19. Integrating with other web frameworks

19.1 Introduction

This chapter details Spring's integration with third party web frameworks such as JSF, Struts, WebWork, and Tapestry.

Spring Web Flow

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 Web Flow website.

One of the core value propositions of the Spring Framework is that of enabling *choice*. In a general sense, Spring does not force one to use or buy into any particular architecture, technology, or methodology (although it certainly recommends some over others). This freedom to pick and choose the architecture, technology, or methodology that is most relevant to a developer and his or her development team is arguably most evident in the web area, where Spring provides its own web framework (Spring MVC), while at the same time providing integration with a number of popular third party web frameworks. This allows one to continue to leverage any and all of the skills one may have acquired in a particular web framework such as Struts, while at the same time being able to enjoy the benefits afforded by Spring in other areas such as data access, declarative transaction management, and flexible configuration and application assembly.

Having dispensed with the woolly sales patter (c.f. the previous paragraph), the remainder of this chapter will concentrate upon the meaty details of integrating your favorite web framework with Spring. One thing that is often commented upon by developers coming to Java from other languages is the seeming super-abundance of web frameworks available in Java. There are indeed a great number of web frameworks in the Java space; in fact there are far too many to cover with any semblance of detail in a single chapter. This chapter thus picks four of the more popular web frameworks in Java, starting with the Spring configuration that is common to all of the supported web frameworks, and then detailing the specific integration options for each supported web framework.

> Please note that this chapter does not attempt to explain how to use any of the supported web frameworks. For example, if you want to use Struts for the presentation layer of your web application, the assumption is that you are already familiar with Struts. If you need further details about any of the supported web frameworks themselves, please do consult Section 19.7, “Further Resources” at the end of this chapter.

19.2 Common configuration

Before diving into the integration specifics of each supported web framework, let us first take a look at the Spring configuration that is *not* specific to any one web framework. (This section is equally applicable to Spring's own web framework, Spring MVC.)

One of the concepts (for want of a better word) espoused by (Spring's) lightweight application model is that of a layered architecture. Remember that in a 'classic' layered architecture, the web layer is but one of many layers; it serves as one of the entry points into a server side application and it delegates to service objects (facades) defined in a service layer to satisfy business specific (and presentation-technology agnostic) use cases. In Spring, these service objects, any other business-specific objects, data access objects, etc. exist in a distinct 'business context', which contains *no* web or presentation layer objects (presentation objects such as Spring MVC controllers are typically configured in a distinct 'presentation context'). This section details how one configures a Spring container (a `WebApplicationContext`) that contains all of the 'business beans' in one's application.

On to specifics: all that one need do is to declare a `ContextLoaderListener` in the standard Java EE servlet `web.xml` file of one's web application, and add a `contextConfigLocation` <context-param/> section (in the same file) that defines which set of Spring XML configuration files to load.

Find below the <listener/> configuration:

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

</listener>
```

Find below the <context-param/> configuration:

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

If you don't specify the `contextConfigLocation` context parameter, the `ContextLoaderListener` will look for a file called `/WEB-INF/applicationContext.xml` to load. Once the context files are loaded, Spring creates a `WebApplicationContext` object based on the bean definitions and stores it in the `ServletContext` of the web application.

All Java web frameworks are built on top of the Servlet API, and so one can use the following code snippet to get access to this 'business context' `ApplicationContext` created by the `ContextLoaderListener`.

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

The `WebApplicationContextUtils` class is for convenience, so you don't have to remember the name of the `ServletContext` attribute. Its *getWebApplicationContext()* method will return `null` if an object doesn't exist under the `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` key. Rather than risk getting

`NullPointerExceptions` in your application, it's better to use the `getRequiredWebApplicationContext()` method. This method throws an exception when the `ApplicationContext` is missing.

Once you have a reference to the `WebApplicationContext`, you can retrieve beans by their name or type. Most developers retrieve beans by name and then cast them to one of their implemented interfaces.

Fortunately, most of the frameworks in this section have simpler ways of looking up beans. Not only do they make it easy to get beans from a Spring container, but they also allow you to use dependency injection on their controllers. Each web framework section has more detail on its specific integration strategies.

19.3 JavaServer Faces 1.1 and 1.2

JavaServer Faces (JSF) is the JCP's standard component-based, event-driven web user interface framework. As of Java EE 5, it is an official part of the Java EE umbrella.

For a popular JSF runtime as well as for popular JSF component libraries, check out the Apache MyFaces project. The MyFaces project also provides common JSF extensions such as MyFaces Orchestra: a Spring-based JSF extension that provides rich conversation scope support.

> Spring Web Flow 2.0 provides rich JSF support through its newly established Spring Faces module, both for JSF-centric usage (as described in this section) and for Spring-centric usage (using JSF views within a Spring MVC dispatcher). Check out the Spring Web Flow website for details!

The key element in Spring's JSF integration is the JSF 1.1 `VariableResolver` mechanism. On JSF 1.2, Spring supports the `ELResolver` mechanism as a next-generation version of JSF EL integration.

19.3.1 DelegatingVariableResolver (JSF 1.1/1.2)

The easiest way to integrate one's Spring middle-tier with one's JSF web layer is to use the `DelegatingVariableResolver` class. To configure this variable resolver in one's application, one will need to edit one's *faces-context.xml* file. After the opening `<faces-config/>` element, add an `<application/>` element and a `<variable-resolver/>` element within it. The value of the variable resolver should reference Spring's `DelegatingVariableResolver`; for example:

