18. View technologies

18.1 Introduction

One of the areas in which Spring excels is in the separation of view technologies from the rest of the MVC framework. For example, deciding to use Velocity or XSLT in place of an existing JSP is primarily a matter of configuration. This chapter covers the major view technologies that work with Spring and touches briefly on how to add new ones. This chapter assumes you are already familiar with Section 17.5, "Resolving views" which covers the basics of how views in general are coupled to the MVC framework.

18.2 JSP & JSTL

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views. Using JSP or JSTL is done using a normal view resolver defined in the `WebApplicationContext`. Furthermore, of course you need to write some JSPs that will actually render the view.

Setting up your application to use JSTL is a common source of error, mainly caused by confusion over the different servlet spec., JSP and JSTL version numbers, what they mean and how to declare the taglibs correctly. The article How to Reference and Use JSTL in your Web Application provides a useful guide to the common pitfalls and how to avoid them. Note that as of Spring 3.0, the minimum supported servlet version is 2.4 (JSP 2.0 and JSTL 1.1), which reduces the scope for confusion somewhat.

18.2.1 View resolvers

Just as with any other view technology you're integrating with Spring, for JSPs you'll need a view resolver that will resolve your views. The most commonly used view resolvers when developing with JSPs are the `InternalResourceViewResolver` and the `ResourceBundleViewResolver`. Both are declared in the `WebApplicationContext`:

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

# And a sample properties file is uses (views.properties in WEB-INF/classes):
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp
```

```
productList.(class)=org.springframework.web.servlet.view.JstlView
productList.url=/WEB-INF/jsp/productlist.jsp
```

As you can see, the `ResourceBundleViewResolver` needs a properties file defining the view names mapped to 1) a class and 2) a URL. With a `ResourceBundleViewResolver` you can mix different types of views using only one resolver.

```
<bean id="viewResolver" 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>
```

The `InternalResourceBundleViewResolver` can be configured for using JSPs as described above. As a best practice, we strongly encourage placing your JSP files in a directory under the `'WEB-INF'` directory, so there can be no direct access by clients.

18.2.2 'Plain-old' JSPs versus JSTL

When using the Java Standard Tag Library you must use a special view class, the `JstlView`, as JSTL needs some preparation before things such as the I18N features will work.

18.2.3 Additional tags facilitating development

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags have *HTML escaping* features to enable or disable escaping of characters.

The tag library descriptor (TLD) is included in the `spring-webmvc.jar`. Further information about the individual tags can be found in the appendix entitled Appendix G, *spring.tld*.

18.2.4 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.

Configuration

The form tag library comes bundled in `spring-webmvc.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.

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="2">
                  <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>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value="Potter"/></td>
        </tr>
        <tr>
            <td colspan="2">
              <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="2">
                  <input type="submit" value="Save Changes" />
              </td>
          </tr>
      </table>
  </form:form>
```

The `input` tag

This tag renders an HTML 'input' tag using the bound value and type='text' by default. For an example of this tag, see the section called “The `form` tag”. Starting with Spring 3.1 you can use other types such HTML5-specific types like 'email', 'tel', 'date', and others.

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>
        </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>
        </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>
            </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>
  </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* ".

The `checkboxes` tag

This tag renders multiple HTML 'input' tags with type 'checkbox'.

Building on the example from the previous `checkbox` tag section. Sometimes you prefer not to have to list all the possible hobbies in your JSP page. You would rather provide a list at runtime of the available options and pass that in to the tag. That is the purpose of the `checkboxes` tag. You pass in an `Array`, a `List` or a `Map` containing the available options in the "items" property. Typically the bound property is a collection so it can hold multiple values selected by the user. Below is an example of the JSP using this tag:

```
<form:form>
      <table>
          <tr>
              <td>Interests:</td>
              <td>
                  <%-- Property is of an array or of type java.util.Collection --%>
                  <form:checkboxes path="preferences.interests" items="${interestList}"/>
              </td>
          </tr>
      </table>
  </form:form>
```

This example assumes that the "interestList" is a `List` available as a model attribute containing strings of the values to be selected from. In the case where you use a Map, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

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>
  </tr>
```

The `radiobuttons` tag

This tag renders multiple HTML 'input' tags with type 'radio'.

Just like the `checkboxes` tag above, you might want to pass in the available options as a runtime variable. For this usage you would use the `radiobuttons` tag. You pass in an `Array`, a `List` or a `Map` containing the available options in the "items" property. In the case where you use a Map, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

```
<tr>
      <td>Sex:</td>
      <td><form:radiobuttons path="sex" items="${sexOptions}"/></td>
  </tr>
```

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>
```

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>
  </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="selected">Herbology</option>
          <option value="Quidditch">Quidditch</option></select>
      </td>
  </tr>
```

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="selected">Gryffindor</option>
              <option value="Hufflepuff">Hufflepuff</option>
              <option value="Ravenclaw">Ravenclaw</option>
              <option value="Slytherin">Slytherin</option>
          </select>
      </td>
   </tr>
```

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>
  </tr>
```

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

```
<tr>
      <td>Country:</td>
      <td>
          <select name="country">
              <option value="-">--Please Select</option>
```

```
                <option value="AT">Austria</option>
                <option value="UK" selected="selected">United Kingdom</option>
                <option value="US">United States</option>
            </select>
        </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".

The `items` attribute is typically populated with a collection or array of item objects. `itemValue` and `itemLabel` simply refer to bean properties of those item objects, if specified; otherwise, the item objects themselves will be stringified. Alternatively, you may specify a `Map` of items, in which case the map keys are interpreted as option values and the map values correspond to option labels. If `itemValue` and/or `itemLabel` happen to be specified as well, the item value property will apply to the map key and the item label property will apply to the map value.

