable of converting files to

Strings (using a user-defined character set) and there is a `ByteArrayMultipartEditor` which converts files to byte arrays. They function just as the `CustomDateEditor` does.

So, to be able to upload files using a (HTML) form, declare the resolver, a url mapping to a controller that will process the bean, and the controller itself.

```
<beans>
        <!-- lets use the Commons-based implementation of the MultipartResolver
interface -->
    <bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

    <bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /upload.form=fileUploadController
            </value>
        </property>
    </bean>

    <bean id="fileUploadController" class="examples.FileUploadController">
        <property name="commandClass" value="examples.FileUploadBean"/>
        <property name="formView" value="fileuploadform"/>
        <property name="successView" value="confirmation"/>
    </bean>

</beans>
```

After that, create the controller and the actual class to hold the file property.

```
public class FileUploadController extends SimpleFormController {

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors) throws ServletException, IOException {

         // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

         let's see if there's content there
        byte[] file = bean.getFile();
        if (file == null) {
             // hmm, that's strange, the user did not upload anything
        }

         // well, let's do nothing with the bean for now and return
        return super.onSubmit(request, response, command, errors);
    }

    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder
binder)
        throws ServletException {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor
```

```
        binder.registerCustomEditor(byte[].class, new
ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

public class FileUploadBean {

    private byte[] file;

    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }
}
```

As you can see, the `FileUploadBean` has a property typed `byte[]` that holds the file. The controller registers a custom editor to let Spring know how to actually convert the multipart objects the resolver has found to properties specified by the bean. In this example, nothing is done with the `byte[]` property of the bean itself, but in practice you can do whatever you want (save it in a database, mail it to somebody, etc).

An equivalent example in which a file is bound straight to a String-typed property on a (form backing) object might look like:

```
public class FileUploadController extends SimpleFormController {

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors) throws ServletException, IOException {

         // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

         let's see if there's content there
        String file = bean.getFile();
        if (file == null) {
             // hmm, that's strange, the user did not upload anything
        }

         // well, let's do nothing with the bean for now and return
        return super.onSubmit(request, response, command, errors);
    }

    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder
binder)
        throws ServletException {
        // to actually be able to convert Multipart instance to a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class, new
StringMultipartFileEditor());
```

```
        // now Spring knows how to handle multipart object and convert them
    }

}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

Of course, this last example only makes (logical) sense in the context of uploading a plain text file (it wouldn't work so well in the case of uploading an image file).

The third (and final) option is where one binds directly to a `MultipartFile` property declared on the (form backing) object's class. In this case one does not need to register any custom `PropertyEditor` because there is no type conversion to be performed.

```
public class FileUploadController extends SimpleFormController {

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors) throws ServletException, IOException {

         // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

         let's see if there's content there
        MultipartFile file = bean.getFile();
        if (file == null) {
             // hmm, that's strange, the user did not upload anything
        }

         // well, let's do nothing with the bean for now and return
        return super.onSubmit(request, response, command, errors);
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
```

```
}
```

13.9. Using Spring's form tag library

As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

Unlike other form/input tag libraries, Spring's form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As you will see in the following examples, the form tags make JSPs easier to develop, read and maintain.

Let's go through the form tags and look at an example of how each tag is used. We have included generated HTML snippets where certain tags require further commentary.

13.9.1. Configuration

The form tag library comes bundled in `spring.jar`. The library descriptor is called `spring-form.tld`.

To use the tags from this library, add the following directive to the top of your JSP page:

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

... where `form` is the tag name prefix you want to use for the tags from this library.

13.9.2. The `form` tag

This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding. It puts the command object in the `PageContext` so that the command object can be accessed by inner tags. *All the other tags in this library are nested tags of the `form` tag.*

Let's assume we have a domain object called `User`. It is a JavaBean with properties such as `firstName` and `lastName`. We will use it as the form backing object of our form controller which returns `form.jsp`. Below is an example of what `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

The `firstName` and `lastName` values are retrieved from the command object placed in the `PageContext` by the page controller. Keep reading to see more complex examples of how inner tags are used with the `form` tag.

The generated HTML looks like a standard form:

