The Java Developer's Guide to Web Development Frameworks

an internet.com Developer eBook

contents

The Java Developer's Guide to Web Development Frameworks

Java Web Development the Wicket Way

By Daniel Carleton

Download the source code for this article at: http://assets.devx.com/sourcecode/20755.zip

The Web application frameworks available today offer a wealth of development styles. Well-known Web development technologies such as JavaServer Faces, ASP.NET, and the Google Web Toolkit use event-driven, component-oriented designs that resemble traditional GUI programming. This approach makes sense as Web applications become more like desktop applications in sophistication and functionality all the time. However, many of these frameworks can be unwieldy, requiring heavy tool support and having steep learning curves. Also, the mingling of code and markup often challenges testability, refactoring, and separation of concerns. A Java Web application framework called Wicket takes a lightweight approach to the component-oriented model to overcome these and other challenges.

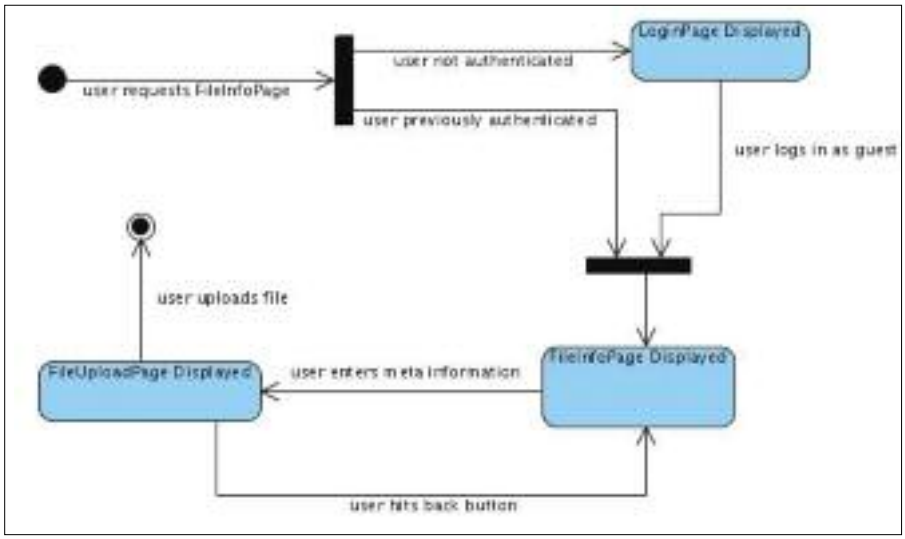
At its core, Wicket is a Java framework for processing markup. It represents Web applications, each page in them, and each component on each page as classes. Wicket then uses a simple naming convention to associate the classes responsible for dynamically manipulating markup with HTML files via the classpath. These files are truly just plain HTML — validating documents devoid of JavaServer Pages (JSP)-style "tag soup," and freely editable by designers.

With markup processing as its base, Wicket extends to support all aspects of modern Web application development by heavily and effectively using the composite and visitor patterns, as well as well-conceived inheritance hierarchies and interfaces.

Using test-driven design to develop a FileDropoff application example, this article illustrates Wicket's approach to testability, authentication and authorization, form handling, page nesting, file uploads with progress feedback, and the classic back-button problem. Throughout the development steps, the example highlights noteworthy pieces of code and describes what's going on behind

“A Java Web application framework called Wicket takes a lightweight approach to the component-oriented model to overcome these and other challenges.”

Figure 1: UML State Diagram for FileDropoff Application Example:



the scenes. For all the details, download the source code and read along.

The FileDropoff use case reads as follows (see Figure 1 for a user workflow):
1. The user logs in.
2. The user enters some meta information about a file on page one.
3. The user uploads the file on page two.
4. The user may use the back button to make corrections to the meta information.

This diagram shows user workflow for the FileDropoff application.

Step 1: Enforcing Page Authorization

Wicket's unique WicketTester class isolates applications in simulated servlet containers, and provides high-level methods for interacting with them. This enables test-driven design with functional tests such as the following, which specifies that anonymous users should be sent to the LoginPage:

```
@Test
public void shouldAuthChallenge()
{
   wicketTester.startPage(FileInfoPage.class);
   wicketTester.assertRenderedPage(LoginPage.class);
}
```

For an introduction to test-driven design, check out the DevX article, Efficient Test-Driven Design with Unitils at http://www.devx.com/Java/Article/35129/0.

To establish authorization and authentication, the application class must extend AuthenticatedWebApplication, specify a login page, and use a WebSession derived from AuthenticatedWebSession as follows:

```
public class ExampleWicketApplication extends AuthenticatedWebApplication
{
   @Override
   protected Class<? extends WebPage> getSignInPageClass()
   {
      return LoginPage.class;
   }
```

```
  @Override
  protected Class<? extends AuthenticatedWebSession> getWebSessionClass()
  {
    return ExampleWebSession.class;
  }

...
```

Sessions have many responsibilities in Wicket; they aren't simply generic containers for persistent variables as in other frameworks. In this case, ExampleWebSession implements the means of authentication (more on the role of sessions later). The following code adds authorization protection to the FileInfoPage:

```
@AuthorizeInstantiation("USER")
public class FileInfoPage extends BasePage
{

...
```

The shouldAuthChallenge test now passes, and you're ready to implement authentication.

Step 2: Enabling Authentication

Now that users are being sent to the LoginPage, you can test to assure that they are able to log in as a guest and get forwarded to the FileInfoPage:

```
  @Test
  public void shouldAllowGuestAuth()
  {
    wicketTester.startPage(FileInfoPage.class);
    wicketTester.assertRenderedPage(LoginPage.class);

    FormTester formTester
      = wicketTester.newFormTester("signInPanel:signInForm");
    formTester.setValue("username", "guest");
    formTester.setValue("password", "guest");
    formTester.submit();

    wicketTester.assertRenderedPage(FileInfoPage.class);
  }
```

This test takes a bit of research. You need to look into the markup of the stock SignInPanel component to determine the ID of the form ("signInForm") and its input IDs ("username" and "password"). Luckily this is simple to do using your IDE's package explorer. You may have noticed that having markup in disparate locations like this could break separation of concerns.

Wicket brings together LoginPage.java and LoginPage.html when rendering the LoginPage. One simple link exists between the two: the wicket:id HTML attribute, which anchors components at specific locations on the page:

```
public class LoginPage extends BasePage
{
  public LoginPage()
  {
    add(new SignInPanel("signInPanel"));
  }
}

<body>
  <wicket:extend>
    <div wicket:id="signInPanel"/>
  </wicket:extend>
</body>
```

The string signInPanel, which places the SignInPanel inside a div in the markup, is one of the only points where you lose type safety in Wicket applications. Luckily, WicketTester helps you catch typos in an automated fashion, and the framework provides detailed debugging output when components in the code don't match up to those specified in the markup.

Next, you implement authentication and authorization inside ExampleWebSession:

```
  @Override
  public boolean authenticate(String userName, String password)
  {
    boolean success = userName.equals("guest") && password.equals("guest");

    if ( success )
      this.userName = userName;

    return success;
  }

  @Override
  public Roles getRoles()
  {
    Roles roles = new Roles();

    if ( isSignedIn() )
      roles.add("USER");

    return roles;
  }
```

This example permits guests to authenticate, and it authorizes them to instantiate the FileInfo page via the USER role. Wicket automatically forwards guests to the original destination, and the "shouldAllowGuestAuth" test now passes.

Step 3: Markup Nesting Through Inheritance

Going back to the markup for LoginPage for a second, did you notice the "wicket:extend" tags? This is Wicket's object-oriented approach to markup nesting. In this case, the BasePage contains the following markup, which acts

as a header and footer for all pages that extend it:

```
<body>
  <h1>FileDropoff</h1>
  <wicket:child/>
  <p>
    <small>Created for DevX.com</small>
  </p>
</body>
```

The "wicket:child" tag is replaced with what's contained in the "wicket:extend" tags of children (see Figure 2).

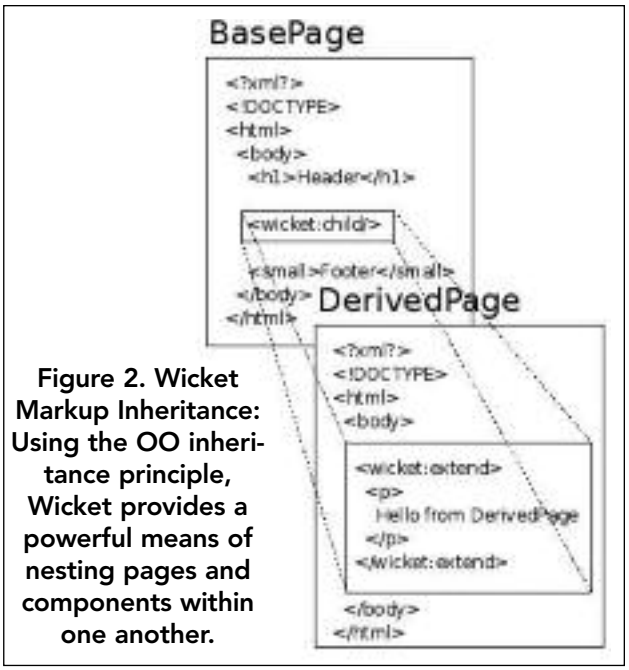


Figure 2. Wicket Markup Inheritance: Using the OO inheritance principle, Wicket provides a powerful means of nesting pages and components within one another.

Step 4: Form Handling, Component Style

Now that users can reach the FileInfoPage, you specify that they should be able to enter some info and advance to the upload page:

```
@Test
public void shouldAcceptInfoAndAdvance()
{
  shouldAllowGuestAuth();

  FormTester formTester = wicketTester.newFormTester("metaDataForm");
  formTester.setValue("title", "Alpine Lakes Trail");
  formTester.setValue("tags", "hike, forest, alpine lakes");
  formTester.submit();

  wicketTester.assertRenderedPage(FileUploadPage.class);
}
```

Now you will implement a component of your own: a simple form. Creating reusable components in Wicket is as simple as writing a small Java class that derives from the framework:

```
public class FileInfoPage extends BasePage
{
  private class FileInfoForm extends Form
  {

...
```

You will need a constructor that accepts a markup ID like SignInPanel does:

```
    public FileInfoForm(String id)
    {
      super(id);

      setModel(new CompoundPropertyModel(new UserContributedFile()));
```

```
        add(new RequiredTextField("title")
          .add(StringValidator.maximumLength(32)));
        add(new TextField("tags")
          .add(StringValidator.maximumLength(32)));
    }
```

Components have backing model objects that generally wrap instances of entity classes (note the call to setModel above). This example has one such class, UserContributedFile, the properties of which the form will allow users to edit. Different types of models facilitate different behaviors for things such as serialization and data access — in this case CompoundPropertyModel is the simplest choice.

Wicket expresses field validation rules as objects. You can see the composite pattern in use above, as you added text fields to your form and then validators to your text fields in chain invocation fashion. You can also imagine how Wicket will use the visitor pattern later to spider the fields and apply the validators on submission.

Wicket's Event-Driven Model and PageMap

In keeping with the use case, it's time to move on to FileUploadPage. Override the default onSubmit method in Form:

```
@Override
protected void onSubmit()
{
   super.onSubmit();

   FileUploadPage fileUploadPage = new FileUploadPage(getModel());
   setResponsePage(fileUploadPage);
}
```

Wicket's event-driven, unmanaged nature is clear here; whatever page you add a FileInfoForm to, submitting that instance of FileInfoForm executes the event handler and sends the user to a newly constructed FileUploadPage. The model object is already populated with validated values at this point, and you pass it to the next page's constructor. This is a departure from other frameworks, which often use the session to marshal objects from one page to the next.

Wicket maintains a server-side cache of the object graphs that represent each page rendered to the user. This makes your application a true state machine, and effectively solves the classic back button problem. For example, after advancing to the FileUploadPage, the user can hit the back button and Wicket will deserialize and render a copy of the object that represents the FileInfoPage the user was just on. The FileInfoForm's model is part of the object graph of the page, and so Wicket deserializes it and binds it to the form with all the values last submitted by the user intact. This is a much more effective way to manage moving back and forth in multipage workflows than juggling session variables.

Dynamic Markup Without JSP-Style Control Structures?

So how do you replicate a foreach loop in Wicket HTML? Decide whether to show or hide a certain block of markup? Conditionally show one design element versus another? You use Java code that acts on the members of a tiny XHTML namespace in your pages. The RepeatingView class allows you to iteratively populate the rows of a table based solely on the wicket:id value of a single row. All components have an isVisible property that you can toggle in your Java code, and you can add different components conditionally to the same location on a page.

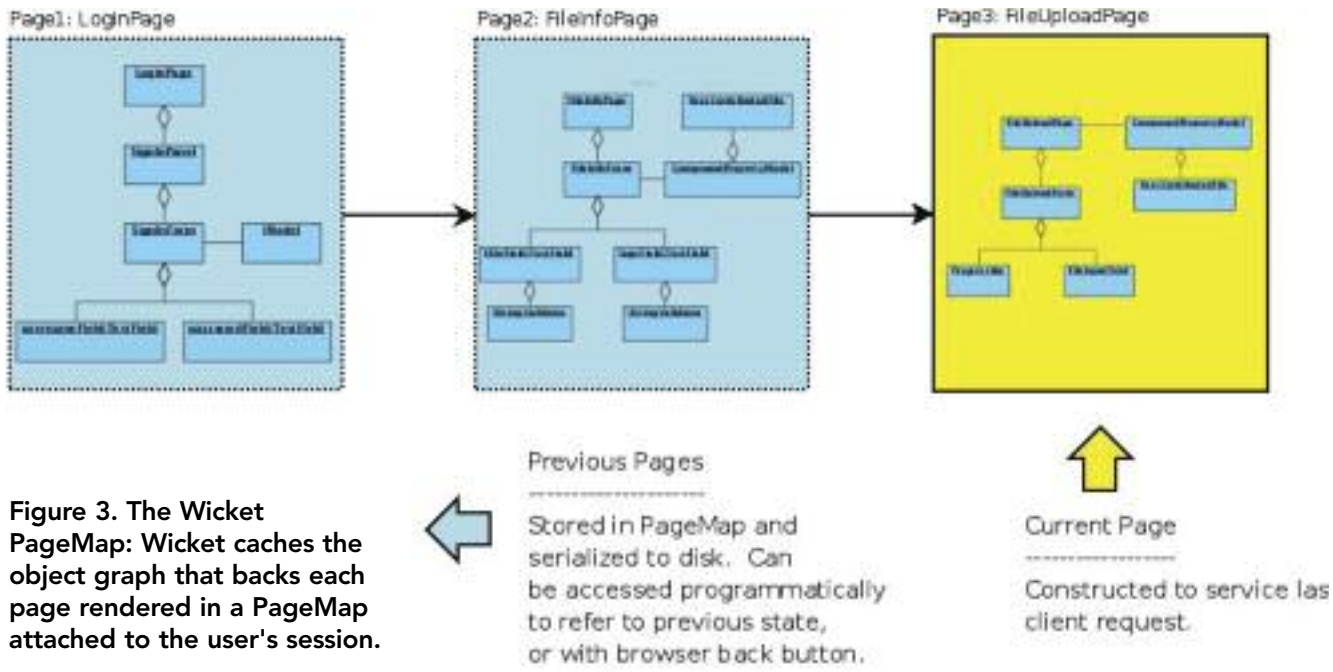


Figure 3. The Wicket PageMap: Wicket caches the object graph that backs each page rendered in a PageMap attached to the user's session.