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>
```

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"/>
```

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 `firstName` 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
- if `path` is omitted - object errors only are displayed

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>
```

HTTP Method Conversion

A key principle of REST is the use of the Uniform Interface. This means that all resources (URLs) can be manipulated using the same four HTTP methods: GET, PUT, POST, and DELETE. For each method, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that is has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or simply do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form). This latter trick is what Spring's `HiddenHttpMethodFilter` does. This filter is a plain Servlet Filter and therefore it can be used in combination with any web framework (not just Spring MVC). Simply add this filter to your web.xml, and a POST with a hidden _method parameter will be converted into the corresponding HTTP method request.

To support HTTP method conversion the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet taken from the updated Petclinic sample

```
<form:form method="delete">
      <p class="submit"><input type="submit" value="Delete Pet"/></p>
</form:form>
```

This will actually perform an HTTP POST, with the 'real' DELETE method hidden behind a request parameter, to be picked up by the `HiddenHttpMethodFilter`, as defined in web.xml:

```
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>httpMethodFilter</filter-name>
    <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

The corresponding @Controller method is shown below:

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {
  this.clinic.deletePet(petId);
  return "redirect:/owners/" + ownerId;
}
```

HTML5 Tags

Starting with Spring 3, the Spring form tag library allows entering dynamic attributes, which means you can enter any HTML5 specific attributes.

In Spring 3.1, the form input tag supports entering a type attribute other than 'text'. This is intended to allow rendering new HTML5 specific input types such as 'email', 'date', 'range', and others. Note that entering type='text' is not required since 'text' is the default type.

18.3 Tiles

It is possible to integrate Tiles - just as any other view technology - in web applications using Spring. The following describes in a broad way how to do this.

NOTE: This section focuses on Spring's support for Tiles 2 (the standalone version of Tiles, requiring Java 5+) in the `org.springframework.web.servlet.view.tiles2` package as as well as Tiles 3 in the `org.springframework.web.servlet.view.tiles3` package. Spring also continues to support Tiles 1.x (a.k.a. "Struts Tiles", as shipped with Struts 1.1+; compatible with Java 1.4) in the original `org.springframework.web.servlet.view.tiles` package.

18.3.1 Dependencies

To be able to use Tiles you have to have a couple of additional dependencies included in your project. The following is the list of dependencies you need.

- `Tiles version 2.1.2 or higher`
- `Commons BeanUtils`
- `Commons Digester`
- `Commons Logging`

18.3.2 How to integrate Tiles

To be able to use Tiles, you have to configure it using files containing definitions (for basic information on definitions and other Tiles concepts, please have a look at http://tiles.apache.org). In Spring this is done using the `TilesConfigurer`. Have a look at the following piece of example ApplicationContext configuration:

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
  <property name="definitions">
```

```
      <list>
        <value>/WEB-INF/defs/general.xml</value>
        <value>/WEB-INF/defs/widgets.xml</value>
        <value>/WEB-INF/defs/administrator.xml</value>
        <value>/WEB-INF/defs/customer.xml</value>
        <value>/WEB-INF/defs/templates.xml</value>
      </list>
    </property>
</bean>
```

As you can see, there are five files containing definitions, which are all located in the `'WEB-INF/defs'` directory. At initialization of the `WebApplicationContext`, the files will be loaded and the definitions factory will be initialized. After that has been done, the Tiles includes in the definition files can be used as views within your Spring web application. To be able to use the views you have to have a `ViewResolver` just as with any other view technology used with Spring. Below you can find two possibilities, the `UrlBasedViewResolver` and the `ResourceBundleViewResolver`.

`UrlBasedViewResolver`

The `UrlBasedViewResolver` instantiates the given `viewClass` for each view it has to resolve.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.tiles2.TilesView"/>
</bean>
```

`ResourceBundleViewResolver`

The `ResourceBundleViewResolver` has to be provided with a property file containing viewnames and viewclasses the resolver can use:

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

```
...
welcomeView.(class)=org.springframework.web.servlet.view.tiles2.TilesView
welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.(class)=org.springframework.web.servlet.view.tiles2.TilesView
vetsView.url=vetsView (again, this is the name of a Tiles definition)

findOwnersForm.(class)=org.springframework.web.servlet.view.JstlView
findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp
...
```

As you can see, when using the `ResourceBundleViewResolver`, you can easily mix different view technologies.

Note that the `TilesView` class for Tiles 2 supports JSTL (the JSP Standard Tag Library) out of the box, whereas there is a separate `TilesJstlView` subclass in the Tiles 1.x support.

`SimpleSpringPreparerFactory` and `SpringBeanPreparerFactory`

As an advanced feature, Spring also supports two special Tiles 2 `PreparerFactory` implementations. Check out the Tiles documentation for details on how to use `ViewPreparer` references in your Tiles definition files.

Specify `SimpleSpringPreparerFactory` to autowire ViewPreparer instances based on specified preparer classes, applying Spring's container callbacks as well as applying configured Spring BeanPostProcessors. If Spring's context-wide annotation-config has been activated, annotations in ViewPreparer classes will be automatically detected and applied. Note that this expects preparer *classes* in the Tiles definition files, just like the default `PreparerFactory` does.

Specify `SpringBeanPreparerFactory` to operate on specified preparer *names* instead of classes, obtaining the corresponding Spring bean from the DispatcherServlet's application context. The full bean creation process will be in the control of the Spring application context in this case, allowing for the use of explicit dependency injection configuration, scoped beans etc. Note that you need to define one Spring bean definition per preparer name (as used in your Tiles definitions).

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/WEB-INF/defs/general.xml</value>
      <value>/WEB-INF/defs/widgets.xml</value>
      <value>/WEB-INF/defs/administrator.xml</value>
      <value>/WEB-INF/defs/customer.xml</value>
      <value>/WEB-INF/defs/templates.xml</value>
    </list>
  </property>

  <!-- resolving preparer names as Spring bean definition names -->
  <property name="preparerFactoryClass"
       value="org.springframework.web.servlet.view.tiles2.SpringBeanPreparerFactory"/>

</bean>
```

