STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

MANNING

Table of Contents

Chapter 7. UI component tags. Struts 2 in Action, ISBN: 9781933988078

7 UI component tags

This chapter covers

- Generating HTML with UI component tags
- Building forms and beyond
- Snazzing it up with templates and themes
- Surveying the components

In the previous chapter, we introduced the Struts 2 tag API. As we saw, this high-level API provides a common set of tag functionality that you can access from any of three view-layer technologies: JSPs, Velocity, and FreeMarker templates. Learn the tag API once, and you'll start using it everywhere. We saw tags that helped us pull data from the `ValueStack`, iterate over collections of data, and even control the flow of page rendering with conditional logic of various sorts. Now it's time to start building a user interface with a special set of tags known as the UI component tags.

UI components take some introduction. Each UI component is a functional unit with which the user can interact and enter data. At the heart of each Struts 2 UI component is an HTML form control, such as a text field or select box. But don't be mistaken: these components are much more than just tags that output an HTML input element. They are a higher-level component, of which the HTML element is only the browser manifestation. They integrate all areas of the framework,

from data transfer and validation to internationalization and look and feel. Some of these components even combine more than one HTML form element to build new functional units for your pages.

In addition to all their functional capacity, the UI components are built on a layered, mini-MVC architecture that isolates the HTML markup output of a given component into an underlying FreeMarker template. This empowers the developer to modify the components themselves simply by modifying these underlying templates. This flexibility can provide developers with a powerful tool for meeting the finicky demands of user-interface requirements while still leveraging the reusability of a component-based architecture. This chapter will familiarize you with the UI component architecture, as well as provide a hands-on tour of the various component tags and demonstrate their use with sample code from the Struts 2 Portfolio.

This chapter might seem long, but the UI components have a lot to offer, and much of this chapter's later parts are reference-oriented. You probably won't need to read about the more advanced UI components until you need to use them.

7.1 Why we need UI component tags

I used to say that I'd never be a front-end developer. All of those hackish browser workarounds and convoluted JavaScript tangles, a million different ways to solve common problems, a dozen ways to solve each eccentric browser-specific problem—this is no place for anyone with an inclination toward order. Now things are clearing up as browsers begin to reach a higher level of standards compliance. You can now count on a large share of browsers behaving as you'd expect most of the time.

But this increasing compliance to standards hasn't made life simpler. Instead of taking a moment to appreciate the calm, developers have taken advantage of the stabilizing front-end platform to pile on a whole new wave of front-end complexity. Each day, dynamic front ends, powered by Ajax and other rich-client technologies, get closer to becoming commonplace requirements. A new web application framework, such as Struts 2, would be in grave error if it didn't lay the architectural foundations for managing the increasing complexity of the front end. The Struts 2 UI components pull this off quite well. In this chapter, we'll see how.

7.1.1 More than just form elements

As most of the work of an HTML user interface is accomplished by forms and form elements, many of the UI component tags correlate to HTML elements. There's a Struts 2 UI component for all of the commonly used HTML form elements. Moving past this one-to-one relationship, the more complex UI components wrap a couple of HTML elements into a single unit to create a new form widget. If you like, you can think of the rendering of the corresponding HTML element(s) as the foundation of the UI component. But, ultimately, the UI components are more than the sum of their underlying HTML elements. The following list summarizes some of the things that a UI component can do for you:

- Generate HTML markup
- Bind HTML form fields to Java-side properties
- Tie into framework type conversion
- Tie into framework validation
- Tie into framework internationalization

In this section, we'll explain each of these functional roles. We'll start with generating the markup.

GENERATING THE HTML MARKUP

First of all, each UI component tag does indeed generate a wad of HTML markup. This markup defines the corresponding HTML element and frequently some additional layout markup. This is the simplest aspect of the component tags. For instance, the Struts 2 `textfield` tag creates an HTML text input element. To parameterize the output of the component, the Struts 2 tag exposes various attributes to the developer. Many of these attributes mirror those found on the HTML element itself, frequently in a one-to-one fashion. However, as is typical for Struts 2, you won't usually need to define many of the attributes by hand. Their values will be deduced by the framework based on convention and intelligent defaults. This makes your page code much cleaner, faster, and more reusable.