Wicket caches the object graph that backs each page rendered in a PageMap attached to the user's session (see Figure 3). These previously visited pages are eventually garbage collected, but in the meantime the user can access them with the back button, or by utilizing the workflow code to repeat previous steps.

File Upload with AJAX Progress Feedback

The "shouldAcceptInfoAndAdvance" test now passes, and users can get as far as the FileUploadPage. The last step is for them to be able to upload their files, which you specify with the following test:

```
@Test
public void shouldAcceptFileUpload()
{
   shouldAcceptMetaAndAdvance();

   FormTester formTester = wicketTester.newFormTester("fileUploadForm");
   formTester.setFile("fileInput", TEST_UPLOAD_FILE, "image/jpeg");
   formTester.submit();

   // for simplicity we store the previously collected meta information in
   // the file name.
   String uploadedFilePath = TEST_UPLOAD_FOLDER.getAbsolutePath()
     .concat( File.separator )
     .concat("guest-Alpine Lakes Trail-hike+forest+alpine lakes.jpg");

   java.io.File uploadedFile = new java.io.File(uploadedFilePath);

   Assert.assertTrue
     ("File not deposited in upload folder or incorrectly named.",
         uploadedFile.exists());

   uploadedFile.delete();
```

```
   }
}
```

Now you implement your second Form component: FileUploadForm. You can read the onSubmit event handler later, but for now take a closer look at the constructor and Wicket's AJAX support:

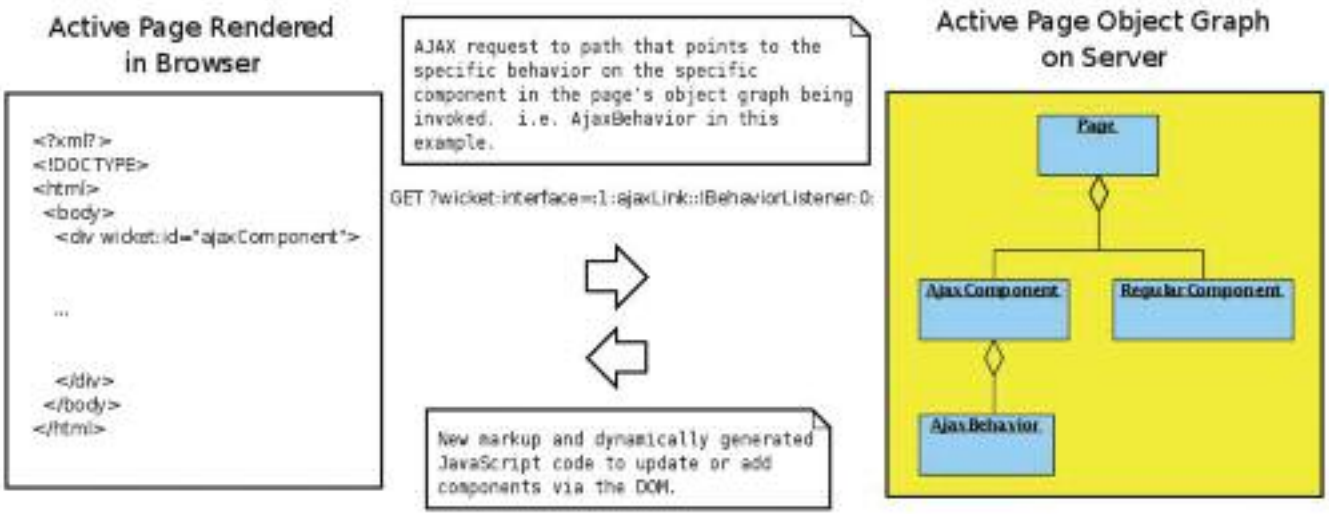
```
public class FileUploadPage extends BasePage
{
   private class FileUploadForm extends Form
   {
      private FileUploadField fileUploadField;

      public FileUploadForm(String id)
      {
         super(id);

         setOutputMarkupId(true);

         setMultiPart(true);
         setMaxSize(Bytes.megabytes(3));

         add(fileUploadField = new FileUploadField("fileInput"));
         add(new UploadProgressBar("progress", this));
      }

  ...
```

Is Wicket the Anti-REST? Does It Scale?

With all the positive talk about RESTful architecture these days, Wicket seems pretty state heavy on the server side. However, you can cluster Wicket servers for scalability, and "mount" UrlCodingStrategy-derived classes to your application to support REST-style URIs. (Investigate Terracotta and wicket-jetty-cluster for details on clustering.) Also, the Wicket team is aware of these concerns, and is working on the means to offload more state maintenance to the client in future versions.

Wicket implements AJAX in a manner consistent with the rest of its design (see Figure 4). By maintaining the full object graph of each page on the server side, the browser can easily communicate with individual components after a page is rendered. Each component on the client side can communicate with its server-side counterpart. Responses can update the state of existing components on the page, or add entirely new ones. Components are addressed using special strings that path into the object graph. Wicket communicates state changes, which can include updates to or additions of one or more components, back to the browser DOM using a thin JavaScript library.

Figure 4. AJAX the Wicket Way: Here is a rough overview of how Wicket implements AJAX.



The UploadProgressBar component in this example works in a somewhat involved manner, but it's still a good example of how Wicket can be extended to accommodate complex AJAX functionality without breaking encapsulation. Simply add the progress bar component to your form and configure the application to use UploadWebRequests in ExampleWicketApplication with this override:

```
  @Override
  protected WebRequest newWebRequest(HttpServletRequest servletRequest)
  {
     return new UploadWebRequest(servletRequest);
  }

...
```

FileDropoff Test Drive

Now that all of your tests have passed, it's time to fire up your Wicket application and try it out in a browser. Since you used a test-driven methodology, you can expect things to work immediately because automated functional tests are already exercising the code. In a real development situation, you should write tests to verify error condition behavior as well (invalid field values, and so on). Try deploying the application remotely or using very large files to see the progress bar in action—uploading to your local disk may go too fast.

Wicket requires only a servlet container to run, and it encourages the use of the Jetty container, whose lightweight footprint is further reduced by the exclusion of a JSP engine. The FileDropoff example was created using the Wicket Maven archetype. You can launch it by simply running mvn jetty:run. Alternatively, the archetype also creates a Start class, which embeds Jetty and can be conveniently debugged as a normal Java application.

Wicket recently became an Apache Software Foundation project, and it is approaching a major release. It represents a good option for lightweight, component-oriented Web application development using pure Java and object-oriented design principles and patterns. ■

How Can Designers Work on Pages Composed Using Separate HTML Files?

If you use third-party components that include markup on your Wicket pages, some of the HTML originates from inside JAR files. Also, if you use custom components that encapsulate their own markup, you remove that HTML from the pages your designers are expected to customize. Thankfully, Wicket has a solution in the form of the <wicket:remove> tag.

Wicket removes anything inside a <wicket:remove> tag during markup processing, which allows you to place markup on pages that will be visible only during design work, when the files are viewed and edited raw. The following is the markup from SignInPanel copied into a <wicket:remote> block under the place where the actual component will be injected by Wicket during processing:

```
<div wicket:id="signInPanel"/>
<wicket:remove>
<span id="feedback" style="display: none;">
<ul>
<li class="feedbackPanelERROR">
```

continued

How Can Designers Work on Pages Composed Using Separate HTML Files?

```
<span class="feedbackPanelERROR">
field 'password' is required.
</span>
</li>
</ul>
</span>
<form id="signInForm2" onsubmit="return mockSignIn()">
<div style="display:none">
<input type="hidden" name="signInForm2_hf_0" id="signInForm2_hf_0" />
</div>
<table>
<tr>
<td align="right">Username:</td>
<td>
<input name="username" value="" type="text" size="30"/>
</td>
</tr>
<tr>
<td align="right">Password:</td>
<td>
<input name="password" value="" type="password" size="30"/>
</td>
</tr>
<tr>
<td></td>
<td>
<input name="rememberMeRow:rememberMe" type="checkbox"
checked="checked"/> Remember Me </td>
</tr>
<tr>
<td></td>
<td>
<input type="submit" name="submit" value="Sign In"/>
<input type="reset" value="Reset"/>
</td>
</tr>
</table>
</form>
</wicket:remove>
```

This markup allows a designer to see the entire LoginPage as it will appear to users. The contents of the <wicket:remove> block is indeed removed before the page goes to the user so they can't customize the markup inside. However, they shouldn't need to, as CSS is the preferred means to change its look and feel.

Also, you can mock up behavior using inline JavaScript to show the designer what feedback messages will look like, and you can even simulate activity for the progress bar and customize its design outside of the application. Once the designer is finished, you simply drop the markup back into the application. No further modification is required. This is what separation of concerns is all about. ■

Struts to Stripes — A Road Worth Traveling

By Rick Smith

Porting an existing Java Web application to a new framework is probably not at the top of most developers' fun-stuff-to-do lists. In addition to the time of learning a new Web framework, the tedious process of converting things like tags, internationalization systems, and validation can force even the most courageous among us to think twice. I recently faced such a challenge when considering a move from Struts.

Jupiterimages

The first question in any decision about porting an application should be "Why not stay with the framework I have?" In my case, Struts was a stable, well-documented framework with a large developer community, but configuration was cumbersome and the separation of forms, actions, application flow, and validation sometimes made following a thread through an application like trying to untangle a knotted fishing line. This tendency only got worse as my Struts applications grew. Eventually, from a maintenance perspective alone, migrating to a new framework made sense.

None of the frameworks I first considered (Java ServerFaces, Tapestry, WebWorks, Spring MVC) convinced me that their potential benefit outweighed the cost of porting from Struts. Some, like JSF, were not view friendly. Others, like Tapestry and WebWorks, had per-page internationalization systems that looked cumbersome. And Spring MVC didn't look much better than Struts from a configuration perspective. The framework I selected needed to justify the time spent learning it and the effort of actually porting the code; it had to take me to a better place—a place where code was easier to write, troubleshoot, and maintain. From that perspective, these alternatives looked more like tradeoffs than saviors.

Stripes to the Rescue!

Then I happened upon the Stripes Framework. Like many in the Java community, I had been following the Ruby on Rails (RoR) phenomenon. For me, Stripes was the closest of the Java MVC frameworks to the RoR phi-

"In addition to the time of learning a newWeb framework, the tedious process of converting things like tags, internationalization systems, and validation can force even themost courageous among us to think twice."

losophy: simple, elegant, and requiring minimal configuration. In addition to its simplicity, Stripes looked familiar to a Struts veteran like me. The application flow and many of the naming conventions were similar. Stripes' ActionBeans were like Strut's Actions, and ForwardResolutions looked a lot like ActionForwards. With this framework, I would not have to throw away all of my hard-earned Struts knowledge.

Something else that appealed to me was the Stripes documentation. Like the framework itself, it was clear, clean, and concise. The tag library docs and API were well documented, and just about every feature of the framework had sample code. This excellent documentation, along with the ability for me to capitalize on my existing Struts knowledge made me confident I could quickly get up to speed with the Stripes framework.

Stripes also contained features that made it a good AJAX platform, including a streaming resolution that allows for improved error handling in AJAX implementations. For me, however, the deciding factor ultimately was that I could clearly see it making my life easier. I estimated that I could reduce the total lines of code in the action/configuration/validation areas of my application by about half. Less code meant fewer bugs, faster development time, and easier troubleshooting.

The Porting Process

I started my port from the view layer and worked my way back to actions. I had no real logic to this approach; I had to start somewhere and views were as good a starting point as any.

JavaServer Pages
Stripes, like Struts, uses JSPs for its view layer. I was pleasantly surprised to find that the Stripes tag library is similar to Struts' HTML taglib. In fact, I was able to upgrade many of my tags using the universal replace.

Stripes relies on JSTL for logic in the JSP view. I was using a mix of Struts logic tags and JSTL in my application, so I was ahead of the game. By porting all of my logic tags to JSTL, I was also able to take advantage of JSTL's superior handling of if/else and case statements, which are either rudimentary or non-existent in the Struts logic taglib.

Internationalization
The next surprise came when I was porting my Struts' message resources. On the configuration side, all this operation required was renaming my Struts message resource files. Inside my JSPs, I was able to replace 100 percent of my Struts message tags (e.g., `<bean:message key="buttons.save"/>`) with JSTL format tags (e.g., `<fmt:message key="buttons.save"/>`) using universal replace. The JSTL format tag also supports message resource bundling available in Struts.

Forms
The most rewarding part of my port was getting rid of my Struts Action Forms, which can require extensive XML markup and tedious conversions in the Action class as the following sample shows:

```
<form-bean name="employeeUpdateForm"
type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="employeeid" type="java.lang.Long" />
      <form-property name="firstname" type="java.lang.String" />
      <form-property name="lastname" type="java.lang.String" />
      <form-property name="phone" type="java.lang.String" />
      <form-property name="email" type="java.lang.String" />
      <form-property name="phone" type="java.lang.String" />
      <form-property name="socialsecurity" type="java.lang.String" />
      <form-property name="birthdate" type="java.lang.String" />
      <form-property name="salary " type="java.lang.String" />
</form-bean>

public ActionForward executeAction(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) throws Exception {

        Employee employee=new Employee();
```

```
        DynaValidatorForm eaf = (DynaValidatorForm) form;
        employee.setFirstname(eaf.getString("firstname"));
        employee.setLastname(eaf.getString("lastname"));
        employee.setPhone (eaf.getString("phone"));
        employee.setEmail (eaf.getString("email"));
        employee.setEmployeeid ((Integer)eaf.get("employeeid"));

employee.setSocialsecurity(Long.parseLong(eaf.getString("socialsecurity")));

employee.setBirthdate(MyUtils.convertStringToDate(eaf.getString("birthdate")));

employee.setSalary(MyUtils.convertStringToBigDecimal(eaf.getString("salary")));
        EmployeeDAOService.updateEmployee(employee);
        return new ActionForward(mapping.getForward());
    }
```

Stripes form handling, on the other hand, allows you to use your domain object as a form:

```
public class UpdateEmployeeActionBean implements ActionBean {
      private ActionBeanContext context;

      private Employee employee;

      public ActionBeanContext getContext() {
            return context;
      }

      public void setContext(ActionBeanContext context) {
            this.context = context;
      }

      public void setEmployee(Employee employee) {
            this.employee = employee;
      }

      public Employee getEmployee() {
            return this.employee;
      }

      @DefaultHandler
      public Resolution update() {
            EmployeeDAOService.updateEmployee(employee);
            return new ForwardResolution("/employees/updateEmployee.jsp");
      }
```

In most cases, I was able to embed a copy of my domain object as a property of my Stripes ActionBean class and include only the code involved in moving that object to and from my persistence layer. I threw away all of the form handling in Struts Actions, including initial configuration, casting the form to the appropriate class, and copying and converting that data to and from the domain object—about 30 percent of the code in most action classes I've seen. It was not needed in Stripes.