18.4 Velocity & FreeMarker

Velocity and FreeMarker are two templating languages that can be used as view technologies within Spring MVC applications. The languages are quite similar and serve similar needs and so are considered together in this section. For semantic and syntactic differences between the two languages, see the FreeMarker web site.

18.4.1 Dependencies

Your web application will need to include `velocity-1.x.x.jar` or `freemarker-2.x.jar` in order to work with Velocity or FreeMarker respectively and `commons-collections.jar` is required for Velocity. Typically they are included in the `WEB-INF/lib` folder where they are guaranteed to be found by a Java EE server and added to the classpath for your application. It is of course assumed that you already have the `spring-webmvc.jar` in your `'WEB-INF/lib'` directory too! If you make use of Spring's 'dateToolAttribute' or 'numberToolAttribute' in your Velocity views, you will also need to include the `velocity-tools-generic-1.x.jar`

18.4.2 Context configuration

A suitable configuration is initialized by adding the relevant configurer bean definition to your `'*-servlet.xml'` as shown below:

```
<!--
  This bean sets up the Velocity environment for us based on a root path for templates.
  Optionally, a properties file can be specified for more control over the Velocity
  environment, but the defaults are pretty sane for file based template loading.
-->
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
</bean>

<!--

  View resolvers can also be configured with ResourceBundles or XML files. If you need
  different view resolving based on Locale, you have to use the resource bundle resolver.

-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".vm"/>
</bean>
```

```
<!-- freemarker config -->
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>

<!--

  View resolvers can also be configured with ResourceBundles or XML files. If you need
  different view resolving based on Locale, you have to use the resource bundle resolver.

-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".ftl"/>
</bean>
```

For non web-apps add a `VelocityConfigurationFactoryBean` or a `FreeMarkerConfigurationFactoryBean` to your application context definition file.

18.4.3 Creating templates

Your templates need to be stored in the directory specified by the `*Configurer` bean shown above. This document does not cover details of creating templates for the two languages - please see their relevant websites for information. If you use the view resolvers highlighted, then the logical view names relate to the template file names in similar fashion to `InternalResourceViewResolver` for JSP's. So if your controller returns a ModelAndView object containing a view name of "welcome" then the resolvers will look for the `/WEB-INF/freemarker/welcome.ftl` or `/WEB-INF/velocity/welcome.vm` template as appropriate.

18.4.4 Advanced configuration

The basic configurations highlighted above will be suitable for most application requirements, however additional configuration options are available for when unusual or advanced requirements dictate.

velocity.properties

This file is completely optional, but if specified, contains the values that are passed to the Velocity runtime in order to configure velocity itself. Only required for advanced configurations, if you need this file, specify its location on the `VelocityConfigurer` bean definition above.

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="configLocation" value="/WEB-INF/velocity.properties"/>
</bean>
```

Alternatively, you can specify velocity properties directly in the bean definition for the Velocity config bean by replacing the "configLocation" property with the following inline properties.

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="velocityProperties">
    <props>
      <prop key="resource.loader">file</prop>
      <prop key="file.resource.loader.class">
        org.apache.velocity.runtime.resource.loader.FileResourceLoader
      </prop>
      <prop key="file.resource.loader.path">${webapp.root}/WEB-INF/velocity</prop>
      <prop key="file.resource.loader.cache">false</prop>
    </props>
  </property>
</bean>
```

Refer to the API documentation for Spring configuration of Velocity, or the Velocity documentation for examples and definitions of the `'velocity.properties'` file itself.

FreeMarker

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker `Configuration` object managed by Spring by setting the appropriate bean properties on the `FreeMarkerConfigurer` bean. The `freemarkerSettings` property requires a `java.util.Properties` object and the `freemarkerVariables` property requires a `java.util.Map`.

```
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
  <property name="freemarkerVariables">
    <map>
      <entry key="xml_escape" value-ref="fmXmlEscape"/>
    </map>
  </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

See the FreeMarker documentation for details of settings and variables as they apply to the `Configuration` object.

18.4.5 Bind support and form handling

Spring provides a tag library for use in JSP's that contains (amongst other things) a `<spring:bind/>` tag. This tag primarily enables forms to display values from form backing objects and to show the results of failed validations from a `Validator` in the web or business tier. From version 1.1, Spring now has support for the same functionality in both Velocity and FreeMarker, with additional convenience macros for generating form input elements themselves.

The bind macros

A standard set of macros are maintained within the `spring-webmvc.jar` file for both languages, so they are always available to a suitably configured application.

Some of the macros defined in the Spring libraries are considered internal (private) but no such scoping exists in the macro definitions making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to be directly calling from within your templates. If you wish to view the macro code directly, the files are called spring.vm / spring.ftl and are in the packages `org.springframework.web.servlet.view.velocity` or `org.springframework.web.servlet.view.freemarker` respectively.

Simple binding

In your html forms (vm / ftl templates) that act as the 'formView' for a Spring form controller, you can use code similar to the following to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. Note that the name of the command object is "command" by default, but can be overridden in your MVC configuration by setting the 'commandName' bean property on your form controller. Example code is shown below for the `personFormV` and `personFormF` views configured earlier;