```
<faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en</supported-locale>
      <supported-locale>es</supported-locale>
    </locale-config>
    <message-bundle>messages</message-bundle>
  </application>
</faces-config>
```

The `DelegatingVariableResolver` will first delegate value lookups to the default resolver of the underlying JSF implementation and then to Spring's 'business context' `WebApplicationContext`. This allows one to easily inject dependencies into one's JSF-managed beans.

Managed beans are defined in one's `faces-config.xml` file. Find below an example where `#{userManager}` is a bean that is retrieved from the Spring 'business context'.

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>com.whatever.jsf.UserList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

19.3.2 SpringBeanVariableResolver (JSF 1.1/1.2)

`SpringBeanVariableResolver` is a variant of `DelegatingVariableResolver`. It delegates to the Spring's 'business context' `WebApplicationContext` *first* and then to the default resolver of the underlying JSF implementation. This is useful in particular when using request/session-scoped beans with special Spring resolution rules, e.g. Spring `FactoryBean` implementations.

Configuration-wise, simply define `SpringBeanVariableResolver` in your *faces-context.xml* file:

```
<faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.SpringBeanVariableResolver</variable-resolver>
    ...
  </application>
</faces-config>
```

19.3.3 SpringBeanFacesELResolver (JSF 1.2+)

`SpringBeanFacesELResolver` is a JSF 1.2 compliant `ELResolver` implementation, integrating with the standard Unified EL as used by JSF 1.2 and JSP 2.1. Like `SpringBeanVariableResolver`, it delegates to the Spring's 'business context' `WebApplicationContext` *first*, then to the default resolver of the underlying JSF implementation.

Configuration-wise, simply define `SpringBeanFacesELResolver` in your JSF 1.2 *faces-context.xml* file:

```
<faces-config>
  <application>
    <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
    ...
  </application>
</faces-config>
```

19.3.4 FacesContextUtils

A custom `VariableResolver` works well when mapping one's properties to beans in *faces-config.xml*, but at times one may need to grab a bean explicitly. The `FacesContextUtils` class makes this easy. It is similar to `WebApplicationContextUtils`, except that it takes a `FacesContext` parameter rather than a `ServletContext` parameter.

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

19.4 Apache Struts 1.x and 2.x

Struts used to be the *de facto* web framework for Java applications, mainly because it was one of the first to be released (June 2001). It has now been renamed to *Struts 1* (as opposed to Struts 2). Many applications still use it. Invented by Craig McClanahan, Struts is an open source project hosted by the Apache Software Foundation. At the time, it greatly simplified the JSP/Servlet programming paradigm and won over many developers who were using proprietary frameworks. It simplified the programming model, it was open source (and thus free as in beer), and it had a large community, which allowed the project to grow and become popular among Java web developers.

The following section discusses Struts 1 a.k.a. "Struts Classic".
Struts 2 is effectively a different product - a successor of WebWork 2.2 (as discussed in Section 19.5, “WebWork 2.x”), carrying the Struts brand now. Check out the Struts 2 Spring Plugin for the built-in Spring integration shipped with Struts 2. In general, Struts 2 is closer to WebWork 2.2 than to Struts 1 in terms of its Spring integration implications.

To integrate your Struts 1.x application with Spring, you have two options:

- Configure Spring to manage your Actions as beans, using the `ContextLoaderPlugin`, and set their dependencies in a Spring context file.

- Subclass Spring's `ActionSupport` classes and grab your Spring-managed beans explicitly using a *getWebApplicationContext()* method.

19.4.1 ContextLoaderPlugin

The `ContextLoaderPlugin` is a Struts 1.1+ plug-in that loads a Spring context file for the Struts `ActionServlet`. This context refers to the root `WebApplicationContext` (loaded by the `ContextLoaderListener`) as its parent. The default name of the context file is the name of the mapped servlet, plus *-servlet.xml*. If `ActionServlet` is defined in web.xml as `<servlet-name>action</servlet-name>`, the default is */WEB-INF/action-servlet.xml*.

To configure this plug-in, add the following XML to the plug-ins section near the bottom of your *struts-config.xml* file:

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

The location of the context configuration files can be customized using the '`contextConfigLocation`' property.

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
      value="/WEB-INF/action-servlet.xml,/WEB-INF/applicationContext.xml"/>
</plug-in>
```

It is possible to use this plugin to load all your context files, which can be useful when using testing tools like StrutsTestCase. StrutsTestCase's `MockStrutsTestCase` won't initialize Listeners on startup so putting all your context files in the plugin is a workaround. (A bug has been filed for this issue, but has been closed as 'Wont Fix').

After configuring this plug-in in *struts-config.xml*, you can configure your `Action` to be managed by Spring. Spring (1.1.3+) provides two ways to do this:

- Override Struts' default `RequestProcessor` with Spring's `DelegatingRequestProcessor`.
- Use the `DelegatingActionProxy` class in the `type` attribute of your `<action-mapping>`.

Both of these methods allow you to manage your Actions and their dependencies in the *action-servlet.xml* file. The bridge between the Action in *struts-config.xml* and *action-servlet.xml* is built with the action-mapping's "path" and the bean's "name". If you have the following in your *struts-config.xml* file:

```
<action path="/users" .../>
```

You must define that Action's bean with the "/users" name in *action-servlet.xml*:

```
<bean name="/users" .../>
```

DelegatingRequestProcessor

To configure the `DelegatingRequestProcessor` in your *struts-config.xml* file, override the "processorClass" property in the <controller> element. These lines follow the <action-mapping> element.

```
<controller>
  <set-property property="processorClass"
      value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>
```

After adding this setting, your Action will automatically be looked up in Spring's context file, no matter what the type. In fact, you don't even need to specify a type. Both of the following snippets will work:

```
<action path="/user" type="com.whatever.struts.UserAction"/>
<action path="/user"/>
```

If you're using Struts' *modules* feature, your bean names must contain the module prefix. For example, an action defined as `<action path="/user"/>` with module prefix "admin" requires a bean name with `<bean name="/admin/user"/>`.

If you are using Tiles in your Struts application, you must configure your <controller> with the `DelegatingTilesRequestProcessor` instead.

DelegatingActionProxy

If you have a custom `RequestProcessor` and can't use the `DelegatingRequestProcessor` or `DelegatingTilesRequestProcessor` approaches, you can use the `DelegatingActionProxy` as the type in your action-mapping.

```
<action path="/user" type="org.springframework.web.struts.DelegatingActionProxy"
    name="userForm" scope="request" validate="false" parameter="method">
  <forward name="list" path="/userList.jsp"/>
  <forward name="edit" path="/userForm.jsp"/>
</action>
```

The bean definition in *action-servlet.xml* remains the same, whether you use a custom `RequestProcessor` or the `DelegatingActionProxy`.

If you define your `Action` in a context file, the full feature set of Spring's bean container will be available for it: dependency injection as well as the option to instantiate a new `Action` instance for each request. To activate the latter, add *scope="prototype"* to your Action's bean definition.

```
<bean name="/user" scope="prototype" autowire="byName"
    class="org.example.web.UserAction"/>
```

19.4.2 ActionSupport Classes

As previously mentioned, you can retrieve the `WebApplicationContext` from the `ServletContext` using the `WebApplicationContextUtils` class. An easier way is to extend Spring's `Action` classes for Struts. For example, instead of subclassing Struts' `Action` class, you can subclass Spring's `ActionSupport` class.