In short, embed the domain object as a property of your ActionBean class, provide it getter and setter methods, and voilà! The whole enchilada—including

lists—is exposed to the form in the HTML view. What I could do with forms I could also do with query-string parameters. I simply made those parameters a property of my ActionBean and they were automatically copied into those fields if they were part of the request.

Validation

Porting Struts validation to Stripes required more work than forms or tags. For my application, I had to rewrite validation configuration in the validation.xml file with Java 5.0 annotations inside the Stripes ActionBean class. Stripes also gives you a nice type-based validation for free. With no user configuration whatsoever, Stripes will return HTML forms to the user when they enter the wrong value type (e.g., a character in a number or date field). Forms can be returned automatically with a user-friendly message and the offending field highlighted.

Application Flow

Converting the control flow of my Struts application was probably the one place that required a break from the Struts way of thinking. In Struts, control flow—the binding of URL requests, actions, and the resulting view—is rendered in XML markup and centralized in the struts-config.xml file. This rendering outside the action layer makes Struts bindings flexible. They are not hard-coded inside the action layer and a single action can easily be coupled with different input URLs and forwards. The downside of this approach is that Struts configurations can quickly grow large and cumbersome. The separation of control flow from the action layer can also make debugging all the way through the request cycle difficult.

Stripes offers three different ways to map requests to the action layer:

1. Explicitly bind an ActionBean to a URL using annotations
2. Allow Stripes during startup to guess the binding of its ActionBeans based on the similarity between the ActionBean class path and the application URLs
3. Bind a JSP to any ActionBean, or call any method of a Java class in the application class path, using the Stripes useBean tag

While the first two approaches seem somewhat hard-coded compared with Struts configuration, the useBean tag provides a lot of flexibility. With it, JSPs can access multiple ActionBeans or classes to get just what they need.

Easy Journey to the Right Destination

When choosing a new framework, the ease of the migration—both in learning the new framework and porting your existing code—are factors to consider, but they should not be given too much weight. Yes, you have made a large investment in learning an existing framework and it would be nice to retain some of that investment in your next MVC platform. And yes, it would be nice if you could port your application in weeks, not months. But no matter how easy or pleasant the journey, you should decide first if the destination is a place you want to go. For me, the ability to nearly half the amount of code in my action layer and centralize forms, configuration, and validation in one place were the major factors in my decision. The quality of Stripes' documentation and the other stuff was just icing on the cake. ■

Rapid Java Web Application Development with Tapestry

By John Ferguson Smart

Download the source code for this article at: http://assets.devx.com/sourcecode/14751.zip

The J2EE world is full of development frameworks, all designed to simplify tedious low-level programming tasks and allow the programmer to get on with more interesting business-related stuff. The most well known probably is Struts, the Model-View-Controller (MVC) framework based largely on Struts Action. More recent frameworks are shifting away from the Struts approach in favor of a higher-level, more object-oriented approach that is component-based and event-driven. Among the most interesting of this new generation of frameworks are JavaServer Faces (JSF), which is backed by industry giants such as Sun, and a dynamic and innovative outsider from the Apache Jakarta project called Tapestry.

Jupiterimages

Introducing Tapestry

Tapestry is an open-source framework for object-oriented, component-based Java Web application development. Simply put, instead of dealing with the Servlet API or with Struts Actions, the Tapestry programmer stores user data with object properties and handles user actions with event-handling methods.

Another major feature of Tapestry is its use of HTML page templates. In Tapestry, each page is an HTML template containing browser-friendly HTML tags. Unlike JSP, JSTL, or JSF pages, creating Tapestry pages is relatively easy using common Web design tools, and you can preview them in a Web browser.

This article demonstrates a few of the main features of Tapestry, and shows how Tapestry 4, released in December 2005, makes things even easier than previous versions.

Setting Up Tapestry

Tapestry is built on the standard Servlet API, which means it will run on any Java servlet container or application server. You just need to set up the Tapestry servlet in your *Web.xml file*, as illustrated here:

> Instead of dealing with the Servlet API or with Struts Actions, the Tapestry programmer stores user data with object properties and handles user actions with event-handling methods.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Web-app
       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
       "http://java.sun.com/dtd/Web-app_2_3.dtd">

<Web-app>
     <display-name> Introduction to Tapestry Tutorial</display-name>
     <servlet>
          <servlet-name> app</servlet-name>
          <servlet-class>org.apache.tapestry.ApplicationServlet</servlet-class>
          <load-on-startup>0</load-on-startup>
     </servlet>
     <servlet-mapping>
          <servlet-name>app</servlet-name>
          <url-pattern>/app</url-pattern>
     </servlet-mapping>
</Web-app>
```

You can set the servlet name to anything you like. However, modifying the url-pattern is a bit trickier, so you should leave this as is ("/app").

The Example Application

The best way to get a feel of Tapestry is to work through some examples. In this tutorial, you will work on a Web site designed to sell discount plane tickets for various destinations. The application uses a simple business model (see Figure 1):

- Promotional offers are represented by the `DestinationOffer` class, which contains the city of destination and the price.
- The `FeatureDestinations` interface provides business methods concerning a set of daily promotional offers.
- The `FeatureDestinationsImpl` class is a memory-based implementation of this interface.

Your First Tapestry Page

Let's start off with a simple page. Every Tapestry application has a home page called (appropriately enough) Home. Your home page will display a randomly chosen feature destination. It also will provide a link back to the same page in order to display another offer. The Home page requires two main things:

- A page template, containing a mixture of HTML code and dynamic Tapestry components (more about components shortly)
- A corresponding Java class, which provides data for the dynamic parts of the page

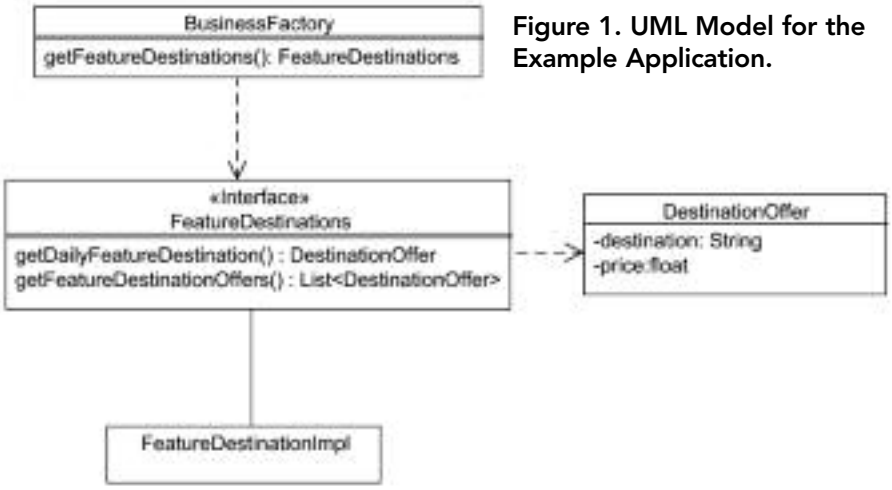


Figure 1. UML Model for the Example Application.

In Tapestry 3, each page also needed a page specification file. This file is an XML file describing the mapping

between the page template and the Java class. Although page specifications can be useful, and even necessary in more complex pages, Tapestry 4 uses Java 5 annotations and extra tag attributes to greatly reduce the need to write one for each and every page. You won't need to worry about them for the examples in this article.

The Page Template

Let's start by looking at the page template:

```
<html>
   <head>
      <title>Tutorial: Introduction to Tapestry</title>
   </head>

   <body>
         <h3>Online Travel Discounts</h3>
         <h4>One of today's feature destinations:</h4>
         <p>
         A trip to
         <span jwcid="@Insert"
value="ognl:featureDestination.destination">Paris</span>
         for only $<span jwcid="@Insert"
```

Where Does Tapestry Look for Its Classes?

Tapestry 4 makes life easier by allowing you to telling it where to look for class pages. You do this by configuring a special *application specification file*, which is an (optional) XML configuration file that contains application-wide parameters. Tapestry expects this file to have the name of the application (with the extension ".application"), and will look for it in the WEB-INF directory. In the example application, the *application specification* file (called *app.application*) is as follows:

```
<!DOCTYPE application PUBLIC
"-//Apache Software Foundation//Tapestry Specification 4.0//EN"
"http://jakarta.apache.org/tapestry/dtd/Tapestry_4_0.dtd">
<application>
<meta key="org.apache.tapestry.page-class-packages"
value="com.wakaleo.tutorials.tapestry.pages"/>
</application>
```

A Tapestry page class typically provides the following:

- Getters and setters for the objects used in the page
- Methods that are called at different points in the page lifecycle
- Methods that will be invoked by user actions on the page

The following is the class for the example Home page, which represents only the first two types of methods:

```
public abstract class Home extends BasePage implements PageBeginRenderListener
{
     public abstract DestinationOffer getFeatureDestination();
     public abstract void setFeatureDestination(DestinationOffer
featureDestination);
```

continued

```
value="ognl:featureDestination.price">199</span>
        </p>
 </body>
</html>
```

The first thing you may notice is that the template is very close to normal HTML. In fact, you can safely display it in a Web browser (see Figure 2). Because it uses Normal HTML tags instead of JSP, JSTL, or JSF tags, a non-Java-savvy Webmaster can easily build and maintain the page templates using ordinary Web design tools. JSF pages, on the other hand, use tags that are quite alien to a traditional HTML Web designer, and that cannot be previewed in a Web browser. Java developers also generally find it more convenient to be able to preview pages without having to deploy to an application server.

Figure 2. Previewing a Tapestry Page.

Tapestry Components

The next things you may notice are the rather strange-looking tag attributes ("jwcid" and the like). These attributes

Where Does Tapestry Look for Its Classes?

```
    public void pageBeginRender(PageEvent event) {
       setFeatureDestination(BusinessFactory
                                          .getFeatureDestinations()
                                          .getDailyFeatureDestination());
    }
}
```

One of the quirks of Tapestry is that page properties are often abstract, rather than being classic JavaBean properties. Another important thing to know is that, for performance reasons, Tapestry pages are pooled and can be shared between users. So you must re-initialize any user-specific data before reusing the page. By declaring abstract getters and setters for a property, you let Tapestry handle all the nitty-gritty details of property cleanup and initialization.

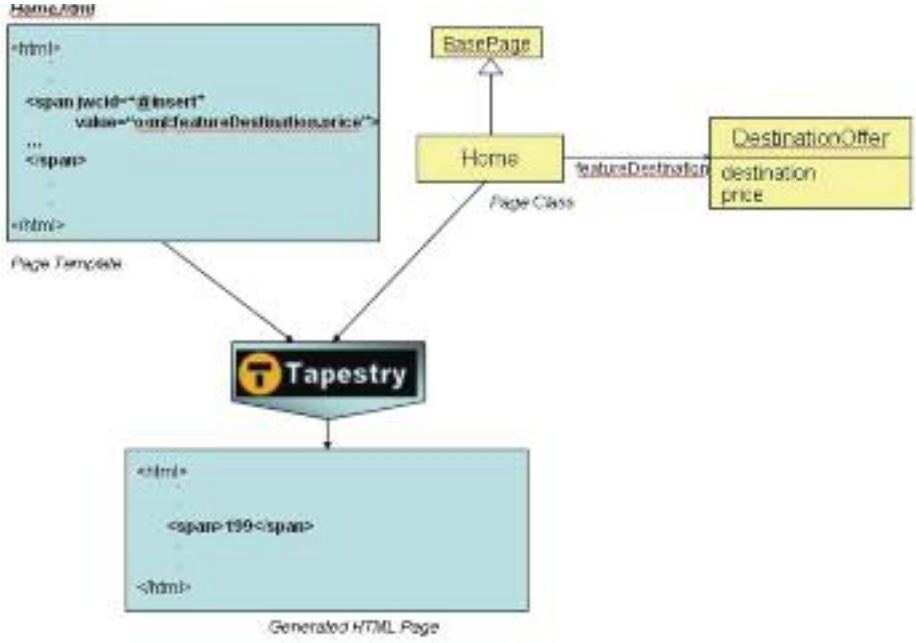
Tapestry also provides a number of interfaces, which you can use to implement methods that are called at certain points in the page lifecycle. One of the most useful is the `PageBeginRenderListener` interface, which along with the `pageBeginRender()` method, initialize page properties before the page is generated. Tapestry calls the pageBeginRender() method just before the page is displayed. The example initializes the featureDestination page attribute with a value retrieved from the appropriate business class.

Working with Links and Listeners

One of the major innovations of Tapestry (and other more recent frameworks such as JSF) is that HTML links, submit buttons, and the like are mapped directly to arbitrary Java class methods. In Tapestry, navigation logic and data submission are placed in these "listener" methods or, for simpler cases, directly configured in the Tapestry component itself.

continued

Figure 3. Generating a Tapestry Page.



identify Tapestry components, which are responsible for generating (or "rendering") the dynamic parts of the page (see Figure 3).

Tapestry components are embedded in ordinary HTML tags, as in the following example:

```
<span jwcid="@Insert" value="ognl:featureDestination.price">199</span>
```

The jwcid attribute stands for Java Web Component ID. It identifies the dynamic component you are interested in. The component here is an Insert component, which inserts data retrieved from a property of the corresponding

Where Does Tapestry Look for Its Classes?

For example, suppose you want to add a button that redisplays the Home screen with a new featured destination. Defining links to other pages is an important part of any Web application, and Tapestry provides several convenient ways of doing so. The easiest is to simply create a link to another Tapestry page, using the PageLink component, as shown here:

```
<a href="#" jwcid="@PageLink" page="Home">Show another feature destination!</a>
```

This will generate a correct href reference to the application Home page. One of the nice things about links in Tapestry is that you almost never have to worry about the exact form of the URL. In this case, you don't even need to add a method to the page class.

Now, suppose you want to add a "Show Details" button to your Home page. You would need to add a link to the details page and provide a parameter that indicates which offer to display. Rather than forcing you to painstakingly building the URL yourself, Tapestry uses a more object-oriented approach: invoking listener methods. To use this function, you add a new method (say, `showDetails()`) to the `Home` Java class with the following signature:

continued

Java class. The value attribute indicates where the dynamic data is to be found. The ognl prefix stands for Object Graph Navigation Language, a powerful expression language that is useful for identifying properties within Java objects. In this particular case, you are fetching the price attribute nested in the featureDestination attribute of the Java class associated with this page. The contents of the tag (199) are just there for previewing purposes; it will be replaced at runtime by the dynamically generated text.

There are 50 or so built-in Tapestry components, which generally cover most Web developer needs, including loops, conditions, and date input fields. If you really need to, you can also write your own quite easily.

The Page Class

Most Tapestry pages with any significant dynamic content have a corresponding Java class, which works with the page template to fill in the dynamic parts of the page. The class should have the same name as the page, and, for convenience, extend the `BasePage` class.

A Tapestry page class typically provides the following:

- Getters and setters for the objects used in the page
- Methods that are called at different points in the page lifecycle
- Methods that will be invoked by user actions on the page

Where Does Tapestry Look for Its Classes?