```
<form method="POST">
    <table>
      <tr>
          <td>First Name:</td>
          <td><input name="firstName" type="text" value="Harry"/></td>
          <td></td>
      </tr>
      <tr>
          <td>Last Name:</td>
          <td><input name="lastName" type="text" value="Potter"/></td>
          <td></td>
      </tr>
      <tr>
          <td colspan="3">
            <input type="submit" value="Save Changes" />
          </td>
      </tr>
    </table>
</form>
```

The preceding JSP assumes that the variable name of the form backing object is `'command'`. If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

```
<form:form commandName="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

13.9.3. The `input` tag

This tag renders an HTML 'input' tag with type 'text' using the bound value. For an example of this tag, see Section 13.9.2, “The form tag”.

13.9.4. The `checkbox` tag

This tag renders an HTML 'input' tag with type 'checkbox'.

Let's assume our `User` has preferences such as newsletter subscription and a list of hobbies. Below is an example of the `Preferences` class:

```
public class Preferences {

    private boolean receiveNewsletter;

    private String[] interests;

    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

The `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
            <td> </td>
        </tr>

        <tr>
            <td>Interests:</td>
            <td>
                <%-- Approach 2: Property is of an array or of type
java.util.Collection --%>
                Quidditch: <form:checkbox path="preferences.interests"
value="Quidditch"/>
```

```
                Herbology: <form:checkbox path="preferences.interests"
value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox
path="preferences.interests"
                    value="Defence Against the Dark Arts"/>
            </td>
            <td> </td>
        </tr>
        <tr>
            <td>Favourite Word:</td>
            <td>
                <%-- Approach 3: Property is of type java.lang.Object --%>
                Magic: <form:checkbox path="preferences.favouriteWord"
value="Magic"/>
            </td>
            <td> </td>
        </tr>
    </table>
</form:form>
```

There are 3 approaches to the `checkbox` tag which should meet all your checkbox needs.

- Approach One - When the bound value is of type `java.lang.Boolean`, the `input(checkbox)` is marked as 'checked' if the bound value is `true`. The `value` attribute corresponds to the resolved value of the `setValue(Object)` value property.
- Approach Two - When the bound value is of type `array` or `java.util.Collection`, the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` value is present in the bound `Collection`.
- Approach Three - For any other bound value type, the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` is equal to the bound value.

Note that regardless of the approach, the same HTML structure is generated. Below is an HTML snippet of some checkboxes:

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="checkbox"
value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox"
value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input name="preferences.interests"
type="checkbox"
            value="Defence Against the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
    <td> </td>
</tr>
```

What you might not expect to see is the additional hidden field after each checkbox. When a checkbox in an HTML page is *not* checked, its value will not be sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML in order for

Spring form data binding to work. The `checkbox` tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore ("_") for each checkbox. By doing this, you are effectively telling Spring that "*the checkbox was visible in the form and I want my object to which the form data will be bound to reflect the state of the checkbox no matter what*".

13.9.5. The `radiobutton` tag

This tag renders an HTML 'input' tag with type 'radio'.

A typical usage pattern will involve multiple tag instances bound to the same property but with different values.

```
<tr>
    <td>Sex:</td>
    <td>Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/> </td>
    <td> </td>
</tr>
```

13.9.6. The `password` tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" />
    </td>
</tr>
```

Please note that by default, the password value is *not* shown. If you do want the password value to be shown, then set the value of the `'showPassword'` attribute to true, like so.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq" showPassword="true" />
    </td>
</tr>
```

13.9.7. The `select` tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested `option` and `options` tags.

Let's assume a `User` has a list of skills.

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
    <td></td>
</tr>
```

If the `User's` skill were in Herbology, the HTML source of the 'Skills' row would look like:

```
<tr>
    <td>Skills:</td>
    <td><select name="skills" multiple="true">
        <option value="Potions">Potions</option>
        <option value="Herbology" selected="true">Herbology</option>
        <option value="Quidditch">Quidditch</option></select></td>
    <td></td>
</tr>
```

13.9.8. The `option` tag

This tag renders an HTML 'option'. It sets 'selected' as appropriate based on the bound value.