As a simple example, let's look at the Struts 2 `textfield` component. Consider the following Struts 2 `textfield` tag:

```
<s:textfield name="username" label="Username"/>
```

As you can see, the only attributes that we set are `name` and `label`, but the output, shown in the following snippet, deduces a lot of information from these two attributes:

```
<td class="tdLabel">
  <label for="Register_username" class="label">Username:</label>
</td>
<td>
  <input type="text" name="username" value="" id="Register_username"/>
</td>
```

First of all, don't be confused by the table markup. This is done by the XHTML theme, one of the themes you can choose to determine the layout style used when rendering the UI component. We'll cover it in detail in section 7.2. For now, it's enough to know that the XHTML theme uses a table to control the layout of form elements. It also adds some class attributes for CSS control, and it can even generate a simple default stylesheet for these styles, if you like. Later, when we cover themes, you'll see that you can choose other themes for rendering your tags, including a theme that will use pure CSS for form layout.

For now, let's not think about the layout. If we look at the HTML elements themselves, we can see that the `textfield` component tag produced two: a `label` and an `input`. As you'd expect, it generates the HTML input field, with `type` set to `text`. It also creates an `id` by concatenating the enclosing form's `name` with the input field's `name`; note that we

haven't shown the enclosing form in this example. Since CSS and JavaScript both depend so heavily on the existence of `id`s, this has become a foundational feature of the component tags. Additionally, the `textfield` tag, following HTML best practices, creates a `label` element to be used for displaying the name of the field to the user.

While this markup generation eases your development, it's not enough to warrant the use of the term *component*. Don't get misled into equating the UI component with a tag that just generates the markup for an HTML element. The `textfield` tag does much more than create the markup shown.

WARNING Don't freak out! Many developers start to twitch when we tell them that the UI component tags will generate the HTML markup for them. If you're getting jittery, just focus on your breathing and hear us out. The Struts 2 UI component tags were carefully designed as a mini-MVC UI component framework of their own. The HTML view of each component is isolated into a single FreeMarker template that can easily be edited. Thus, you can tweak the Struts 2 UI components to generate HTML that meets your own idiosyncratic requirements. Now, rather than being a one-off hack, your quirky HTML will be a quirky component. You will, if you like, be able to share your quirky markup across many pages and projects.

You probably expected that the component tags would generate the markup for the corresponding HTML elements. We'll now move on to the other services provided by the UI components that integrate them with the whole framework and truly set them apart.

BINDING FORM FIELDS TO VALUESTACK PROPERTIES

Creating the HTML elements is just the beginning. Once the HTML elements are in place, the UI components wire these elements into all the rich functionality of the Struts 2 framework. For starters, they bind the form input fields to the properties on the `ValueStack`. This binding lays the foundation for a bidirectional flow of data between UI components and the domain model objects on the `ValueStack`. We've already explored this in some depth, but we'll now reiterate the role that the UI components play in this.

When you're looking at a form generated by Struts 2 UI component tags, the form fields you see are tied to the back-end Java properties in two directions. First, `ValueStack` data can flow into your form for prepopulation, if you desire. Then, when the form is posted, the data from those form fields will flow back into the framework and be automatically transferred onto the `ValueStack`. When we covered actions in chapter 3, we discussed the incoming data at length. We'll now go through an example to show how the full binding of UI components to the `ValueStack` also works in the outgoing mode to power easy prepopulation of your forms.

As we've already learned, the name of a UI component is what binds the component to a `ValueStack` property. The name, as it's interpreted as an OGNL expression, is used to locate a property on the stack. During rendering, a UI component will pull the value from the stack, if it exists, to prepopulate the form. On submission, that same name will be used to locate the target of the framework's automatic data transfer. Let's look at an example of how this works.

In order to show both the form prepopulation and submission phases, we've added an Update Account feature to the Struts 2 Portfolio application. Go to the chapter 7 section of the sample application and log in to an account. From there, choose to update your account details. When you do, you'll be presented with an account update form that's prepopulated with your existing account details, shown in figure 7.1.

Figure 7.1 The account form is prepopulated with data from properties on the `ValueStack`.

As you can see, the form has been prepopulated. Clearly, the UI components have already been bound to Java-side properties. Before we edit anything, let's look at how we've built the actions and JSP result pages to drive this account update example.