```
public IPage showDetails(DestinationOffer destination)
```

Then you call this method using the DirectLink component and indicate where the method parameters should be with the parameters attribute:

```
<a href="#" jwcid="@DirectLink"
             listener="listener:showDetails"
             parameters="ognl:{  featureDestination }">Show Details</a>
```

Tapestry 4 is very flexible about listener methods. A listener method can be just an ordinary Java method, with or without parameters or a return type.

In this case, you want the method to go to the ShowDetails page. To do this, the method must return an instance of the `IPage` interface, which each Tapestry page implements. Using Java 5, you can use Tapestry 4 annotations to inject an appropriate getter method to reference the target page, as follows:

```
@InjectPage("ShowDetails")
public abstract ShowDetails getShowDetailsPage();

public IPage showDetails(DestinationOffer destination) {
  ShowDetails showDetailsPage = getShowDetailsPage();
  showDetailsPage.setOffer(destination);
  return showDetailsPage;
}
```

continued

The following is the class for the example Home page, which represents only the first two types of methods:

```
public abstract class Home extends BasePage implements PageBeginRenderListener
{
    public abstract DestinationOffer getFeatureDestination();
    public abstract void setFeatureDestination(DestinationOffer
featureDestination);

    public void pageBeginRender(PageEvent event) {
       setFeatureDestination(BusinessFactory
                                          .getFeatureDestinations()
                                          .getDailyFeatureDestination());
    }
}
```

One of the quirks of Tapestry is that page properties are often abstract, rather than being classic JavaBean properties. Another important thing to know is that, for performance reasons, Tapestry pages are pooled and can be shared between users. So you must re-initialize any user-specific data before reusing the page. By declaring abstract getters and setters for a property, you let Tapestry handle all the nitty-gritty details of property cleanup and initialization.

Where Does Tapestry Look for Its Classes?

In a nutshell, this method will initialize the ShowDetails Page with the selected offer and then transfer control to this page.

Working with Lists

Tapestry provides an impressive number of useful components out of the box. To illustrate these standard components, suppose you now want a page that lists all your feature destinations. Each feature destination should have a link to the Details page. To do this, you would use the Tapestry For component, as illustrated here:

```
<table cellspacing="6">
  <tr>
   <td>Destination</td>
   <td>Price</td>
   <td></td>
  </tr>
  <tr>
   <td colspan="3"><hr/></td>
  </tr>
  <tr jwcid="@For" source="ognl:featureDestinationOffers" value="ognl:offer"
element="tr">
    <td><span jwcid="@Insert" value="ognl:offer.destination"/></td>
    <td>$<span jwcid="@Insert" value="ognl:offer.price"/></td>
      <td>
        <a href="#" jwcid="@DirectLink"
               listener="listener:showDetails"
               parameters="ognl:{  offer }">Show Details</a>
```

continued

Tapestry also provides a number of interfaces, which you can use to implement methods that are called at certain points in the page lifecycle. One of the most useful is the `PageBeginRenderListener` interface, which along with the `pageBeginRender()` method, initialize page properties before the page is generated. Tapestry calls the `pageBeginRender()` method just before the page is displayed. The example initializes the `featureDestination` page attribute with a value retrieved from the appropriate business class.

Where Does Tapestry Look for Its Classes?

```
   </td>
  </tr>
  <tr jwcid="$remove$">
    <td>Paris</td>
    <td>199</td>
    <td><a href="#">Show Details</a></td>
  </tr>
  <tr jwcid="$remove$">
    <td>Rome</td>
    <td>299</td>
  </tr>
</table>
```

The `source` attribute specifies the collection or list of objects to be iterated. The value attribute specifies a placeholder variable used to store the current object value for each iteration.

Astute readers will notice two lines at the end of the table containing dummy data and a strange looking $remove$ component. The remove component allows a Web designer to add HTML tags that can be used for previewing, but which will be discarded during page rendering. This is very useful for correctly formatting and previewing large tables.

The corresponding Java class has to provide getters and setters for the attributes used in the `For` component. You will also need a `showDetails()` method for the DirectLink component, similar to the one you used in the Home page. The following is the full class:

```
public abstract class FeatureDestinationList extends BasePage implements
PageBeginRenderListener

{
    public abstract List<DestinationOffer> getFeatureDestinationOffers();
    public abstract void setFeatureDestinationOffers(List<DestinationOffer>
offers);

    public abstract DestinationOffer getOffer();
    public abstract void setOffer(DestinationOffer offer);

    public void pageBeginRender(PageEvent event) {
      setFeatureDestinationOffers(BusinessFactory
                                  .getFeatureDestinations()
                                  .getFeatureDestinationOffers());
    }
```

continued

Where Does Tapestry Look for Its Classes?

```
    @InjectPage("ShowDetails")
    public abstract ShowDetails getShowDetailsPage();

    public IPage showDetails(DestinationOffer destination) {
      ShowDetails showDetailsPage = getShowDetailsPage();
      showDetailsPage.setOffer(destination);
      return showDetailsPage;
    }
}
```

Figure 4 shows the final page.

Figure 4. A List Page.

Localization

Localization in Tapestry is a remarkably simple task. Each page or component needing to be localized has its own properties file in the WEB-INF directory of your Web application and contains the messages to be displayed. To use them, you can use a special attribute of the span tag as follows:

```
<span key="title">Online Travel Discounts</span>
```

The contents of the tag are for previewing purposes only; this text is discarded at runtime and replaced by the localized text.

If you need to use a formatted message, you instead use the Insert component and the special messages object accessible in every page:

```
  <span jwcid="@Insert"
        value="ognl:messages.format('feature-destination',

featureDestination.destination,

featureDestination.price)">
    A trip to Paris for only $199
  </span>
```

The first parameter is the name of the message key; the following parameters are message-formatting parameters.

The final page template, complete with localization, is presented here:

```
<html>
   <head>
     <title><span key="page-title">Tutorial: Introduction to
Tapestry</span></title>
   </head>
<body>
```

continued

Where Does Tapestry Look for Its Classes?

```
    <h3><span key="title">Online Travel Discounts</span></h3>
    <h4>
       <span key="todays-feature-destination">One of today's feature destinations:
       </span>
    </h4>
    <p>
       <span jwcid="@Insert"
                 value="ognl:messages.format('feature-destination',
                                        featureDestination.destination,
                                        featureDestination.price)">
           A trip to Paris for only $199
       </span>
       <a href="#" jwcid="@DirectLink"
                listener="listener:showDetails"
                parameters="ognl:{  featureDestination }">
           <span key="show-details">Show Details</span>
       </a>
    </p>

    <p>
       <a href="#" jwcid="@PageLink" page="Home">
           <span key="show-another">Show another feature destination!</span>
       </a>
     </p>
    <p>
      <a href="#" jwcid="@PageLink" page="FeatureDestinationList">
          <span key="show-all">Show all your feature destinations!</span>
      </a>
    </p>
</body>
</html>
```

As you can see, it is still quite readable and can still be previewed in a standard Web browser. You can also localize your page templates, which may be useful for pages with a lot of static text that needs to be localized. For example, Home_fr.html would be the French translation of the Home.html template (see Figure 5).

Figure 5. Template Localization.

Many More Pros Than Cons

Tapestry is a clean, efficient, and easy-to-use (dare I say "fun"?) framework. In the tradition of Hibernate and Spring, and as opposed to EJB and JSF, it is based on real user needs and experience rather than being designed to a committee-specified standard. Its programming model, although not devoid of a few quirks and idiosyncrasies, is globally clean and elegant. Its

continued

Where Does Tapestry Look for Its Classes?

architecture (especially where session management and component pooling are concerned) is efficient and highly scaleable. It provides a large number of very useable standard components out of the box, and custom components are relatively easy to develop if necessary. And important Web-development requirements such as localization are also well integrated.

On the negative side, it is less well known than Struts or JSF, and not backed by any industry giants. Documentation is not as abundant as it is for Struts or JSF. And although the Tapestry community is active and growing fast, experienced Tapestry developers are probably relatively hard to find.

Nevertheless, Tapestry is a fine piece of architecture and is well worth consideration for your next Web development project. ■

Face Up to Web Application Design Using JSF and MyFaces

By Javid Jamae

Download the source code for this article at: http://assets.devx.com/sourcecode/13183.zip

If you've worked on more than one Web application with different teams, you've probably worked with more than one Web application framework. J2EE always provided Web technologies, but never defined a Web application framework for managing page navigation, request lifecycle, request validation, etc. Developers had to develop these features themselves or utilize one of many open-source frameworks such as Struts and Spring MVC. Enter JavaServer Faces.

JSF is specified in JSR 127 (see Related Resources section in the left column) and "defines an architecture and APIs that simplify the creation and maintenance of Java Server application GUIs." This article will discuss the basic ideas behind JavaServer Faces and then show you an example using the Apache MyFaces JSF implementation.

What is JavaServer Faces?

JavaServer Faces (JSF) is a Web application framework that can be used to link and manage user interface events. A JSF application is just like any other Java Web application in that it runs in a servlet container (such as Tomcat, WebLogic, WebSphere, etc.).

JSF is different than most Java Web-application frameworks, such as Struts, which treat each page request as a single event. JSF pages are comprised of multiple components that can each trigger individual events. Each component represents one or more Web page elements capable of generating dynamic output or providing user input. In JSF parlance, an individual component such as a text box or a button is said to be a simple component. A compound component is one that is comprised of multiple elements such as a table. The JSF component model is similar to the component model that traditional non-Web MVC frameworks such as Swing use. One benefit of this component model is that it should foster the

"JSF pages are comprised of multiple components that can each trigger individual events. Each component represents one or more Web page elements capable of generating dynamic output or providing user input."

creation of development tools that allow user interface designers to drag and drop components into a layout. Then, after creating a visual layout of the components, the developer(s) can write event-handling code that will allow the view components to interact with the application model. This could allow for faster GUI creation for prototyping and while doing rapid application development (RAD).

Other Web frameworks are usually tightly coupled to JSP as the view-rendering technology, but JSF is not bound to JSP. JSF calls for a strict decoupling of components from their view rendering. Rendering is done by using a rendering kit. JSP is a required rendering kit specified in the JSF specification and can be used to render JSF components into HTML, but developers can also use custom rendering kits to render views.

At the time of writing this article, there were a few JSF implementations available for use: a reference implementation available from Sun, Apache MyFaces, Smile, and ECruiser. Thus far, it looks like Apache MyFaces is the most widely used (or at least most widely referenced) implementation, so I'll use it for my example.

HiLo Example

I'm going to show you an example JSF application

Listing 1. Web.xml

```
<?xml version="1.0" ?>
<!DOCTYPE Web-app (View Source for full doctype...)>
<Web-app>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml</param-value>
  </context-param>
  <listener>
    <listener-
class>net.sourceforge.myfaces.Webapp.StartupServletContextListener</listener-
class>
  </listener>
  <!--  Faces Servlet -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.Webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!--  Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.jsf</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</Web-app>
```

using the MyFaces JSF implementation and JSP as the rendering technology. I will create an application that will play HiLo with you. HiLo is a game where the computer picks a random number and you must guess what the number is. Each time you guess, it will tell you if you've guessed too high or too low, until you guess the exact number.

The application will have a greeting screen that will ask you your name and the highest possible number that the computer can pick. After submitting this page, you will go into the main game page where you can start guessing numbers. If you guess a number that is out of range, it will give you an error. The game will keep a count of how many guesses you've made.

In order to run the application, you'll need to have a Servlet engine. I wrote and tested the example using Tomcat 5.0.28 and the MyFaces 1.0.7. Unfortunately, I could not get MyFaces to work with Tomcat 5.5, though I was able to get a few JSF examples to work on Tomcat 5.5 using Sun's reference implementation. *Note: If you want to try the Sun reference implementation, you can download it at http://java.sun.com/j2ee/javaserverfaces/download.html.*

The code for the application is packaged as a war file with source code included, so you can deploy the application immediately to see it in action, and you have the source available so that you can look through it. The war file includes all the jar files from the MyFaces distribution.

Looking at the HiLo Code

Take a look at the code in Listing 1. My Web deployment descriptor, Web.xml, tells my container about the JSF controller Servlet.

Listing 2. Faces-config.xml

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/Web-facesconfig_1_1.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>/pages/greeting.jsp</from-view-id>
    <navigation-case>
      <from-outcome>play</from-outcome>
      <to-view-id>/pages/game.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/pages/game.jsp</from-view-id>
    <navigation-case>
      <from-outcome>high</from-outcome>
      <to-view-id>/pages/game.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>low</from-outcome>
      <to-view-id>/pages/game.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>correct</from-outcome>
      <to-view-id>/pages/win.jsp</to-view-id>
    </navigation-case>
```

continued

The servlet-mapping and servlet blocks specify that any URL that ends in a .jsf extension should be redirected through a servlet called javax.faces.Webapp.FacesServlet. This is a controller servlet that manages the request processing lifecycle for the Web application. All JSF requests will route through this controller servlet.

The servlet architecture allows for custom classes to be registered as event listeners. A context listener is one such listener that is triggered when application-context lifecycle events occur. As you can see, there is a context listener that is registered in the deployment descriptor. This listener is required in order to initialize the JSF application framework.

There are two context-param blocks. The javax.faces.CONFIG_FILES parameter defines where the JSF configuration file exists. The faces-config.xml file contains all of the configuration information for a JSF application. This is where you define how one page will navigate to the next as well as where you specify managed beans, which are model objects that JSF uses to get and set data from the UI. The second parameter specifies whether state should be maintained on the client side or on the server side.

The application has four JSPs:

1. index.jsp—This JSP exists in the root of the Web application and merely routes the user to the greeting page
2. greeting.jsp—This page greets the user, asks their name, and asks them for the upper limit with which they want to play HiLo
3. game.jsp—The main game page asks the user for a guess and tells the user if they've guessed high or low
4. win.jsp—This is the page that the user is directed to if they win the game

The way that these JSPs link together is defined in the faces-config.xml file, which is shown in Listing 2.