```
<!-- velocity macros are automatically available -->
<html>
...
<form action="" method="POST">
  Name:
  #springBind( "command.name" )
  <input type="text"
    name="${status.expression}"
    value="$!status.value" /><br>
  #foreach($error in $status.errorMessages) <b>$error</b> <br> #end
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

```
<!-- freemarker macros have to be imported into a namespace.  We strongly
recommend sticking to 'spring' -->
<#import "/spring.ftl" as spring />
<html>
...
<form action="" method="POST">
  Name:
  <@spring.bind "command.name" />
  <input type="text"
    name="${spring.status.expression}"
    value="${spring.status.value?default("")}" /><br>
  <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

`#springBind` / `<@spring.bind>` requires a 'path' argument which consists of the name of your command object (it will be 'command' unless you changed it in your FormController properties) followed by a period and the name of the field on the command object you wish to bind to. Nested fields can be used too such as "command.address.street". The `bind` macro assumes the default HTML escaping behavior specified by the ServletContext parameter `defaultHtmlEscape` in web.xml

The optional form of the macro called `#springBindEscaped` / `<@spring.bindEscaped>` takes a second argument and explicitly specifies whether HTML escaping should be used in the status error messages or values. Set to true or false as required. Additional form handling macros simplify the use of HTML escaping and these macros should be used wherever possible. They are explained in the next section.

Form input generation macros

Additional convenience macros for both languages simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and they can be mixed and matched with simple HTML or calls direct to the spring bind macros highlighted previously.

The following table of available macros show the VTL and FTL definitions and the parameter list that each takes.

Table 18.1. Table of macro definitions

macro	VTL definition	FTL definition

macro	VTL definition	FTL definition
message (output a string from a resource bundle based on the code parameter)	`#springMessage($code)`	`<@spring.message code/>`
messageText (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter)	`#springMessageText($code $text)`	`<@spring.messageText code, text/>`
url (prefix a relative URL with the application's context root)	`#springUrl($relativeUrl)`	`<@spring.url relativeUrl/>`
formInput (standard input field for gathering user input)	`#springFormInput($path $attributes)`	`<@spring.formInput path, attributes, fieldType/>`
formHiddenInput * (hidden input field for submitting non-user input)	`#springFormHiddenInput($path $attributes)`	`<@spring.formHiddenInput path, attributes/>`
formPasswordInput * (standard input field for gathering passwords. Note that no value will ever be populated in fields of this type)	`#springFormPasswordInput($path $attributes)`	`<@spring.formPasswordInput path, attributes/>`
formTextarea (large text field for gathering long, freeform text input)	`#springFormTextarea($path $attributes)`	`<@spring.formTextarea path, attributes/>`

macro	VTL definition	FTL definition
formSingleSelect (drop down box of options allowing a single required value to be selected)	`#springFormSingleSelect( $path $options $attributes)`	`<@spring.formSingleSelect path, options, attributes/>`
formMultiSelect (a list box of options allowing the user to select 0 or more values)	`#springFormMultiSelect($path $options $attributes)`	`<@spring.formMultiSelect path, options, attributes/>`
formRadioButtons (a set of radio buttons allowing a single selection to be made from the available choices)	`#springFormRadioButtons($path $options $separator $attributes)`	`<@spring.formRadioButtons path, options separator, attribut`
formCheckboxes (a set of checkboxes allowing 0 or more values to be selected)	`#springFormCheckboxes($path $options $separator $attributes)`	`<@spring.formCheckboxes path, options, separator, attribute`
formCheckbox (a single checkbox)	`#springFormCheckbox($path $attributes)`	`<@spring.formCheckbox path, attributes/>`
showErrors (simplify display of validation errors for the bound field)	`#springShowErrors($separator $classOrStyle)`	`<@spring.showErrors separator, classOrStyle/>`

* In FTL (FreeMarker), these two macros are not actually required as you can use the normal `formInput` macro, specifying '`hidden`' or '`password`' as the value for the `fieldType` parameter.

The parameters to any of the above macros have consistent meanings:

- path: the name of the field to bind to (ie "command.name")
- options: a Map of all the available values that can be selected from in the input field. The keys to the map represent the values that will be POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from the

corresponding values posted back by the form. Usually such a map is supplied as reference data by the controller. Any Map implementation can be used depending on required behavior. For strictly sorted maps, a `SortedMap` such as a `TreeMap` with a suitable Comparator may be used and for arbitrary Maps that should return values in insertion order, use a `LinkedHashMap` or a `LinkedMap` from commons-collections.

- separator: where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (ie "<br>").
- attributes: an additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field you may supply attributes as 'rows="5" cols="60"' or you could pass style information such as 'style="border:1px solid silver"'.
- classOrStyle: for the showErrors macro, the name of the CSS class that the span tag wrapping each error will use. If no information is supplied (or the value is empty) then the errors will be wrapped in <b></b> tags.

Examples of the macros are outlined below some in FTL and some in VTL. Where usage differences exist between the two languages, they are explained in the notes.

Input Fields