First of all, we've broken the process of updating the account into two actions. The first action retrieves the current account data and builds a prepopulated form. The second action, which the form will submit to, will accept the revised account information, validate it, and persist it. The use of two actions is perhaps inelegant, but it serves as a better illustration of the prepopulation and submission phases of the component. In chapter 15, we explore a common technique of combining these phases into a single action that can handle all aspects of data manipulation for a given data object, a.k.a. the Create-Read-Update-Delete (CRUD) action.

The first action, `UpdateAccountForm`, will build a prepopulated form. All this action needs to do is expose the appropriate `User` object. We'll do this by exposing the `User` object as a domain model object, à la `ModelDriven` actions. Listing 7.1 shows the `UpdateAccountForm` action in full.

Listing 7.1 The `UpdateAccountForm` exposes a `User` to prepopulate the form

```
public class UpdateAccountForm extends ActionSupport
  implements UserAware, ModelDriven {                    (1)

  public String execute(){                               (2)
    return SUCCESS;
  }

  private User user;

  public void setUser(User user) {                       (3)
    this.user = user;
  }

  public Object getModel() {                             (4)
    return user;
  }
}
```

This action implements the `UserAware` interface (1), developed in conjunction with the `AuthenticationInterceptor` we built in chapter 4, to receive an injection of the

current `User` object into a setter method ❸. This action is in the secure package that has that interceptor in its stack. This is great, because it saves the work of having to manually retrieve the `User` object that'll be used to prepopulate our form. We also implement the `ModelDriven` interface ❶, because we'll expose our `User` object as a domain model object rather than as a local JavaBeans property. This, in combination with the automatic injection of the `User` as described previously, allows us to use an extremely elegant syntax where we implement our `getModel()` to just return the already-injected `user` ❹.

Since all this action needs to do is retrieve and expose the `User` object for the purpose of form prepopulation, we're done! But we haven't even hit the `execute()` method yet! No problem; if we've made it this far, we can assume success ❷. Any problems would've been intercepted at the levels of data conversion or validation. What's our result?

As you would see if you consulted manning/chapterSeven/chapterSeven.xml, the `SUCCESS` result points to the UpdateAccountForm.jsp page. This page presents a prepopluated and editable form of user account information. This page contains the UI component tags that'll build our account update form. The essential markup from the UpdateAccountForm.jsp page is shown in the following snippet:

```
<s:form action="UpdateAccount">
  <s:textfield name="username" label="Username" readonly="true"/>      ❶
  <s:password name="password" label="Password"/>
  <s:textfield name="portfolioName" label="Enter a name."/>
  <s:submit/>
</s:form>
```

The most important aspect of these tags is the `name` attribute. The `name` attribute is what binds each component to the properties exposed on the `ValueStack`, thus allowing data from the current stack to flow into the form fields during the page rendering prepopulation stage, and also allowing data from this form's eventual submission to flow from that request onto the receiving action's `ValueStack` properties. This is what we mean when we say that a UI component binds the form field to Java-side properties on the `ValueStack`. And this works, ultimately, because the name of an input field is interpreted by the framework as an OGNL expression that ties everything together.

Since our `UpdateAccountForm` action exposes the `User` object via the `ModelDriven` interface's `getModel()` method, our `name` attribute OGNL can be simple, top-level references. For example, we can bind to the username of our model object with a simple `username` ❶. With this binding in place, this `textfield` component will pull the value off of the `ValueStack`'s `username` property, which is actually our user's `username` property, and use that value in the creation of the actual form field. In other words, it'll write that value into the input field's `value` attribute. The rest of the form components are handled similarly. Here's the resulting HTML source:

```
<input type="text" name="username" value="Arty" readonly="readonly"
   id="UpdateAccount_username"/>
```

```
<input type="password" name="password" id="UpdateAccount_password"/>

<input type="hidden" name="__checkbox_receiveJunkMail" value="true" />
```

Voilà, these are our prepopulated form fields. For ease of reading, we've stripped out all of the markup except the form input elements themselves. As you can see, the input elements have had their `value` attribute set with the data from the bound property on the `ValueStack`. Refer back to figure 7.1 to see how this looks on the page. As you can see, UI components, in addition to just generating the HTML element, make easy work of form prepopulation.