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

If the `User's` house was in Gryffindor, the HTML source of the 'House' row would look like:

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor" selected="true">Gryffindor</option>
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
 </tr>
```

13.9.9. The `options` tag

This tag renders a list of HTML 'option' tags. It sets the 'selected' attribute as appropriate based on the bound value.

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code"
itemLabel="name"/>
        </form:select>
    </td>
    <td></td>
</tr>
```

If the `User` lived in the UK, the HTML source of the 'Country' row would look like:

```
 <tr>
    <td>Country:</td>
    <tr>
        <td>Country:</td>
        <td>
            <select name="country">
                <option value="-">--Please Select</option>
                <option value="AT">Austria</option>
                <option value="UK" selected="true">United Kingdom</option>
                <option value="US">United States</option>
            </select>
        </td>
        <td></td>
    </tr>
    <td></td>
</tr>
```

As the example shows, the combined usage of an `option` tag with the `options` tag generates the same standard HTML, but allows you to explicitly specify a value in the JSP that is for display only (where it belongs) such as the default string in the example: "-- Please Select".

13.9.10. The `textarea` tag

This tag renders an HTML 'textarea'.

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20" /></td>
    <td><form:errors path="notes" /></td>
</tr>
```

13.9.11. The `hidden` tag

This tag renders an HTML 'input' tag with type 'hidden' using the bound value. To submit an unbound hidden value, use the HTML `input` tag with type 'hidden'.

```
<form:hidden path="house" />
```

If we choose to submit the 'house' value as a hidden one, the HTML would look like:

```
<input name="house" type="hidden" value="Gryffindor"/>
```

13.9.12. The `errors` tag

This tag renders field errors in an HTML 'span' tag. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

Let's assume we want to display all error messages for the `firstName` and `lastName` fields once we submit the form. We have a validator for instances of the `User` class called `UserValidator`.

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
```

```
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required",
"Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required",
"Field is required.");
    }
}
```

The `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName" /></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

If we submit a form with empty values in the `firstHame` and `lastName` fields, this is what the HTML would look like:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
```

```
        </tr>
    </table>
</form>
```

What if we want to display the entire list of errors for a given page? The example below shows that the `errors` tag also supports some basic wildcarding functionality.

- `path="*"` - displays all errors
- `path="lastName*"` - displays all errors associated with the `lastName` field

The example below will display a list of errors at the top of the page, followed by field-specific errors next to the fields:

```
<form:form>
    <form:errors path="*" cssClass="errorBox" />
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <td><form:errors path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

The HTML would look like:

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br/>Field is
required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
</form>
```

13.10. Handling exceptions

Spring provides `HandlerExceptionResolvers` to ease the pain of unexpected exceptions occurring while your request is being handled by a controller which matched the request. `HandlerExceptionResolvers` somewhat resemble the exception mappings you can define in the web application descriptor `web.xml`. However, they provide a more flexible way to handle exceptions. They provide information about what handler was executing when the exception was thrown. Furthermore, a programmatic way of handling exception gives you many more options for how to respond appropriately before the request is forwarded to another URL (the same end result as when using the servlet specific exception mappings).

Besides implementing the `HandlerExceptionResolver` interface, which is only a matter of implementing the `resolveException(Exception, Handler)` method and returning a `ModelAndView`, you may also use the `SimpleMappingExceptionResolver`. This resolver enables you to take the class name of any exception that might be thrown and map it to a view name. This is functionally equivalent to the exception mapping feature from the Servlet API, but it's also possible to implement more fine grained mappings of exceptions from different handlers.

13.11. Convention over configuration

For a lot of projects, sticking to established conventions and having reasonable defaults is just what they (the projects) need... this theme of convention-over-configuration now has explicit support in Spring Web MVC. What this means is that if you establish a set of naming conventions and suchlike, you can *substantially* cut down on the amount of configuration that is required to set up handler mappings, view resolvers, `ModelAndView` instances, etc. This is a great boon with regards to rapid prototyping, and can also lend a degree of (always good-to-have) consistency across a codebase should you choose to move forward with it into production.

Tip

The Spring distribution ships with a web application that showcases the convention over configuration support described in this section. The application can be found in the `'samples/showcases/mvc-convention'` directory.

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

13.11.1. The Controller - `ControllerClassNameHandlerMapping`

The `ControllerClassNameHandlerMapping` class is a `HandlerMapping` implementation that uses a convention to determine the mapping between request URLs and the `Controller` instances that are to handle those requests.

An example; consider the following (simplistic) `Controller` implementation. Take especial notice of the *name* of the class.

```
public class ViewShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) {
        // the implementation is not hugely important for this example...
    }
```

```
}
```

Here is a snippet from the attendent Spring Web MVC configuration file...

```
<bean
class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMappi
ng"/>

<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">
    <!-- inject dependencies as required... -->
</bean>
```

The `ControllerClassNameHandlerMapping` finds all of the various handler (or `Controller`) beans defined in its application context and strips `'Controller'` off the name to define its handler mappings.

Let's look at some more examples so that the central idea becomes immediately familiar.

- `WelcomeController` maps to the `'/welcome*'` request URL
- `HomeController` maps to the `'/home*'` request URL
- `IndexController` maps to the `'/index*'` request URL
- `RegisterController` maps to the `'/register*'` request URL
- `DisplayShoppingCartController` maps to the `'/displayshoppingcart*'` request URL

  (Notice the casing - all lowercase - in the case of camel-cased `Controller` class names.)

In the case of `MultiActionController` handler classes, the mappings generated are (ever so slightly) more complex, but hopefully no less understandable. Some examples (all of the `Controller` names in this next bit are assumed to be `MultiActionController` implementations).

- `AdminController` maps to the `'/admin/*'` request URL
- `CatalogController` maps to the `'/catalog/*'` request URL

If you follow the pretty standard convention of naming your `Controller` implementations as `xxx`**`Controller`**, then the `ControllerClassNameHandlerMapping` will save you the tedium of having to firstly define and then having to maintain a potentially *looooong* `SimpleUrlHandlerMapping` (or suchlike).

The `ControllerClassNameHandlerMapping` class extends the `AbstractHandlerMapping` base class so you can define `HandlerInterceptor` instances and everything else just like you would with many other `HandlerMapping` implementations.

13.11.2. The Model - `ModelMap` (`ModelAndView`)

The `ModelMap` class is an essentially glorified `Map` that can make adding objects that are to be displayed in (or on) a `View` adhere to a common naming convention. Consider the following `Controller` implementation; notice that objects are added to the `ModelAndView` without any associated name being specified.

```
public class DisplayShoppingCartController implements Controller {
```

```
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) {

        List cartItems = // get a List of CartItem objects
        User user = // get the User doing the shopping

        ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- the
logical view name

        mav.addObject(cartItems); <-- look ma, no name, just the object
        mav.addObject(user); <-- and again ma!