```
<!-- the Name field example from above using form macros in VTL -->
...
    Name:
    #springFormInput("command.name" "")<br>
    #springShowErrors("<br>" "")<br>
```

The formInput macro takes the path parameter (command.name) and an additional attributes parameter which is empty in the example above. The macro, along with all other form generation macros, performs an implicit spring bind on the path parameter. The binding remains valid until a new bind occurs so the showErrors macro doesn't need to pass the path parameter again - it simply operates on whichever field a bind was last created for.

The showErrors macro takes a separator parameter (the characters that will be used to separate multiple errors on a given field) and also accepts a second parameter, this time a class name or style attribute. Note that FreeMarker is able to specify default values for the attributes parameter, unlike Velocity, and the two macro calls above could be expressed as follows in FTL:

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

Output is shown below of the form fragment generating the name field, and displaying a validation error after the form was submitted with no value in the field. Validation occurs through Spring's Validation framework.

The generated HTML looks like this:

```
Name:
  <input type="text" name="name" value=""
>
<br>
  <b>required</b>
<br>
<br>
```

The formTextarea macro works the same way as the formInput macro and accepts the same parameter list. Commonly, the second parameter (attributes) will be used to pass style information or rows and cols attributes for the textarea.

Selection Fields

Four selection field macros can be used to generate common UI value selection inputs in your HTML forms.

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes

Each of the four macros accepts a Map of options containing the value for the form field, and the label corresponding to that value. The value and the label can be the same.

An example of radio buttons in FTL is below. The form backing object specifies a default value of 'London' for this field and so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'.

```
...
  Town:
  <@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>
```

This renders a line of radio buttons, one for each value in `cityMap` using the separator "". No additional attributes are supplied (the last parameter to the macro is missing). The cityMap uses the same String for each key-value pair in the map. The map's keys are what the form actually submits as POSTed request parameters, map values are the labels that the user sees. In the example above, given a list of three well known cities and a default value in the form backing object, the HTML would be

```
Town:
<input type="radio" name="address.town" value="London"

>
London
<input type="radio" name="address.town" value="Paris"
  checked="checked"
>
Paris
<input type="radio" name="address.town" value="New York"

>
New York
```

If your application expects to handle cities by internal codes for example, the map of codes would be created with suitable keys like the example below.

```
protected Map referenceData(HttpServletRequest request) throws Exception {
  Map cityMap = new LinkedHashMap();
  cityMap.put("LDN", "London");
  cityMap.put("PRS", "Paris");
  cityMap.put("NYC", "New York");

  Map m = new HashMap();
  m.put("cityMap", cityMap);
  return m;
}
```

The code would now produce output where the radio values are the relevant codes but the user still sees the more user friendly city names.

```
Town:
<input type="radio" name="address.town" value="LDN"

>
London
<input type="radio" name="address.town" value="PRS"
  checked="checked"
>
Paris
<input type="radio" name="address.town" value="NYC"

>
New York
```

HTML escaping and XHTML compliance

Default usage of the form macros above will result in HTML tags that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your web.xml as used by Spring's bind support. In order to make the tags XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model where they will be visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

To switch to XHTML compliance for your tags, specify a value of 'true' for a model/context variable named xhtmlCompliant:

```
## for Velocity..
#set($springXhtmlCompliant = true)

<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

Any tags generated by the Spring macros will now be XHTML compliant after processing this directive.

In similar fashion, HTML escaping can be specified per field:

```
<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>
<#-- next field will use HTML escaping -->
<@spring.formInput "command.name" />
```

```
<#assign htmlEscape = false in spring>
<#-- all future fields will be bound with HTML escaping off -->
```

18.5 XSLT

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring Web MVC application.

18.5.1 My First Words

This example is a trivial Spring application that creates a list of words in the `Controller` and adds them to the model map. The map is returned along with the view name of our XSLT view. See Section 17.3, “Implementing Controllers” for details of Spring Web MVC's `Controller` interface. The XSLT view will turn the list of words into a simple XML document ready for transformation.

Bean definitions

Configuration is standard for a simple Spring application. The dispatcher servlet config file contains a reference to a `ViewResolver`, URL mappings and a single controller bean...

```
<bean id="homeController"class="xslt.HomeController"/>
```

... that encapsulates our word generation logic.

Standard MVC controller code

The controller logic is encapsulated in a subclass of `AbstractController`, with the handler method being defined like so...

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

    Map map = new HashMap();
    List wordList = new ArrayList();

    wordList.add("hello");
    wordList.add("world");

    map.put("wordList", wordList);

    return new ModelAndView("home", map);
}
```

So far we've done nothing that's XSLT specific. The model data has been created in the same way as you would for any other Spring MVC application. Depending on the configuration of the application now, that list of words could be rendered by JSP/JSTL by having them added as request attributes, or they could be handled by Velocity by adding the object to the `VelocityContext`. In order to have XSLT render them, they of course have to be converted into an XML document somehow. There are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

Convert the model data to XML

In order to create a DOM document from our list of words or any other model data, we must subclass the (provided) `org.springframework.web.servlet.view.xslt.AbstractXsltView` class. In doing so, we must also typically implement the abstract method `createXsltSource(..)` method. The first parameter passed to this method is our model map. Here's the complete listing of the `HomePage` class in our trivial word application:

```java
package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

    protected Source createXsltSource(Map model, String rootName, HttpServletRequest
        request, HttpServletResponse response) throws Exception {

        Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element root = document.createElement(rootName);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(nextWord);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }
        return new DOMSource(root);
    }

}
```

A series of parameter name/value pairs can optionally be defined by your subclass which will be added to the transformation object. The parameter names must match those defined in your XSLT template declared with `<xsl:param name="myParam">defaultValue</xsl:param>`. To specify the parameters, override the `getParameters()` method of the `AbstractXsltView` class and return a `Map` of the name/value pairs. If your parameters need to derive information from the current request, you can override the `getParameters(HttpServletRequest request)` method instead.

Defining the view properties

The views.properties file (or equivalent xml definition if you're using an XML based view resolver as we did in the Velocity examples above) looks like this for the one-view application that is 'My First Words':