The root-element is faces-config. There are three navigation-rule blocks and a managed-bean block. The navigation rules specify a from-view-id and one or more navigation-case blocks. The from-view-id is the page that you are currently on. Each navigation-case tells the framework where it should forward the user depending on the from-outcome that results from the action taken. For example, to get from the greeting page to the game page, a **play** action must occur.

The managed bean (also known as a form-backing bean) is the model of the Model View Controller pattern. The views interact with the managed bean, and the framework maintains all the necessary state for the scope defined in this definition. The managed-bean-name is the name that the views will use to reference the bean. The managed-bean-class is the fully-qualified class name for the bean. Have a look at the managed bean code shown in Listing 3.

Listing 2. Faces-config.xml

```
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/pages/win.jsp</from-view-id>
    <navigation-case>
      <from-outcome>restart</from-outcome>
      <to-view-id>/pages/greeting.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <managed-bean-name>gameBean</managed-bean-name>
    <managed-bean-class>com.devx.jsfexample.ui.HiLoGameBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

The JSF framework will use the managed bean to store state information for the user. The bean has a reference to our actual domain object HiLoGame. This is a good layer separation to try to maintain when developing Web applications, because it helps keep view logic and properties out of your domain classes. This allows your domain classes to remain generic enough to be used by multiple views. It also makes it easier to introduce a service layer (perhaps stateless session EJBs) into your applications, because you don't have to go back and refactor the view code out of your domain.

When the framework sets the maxValue, the game is reset, and when it sets the guess, the HiLoGame domain object is called and the return value is stored in guessOutcome for retrieval by JSP components. The guess outcome will either be *high*, *low*, or *correct*. This outcome will be used both to display to the user, and to determine which navigation path to take based on the outcome of the guess. This will become more clear when I explain the game JSP.

Now let's look at the JSPs. The first JSP I'll look at is index.jsp:

```
<html>
<body>
  <jsp:forward
page="/pages/greeting.jsf" />
</body>
</html>
```

Listing 3. Managed Bean Code

```
package com.devx.jsfexample.ui;

import com.devx.jsfexample.hilo.HiLoGame;

public class HiLoGameBean
{
      private int                guess;
      private HiLoGame  game;
      private String           name;
      private String           guessOutcome     = "noGuessesYet";

      public HiLoGameBean()
      {
            game = new HiLoGame( 10 );
      }

      public String getName()
      {
            return name;
      }

      public void setName( String name )
      {
            this.name = name;
      }

      public int getGuess()
      {
            return guess;
      }

      public void setGuess( int guess )
```

continued

This forwarding JSP will direct control to the controller servlet because the URL it is forwarding ends in .jsf. No navigation rules need to be setup for this forward to work. The controller will determine that it needs to forward the user to the greeting page by looking at the URL. The controller will render the greeting.jsp page, using the data from the managed-bean.

Now let's look at the greeting page (see Listing 4 and Figure 1). The first thing you'll notice is that I've included two tag libraries: html and core. These custom tag libraries are a JSP implementation of a JSF rendering kit.

The core tags are provided to developer to perform core actions that are independent of

Figure 1. Greeting.jsp: The greeting page gets you started with the HiLo game.

Listing 3. Managed Bean Code

```
{
      this.guess = guess;
      guessOutcome = game.guess( guess );
}

public void setMaxValue( int max )
{
      game.setMaxRange( max );
      game.resetGame();
      guess = 0;
}

public int getMaxValue()
{
      return game.getMaxRange();
}

public String getGuessOutcome()
{
      return guessOutcome;
}

public int getNextGuessNumber()
{
      return game.getCount() + 1;
}

public boolean getAlreadyGuessedOnce()
{
      return game.getCount() > 0;
```

a rendering kit. These tags will not display anything to the user. The html tag library is mainly used to render html elements (text, form elements, etc.).

The first tag you see is a core tag called <f:loadBundle>. This tag provides JSF with internationalization support. It loads a properties file and makes its values available in a variable called msg. In a production application, I would have probably made better use of this bundle, but I only reference it once here so that you can see how it works. The bundle is referenced by the first <h:outputText> tag. This HTML tag actually renders text to the screen. The value of the text is a variable that references the bundle that was just created.

JSF uses a templating mechanism to provide variables. Any attribute value that starts with a hash sign and is wrapped in brackets is dynamically substituted in.

The <f:view> core tag tells the JSP that I am going to start using JSF components. All JSF components must be nested inside of an <f:view> tag in a JSP-rendered JSF file.

The <h:form> tag represents a form element. The form action is defined in the <h:commandButton> element. This is the action that is passed to the controller so that it can determine which page it should forward the user to. As you'll see in the game page, the action can also be a variable, thus providing dynamic redirection.

Validation can be achieved in a few ways. The <h:inputText> tag for the name field will render a

Listing 4. Greeting.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<f:loadBundle basename="com.devx.jsfexample.ui.messages" var="msg"/>

<html>
    <head><title>Greeting page</title></head>
    <body>
    <f:view>
        <h2><h:outputText value="#{msg.greeting}"/></h2>
        I'll pick a random number between 1 and the number you
        specify and you try to guess the number I've selected.
        I'll let you know if your guess is high or low.
        <br/><br/>
        <h:form>
            <h:outputLabel for="name">
                <h:outputText value="Name:" />
            </h:outputLabel>
            <h:inputText id="name" value="#{gameBean.name}" required="true" />
            <br/>
            <br/>
            <h:outputLabel for="maxValue">
                <h:outputText value="Max value to play with:" />
            </h:outputLabel>
            <h:inputText id="maxValue" value="#{gameBean.maxValue}"
required="true">
                <f:validateLongRange minimum="10" maximum="1000000" />
            </h:inputText>
            <br/>
```

continued

textbox, and it has a required attribute that tells it that a value must be provided by the user. The maxValue <h:inputText> tag also validates that the field is filled out, but it also has a nested core tag that provides range validation. In this case, the user is required to provide a number between 10 and 1 million. If validation fails, error messages are presented to the user using the <h:message> tag. I have defined two tags, one for maxValue errors and another for name errors. If validation fails for either of these input fields, the user will get an error message.

Game.jsp (see Listing 5 and Figure 2) is similar to greeting.jsp except for a few things. The first thing is the <h:panelGroup> component tag. A panel group is essentially a component container. A panel group is comprised of sub-components and can determine whether or not all of the subcomponents are rendered or not. In this case, an output is rendered that tells the user if their guess is high or low, but only if the user has already made a guess.

The other thing to notice about this JSP is that the command button action is not hard coded. The gameBean provides a method called getGuessOutcome that is called when the action is invoked. Note that an action must reference the actual name of the method, not just using the property name like other HTML components do. This is because an action can call any method on a managed bean, not just JavaBean style methods. The

Figure 2. Game.jsp: The HiLo game is a good example of how variable results can be handled dynamically.

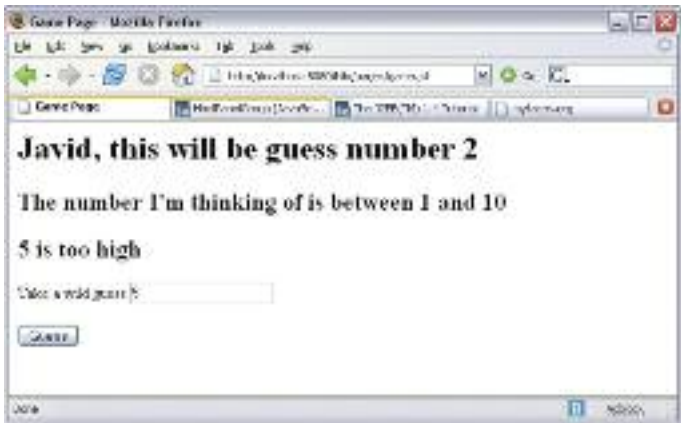


Figure 3. Win.jsp: The final .jsp page in the game is designed to be able to restart the game by sending the user back to index.jsp.

Listing 4. Greeting.jsp

```
            <br/>
            <h:commandButton value="Start Game!" action="play"/>
            <br/>
            <br/>
            <h:message style="color: red; font-family: 'New Century Schoolbook',
serif; font-style: oblique; text-decoration: overline"
                       id="errors1"
                       for="maxValue"/>
            <br/>
            <h:message style="color: red; font-family: 'New Century Schoolbook',
serif; font-style: oblique; text-decoration: overline"
                       id="errors2"
                       for="name"/>
        </h:form>
    </f:view>
    </body>
</html>
```

getGuessOutcome method will return a String that the framework will compare to the navigation-rule in the faces-config.xml file to determine where to forward the user.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<body>
     <h1>YOU WIN!</h1>

     <br/>
     <br/>

     <f:view>
          <h:form>
               <h:commandButton action="restart" value="Play
Again"></h:commandButton>
          </h:form>
     </f:view>
</body>
</html>
```

Listing 5. Game.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
     <head><title>Game Page</title></head>

     <body>
     <f:view>

          <h1><h:outputText value="#{ gameBean.name} "/>, this will be guess number
<h:outputText value="#{ gameBean.nextGuessNumber} "/></h1>
          <h2>The number I'm thinking of is between 1 and <h:outputText
value="#{ gameBean.maxValue} "/></h2>

          <h2>
          <h:panelGroup rendered="#{ gameBean.alreadyGuessedOnce} ">
               <h:outputText value="#{ gameBean.guess} "/><h:outputText value=" is
too "/><h:outputText value="#{ gameBean.guessOutcome} "/>
          </h:panelGroup>
          </h2>

          <h:form>

               <h:outputLabel for="guess">Take a wild guess</h:outputLabel>
               <h:inputText id="guess" value="#{ gameBean.guess} ">
```

continued

This JSP is very simple. It simply has a command button that triggers the restart navigation case, which forwards the user back to the greeting page.

JSF and JSTL

Though it is supported, many people are now advocating that JSF tags should not be mixed with Java Standard Tag Library (JSTL) tags in the same JSP. The reason for this is that JSF provides a slightly different model for separating the view from the view logic. Using standard JSTL tags, a lot of the programmatic flow is actually embedded in the JSP itself. With JSF, the goal is to push most of the programmatic logic into the bean classes rather than in the JSP itself, so your JSPs tend to remain more declarative in nature. One advantage to abiding by this separation of view and view logic is that it makes it simpler to switch rendering kits if the need arises. There are equivalent ways to accomplish most things in JSF that you may have previously done using JSTL.

This said, a pragmatic approach to introducing JSF into an application is to incrementally add new JSPs or change existing JSPs to use JSF custom tags instead of standard custom tags. The DevX article, "Mixing JSTL and JSF in Web Applications," discusses this approach. You can read it at: http://www.devx.com/Java/Article/21020/0.

Is JSP the right choice?

Server page technologies (like JSP) have always been a little difficult to work with because of their in between nature. A server page is a concoction of HTML embedded with Java code. Custom tags help to minimize the amount of code in JSPs, but there is still a fundamental problem in that somebody has to write the JSP. This JSP author must either be a UI designer or a developer that learns how to convert UI mock-ups into JSPs.

In my experience, the creation of JSPs never seems natural to either the developer or the UI designer. Developers don't want to have to deal with problems involved with text and images not lining up right, and UI designers don't want to deal with case-sensitive variables and iteration logic. JSPs also seem to be a recurring point of failure because they are hard to unit test and they are affected by both code changes and look-and-feel changes. These are all indications that JSP

Listing 5. Game.jsp

```
                <f:validateLongRange minimum="1" maximum="#{gameBean.maxValue}"/>
            </h:inputText>

            <br/>
            <br/>

            <h:commandButton value="Guess"
action="#{gameBean.getGuessOutcome}"/>

            <br/>
            <br/>

            <h:message style="color: red; font-family: 'New Century Schoolbook',
serif; font-style: oblique; text-decoration: overline"
                       id="errors1"
                       for="guess"/>

        </h:form>

    </f:view>
    </body>
</html>
```

does not give you a true separation of view from view logic.

People have been using JSP because it is a standard and because there is a lot of support for it. Both are obviously great reasons, but not too many people even question using JSP as a view technology because they are used to it and/or because they have never worked with any other view technologies.

JSP is a required part of the JSF specification because the specification writers wanted to ensure an easy transition into JSF for Web developers who are familiar with existing Java technologies. But, as some people have discovered, the combination of the two technologies has its own set of problems. Hans Bergsten, author of "JavaServer Faces" (O'Reilly) has written an article about some of the technical issues that arise when using the technologies together.

Some highlights of the article are:

- JSPs are processed in order from top to bottom in order, but for JSF to work well components should be created, processed, and rendered separately. Mixing the two causes JSF component rendering to behave erratically.
- It's not safe to assume that a construct that is valid in a JSP page is necessarily valid in a JSF.
- JSP is supported in JSF to allow an easier segue into JSF development, but the complications of mixing the two don't necessarily make JSP the best choice.

As an alternative to using JSPs, Bergsten shows how one can almost entirely separate the view from the underlying view logic by creating a custom rendering kit.

A mockup is a UI prototype that a graphic designer or a Web designer puts together so that people can get an idea of what the Web application will look like. I've heard one too many developer wish that they could just read the mockup and programmatically change the elements inside of it, without having to manually "translate" it into a JSP. Bergsten's template-based rendering kit allows you to do exactly that. The HTML mockup need only be injected with identifiers specified in span tags and id attributes. These identifiers represent JSF components. An XML file is provided that tells the rendering engine which component to replace the identified HTML elements with. The rendering kit will parse the HTML page using a DOM parser, replace identified tags with matching components that are specified in the XML file, and then translate the DOM back into HMTL using XSLT.

Using this model, UI designers are free to modify the layout of the page using their favorite WYSIWYG editor and the changes can be reflected in the system immediately. The only thing that they must worry about is to leave the id attributes and span elements intact. Developers would no longer have to worry about UI layout and can focus on developing the view logic and view components.

I would like to see this kind of rendering kit become more popular, if not become the de facto standard for Web development. I'm sure that it has its limitations as well, but I think it facilitates easier cooperation between UI designers and developers and it provides a better separation of layers.

Is JSF Ready for the Mainstream?

Today, if a client asked me to give them a recommendation on whether or not to use JSF for a mission critical application, my answer would be no. If a client asked me to build a small Web application that would be used for internal operations, I would probably build it using JSF. My point here is that I think JSF has potential to become the framework of choice for Web developers in the next several years, but it is not quite there today. The specification is still being updated and existing implementations are not mature yet. It is also unclear how developers will react to the integration problems that exist with JSP and JSF. It's uncertain whether developers will be willing to shift to a different view technology, whether they will just chose to abandon JSF, or whether JSF and JSP will undergo changes in order to facilitate better integration.

What about Struts?