The `ActionSupport` class provides additional convenience methods, like *getWebApplicationContext()*. Below is an example of how you might use this in an Action:

```
public class UserAction extends DispatchActionSupport {

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response) throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        WebApplicationContext ctx = getWebApplicationContext();
        UserManager mgr = (UserManager) ctx.getBean("userManager");
        // talk to manager for business logic
        return mapping.findForward("success");
    }
}
```

Spring includes subclasses for all of the standard Struts Actions - the Spring versions merely have *Support* appended to the name:

- `ActionSupport`,
- `DispatchActionSupport`,
- `LookupDispatchActionSupport` and
- `MappingDispatchActionSupport`.

The recommended strategy is to use the approach that best suits your project. Subclassing makes your code more readable, and you know exactly how your dependencies are resolved. In contrast, using the `ContextLoaderPlugin` allows you to easily add new dependencies in your context XML file. Either way, Spring provides some nice options for integrating with Struts.

19.5 WebWork 2.x

From the WebWork homepage:

“ *WebWork is a Java web-application development framework. It is built specifically with developer productivity and code simplicity in mind, providing robust support for building reusable UI templates, such as form controls, UI themes, internationalization, dynamic form parameter mapping to JavaBeans, robust client and server side validation, and much more.* ”

Web work's architecture and concepts are easy to understand, and the framework also has an extensive tag library as well as nicely decoupled validation.

One of the key enablers in WebWork's technology stack is an IoC container to manage Webwork Actions, handle the "wiring" of business objects, etc. Prior to WebWork version 2.2, WebWork used its own proprietary IoC container (and provided integration points so that one could integrate an IoC container such as Spring's into the mix). However, as of WebWork version 2.2, the default IoC container that is used within WebWork *is* Spring. This is obviously great news if one is a Spring developer, because it means that one is immediately familiar with the basics of IoC configuration, idioms, and suchlike within WebWork.

Now in the interests of adhering to the DRY (Don't Repeat Yourself) principle, it would be foolish to document the Spring-WebWork integration in light of the fact that the WebWork team have already written such a writeup. Please consult the Spring-WebWork integration page on the WebWork wiki for the full lowdown.

Note that the Spring-WebWork integration code was developed (and continues to be maintained and improved) by the WebWork developers themselves. So please refer first to the WebWork site and forums if you are having issues with the integration. But feel free to post comments and queries regarding the Spring-WebWork integration on the Spring support forums, too.

19.6 Tapestry 3.x and 4.x

From the Tapestry homepage:

“ *Tapestry is an open-source framework for creating dynamic, robust, highly scalable web applications in Java. Tapestry complements and builds upon the standard Java Servlet API, and so it works in any servlet container or application server.* ”

While Spring has its own powerful web layer, there are a number of unique advantages to building an enterprise Java application using a combination of Tapestry for the web user interface and the Spring container for the lower layers. This section of the web integration chapter attempts to detail a few best practices for combining these two frameworks.

A *typical* layered enterprise Java application built with Tapestry and Spring will consist of a top user interface (UI) layer built with Tapestry, and a number of lower layers, all wired together by one or more Spring containers. Tapestry's own reference documentation contains the following snippet of best practice advice. (Text that the author of this Spring section has added is contained within `[]` brackets.)

“ *A very successful design pattern in Tapestry is to keep pages and components very simple, and **delegate** as much logic as possible out to HiveMind [or Spring, or whatever] services. Listener methods should ideally do little more than marshal together the correct information and pass it over to a service.* ”

The key question then is: how does one supply Tapestry pages with collaborating services? The answer, ideally, is that one would want to dependency inject those services directly into one's Tapestry pages. In Tapestry, one can effect this dependency injection by a variety of means. This section is only going to enumerate the dependency injection means afforded by Spring. The real beauty of the rest of this Spring-Tapestry integration is that the elegant and flexible design of Tapestry itself makes doing this dependency injection of Spring-managed beans a cinch. (Another nice thing is that this Spring-Tapestry integration code was written - and continues to be maintained - by the Tapestry creator Howard M. Lewis Ship, so hats off to him for what is really some silky smooth integration).