```
home.(class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words
```

Here, you can see how the view is tied in with the `HomePage` class just written which handles the model domification in the first property `'.(class)'`. The `'stylesheetLocation'` property points to the XSLT file which will handle the XML transformation into HTML for us and the final property `'.root'` is the name that will be used as the root of the XML document. This gets passed to the `HomePage` class above in the second parameter to the `createXsltSource(..)` method(s).

Document transformation

Finally, we have the XSLT code used for transforming the above document. As shown in the above `'views.properties'` file, the stylesheet is called `'home.xslt'` and it lives in the war file in the `'WEB-INF/xsl'` directory.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>
                <h1>My First Words</h1>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="word">
        <xsl:value-of select="."/><br/>
    </xsl:template>

</xsl:stylesheet>
```

18.5.2 Summary

A summary of the files discussed and their location in the WAR file is shown in the simplified WAR structure below.

```
ProjectRoot
  |
  +- WebContent
      |
      +- WEB-INF
          |
          +- classes
          |    |
          |    +- xslt
          |    |   |
          |    |   +- HomePageController.class
          |    |   +- HomePage.class
          |    |
          |    +- views.properties
          |
          +- lib
          |   |
          |   +- spring-*.jar
          |
          +- xsl
          |   |
          |   +- home.xslt
          |
          +- frontcontroller-servlet.xml
```

You will also need to ensure that an XML parser and an XSLT engine are available on the classpath. JDK 1.4 provides them by default, and most Java EE containers will also make them available by default, but it's a possible source of errors to be aware of.

18.6 Document views (PDF/Excel)

18.6.1 Introduction

Returning an HTML page isn't always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the correct content type to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.

In order to use Excel views, you need to add the 'poi' library to your classpath, and for PDF generation, the iText library.

18.6.2 Configuration and setup

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

Document view definitions

First, let's amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier:

```
home.(class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.(class)=excel.HomePage

pdf.(class)=pdf.HomePage
```

If you want to start with a template spreadsheet or a fillable PDF form to add your model data to, specify the location as the 'url' property in the view definition

Controller code

The controller code we'll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

Subclassing for Excel views

Exactly as we did for the XSLT example, we'll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass of `org.springframework.web.servlet.view.document.AbstractExcelView` (for Excel files generated by POI) or `org.springframework.web.servlet.view.document.AbstractJExcelView` (for JExcelApi-generated Excel files) and implementing the `buildExcelDocument()` method.

Here's the complete listing for our POI Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(
        Map model,
        HSSFWorkbook wb,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        HSSFSheet sheet;
        HSSFRow sheetRow;
        HSSFCell cell;

        // Go to the first sheet
        // getSheetAt: only if wb is created from an existing document
        // sheet = wb.getSheetAt(0);
        sheet = wb.createSheet("Spring");
        sheet.setDefaultColumnWidth((short) 12);

        // write a text at A1
        cell = getCell(sheet, 0, 0);
        setText(cell, "Spring-Excel test");

        List words = (List) model.get("wordList");
        for (int i=0; i < words.size(); i++) {
            cell = getCell(sheet, 2+i, 0);
            setText(cell, (String) words.get(i));

        }
    }
}
```

And the following is a view generating the same Excel file, now using JExcelApi:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractJExcelView {

    protected void buildExcelDocument(Map model,
        WritableWorkbook wb,
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {

        WritableSheet sheet = wb.createSheet("Spring", 0);

        sheet.addCell(new Label(0, 0, "Spring-Excel test"));

        List words = (List) model.get("wordList");
        for (int i = 0; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String) words.get(i)));
        }
    }
}
```

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

If you now amend the controller such that it returns `xl` as the name of the view (`return new ModelAndView("xl", map);`) and run your application again, you should find that the Excel spreadsheet is created and downloaded automatically when you request the same page as before.

Subclassing for PDF views

The PDF version of the word list is even simpler. This time, the class extends `org.springframework.web.servlet.view.document.AbstractPdfView` and implements the `buildPdfDocument()` method as follows:

```
package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

    protected void buildPdfDocument(
        Map model,
        Document doc,
        PdfWriter writer,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        List words = (List) model.get("wordList");

        for (int i=0; i<words.size(); i++)
            doc.add( new Paragraph((String) words.get(i)));

    }
}
```

Once again, amend the controller to return the `pdf` view with `return new ModelAndView("pdf", map);`, and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

18.7 JasperReports

JasperReports (http://jasperreports.sourceforge.net) is a powerful open-source reporting engine that supports the creation of report designs using an easily understood XML file format. JasperReports is capable of rendering reports in four different formats: CSV, Excel, HTML and PDF.

18.7.1 Dependencies

Your application will need to include the latest release of JasperReports, which at the time of writing was 0.6.1. JasperReports itself depends on the following projects:

- BeanShell
- Commons BeanUtils
- Commons Collections
- Commons Digester
- Commons Logging

- iText
- POI

JasperReports also requires a JAXP compliant XML parser.

18.7.2 Configuration

To configure JasperReports views in your Spring container configuration you need to define a `ViewResolver` to map view names to the appropriate view class depending on which format you want your report rendered in.

Configuring the `ViewResolver`

Typically, you will use the `ResourceBundleViewResolver` to map view names to view classes and files in a properties file.