Craig McClanahan is the original creator of Struts and is also the specification lead for JSR-127. McClanahan talks about how Struts and JSF will work together on the Struts Web page (see Related Resources, left column). It seems that the short-term vision for Struts is for it to easily integrate with JSF by providing a custom-

tag-based rendering kit. The long-term vision is unclear, though I envision future versions of Struts becoming pure JSF implementations, and eventually dropping support for the existing MVC framework. But this is all speculation; so don't bet money on it. ■

Simplify Your Web App Development Using the Spring MVC Framework

By Javid Jamae

This article is going to introduce you to the Spring model-view- controller (MVC) framework by walking you through the creation of a simple stock-trading Web application.

Download the source code for this article at: http://assets.devx.com/sourcecode/11237.zip

MVC is a pattern that helps separate presentation from business logic. In short, in an MVC application controllers handle all Web requests. A "controller" is responsible for interpreting the user's request and interacting with the application's business objects in order to fulfill the request. These business objects are represented as the "model" part of the MVC. Based on the outcome of the request execution, the controller decides which "view" to forward the model to. The view uses the data in the model to create the presentation that is returned to the user.

If you've ever worked on a Web application, chances are that you've worked with either a custom MVC framework or an existing framework, such as Struts. Struts is in fairly widespread use in the Java world, but the Spring MVC framework promises to provide a simpler alternative to Struts.

In this article I will walk you through the development of a Web-based stock-trading application. This application will have three primary workflows each of which will cover a different type of controller available in Spring. After reading this article, you should have enough technical and business knowledge to build a full-fledged stock-trading application and compete head-on with existing discount brokerage firms. … OK, not really, but you will at least have enough knowledge to get you started on your own Web projects using Spring.

You can download the Spring distribution from http://www.springframework.org/download.html. If you want to see the application in action, download the

> In an MVC application controllers handle allWeb requests. A "controller" is responsible for interpreting the user's request and interacting with the application's business objects in order to fulfill the request.

source code for this article, which includes a deployable war file.

Note: The code in this article has only been tested on Tomcat 5.5.

The Platform

There are many different Web application servers and development environments to choose from. I won't describe any single environment here because I want to focus the discussion on the Spring framework and not the underlying tools. However, I developed this sample application using Tomcat 5.5, Java 1.5, Spring 1.1, and Eclipse 3.0 using the Sysdeo Tomcat plugin. I have used Spring with other application servers and with different versions of Java 1.4.x without any big surprises, so you should be safe if you have a slightly different environment.

Getting Started

Before diving into Spring, set up a mechanism for deploying your application code into your Web server and set up an application context if necessary. In Tomcat 5.5, the application context automatically takes on the name of the war file (or expanded war directory) that you deploy. My war file is called "tradingapp.war" and my context is consequently "/tradingapp."

Here is the basic directory structure that you need to create for your Web app:

```
/WEB-INF
          /src              (java classes will go in here)
          /jsp              (application jsps will go in here)
          /lib              (jar files that our app depends on will go in here)
          /classes       (compiled class files will go in here)
```

In order to test out the application server setup, create a file called index.jsp (as follows) and put it in the root of your application directory:

```
/index.jsp
<html>
<head><title>Trading App Test</title></head>
<body>
Trading App Test
</body>
</html>
```

Try to pull up the page by going to http://localhost:8080/tradingapp/index.jsp (see Figure 1). (The port may vary depending on the application server you are using.)

On most application servers, users can access files in the root directory of the context. Hence, you were able to hit the index.jsp file directly. In an MVC architecture, you want the controller to handle all incoming requests. In order to do this, it is a good idea to keep your JSP files in a place where they are not directly accessible to the user. That is why we will put all of our application JSP files in the WEB-INF/jsp directory. We'll come back to this soon, but for now let's find out how to access a Spring controller.

Figure 1. Clear the Deck: Pull up the index.jsp page to make sure that you can retrieve a simple JSP page.

Accessing a Controller

Like in other Web frameworks, Spring uses a servlet called DispatcherServlet to route all incoming requests (this pattern is sometimes called the Front Controller pattern). This servlet should be defined in the Web deployment descriptor as shown here:

```
/WEB-INF/Web.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
 'http://java.sun.com/dtd/Web-app_2_3.dtd'>

<Web-app>
      <servlet>
            <servlet-name>tradingapp</servlet-name>
            <servlet-class>
                  org.springframework.Web.servlet.DispatcherServlet
            </servlet-class>
            <load-on-startup>1</load-on-startup>
      </servlet>

      <servlet-mapping>
            <servlet-name>tradingapp</servlet-name>
            <url-pattern>*.htm</url-pattern>
      </servlet-mapping>

      <welcome-file-list>
            <welcome-file>index.jsp</welcome-file>
      </welcome-file-list>

</Web-app>
```

In the Web.xml file I've defined a servlet mapping that forces any URL that ends in .htm to reroute to the tradingapp Servlet (the DispatcherServlet). This servlet analyzes the request URL and determines which controller to pass control on to by using a URL mapping defined in a Spring XML file. This Spring XML file must exist in the /WEB-INF directory and it must have the same name as the servlet name that you defined in the Web.xml with a "-servlet.xml" appended to it. Thus, we will create a file in the /WEB-INF directory called "tradingapp-servlet.xml."

Here is the Spring XML file that the DispatcherServlet will use:

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

<beans>
      <bean id="portfolioController"
            class="com.devx.tradingapp.Web.portfolio.PortfolioController">
      </bean>

      <!-- you can have more than one handler defined -->
      <bean id="urlMapping"
```

```
          class="org.springframework.Web.servlet.handler.SimpleUrlHandlerMapping">
          <property name="urlMap">
               <map>
                    <entry key="/portfolio.htm">
                         <ref bean="portfolioController" />
                    </entry>
               </map>
          </property>
     </bean>
</beans>
```

I called the bean urlMapping, but Spring should pick up the mapping regardless of what you name it. In fact, you can have multiple handler-mapping objects defined. The SimpleUrlHandlerMapping class has a map property called urlMap that maps URLs to objects that implement the Controller interface. When the DispatcherServlet looks in this bean definition file, it will load the mapping and use it to determine which controller to route to. I've created a reference to the PortfolioController.

There are several different types of controllers available in the Spring framework. In this article, I will show you three different types:
1. For the portfolio page, I will create the simplest type of controller: one that implements the Controller interface directly.
2. Next I'll create a logon form that will use a SimpleFormController class.
3. Last, I'll create a trade wizard using the AbstractWizardFormController class.

The Portfolio Page

I first want to change the index.jsp page to redirect to my portfolio page by going through the PortfolioController. I'll also create a JSP called include.jsp that all of my JSP files can include in order to inherit a common set of properties and tag library definitions.

```
/index.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<core:redirect url="/portfolio.htm"/>

/WEB-INF/jsp/include.jsp
<%@ page session="false"%>
```

Now I'll create a simple portfolio view and have my controller route to it just to make sure everything is working.

```
/WEB-INF/jsp/portfolio.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
<head><title>Portfolio Page</title></head>
<body>
Portfolio Page
</body>
</html>
```

Before writing the Controller, you should add a few jar files to your /WEB-INF/lib directory:

Jar	Description	Source
Spring.jar	Main spring jar file	[spring-dist]/dist
log4j-1.2.8.jar	log4j logging package	[spring-dist]/lib/log4j
commons-logging.jar	Jakarta Commons logging package	[spring-dist]/lib/jakarta-commons
jstl.jar	Java standard tag library	[spring-dist]/lib/j2ee
standard.jar	Jakarta standard tag library	[spring-dist]/lib/jakarta-taglibs
taglibs-string.jar	Jakarta tag library used for String manipulation	http://jakarta.apache.org/taglibs/
commons-lang-2.0.jar	http://www.neuro-tech.net/archives/000032.html	http://jakarta.apache.org/commons
jfl.jar	Financial library used to get stock-quotes online	http://www.neuro-tech.net/archives/000032.html

Most of these jars are available in the Spring distribution. All of them are available with the downloadable code for this article.

Now create a class called PortfolioController:

```
/WEB-INF/src/com/devx/tradingapp/Web/PortfolioController.java
package com.devx.tradingapp.Web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.Web.servlet.ModelAndView;
import org.springframework.Web.servlet.mvc.Controller;

public class PortfolioController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) {

        return new ModelAndView("/WEB-INF/jsp/portfolio.jsp");
    }

}
```

The Controller interface defines a single method signature:

```
public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws java.lang.Exception;
```

The object that is returned by the handleRequest method is of the type ModelAndView. As you probably guessed, the ModelAndView object represents the Model and View in the MVC pattern. ModelAndView has several constructors. The one we're using right now just takes a string that represents the view that we want to forward to. Because our portfolio.jsp page doesn't have any dynamic content (yet), we don't need to return a model object.

Compile the PortfolioController and make sure the compiled file is under the WEB-INF/classes directory structure (i.e. /WEB-INF/classes/com/devx/tradingapp/Web/PortfolioController.class). Now you can deploy your application and pull up the index.jsp page again. Go to http://localhost:8080/tradingapp/index.jsp and you should see what is shown in Figure 2.

Now I'll make it a little easier to specify views from within the Controller objects. The goal is to avoid having to type out the full path to a JSP inside of my Controller code. I can achieve this through use of a ViewResolver that I'll define in my tradingapp-servlet.xml file. The ViewResolver appends a prefix and suffix to the view that the Controller returns.

```
/WEB-INF/tradingapp-servlet.xml
    <bean id="viewResolver"
class="org.springframework.Web.servlet.view.InternalResourceViewResolver">
        <property
name="viewClass"><value>org.springframework.Web.servlet.view.JstlView</value></prop
erty>
        <property name="prefix"><value>/WEB-INF/jsp/</value></property>
        <property name="suffix"><value>.jsp</value></property>
    </bean>
```

Now I can simplify my PortfolioController:

```
/WEB-INF/src/com/devx/tradingapp/Web/PortfolioController.java
package com.devx.tradingapp.Web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.Web.servlet.ModelAndView;
import org.springframework.Web.servlet.mvc.Controller;

public class PortfolioController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) {

        return new ModelAndView("portfolio");
    }

}
```

Figure 2. Back to the Portfolio: The index.jsp page should now redirect you to the Portfolio Page.

Just to make sure that the Portfolio Page still loads after this change, reload the page in your Web browser (you may have to reload the application context or restart your application server if your application server does not support hot-deploy functionality).

Now I want to make the portfolio page more interesting! First, I'll add some custom-tag library definitions to the include.jsp file. Using custom tags helps me keep my presentation logic separate from the presentation itself.

```
/WEB-INF/jsp/include.jsp
<%@ page session="false"%>

<%@ taglib prefix="core" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<%@ taglib prefix="str" uri="http://jakarta.apache.org/taglibs/string-1.1" %>
```

Next, update your PortfolioController (see the code in Listing 1).

Listing 1

Listing 1. /WEB-INF/src/com/devx/tradingapp/Web/PortfolioController.java

```
/WEB-INF/src/com/devx/tradingapp/Web/PortfolioController.java

package com.devx.tradingapp.Web;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import net.neurotech.quotes.Quote;
import net.neurotech.quotes.QuoteException;
import net.neurotech.quotes.QuoteFactory;

import org.springframework.Web.servlet.ModelAndView;
import org.springframework.Web.servlet.mvc.Controller;

import com.devx.tradingapp.business.Portfolio;

public class PortfolioController implements Controller {

    private com.devx.tradingapp.business.Portfolio portfolio;

    public PortfolioController(Portfolio portfolio) {
        this.portfolio = portfolio;
    }

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) {
        Map model = new HashMap();

        List portfolioItems = getPortfolioItems();

        model.put("cash", portfolio.getCash() + "");
        model.put("portfolioItems", portfolioItems);
```

continued

In the new version of the Controller, I've added a model. I use the three-argument constructor for ModelAndView that takes the view, the string name that the JSPs will use to refer to the model, and the model object itself. In Spring, a model is usually just a java.util.Map object. I put two elements on the model, a cash amount, and a list of portfolio line items. Each line item is of type PortfolioItemBean, a JavaBean that I created. This bean is primarily for View purposes and is created using data from the underlying business object called Portfolio. Listing 2 shows these classes.

You'll also notice that I'm using a class called QuoteFactory to obtain a Quote object using a stock symbol. These classes are part of the Neurotech Java Financial Library, a simple, open source API that can be used to retrieve stock quotes from the Web as well as other simple financial tasks.

Listing 3 shows the updated tradingapp-servlet.xml file and the portfolio.jsp file. If all goes well, when you deploy and reload the page you should see something very similar to Figure 3.

Listing 1

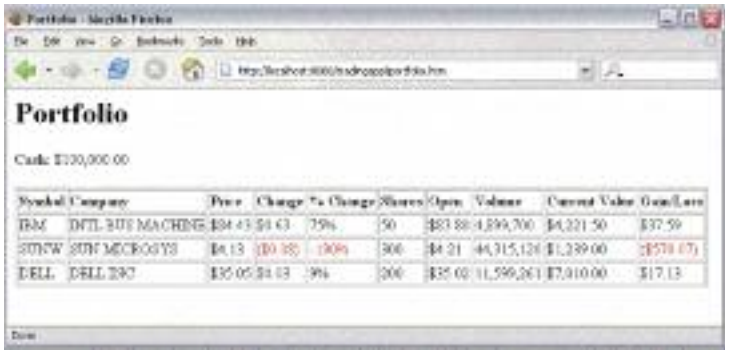
```
        return new ModelAndView("portfolio", "model", model);
    }

    private List getPortfolioItems() {
        List portfolioItems = new ArrayList();

        Iterator symbolIter = portfolio.getSymbolIterator();

        while (symbolIter.hasNext()) {
            String symbol = (String) symbolIter.next();

            int shares = portfolio.getNumberOfShares(symbol);
            QuoteFactory quoteFactory = new QuoteFactory();

            Quote quote = null;

            try {
                quote = quoteFactory.getQuote(symbol);
            } catch (QuoteException e) {
                quote = new Quote(this.getClass().getName()) {
                };
            }

            PortfolioItemBean portfolioItem = new PortfolioItemBean();
            portfolioItem.setSymbol(symbol);
            portfolioItem.setShares(shares);
            portfolioItem.setQuote(quote);
            portfolioItem.setCurrentValue(shares * quote.getValue());
            portfolioItem.setGainLoss(shares * quote.getPctChange());
            portfolioItems.add(portfolioItem);
        }
        return portfolioItems;
    }

}
```

The cool thing is that the PortfolioController retrieves quotes from the Web using the Java Financial Library, so if you keep refreshing the page during trading hours, you'll get updated quotes.

In summary, here is the flow to this page:

1. The user goes to the portfolio.htm page.
2. He is routed to the dispatcher Servlet (because all .htm pages are directed to the dispatcher Servlet).
3. The DispatcherServlet loads the tradingapp-servlet.xml file and routes the user to the PortfolioController.

Figure 3. Improving the Portfolio: The portfolio view uses custom tags to display model data that is provided by the PortfolioController.



Listing 2

```
/WEB-INF/src/com/devx/tradingapp/Web/PortfolioBeanItem.java
/WEB-INF/src/com/devx/tradingapp/business/Portfolio.java
/WEB-INF/src/com/devx/tradingapp/Web/PortfolioBeanItem.java
package com.devx.tradingapp.Web;

import net.neurotech.quotes.Quote;

public class PortfolioItemBean {
     private String symbol;

     private int shares;

     private Quote quote;

     private double currentValue;

     private double gainLoss;

     public int getShares() {
          return shares;
     }

     public void setShares(int quantity) {
          this.shares = quantity;
     }

     public String getSymbol() {
          return symbol;
     }

     public void setSymbol(String symbol) {
          this.symbol = symbol;
     }

     public double getCurrentValue() {
```

continued

4. The PortfolioController creates the model and view and passes control back to the DispatcherServlet.
5. The DispatcherServlet makes the model data accessible via session or request parameters.
6. The DispatcherServlet routes to the JSP pages.
7. The JSP is rendered and the presentation is sent back to the user.

The Logon Page

Now you know how to create a page with dynamically updated content, but what good is a Web application that has no way to capture user input from an HTML form? We could implement the Controller interface in order to do form processing, but that would require us to do a lot of manual processing or request parameters and storing things on the session. Personally, I've hand coded more form-processing code in this fashion than I ever hope to again. Thus, out of sheer laziness, I will show you how Spring simplifies form handling via the SimpleFormController by building a logon page.

Listing 4 shows the logon JSP. The <spring:bind> tag and the <spring:hasBindErrors> tag are in the spring.tld tag library descriptor that comes with Spring. Add this file to your WEB-INF directory and then add

Listing 2

```
        return currentValue;
    }

    public void setCurrentValue(double currentValue) {
        this.currentValue = currentValue;
    }

    public Quote getQuote() {
        return quote;
    }

    public void setQuote(Quote quote) {
        this.quote = quote;
    }

    public double getGainLoss() {
        return gainLoss;
    }

    public void setGainLoss(double valueChange) {
        this.gainLoss = valueChange;
    }
}



/WEB-INF/src/com/devx/tradingapp/business/Portfolio.java

package com.devx.tradingapp.business;

import java.util.Iterator;
import java.util.Map;

public class Portfolio {
```

continued

the following to your Web.xml and include.jsp files.