HEADS-UP Though we've made it sound like the `name` attribute is responsible for the prepopulation, it's actually the `value` attribute at work. We've glossed over this a bit thanks to the intelligent default behavior of the framework. In truth, the UI components expose a `value` attribute, which takes an OGNL expression pointing to a Java property that should be used to fill in the value of the form field during prepopulation. You can do it this way if you like. However, since you'll almost always prepopulate from a property with the same name as the property that'll be the target of the posted data, the framework will simply propagate the `name` attribute's OGNL over to the `value` attribute. If you ever need to prepopulate from a different property than you'll submit to, feel free to set the `value` attribute independently from the `name` attribute.

The other side of the UI component data binding is the incoming request parameters from the submitted form. We've spent a lot of time on that already, so we won't rehash how the automatic data transfer works. We'll summarize this section by noting that the UI component does indeed bind itself, via the `name` attribute as an OGNL expression, to both outgoing and incoming properties on the `ValueStack` to achieve both prepopulation of forms and automatic data transfer when a form is posted.

In addition to providing built-in binding to `ValueStack` properties, UI components also provide integration with several framework features, including type conversion, validation, and internationalization.

INTEGRATION WITH TYPE CONVERSION, VALIDATION, AND INTERNATIONALIZATION

We've already seen how the UI components can bind a form field to a property on the `ValueStack` to make the data flow effortlessly from the user interface to the back-end Java code. There are several other functions encapsulated in the UI components. In this section, we'll discuss how the components tie into the framework's type conversion, validation, and internationalization mechanisms.

For type conversion and validation, the UI components automatically handle error messages when there's a problem. As you've seen, problems with type conversion or validation cause the request to return the user back to the input form. When this happens, the UI components will automatically detect the presence of errors associated with themselves, and display error messages accordingly. You can customize these error messages and even tie them into the internationalization features to provide

localized error messages to your users. For a full discussion of these functionalities, please consult chapters 5 and 10.

As we've indicated, UI components can tap into the internationalization mechanism to provide localized error messages. They can also tap into the internationalization to provide localized label names for your UI components. This feature uses the `key` attribute provided by all UI components. When using the `key` attribute, you can simplify your form field markup to a high level. You can set just this attribute and it'll pull a localized label from a framework `ResourceBundle` and intelligently deduce the `name` and `value` attributes to fully complete the component's bindings. This is fully explored in chapter 11.

Now that we've convinced you that the Struts 2 UI components provide much more than just an HTML element on the page, let's learn how to use them. The first thing we need to do is explain that mini-MVC architecture in a bit more depth.

7.2 Tags, templates, and themes

We've covered the functional aspects of the UI components. Now we need to say something about their unique architecture. You might be thinking that an architecture for tags is a bit much. We understand that response. But once you see what the tag architecture provides, we think you'll fully appreciate the effort. The most important benefit of the mini-MVC architecture of the UI components is reusable customization.

Let's start with a high-level introduction. Figure 7.2 shows the architecture of the UI component framework.

The UI components are built on a layered architecture. As a developer, you need to be clear about three things: *tags*, *templates*, and *themes*. At the top of figure 7.2, the component API is exposed as a JSP tag in a page. This tag is first processed in its native environment—the JSP tag is processed as a JSP tag. As we learned in the previous chapter,