19.6.1 Injecting Spring-managed beans

Assume we have the following simple Spring container definition (in the ubiquitous XML format):

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

<beans>
    <!-- the DataSource -->
    <jee:jndi-lookup id="dataSource" jndi-name="java:DefaultDS"/>
```

```
<bean id="hibSessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager"/>

<bean id="mapper"
      class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
    <property name="sessionFactory" ref="hibSessionFactory"/>
</bean>

<!-- (transactional) AuthenticationService -->
<bean id="authenticationService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="target">
        <bean class="com.whatever.services.service.user.AuthenticationServiceImpl">
            <property name="mapper" ref="mapper"/>
        </bean>
    </property>
    <property name="proxyInterfacesOnly" value="true"/>
    <property name="transactionAttributes">
        <value>
            *=PROPAGATION_REQUIRED
        </value>
    </property>
</bean>

<!-- (transactional) UserService -->
<bean id="userService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
```

```
        <property name="target">
            <bean class="com.whatever.services.service.user.UserServiceImpl">
                <property name="mapper" ref="mapper"/>
            </bean>
        </property>
        <property name="proxyInterfacesOnly" value="true"/>
        <property name="transactionAttributes">
            <value>
                *=PROPAGATION_REQUIRED
            </value>
        </property>
    </bean>

</beans>
```

Inside the Tapestry application, the above bean definitions need to be loaded into a Spring container, and any relevant Tapestry pages need to be supplied (injected) with the `authenticationService` and `userService` beans, which implement the `AuthenticationService` and `UserService` interfaces, respectively.

At this point, the application context is available to a web application by calling Spring's static utility function `WebApplicationContextUtils.getApplicationContext(servletContext)`, where servletContext is the standard `ServletContext` from the Java EE Servlet specification. As such, one simple mechanism for a page to get an instance of the `UserService`, for example, would be with code such as:

```
WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
    getRequestCycle().getRequestContext().getServlet().getServletContext());
UserService userService = (UserService) appContext.getBean("userService");
// ... some code which uses UserService
```

This mechanism does work. Having said that, it can be made a lot less verbose by encapsulating most of the functionality in a method in the base class for the page or component. However, in some respects it goes against the IoC principle; ideally you would like the page to not have to ask the context for a specific bean by name, and in fact, the page would ideally not know about the context at all.

Luckily, there is a mechanism to allow this. We rely upon the fact that Tapestry already has a mechanism to declaratively add properties to a page, and it is in fact the preferred approach to manage all properties on a page in this declarative fashion, so that Tapestry can properly manage their lifecycle as part of the page and component lifecycle.

This next section is applicable to Tapestry 3.x. If you are using Tapestry version 4.x, please consult the section entitled the section called “Dependency Injecting Spring Beans into Tapestry pages - Tapestry 4.x style”.

Dependency Injecting Spring Beans into Tapestry pages

First we need to make the `ApplicationContext` available to the Tapestry page or Component without having to have the `ServletContext`; this is because at the stage in the page's/component's lifecycle when we need to access the `ApplicationContext`, the `ServletContext` won't be easily available to the page, so we can't use `WebApplicationContextUtils.getApplicationContext(servletContext)` directly. One way is by defining a custom version of the Tapestry `IEngine` which exposes this for us:

```
package com.whatever.web.xportal;

// import ...

public class MyEngine extends org.apache.tapestry.engine.BaseEngine {

    public static final String APPLICATION_CONTEXT_KEY = "appContext";

    /**
     * @see org.apache.tapestry.engine.AbstractEngine#setupForRequest(org.apache.tapestry.request.RequestConte
     */
    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);

        // insert ApplicationContext in global, if not there
        Map global = (Map) getGlobal();
        ApplicationContext ac = (ApplicationContext) global.get(APPLICATION_CONTEXT_KEY);
        if (ac == null) {
            ac = WebApplicationContextUtils.getWebApplicationContext(
                context.getServlet().getServletContext()
            );
            global.put(APPLICATION_CONTEXT_KEY, ac);
        }
    }
}
```

This engine class places the Spring Application Context as an attribute called "appContext" in this Tapestry app's 'Global' object. Make sure to register the fact that this special IEngine instance should be used for this Tapestry application, with an entry in the Tapestry application definition file. For example:

```
file: xportal.application:
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">
<application
    name="Whatever xPortal"
    engine-class="com.whatever.web.xportal.MyEngine">
</application>
```

Component definition files

Now in our page or component definition file (*.page or *.jwc), we simply add property-specification elements to grab the beans we need out of the `ApplicationContext`, and create page or component properties for them. For example:

```
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>
```

The OGNL expression inside the property-specification specifies the initial value for the property, as a bean obtained from the context. The entire page definition might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="com.whatever.web.xportal.pages.Login">

    <property-specification name="username" type="java.lang.String"/>
    <property-specification name="password" type="java.lang.String"/>
    <property-specification name="error" type="java.lang.String"/>
    <property-specification name="callback" type="org.apache.tapestry.callback.ICallback" persistent="yes"/>
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>

    <bean name="delegate" class="com.whatever.web.xportal.PortalValidationDelegate"/>

    <bean name="validator" class="org.apache.tapestry.valid.StringValidator" lifecycle="page">
        <set-property name="required" expression="true"/>
        <set-property name="clientScriptingEnabled" expression="true"/>
    </bean>
    <component id="inputUsername" type="ValidField">
        <static-binding name="displayName" value="Username"/>
        <binding name="value" expression="username"/>
        <binding name="validator" expression="beans.validator"/>
    </component>

    <component id="inputPassword" type="ValidField">
        <binding name="value" expression="password"/>
       <binding name="validator" expression="beans.validator"/>
       <static-binding name="displayName" value="Password"/>
       <binding name="hidden" expression="true"/>
    </component>

</page-specification>
```

Adding abstract accessors

Now in the Java class definition for the page or component itself, all we need to do is add an abstract getter method for the properties we have defined (in order to be able to access the properties).

```
// our UserService implementation; will come from page definition
public abstract UserService getUserService();
// our AuthenticationService implementation; will come from page definition
public abstract AuthenticationService getAuthenticationService();
```

For the sake of completeness, the entire Java class, for a login page in this example, might look like this:

```
package com.whatever.web.xportal.pages;

/**
 *  Allows the user to login, by providing username and password.
 *  After successfully logging in, a cookie is placed on the client browser
 *  that provides the default username for future logins (the cookie
 *  persists for a week).
 */
public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    /** the key under which the authenticated user object is stored in the visit as */
    public static final String USER_KEY = "user";

    /** The name of the cookie that identifies a user **/
    private static final String COOKIE_NAME = Login.class.getName() + ".username";
    private final static int ONE_WEEK = 7 * 24 * 60 * 60;

    public abstract String getUsername();
    public abstract void setUsername(String username);

    public abstract String getPassword();
    public abstract void setPassword(String password);

    public abstract ICallback getCallback();
    public abstract void setCallback(ICallback value);

    public abstract UserService getUserService();
    public abstract AuthenticationService getAuthenticationService();


    protected IValidationDelegate getValidationDelegate() {
        return (IValidationDelegate) getBeans().getBean("delegate");
    }

    protected void setErrorField(String componentId, String message) {
        IFormComponent field = (IFormComponent) getComponent(componentId);
        IValidationDelegate delegate = getValidationDelegate();
        delegate.setFormComponent(field);
        delegate.record(new ValidatorException(message));
    }

    /**
```

```
     *  Attempts to login.
     * <p>
     *  If the user name is not known, or the password is invalid, then an error
     *  message is displayed.
     **/
    public void attemptLogin(IRequestCycle cycle) {

        String password = getPassword();

        // Do a little extra work to clear out the password.
        setPassword(null);
        IValidationDelegate delegate = getValidationDelegate();

        delegate.setFormComponent((IFormComponent) getComponent("inputPassword"));
        delegate.recordFieldInputValue(null);

        // An error, from a validation field, may already have occurred.
        if (delegate.getHasErrors()) {
            return;
        }

        try {
            User user = getAuthenticationService().login(getUsername(), getPassword());
           loginUser(user, cycle);
        }
        catch (FailedLoginException ex) {
            this.setError("Login failed: " + ex.getMessage());
            return;
        }
    }

    /**
     *  Sets up the {@link User} as the logged in user, creates
     *  a cookie for their username (for subsequent logins),
     *  and redirects to the appropriate page, or
     *  a specified page).
     **/
    public void loginUser(User user, IRequestCycle cycle) {

        String username = user.getUsername();

        // Get the visit object; this will likely force the
        // creation of the visit object and an HttpSession
        Map visit = (Map) getVisit();
        visit.put(USER_KEY, user);

        // After logging in, go to the MyLibrary page, unless otherwise specified
        ICallback callback = getCallback();

        if (callback == null) {
            cycle.activate("Home");
        }
        else {
            callback.performCallback(cycle);
        }

        IEngine engine = getEngine();
        Cookie cookie = new Cookie(COOKIE_NAME, username);
        cookie.setPath(engine.getServletPath());
        cookie.setMaxAge(ONE_WEEK);

        // Record the user's username in a cookie
        cycle.getRequestContext().addCookie(cookie);
        engine.forgetPage(getPageName());
    }

    public void pageBeginRender(PageEvent event) {
        if (getUsername() == null) {
            setUsername(getRequestCycle().getRequestContext().getCookieValue(COOKIE_NAME));
        }
    }
}
```