```
/WEB-INF/Web.xml
     <taglib>
          <taglib-uri>/spring</taglib-uri>
          <taglib-location>/WEB-INF/spring.tld</taglib-location>
     </taglib>

/WEB-INF/jsp/include.jsp
<%@ page session="false"%>

<%@ taglib prefix="core" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<%@ taglib prefix="str" uri="http://jakarta.apache.org/taglibs/string-1.1" %>
<%@ taglib prefix="spring" uri="/spring" %>
```

Listing 2

```
    private float cash;

    //maps symbol string to shares
    private Map sharesPerSymbol;

    public Portfolio(float cash, Map sharesPerSymbol) {
        this.cash = cash;
        this.sharesPerSymbol = sharesPerSymbol;
    }

    public float getCash() {
        return cash;
    }

    public boolean contains(String symbol) {
        return sharesPerSymbol.containsKey(symbol);
    }

    public int getNumberOfShares(String symbol) {
        Object shares = sharesPerSymbol.get(symbol);

        if (shares instanceof String) {
            return Integer.parseInt((String) shares);
        } else if (shares instanceof Integer) {

            return ((Integer) shares).intValue();
        } else {
            throw new RuntimeException("Application error");
        }
    }

    public Iterator getSymbolIterator() {
```

continued

The bind tag tells Spring to bind the enclosed form element to the bean property specified by the path attribute. For example the username input parameter will be bound to the username field on a bean named "credentials." The credentials bean is a special type of bean called a "command" or a "form-backing bean." Spring uses command beans for data binding while processing forms. The name used to reference this bean and its class type is defined in the tradingapp-servlet.xml file in the logonForm bean (see Listing 5).

I've also added a new entry to the handler mapping to point to our logon form when the user navigates (or is routed) to the logon.htm URL. The logon form has several properties:

- commandClass—the class of the object that will be used to represent the data in this form.
- commandName—the name of the command object.
- sessionForm—if set to false, Spring uses a new bean instance (i.e. command object) per request, otherwise it will use the same bean instance for the duration of the session.
- validator—a class that implements Spring's Validator interface, used to validate data that is passed in from the form.
- formView—the JSP for the form, the user is sent here when the controller initially loads the form and when the form has been submitted with invalid data.
- successView—the JSP that the user is routed to if the form submits with no validation errors.

Listing 2

```
        return sharesPerSymbol.keySet().iterator();
    }

    public void buyStock(String symbol, int sharesBought, float purchasePrice) {
        cash -= sharesBought * purchasePrice;
        if (sharesPerSymbol.containsKey(symbol)) {
            int currentShares = getNumberOfShares(symbol);
            sharesPerSymbol.put(symbol, new Integer(currentShares
                    + sharesBought));
        } else {
            sharesPerSymbol.put(symbol, new Integer(sharesBought));
        }
    }

    public void sellStock(String symbol, int sharesSold, float sellPrice) {
        cash += sharesSold * sellPrice;
        int currentShares = getNumberOfShares(symbol);
        int sharesLeft = currentShares - sharesSold;
        if (sharesLeft == 0) {
            sharesPerSymbol.remove(symbol);
        } else {
            sharesPerSymbol.put(symbol, new Integer(sharesLeft));
        }

    }

    public boolean canBuy(int shares, float purchasePrice) {
        if ((shares * purchasePrice) <= cash) {
            return true;
        } else {
            return false;
        }
    }
}
```

Here is the code for the Controller:

```
/WEB-INF/src/com/devx/tradingapp/Web/LogonFormController.java
package com.devx.tradingapp.Web;

import javax.servlet.ServletException;

import org.springframework.Web.servlet.ModelAndView;
import org.springframework.Web.servlet.mvc.SimpleFormController;
import org.springframework.Web.servlet.view.RedirectView;

public class LogonFormController extends SimpleFormController {
    public ModelAndView onSubmit(Object command) throws ServletException {
        return new ModelAndView(new RedirectView(getSuccessView()));
    }

}
```

Listing 3

```
/WEB-INF/tradingapp-servlet.xml
/WEB-INF/jsp/portfolio.jsp
/WEB-INF/tradingapp-servlet.xml
     <bean id="portfolioController"
           class="com.devx.tradingapp.Web.PortfolioController">
          <constructor-arg index="0">
               <ref bean="portfolio"/>
          </constructor-arg>
     </bean>

    <bean id="portfolio" class="com.devx.tradingapp.business.Portfolio">
         <constructor-arg index="0"><value>100000</value></constructor-arg>
         <constructor-arg index="1">
              <map>
                   <entry key="IBM"><value>50</value></entry>
                   <entry key="SUNW"><value>300</value></entry>
                   <entry key="DELL"><value>200</value></entry>
              </map>
         </constructor-arg>
    </bean>
.
.
.

/WEB-INF/jsp/portfolio.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
<head><title>Portfolio</title></head>
<body>
<h1>Portfolio</h1>
<b>Cash:</b> <fmt:formatNumber value="${ model.cash} " type="currency" />
```

continued

There's really not much to it. All the FormController does is forward to the success view. The validator takes care of authenticating the user. If the user provides an invalid username or password, the validator will return the user to the form and display an error message. Listing 6 shows the code for the validator.

The validator has a supports method and a validate method. The supports method is called to see if the validator supports a given object type. We want our validator to be able to validate objects of the type Credentials, because that is the command object that we are using for this form.

The validate method does some checks and calls the rejectValue method on the Errors class if validation fails. This method takes four parameters: the name of the field that failed validation, the key value of the error message that exists in a resource file (I'll explain in a second), a string array of values to substitute into the error message in the resource file, and the default error message that should be displayed if the resource file

Listing 3

```
<br/>
<br/>
<table border="1">
      <tr>
            <td><b>Symbol</b></td>
            <td><b>Company</b></td>
            <td><b>Price</b></td>
            <td><b>Change</b></td>
            <td><b>% Change</b></td>
            <td><b>Shares</b></td>
            <td><b>Open</b></td>
            <td><b>Volume</b></td>
            <td><b>Current Value</b></td>
            <td><b>Gain/Loss</b></td>
      </tr>
      <core:forEach items="${ model.portfolioItems} " var="stock">
      <tr>
            <td><str:upperCase><core:out
value="${ stock.symbol} "/></str:upperCase></td>
            <td><core:out value="${ stock.quote.company} "/></td>
            <td><fmt:formatNumber value="${ stock.quote.value} " type="currency"
/></td>
            <td>
                  <core:choose>
                        <core:when test="${ stock.quote.change >= 0} ">
                              <fmt:formatNumber value="${ stock.quote.change} "
type="currency" />
                        </core:when>
                        <core:otherwise>
                              <font color="red">
                                    <fmt:formatNumber value="${ stock.quote.change} "
type="currency" />
                              </font>
                        </core:otherwise>
                  </core:choose>
            </td>
            <td>
                  <core:choose>
```

continued

cannot be found. If any errors are found on the Errors object after validate is done running, Spring will forward control back to the original form JSP so that the errors can be displayed.

You may have noticed above that I also added a ResourceBundleMessageSource bean definition to the tradingapp-servlet.xml file. This is a reference to a properties file that contains application messages that can be defined and accessed throughout the application. These messages can be accessed in many ways, including being directly accessed from JSPs or, as we just saw, by the Errors object to provide error messages. This file must exist in the top level of your classpath, so I just put it in the WEB-INF/classes directory. I've jumped the gun and included all the error messages that our application will use in the file:

Listing 3

```
                        <core:when test="${ stock.quote.pctChange >= 0} ">
                              <fmt:formatNumber value="${ stock.quote.pctChange} "
type="percent" />
                        </core:when>
                        <core:otherwise>
                              <font color="red">
                                    <fmt:formatNumber
value="${ stock.quote.pctChange} " type="percent" />
                              </font>
                        </core:otherwise>
                  </core:choose>
            </td>
            <td><fmt:formatNumber value="${ stock.shares} "/></td>
            <td><fmt:formatNumber value="${ stock.quote.openPrice} " type="currency"
/></td>
            <td><fmt:formatNumber value="${ stock.quote.volume} "/></td>
            <td><fmt:formatNumber value="${ stock.currentValue} " type="currency"
/></td>
            <td>
                  <core:choose>
                        <core:when test="${ stock.gainLoss >= 0} ">
                              <fmt:formatNumber value="${ stock.gainLoss} "
type="currency" />
                        </core:when>
                        <core:otherwise>
                              <font color="red">
                                    <fmt:formatNumber value="${ stock.gainLoss} "
type="currency" />
                              </font>
                        </core:otherwise>
                  </core:choose>
            </td>
      </tr>
      </core:forEach>
</table>
<br>
<br>
</body>
</html>
```

continued

```
/WEB-INF/classes/messages.properties
error.login.not-specified=User credentials not specified (try guest/guest).
error.login.invalid-user=Username not valid, try 'guest'
error.login.invalid-pass=Password not valid, try 'guest'
error.trade.insufficient-funds=You do not have enough money to place this order
error.trade.not-enough-shares=You do not have that many shares
error.trade.dont-own=You don't own this stock
```

Listing 4

```
/WEB-INF/jsp/logon.jsp
/WEB-INF/jsp/logon.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
<head><title>DevX.com Stock-Trading System Logon</title></head>
<body>

<center>

<h1>Welcome to the DevX.com Stock-Trading System</h1>
<br/>

<form method="post">
     <table width="25%" border="1">
          <tr>
               <td align="center" bgcolor="lightblue">Log on</td>
          </tr>
          <tr>
               <td>
                    <table border="0" width="100%">
                         <tr>
                              <td width="33%" align="right">Username: </td>
                              <td width="66%" align="left">
                                   <spring:bind path="credentials.username">
                                   <input type="text"
                                          name="username"
                                          value="<core:out
value="${ status.value} "/>"/>
                                   </spring:bind>
                              </td>

                         </tr>
                         <tr>
                              <td colspan="2" align="center">
                                   <spring:hasBindErrors name="credentials">
                                   <font color="red"><core:out
value="${ status.errorMessage} "/></font>
                                   </spring:hasBindErrors>
                              </td>
                         </tr>
                         <tr>
```

continued

```
error.trade.invalid-symbol=Invalid ticker symbol: {0}
```

Here is the code for the infamous Credentials class (the command/form-backing object):

```
WEB-INF/src/com/devx/tradingapp/business/Credentials.java
package com.devx.tradingapp.business;

public class Credentials {
    private String username;

    private String password;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
```

Listing 4

```
                                    <td width="33%" align="right">Password: </td>
                                    <td width="66%" align="left">
                                         <spring:bind path="credentials.password">
                                         <input type="password" name="password" />
                                         </spring:bind>
                                    </td>
                               </tr>
                               <tr>
                                    <td colspan="2" align="center">
                                         <spring:hasBindErrors name="credentials">
                                         <font color="red"><core:out
value="${status.errorMessage}"/></font>
                                         </spring:hasBindErrors>
                                    </td>
                               </tr>
                               <tr>
                                    <td align="center" colspan="2">
                                         <input type="submit" alignment="center"
value="Logon">
                                    </td>
                               </tr>
                          </table>

                    </td>
               </tr>
          </table>

</form>

</center>

</body>
</html>
```

```
        this.password = password;
    }
    public String getUsername() {
        return username;
    }

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

    }
}
```

Listing 5

```
/WEB-INF/tradingapp-servlet.xml
/WEB-INF/tradingapp-servlet.xml
    <bean id="logonValidator" class="com.devx.tradingsystem.Web.LogonValidator"/>

     <bean id="logonForm" class="com.devx.tradingsystem.Web.LogonFormController">
        <property name="sessionForm"><value>true</value></property>
        <property name="commandName"><value>credentials</value></property>
        <property
name="commandClass"><value>com.devx.tradingsystem.Web.logon.Credentials</value></
property>
        <property name="validator"><ref bean="logonValidator"/></property>
        <property name="formView"><value>logon</value></property>
        <property name="successView"><value>portfolio.htm</value></property>
    </bean>

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

     <!-- you can have more than one handler defined -->
     <bean id="urlMapping"

class="org.springframework.Web.servlet.handler.SimpleUrlHandlerMapping">
          <property name="urlMap">
               <map>
                    <entry key="/portfolio.htm">
                         <ref bean="portfolioController" />
                    </entry>
                    <entry key="/logon.htm">
                         <ref bean="logonForm"/>
                    </entry>
               </map>
          </property>
     </bean>
.
.
.
```

continued

Now you can change the index.jsp page in the root directory to point to logon.htm.