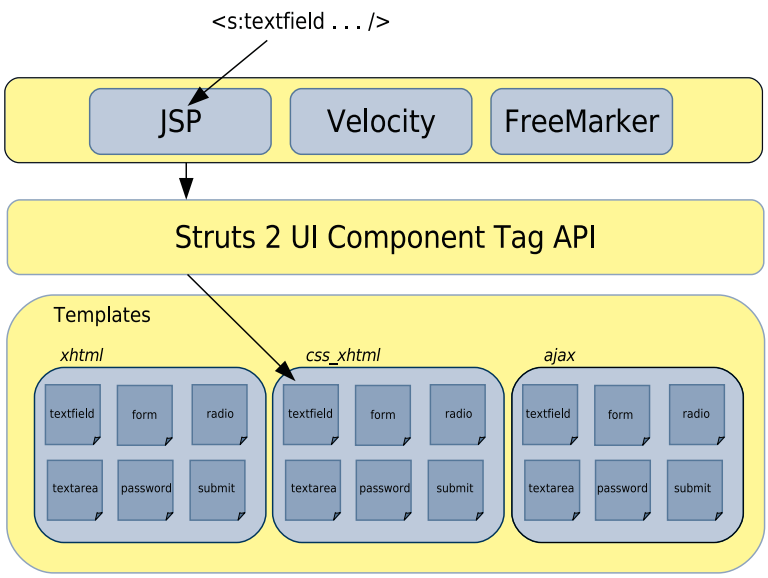

Figure 7.2 Tags, templates, and themes work together to provide feature-rich, flexible, extensible UI components.

the tag API can be utilized from several view-layer technologies. The native tag, such as a JSP Struts 2 `textfield` tag, is just a wrapper around the framework's corresponding UI component API, such as the `org.apache.struts2.components.TextField` component class. As you can see in figure 7.2, the native tag quickly hands control over to this component API for actual processing. The UI component layer processes the logic of the tag and prepares the data. It then hands off the rendering task to the underlying FreeMarker templates.

WARNING Don't be confused by the fact that the UI components use FreeMarker templates for their own rendering. This is an internal detail of the mini-MVC component framework itself; it has no bearing on your choice of JSP, Velocity, or FreeMarker for your view-layer pages. The internal FreeMarker templates are used to generate the output of the tag itself, which'll show up in whatever kind of page you put the tag. If you want to customize the tag output, you'll have to work with the underlying FreeMarker templates. But your view layer isn't bound to FreeMarker just because the framework uses it internally. Incidentally, we'll show you how to customize those templates in chapter 13.

In this way, the markup output of the tag API is layered away from the Java classes that conduct the logic; thus we have a mini-MVC. As with all MVCs, the main benefit is the ability to easily change the "view" of the tag without needing to touch the business logic embedded in the tag's component class. We'll show you how to customize the view of the tags yourself in chapter 13. For now, we'll focus on explaining how to work with the several view options that are bundled with the framework. Several *themes* of templates are available for rendering the tags. The `css_xhtml` theme, for instance, renders the tags with pure CSS-based layout markup.

All of this will be explained in the following sections, which go into more detail about the layers shown in figure 7.2: tags, templates, and themes.

7.2.1 Tags

Tags are quite simple. As we explained in chapter 6, the Struts 2 tag API is high-level API. In other words, the functionality and use of the tags is defined outside the details of a given view-layer technology. The tag API could be implemented in any technology. By default, the framework provides implementations of the tag API for JSP, Velocity, and FreeMarker. In chapter 6, we explained how to use the tags with each of these technologies. Regardless of which you use, the native tag will delegate the processing to the UI component framework. The framework manages the logic and data model for the tag, processing the logic associated with the tag as well as collecting the relative data. The data might come from tag attributes or parameters, as well as from the `ActionContext` and `ValueStack`. This data is all made available to the template that'll render the HTML view of the UI component. The framework uses FreeMarker templates for rendering the UI components.

7.2.2 Templates

As we've said, every tag has a FreeMarker template that'll render its markup. As with most templating technologies, a FreeMarker template looks like a normal text file. Inside the plain text, special FreeMarker directives are included that can dynamically pull in data and render the resulting output. In the case of the UI component tags, a template takes the data model, as collected by the tag logic, and blends that data with the static parts of the template to create the final markup that'll be in your HTML page. Note that regardless of whether the tag was invoked from a JSP or a Velocity template, the same template will receive the same set of parameters and render the resulting markup in exactly the same way.

If you never customize or create your own templates, you don't need to concern yourself with these details in any greater detail than this. However, we think that customizing the tags will appeal to many users. If you want to customize the templates, they are readily accessible in the Struts 2 JAR, or you can override them by inserting your own in the classpath. In chapter 13, we'll provide a demonstration of some techniques for customizing these templates. If you never customize them, you should be fine as long as you understand that the UI components use FreeMarker templates to handle their rendering.