        return mav;
    }
}
```

The `ModelAndView` class uses a `ModelMap` class that is a custom `Map` implementation that automatically generates a key for an object when an object is added to it. The strategy for determining the name for an added object is, in the case of a scalar object such as `User`, to use the short class name of the object's class. Find below some examples of the names that are generated for scalar objects put into a `ModelMap` instance.

- An `x.y.User` instance added will have the name `'user'` generated
- An `x.y.Registration` instance added will have the name `'registration'` generated
- An `x.y.Foo` instance added will have the name `'foo'` generated
- A `java.util.HashMap` instance added will have the name `'hashMap'` generated (you'll probably want to be explicit about the name in this case because `'hashMap'` is less than intuitive).
- Adding `null` will result in an `IllegalArgumentException` being thrown. If the object (or objects) that you are adding could potentially be `null`, then you will also want to be explicit about the name).

What, no automatic pluralisation?

Spring Web MVC's convention over configuration support does not support automatic pluralisation. That is to say, you cannot add a `List` of `Person` objects to a `ModelAndView` and have the generated name be 'people'.

This decision was taken after some debate, with the “Principle of Least Surprise” winning out in the end.

The strategy for generating a name after adding a `Set`, `List` or array object is to peek into the collection, take the short class name of the first object in the collection, and use that with `'List'` appended to the name. Some examples will make the semantics of name generation for collections clearer...

- An `x.y.User[]` array with one or more `x.y.User` elements added will have the name `'userList'` generated
- An `x.y.Foo[]` array with one or more `x.y.User` elements added will have the name `'fooList'` generated

- A `java.util.ArrayList` with one or more `x.y.User` elements added will have the name `'userList'` generated
- A `java.util.HashSet` with one or more `x.y.Foo` elements added will have the name `'fooList'` generated
- An **empty** `java.util.ArrayList` will not be added at all (i.e. the `addObject(..)` call will essentially be a no-op).

13.11.3. The View - `RequestToViewNameTranslator`

The `RequestToViewNameTranslator` interface is responsible for determining a logical `View` name when no such logical view name is explicitly supplied. It has just one implementation, the rather cunningly named `DefaultRequestToViewNameTranslator` class.

The `DefaultRequestToViewNameTranslator` maps request URLs to logical view names in a fashion that is probably best explained by recourse to an example.

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) {
        // process the request...
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }
}

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <!-- this bean with the well known name generates view names for us -->
    <bean id="viewNameTranslator"
class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator"/>

    <bean class="x.y.RegistrationController">
        <!-- inject dependencies as necessary -->
    </bean>

    <!-- maps request URLs to Controller names -->
    <bean
class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMappi
ng"/>

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

</beans>
```

Notice how in the implementation of the `handleRequest(..)` method no `View` or logical view name is ever set on the `ModelAndView` that is returned. It is the `DefaultRequestToViewNameTranslator` that will be tasked with generating a *logical view name* from the URL of the request. In the case of the above `RegistrationController`, which is being used in conjunction with the `ControllerClassNameHandlerMapping`, a request URL of `'http://localhost/registration.html'` will result in a logical view name of `'registration'` being generated by the `DefaultRequestToViewNameTranslator`. This logical view name will then be resolved into the `'/WEB-INF/jsp/registration.jsp'` view by the `InternalResourceViewResolver` bean.

Tip

You don't even need to define a `DefaultRequestToViewNameTranslator` bean explicitly. If you are okay with the default settings of the `DefaultRequestToViewNameTranslator`, then you can rely on the fact that the Spring Web MVC `DispatcherServlet` will actually instantiate an instance of this class if one is not explicitly configured.

Of course, if you need to change the default settings, then you do need to configure your own `DefaultRequestToViewNameTranslator` bean explicitly. Please do consult the quite comprehensive Javadoc for the `DefaultRequestToViewNameTranslator` class for details of the various properties that can be configured.

13.12. Further Resources

Find below links and pointers to further resources about Spring Web MVC.

- The Spring distribution ships with a Spring Web MVC tutorial that guides the reader through building a complete Spring Web MVC-based application using a step-by-step approach. This tutorial is available in the `'docs'` directory of the Spring distribution. An online version can also be found on the Spring Framework website.
- The book entitled “Expert Spring Web MVC and WebFlow” by Seth Ladd and others (published by Apress) is an excellent hardcopy source of Spring Web MVC goodness.