```
/index.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<core:redirect url="/logon.htm"/>
```

Now if you load index.jsp, you will see the logon page, as shown in Figure 4.

The Trade Wizard

Often times we have more than one screen that a user interacts with in order to complete a given task. This sequence of screens is often called a "wizard." The last MVC component I will introduce you to is the AbstractWizardFormController. This controller allows you to carry the same command object through an entire flow of Web pages, thus allowing you to break up the presentation into multiple stages that act on the same model.

For the example, I'll create a wizard that allows a user to trade stock. This wizard will start with a page that takes the order and then goes to a confirmation page where the user will choose to either execute or cancel the order. If the order is cancelled, the user will be

Figure 4. Validating: When the logon page does not validate correctly, the user is returned to the form page where the validation errors are displayed.

Listing 6

```
/WEB-INF/src/com/devx/tradingapp/Web/LogonValidator.java
/WEB-INF/src/com/devx/tradingapp/Web/LogonValidator.java
package com.devx.tradingapp.Web;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

import com.devx.tradingapp.business.Credentials;

public class LogonValidator implements Validator {

    private final Log logger = LogFactory.getLog(getClass());

    public boolean supports(Class clazz) {
        return clazz.equals(Credentials.class);
    }

    public void validate(Object obj, Errors errors) {
        Credentials credentials = (Credentials) obj;
```

continued

returned to the portfolio page, if the order is executed the user is sent to an acknowledgement page to tell them that their order was filled.

First I need a way to get to the trade page, so I'll put a link to the trade page on the portfolio page.

```
/WEB-INF/jsp/portfolio.jsp
<br>
<a href="<core:url value="trade.htm"/>">Make a trade</a><br/>
<a href="<core:url value="logon.htm"/>">Log out</a>
<br>
</body>
</html>
```

Now I'll update the tradingapp-servlet.xml so that it knows what to do when the user clicks on the trade.htm link.

```
/WEB-INF/tradingapp-servlet.xml
     <bean id="tradeForm" class="com.devx.tradingapp.Web. TradeFormController">
        <constructor-arg index="0">
            <ref bean="portfolio"/>
        </constructor-arg>
    </bean>

     <!-- you can have more than one handler defined -->
     <bean id="urlMapping"
          class="org.springframework.Web.servlet.handler.SimpleUrlHandlerMapping">
          <property name="urlMap">
               <map>
                    <entry key="/portfolio.htm">
                         <ref bean="portfolioController" />
```

Listing 6

```
        if (credentials == null) {
            errors.rejectValue("username", "error.login.not-specified", null,
                    "Value required.");
        } else {
            logger.info("Validating user credentials for: "
                    + credentials.getUsername());
            if (credentials.getUsername().equals("guest") == false) {
                errors.rejectValue("username", "error.login.invalid-user",
                        null, "Incorrect Username.");
            } else {
                if (credentials.getPassword().equals("guest") == false) {
                    errors.rejectValue("password", "error.login.invalid-pass",
                            null, "Incorrect Password.");
                }
            }

        }
    }

}
```

```
                    </entry>
                    <entry key="/logon.htm">
                          <ref bean="logonForm"/>
                    </entry>
                    <entry key="/trade.htm">
                          <ref bean="tradeForm"/>
                    </entry>
               </map>
          </property>
     </bean>
```

The TradeFormController should exist as /WEB-INF/src/com/devx/tradingapp/Web/TradeFormController.java. But there is a lot to the TradeFormController, so I'll explain it in segments.

```
public class TradeFormController extends AbstractWizardFormController {

    private Portfolio portfolio;

    public TradeFormController(Portfolio portfolio) {
        this.portfolio = portfolio;
```

Listing 7

```
/WEB-INF/src/com/devx/tradingapp/business/Trade.java
/WEB-INF/src/com/devx/tradingapp/business/Trade.java
package com.devx.tradingapp.business;

public class Trade {

    public static final boolean BUY = true;

    public static final boolean SELL = false;

    private boolean buySell;

    private String symbol;

    private int shares;

    private float price;

    public boolean isBuySell() {
        return buySell;
    }

    public void setBuySell(boolean buySell) {
        this.buySell = buySell;
    }
```

continued

```
        setPages(new String[] {  "trade", "trade-confirm" });
        setCommandName("trade");
    }
```

The TradeFormController extends AbstractWizardFormController. I pass in a Portfolio object so that the trade form knows what my buying limit is and can add and remove stock from the portfolio. The setPages method in the constructor tells the form the sequence of pages we are going to call.

The strings in this array relate to views that are resolved by the ViewResolver just like in the other controllers. These pages are indexed by Spring starting at 0. Thus the trade view is index 0 and the trade-confirm view is index 1. The indexes exist because we may have an event that causes us to skip or go back to a previous page. The setCommandName method sets the name of the command object that we are going to use for this form. This object will be created in the formBackingObject method.

```
    protected Object formBackingObject(HttpServletRequest request) {
        Trade trade = new Trade();
        trade.setBuySell(Trade.BUY);
        return trade;
    }
```

This method will create the command object and set it in its initial state. This method is called before the user is directed to the first page of the wizard. Subsequent submits will call several methods on the abstract controller. The main methods that are called are onBind, validatePage, and getTargetPage.

Listing 7

```
    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public int getShares() {
        return shares;
    }

    public void setShares(int quantity) {
        this.shares = quantity;
    }

    public String getSymbol() {
        return symbol;
    }

    public void setSymbol(String symbol) {
        this.symbol = symbol;
```

continued

```
onBind
    protected void onBind(HttpServletRequest request, Object command,
            BindException errors) {

        Trade trade = (Trade) command;

        if (symbolIsInvalid(trade.getSymbol())) {
            errors.rejectValue("symbol", "error.trade.invalid-symbol",
                    new Object[] {  trade.getSymbol() },
                    "Invalid ticker symbol.");
        } else {
            Quote quote = null;
            try {
                quote = new QuoteFactory().getQuote(trade.getSymbol());
            } catch (QuoteException e) {
                throw new RuntimeException(e);
            }
            trade.setPrice(quote.getValue());
            trade.setSymbol(trade.getSymbol().toUpperCase());
        }
    }
```

The onBind method is called before any validation occurs for each submit. I override the onBind method in order to get a quote on the symbol the user is trying to purchase and set the price and symbol on the command object. Keep in mind that even if onBind adds errors to the BindException, the validatePage method will still be called.

Listing 8

```
/WEB-INF/jsp/trade.jsp
/WEB-INF/jsp/trade-confirm.jsp
/WEB-INF/jsp/trade-acknowledge.jsp
/WEB-INF/jsp/trade.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
<head><title>Trade</title></head>
<body>
<h1>Trade</h1>
<form method="post">

<!-- first bind on the object itself to display global errors - if available -->
<spring:bind path="trade.*">
     <font color="red">
          <core:forEach items="${status.errorMessages}" var="error">
          Error: <core:out value="${error}"/><br/>
          </core:forEach>
    </font>
    <br/>
</spring:bind>

<table border="1">                                                    continued
```

```
protected void validatePage(Object command, Errors errors, int page) {
    Trade trade = (Trade) command;

    if (tradeIsBuy(trade)) {
        if (insufficientFunds(trade)) {
            errors.reject("error.trade.insufficient-funds",
                    "Insufficient funds.");
        }
    } else if (tradeIsSell(trade)) {
        if (portfolio.contains(trade.getSymbol()) == false) {
            errors.rejectValue("symbol", "error.trade.dont-own",
                    "You don't own this stock.");
        } else if (notEnoughShares(trade)) {
            errors.rejectValue("quantity", "error.trade.not-enough-shares",
                    "Not enough shares.");
        }
    }
}
```

validatePage

The validatePage method is called after the onBind method. It acts in the same way as the validator did in the logon controller. The validatePage method is passed the page number for the page in the flow that you are validating. This can be used if you need to do custom validation for each page in the flow. If you wanted to, you could create validator objects for each page and use the validatePage method to delegate to the appropriate validator based on the page index that was passed in.

Listing 8

```
<tr>
      <td></td>
      <td><b>Symbol</b></td>
      <td><b>Shares</b></td>
</tr>
<tr>
      <td>
            <spring:bind path="trade.buySell">
            <input type="radio"
                    name="buySell"
                    value="true"
                    <core:if test="${status.value}">checked</core:if> >
                  Buy
            </input>
            <input type="radio"
                    name="buySell"
                    value="false"
                    <core:if test="${!status.value}">checked</core:if> >
                  Sell
            </input>
            </spring:bind>
      </td>
```

continued

getTargetPage

The getTargetPage method will be called in order to find out which page to navigate to. This method can be overridden in your controller, but I use the default implementation, which looks for a request parameter starting with "_target" and ending with a number (e.g. "_target1"). The JSP pages should provide these request parameters so that the wizard knows which page to go to even when the user uses the Web-browser's back button.

The processFinish method is called if there is a submit that validates and contains the "_finish" request parameter. The processCancel method is called if there is a submit that contains the "_cancel" request parameter.

```
    protected ModelAndView processFinish(HttpServletRequest request,
            HttpServletResponse response, Object command, BindException errors) {
        Trade trade = (Trade) command;

        if (trade.isBuySell() == Trade.BUY) {
            portfolio.buyStock(trade.getSymbol(), trade.getShares(), trade
                    .getPrice());
        } else {
            portfolio.sellStock(trade.getSymbol(), trade.getShares(), trade
                    .getPrice());
        }
        return new ModelAndView("trade-acknowledge", "trade", trade);
    }

    protected ModelAndView processCancel(HttpServletRequest request,
```

Listing 8

```
            <td>
                <spring:bind path="trade.symbol">
                <input type="text" name="symbol" value="<core:out
value="${ status.value} "/>"/>
                </spring:bind>
            </td>
            <td>
                <spring:bind path="trade.shares">
                <input type="text" name="shares" value="<core:out
value="${ status.value} "/>"/>
                </spring:bind>
            </td>
        </tr>
        <tr>
            <td colspan="3" align="center"><input type="submit" alignment="center"
name="_target1" value="Execute Order"></td>
        </tr>
</table>


</form>
<br>
<a href="<core:url value="portfolio.htm"/>">View Portfolio</a><br/>
<a href="<core:url value="logon.htm"/>">Log out</a>
```

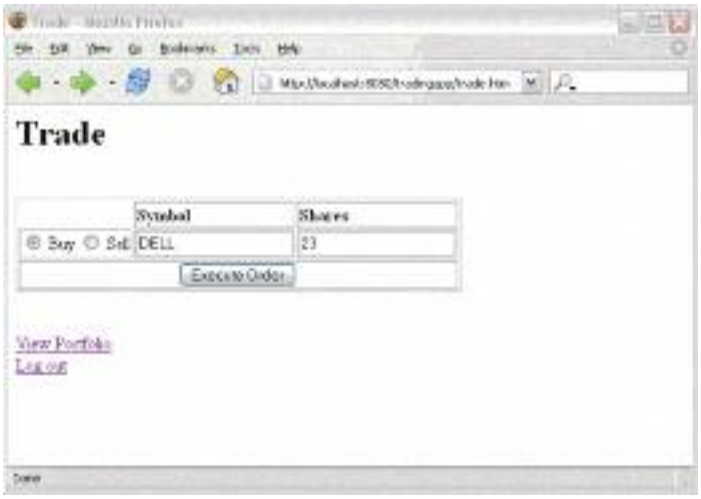
continued

```
            HttpServletResponse response, Object command, BindException errors) {
        return new ModelAndView(new RedirectView("portfolio.htm"));
    }
```

Listing 7 is the code for the Trade class (our command for this form). And now all you need are the JSP pages (see Listing 8 for the three JSP pages).

As you can see in the code, the Spring-specific request parameters are specified in the submit button name attribute. If all goes well, you should be able to pull up the first page of the trade wizard by going to http://localhost:8080/tradingapp/trade.htm (see Figure 5).

Figure 5. Ready to Trade: The first page of the trade wizard allows you place an order for a stock trade.



In summary, here is the flow through this form:

1. The user goes to the trade.htm page.
2. He is routed to the dispatcher servlet (because all .htm pages are directed to the dispatcher servlet).
3. The DispatcherServlet loads the tradingapp-servlet.xml file and routes the user to the TradeFormController.
4. The TradeFormController loads the command bean by calling the formBackingObject method

Listing 8

```
<br>
</body>
</html>


/WEB-INF/jsp/trade-confirm.jsp

<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
<head><title>Trade Confirmation</title></head>
<body>
<h1>Trade Confirmation</h1>
<form method="post">

<table border="1">
      <tr>
            <td></td>
            <td><b>Symbol</b></td>
            <td><b>Shares</b></td>
      </tr>
      <tr>
            <td>
                  <core:choose>
```

continued

and routes the user to the first page defined in the setPages call in the constructor (trade.jsp).
5. The user fills out the form and submits it.
6. The user is directed back to the controller, which parses the target page off of the request parameters from trade.jsp, binds and validates the command object, and forwards the user to the next page in the wizard.
7. If the validator fails, the user is sent back to the form view and error messages are displayed (back to trade.jsp).
8. If the validator doesn't fail, the controller forwards to the trade-confirm.jsp page.
9. The user submits an execute or cancel command, which will in turn tell the controller to execute either the processFinish or processCancel commands, respectively.
10. The user is forwarded to the page that the processFinish or processCancel command sends them to.

Listing 8

```
                    <core:when test="${trade.buySell == true}">Buy</core:when>
                    <core:otherwise>Sell</core:otherwise>
               </core:choose>
          </td>
          <td>
               <core:out value="${trade.symbol}"/>
          </td>
          <td>
               <core:out value="${trade.shares}"/>
          </td>
     </tr>
     <tr>
          <td colspan="3" align="center">
               <input type="submit" alignment="center" name="_finish"
value="Execute Order">
               <input type="submit" alignment="center" name="_cancel"
value="Cancel Order">
          </td>
     </tr>
</table>


</form>
<br>
<a href="<core:url value="logon.htm"/>">Log out</a>
<br>
</body>
</html>


/WEB-INF/jsp/trade-acknowledge.jsp

<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
<head><title>Successful Trade Acknowledgement</title></head>
<body>
<h1>Successful Trade Acknowledgement</h1>
```

continued

I've gone over three different types of controllers provided by Spring, which should be enough to get you started on your own Web projects. I have found Spring to be a very valuable tool in my software development toolbox, and I urge you to discover its vast utility for yourself. ■

Listing 8

```
<form method="post">

<table border="1">
      <tr>
            <td></td>
            <td><b>Symbol</b></td>
            <td><b>Shares</b></td>
      </tr>
      <tr>
            <td>
                  <core:choose>
                        <core:when test="${trade.buySell ==
true}">Bought</core:when>
                        <core:otherwise>Sold</core:otherwise>
                  </core:choose>
            </td>
            <td>
                  <core:out value="${trade.symbol}"/>
            </td>
            <td>
                  <core:out value="${trade.shares}"/>
            </td>
      </tr>
</table>

<h2>Your order was filled</h2>

</form>
<br>
<a href="<core:url value="portfolio.htm"/>">View Portfolio</a><br/>
<a href="<core:url value="logon.htm"/>">Log out</a>
<br>
</body>
</html>
```