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

Here we've configured an instance of the `ResourceBundleViewResolver` class that will look for view mappings in the resource bundle with base name `views`. (The content of this file is described in the next section.)

Configuring the `View`s

The Spring Framework contains five different `View` implementations for JasperReports, four of which correspond to one of the four output formats supported by JasperReports, and one that allows for the format to be determined at runtime:

Table 18.2. JasperReports `View` classes

Class Name	Render Format
`JasperReportsCsvView`	CSV
`JasperReportsHtmlView`	HTML
`JasperReportsPdfView`	PDF
`JasperReportsXlsView`	Microsoft Excel
`JasperReportsMultiFormatView`	The view is decided upon at runtime

Mapping one of these classes to a view name and a report file is a matter of adding the appropriate entries in the resource bundle configured in the previous section as shown here:

```
simpleReport.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
```

Here you can see that the view with name `simpleReport` is mapped to the `JasperReportsPdfView` class, causing the output of this report to be rendered in PDF format. The `url` property of the view is set to the location of the underlying report file.

About Report Files

JasperReports has two distinct types of report file: the design file, which has a `.jrxml` extension, and the compiled report file, which has a `.jasper` extension. Typically, you use the JasperReports Ant task to compile your `.jrxml` design file into a `.jasper` file before deploying it into your application. With the Spring Framework you can map either of these files to your report file and the framework will take care of compiling the `.jrxml` file on the fly for you. You should note that after a `.jrxml` file is compiled by the Spring Framework, the compiled report is cached for the lifetime of the application. Thus, to make changes to the file you will need to restart your application.

Using `JasperReportsMultiFormatView`

The `JasperReportsMultiFormatView` allows for the report format to be specified at runtime. The actual rendering of the report is delegated to one of the other JasperReports view classes - the `JasperReportsMultiFormatView` class simply adds a wrapper layer that allows for the exact implementation to be specified at runtime.

The `JasperReportsMultiFormatView` class introduces two concepts: the format key and the discriminator key. The `JasperReportsMultiFormatView` class uses the mapping key to look up the actual view implementation class, and it uses the format key to lookup up the mapping key. From a coding perspective you add an entry to your model with the format key as the key and the mapping key as the value, for example:

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

  String uri = request.getRequestURI();
  String format = uri.substring(uri.lastIndexOf(".") + 1);

  Map model = getModel();
  model.put("format", format);

  return new ModelAndView("simpleReportMulti", model);
}
```

In this example, the mapping key is determined from the extension of the request URI and is added to the model under the default format key: `format`. If you wish to use a different format key then you can configure this using the `formatKey` property of the `JasperReportsMultiFormatView` class.

By default the following mapping key mappings are configured in `JasperReportsMultiFormatView`:

Table 18.3. `JasperReportsMultiFormatView` Default Mapping Key Mappings

Mapping Key	View Class

Mapping Key	View Class
csv	`JasperReportsCsvView`
html	`JasperReportsHtmlView`
pdf	`JasperReportsPdfView`
xls	`JasperReportsXlsView`

So in the example above a request to URI /foo/myReport.pdf would be mapped to the `JasperReportsPdfView` class. You can override the mapping key to view class mappings using the `formatMappings` property of `JasperReportsMultiFormatView`.

18.7.3 Populating the `ModelAndView`

In order to render your report correctly in the format you have chosen, you must supply Spring with all of the data needed to populate your report. For JasperReports this means you must pass in all report parameters along with the report datasource. Report parameters are simple name/value pairs and can be added to the `Map` for your model as you would add any name/value pair.

When adding the datasource to the model you have two approaches to choose from. The first approach is to add an instance of `JRDataSource` or a `Collection` type to the model `Map` under any arbitrary key. Spring will then locate this object in the model and treat it as the report datasource. For example, you may populate your model like so:

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  model.put("myBeanData", beanData);
  return model;
}
```

The second approach is to add the instance of `JRDataSource` or `Collection` under a specific key and then configure this key using the `reportDataKey` property of the view class. In both cases Spring will wrap instances of `Collection` in a `JRBeanCollectionDataSource` instance. For example:

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  Collection someData = getSomeData();
  model.put("myBeanData", beanData);
  model.put("someData", someData);
  return model;
}
```

Here you can see that two `Collection` instances are being added to the model. To ensure that the correct one is used, we simply modify our view configuration as appropriate:

```
simpleReport.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
simpleReport.reportDataKey=myBeanData
```

Be aware that when using the first approach, Spring will use the first instance of `JRDataSource` or `Collection` that it encounters. If you need to place multiple instances of `JRDataSource` or `Collection` into the model you need to use the second approach.

18.7.4 Working with Sub-Reports

JasperReports provides support for embedded sub-reports within your master report files. There are a wide variety of mechanisms for including sub-reports in your report files. The easiest way is to hard code the report path and the SQL query for the sub report into your design files. The drawback of this approach is obvious: the values are hard-coded into your report files reducing reusability and making it harder to modify and update report designs. To overcome this you can configure sub-reports declaratively, and you can include additional data for these sub-reports directly from your controllers.

Configuring Sub-Report Files

To control which sub-report files are included in a master report using Spring, your report file must be configured to accept sub-reports from an external source. To do this you declare a parameter in your report file like so:

```
<parameter name="ProductsSubReport" class="net.sf.jasperreports.engine.JasperReport"/>
```

Then, you define your sub-report to use this sub-report parameter:

```
<subreport>
    <reportElement isPrintRepeatedValues="false" x="5" y="25" width="325"
        height="20" isRemoveLineWhenBlank="true" backcolor="#ffcc99"/>
    <subreportParameter name="City">
        <subreportParameterExpression><![CDATA[$F{city}]]></subreportParameterExpression>
    </subreportParameter>
    <dataSourceExpression><![CDATA[$P{SubReportData}]]></dataSourceExpression>
    <subreportExpression class="net.sf.jasperreports.engine.JasperReport">
                  <![CDATA[$P{ProductsSubReport}]]></subreportExpression>
</subreport>
```

This defines a master report file that expects the sub-report to be passed in as an instance of `net.sf.jasperreports.engine.JasperReports` under the parameter `ProductsSubReport`. When configuring your Jasper view class, you can instruct Spring to load a report file and pass it into the JasperReports engine as a sub-report using the `subReportUrls` property:

```
<property name="subReportUrls">
    <map>
        <entry key="ProductsSubReport" value="/WEB-INF/reports/subReportChild.jrxml"/>
    </map>
</property>
```

Here, the key of the `Map` corresponds to the name of the sub-report parameter in the report design file, and the entry is the URL of the report file. Spring will load this report file, compiling it if necessary, and pass it into the JasperReports engine under the given key.

Configuring Sub-Report Data Sources

This step is entirely optional when using Spring to configure your sub-reports. If you wish, you can still configure the data source for your sub-reports using static queries. However, if you want Spring to convert data returned in your `ModelAndView` into instances of `JRDataSource` then you need to specify which of the parameters in your `ModelAndView` Spring should convert. To do this, configure the list of parameter names using the `subReportDataKeys` property of your chosen view class:

```
<property name="subReportDataKeys" value="SubReportData"/>
```

Here, the key you supply **must** correspond to both the key used in your `ModelAndView` and the key used in your report design file.

18.7.5 Configuring Exporter Parameters

If you have special requirements for exporter configuration -- perhaps you want a specific page size for your PDF report -- you can configure these exporter parameters declaratively in your Spring configuration file using the `exporterParameters` property of the view class. The `exporterParameters` property is typed as a `Map`. In your configuration the key of an entry should be the fully-qualified name of a static field that contains the exporter parameter definition, and the value of an entry should be the value you want to assign to the parameter. An example of this is shown below:

```
<bean id="htmlReport" class="org.springframework.web.servlet.view.jasperreports.JasperReportsHtmlView">
  <property name="url" value="/WEB-INF/reports/simpleReport.jrxml"/>
  <property name="exporterParameters">
    <map>
      <entry key="net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER">
        <value>Footer by Spring!
          &lt;/td&gt;&lt;td width="50%"&gt;&amp;nbsp; &lt;/td&gt;&lt;/tr&gt;
          &lt;/table&gt;&lt;/body&gt;&lt;/html&gt;
        </value>
      </entry>
    </map>
  </property>
</bean>
```

Here you can see that the `JasperReportsHtmlView` is configured with an exporter parameter for `net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER` which will output a footer in the resulting HTML.

18.8 Feed Views

Both `AbstractAtomFeedView` and `AbstractRssFeedView` inherit from the base class `AbstractFeedView` and are used to provide Atom and RSS Feed views respectfully. They are based on java.net's ROME project and are located in the package `org.springframework.web.servlet.view.feed`.

`AbstractAtomFeedView` requires you to implement the `buildFeedEntries()` method and optionally override the `buildFeedMetadata()` method (the default implementation is empty), as shown below.

```
public class SampleContentAtomView extends AbstractAtomFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Feed feed,
      HttpServletRequest request) {
    // implementation omitted
  }

  @Override
  protected List<Entry> buildFeedEntries(Map<String, Object> model,
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {

    // implementation omitted
  }
}
```

Similar requirements apply for implementing `AbstractRssFeedView`, as shown below.

```
public class SampleContentAtomView extends AbstractRssFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Channel feed,
                                   HttpServletRequest request) {
    // implementation omitted
  }

  @Override
  protected List<Item> buildFeedItems(Map<String, Object> model,
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {
    // implementation omitted
  }

}
```

The `buildFeedItems()` and `buildFeedEntires()` methods pass in the HTTP request in case you need to access the Locale. The HTTP response is passed in only for the setting of cookies or other HTTP headers. The feed will automatically be written to the response object after the method returns.

For an example of creating an Atom view please refer to Alef Arendsen's SpringSource Team Blog entry.

18.9 XML Marshalling View

The `MarhsallingView` uses an XML `Marshaller` defined in the `org.springframework.oxm` package to render the response content as XML. The object to be marshalled can be set explicitly using `MarhsallingView`'s modelKey bean property. Alternatively, the view will iterate over all model properties and marshal only those types that are supported by the `Marshaller`. For more information on the functionality in the `org.springframework.oxm` package refer to the chapter Marshalling XML using O/X Mappers.

18.10 JSON Mapping View

The `MappingJackson2JsonView` (or `MappingJacksonJsonView` depending on the the Jackson version you have) uses the Jackson library's `ObjectMapper` to render the response content as JSON. By default, the entire contents of the model map (with the exception of framework-specific classes) will be encoded as JSON. For cases where the contents of the map need to be filtered, users may specify a specific set of model attributes to encode via the `RenderedAttributes` property. The `extractValueFromSingleKeyModel` property may also be used to have the value in single-key models extracted and serialized directly rather than as a map of model attributes.

JSON mapping can be customized as needed through the use of Jackson's provided annotations. When further control is needed, a custom `ObjectMapper` can be injected through the `ObjectMapper` property for cases where custom JSON serializers/deserializers need to be provided for specific types.

Prev Up Next
17. Web MVC framework Home 19. Integrating with other web frameworks