Dependency Injecting Spring Beans into Tapestry pages - Tapestry 4.x style

Effecting the dependency injection of Spring-managed beans into Tapestry pages in Tapestry version 4.x is *so* much simpler. All that is needed is a single add-on library, and some (small) amount of (essentially boilerplate) configuration. Simply package and deploy this library with the (any of the) other libraries required by your web application (typically in `WEB-INF/lib`).

You will then need to create and expose the Spring container using the method detailed previously. You can then inject Spring-managed beans into Tapestry very easily; if we are using Java 5, consider the `Login` page from above: we simply need to annotate the appropriate getter methods in order to dependency inject the Spring-managed `userService` and `authenticationService` objects (lots of the class definition has been elided for clarity).

```
package com.whatever.web.xportal.pages;

public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    @InjectObject("spring:userService")
    public abstract UserService getUserService();

    @InjectObject("spring:authenticationService")
    public abstract AuthenticationService getAuthenticationService();

}
```

We are almost done. All that remains is the HiveMind configuration that exposes the Spring container stored in the `ServletContext` as a HiveMind service; for example:

```
<?xml version="1.0"?>
<module id="com.javaforge.tapestry.spring" version="0.1.1">

    <service-point id="SpringApplicationInitializer"
```

```
        interface="org.apache.tapestry.services.ApplicationInitializer"
        visibility="private">
        <invoke-factory>
            <construct class="com.javaforge.tapestry.spring.SpringApplicationInitializer">
                <set-object property="beanFactoryHolder"
                    value="service:hivemind.lib.DefaultSpringBeanFactoryHolder" />
            </construct>
        </invoke-factory>
    </service-point>

    <!-- Hook the Spring setup into the overall application initialization. -->
    <contribution
        configuration-id="tapestry.init.ApplicationInitializers">
        <command id="spring-context"
            object="service:SpringApplicationInitializer" />
    </contribution>

</module>
```

If you are using Java 5 (and thus have access to annotations), then that really is it.

If you are not using Java 5, then one obviously doesn't annotate one's Tapestry page classes with annotations; instead, one simply uses good old fashioned XML to declare the dependency injection; for example, inside the `.page` or `.jwc` file for the `Login` page (or component):

```
<inject property="userService" object="spring:userService"/>
<inject property="authenticationService" object="spring:authenticationService"/>
```

In this example, we've managed to allow service beans defined in a Spring container to be provided to the Tapestry page in a declarative fashion. The page class does not know where the service implementations are coming from, and in fact it is easy to slip in another implementation, for example, during testing. This inversion of control is one of the prime goals and benefits of the Spring Framework, and we have managed to extend it throughout the stack in this Tapestry application.

19.7 Further Resources

Find below links to further resources about the various web frameworks described in this chapter.

- The JSF homepage
- The Struts homepage
- The WebWork homepage
- The Tapestry homepage

Prev Up Next

18. View technologies Home 20. Portlet MVC Framework