7.2.3 Themes

It's easy to understand that an underlying template generates the markup for a given tag. However, it's more complicated than that. In fact, each tag has several versions of the underlying template at its disposal. Each of these versions belongs to a different theme. A theme, as shown in figure 7.2, is a group of templates, one for each tag more or less, that'll render the tags in some consistent fashion or manner. For instance, one theme renders the tags to work with HTML table layout, while another renders them to work with CSS. Several themes are bundled with the framework, and you can choose which one you want your tags to use.

There are several ways to specify the theme. By default, all tags will render under the `xhtml` theme. You can change this default theme for the whole application, or you can specify a different theme on a per-page or per-tag basis. In the rendering scenario of figure 7.2, we can see that the `css_xhtml` theme, which uses pure CSS to lay out the elements of a form, has been selected. Right now, Struts 2 comes with four themes for rendering your UI components: `simple`, `xhtml`, `css_xhtml`, and `ajax`. Table 7.1 summarizes the characteristics of these themes.

Table 7.1 The built-in UI component themes

Theme	Description
`simple`	Renders the basic HTML element.
`xhtml`	Renders the UI component using a table to provide the layout.

Table 7.1 The built-in UI component themes *(continued)*

Theme	Description
`css_xhtml`	Renders the UI component using pure CSS to provide the layout.
`ajax`	Extends `xhtml` and provides rich Ajax components. This theme is not quite finished as of this writing… but is a good starting point for building Ajax applications nonetheless. Check the Struts 2 website for more details.

These themes are straightforward. The `simple` theme is rarely used on its own. The `simple` theme does little more than build the basic HTML element itself. While this isn't too useful on its own, it provides a good core for creating more complicated themes. In fact, the other themes that come with Struts 2 all take advantage of this and use the `simple` theme to render the basic HTML element at the core of their more complex markup. They do this by extending the `simple` theme. Extending a theme is a technique you can use to create new themes; again, we'll explore this topic in chapter 13. While the `simple` theme may rarely be used on its own, the other three themes presented in table 7.1 all provide full-featured UI components that integrate fully with the Struts 2 framework.

CHANGING THE THEME

Before moving on to discuss the use of the UI components themselves, we should show how to change the theme under which they render. By default, the components will render under the `xhtml` theme, which uses HTML tables for layout of the forms. Many developers are moving away from the use of HTML tables for layout. If you want to do this, you'll probably want to change the default rendering theme to `css_xhtml`. The default theme is changed by overriding the `default` property. All of the framework's default properties are defined in `default.properties`, found in the Struts 2 core JAR file at `org.apache.struts2`. To override any of the framework's default properties, you only need to create your own properties file, named struts.properties, and place it in the classpath. To change the default theme, just create a struts.properties file with the following property definition:

```
struts.ui.theme=css_xhtml
```

Then just make sure it's on the classpath. Most people put it in the web application at WEB-INF/classes/struts.properties. With this in place, all UI components will render under the `css_xhtml` theme. Now your application will use CSS and `div`s instead of HTML tables for its layout.

You can also change the theme on a more fine-grained level by specifying the `theme` attribute of the components themselves. You can, for instance, specify that a certain `textfield` should be rendered under a certain theme. Or you can specify the theme for a given `form` component, thus causing all components in that form to render with that theme.

That's about all there is to setting themes. And that finishes up our general introduction to tag usage; it's time for specifics. We understand that the UI components can

seem complex at first glance. Admittedly, they're more complex than the tags of previous frameworks, but, on the other hand, they do a whole lot more. There's more to learn, but it's worth it. In particular, keep the mini-MVC in mind. When it comes time to customize your tag output to meet some rather particular requirements, we think you'll find the added sophistication pays off in triple. Now let's meet the UI components and start demonstrating their use with the Struts 2 Portfolio. Any lingering confusion will dissolve with a bit of practical experience.

7.3 UI Component tag reference

Now that we've done a high-level overview of the UI component architecture, it's time to learn to use the UI component tags themselves. In this section, we provide both a reference and a demonstration of the most common UI component tags. For each tag, we'll provide a description and a list of attributes and parameters, and demonstrate usage.

REFERENCE USER TIP

The components in this reference are covered in order of most commonly used to more, shall we say, specialized. Since usage of the components follows certain patterns, our coverage of the first tags in the reference includes much more detailed explanation of these usage patterns. We recommend reading everything through section 7.3.4 in order to learn how the tags work in general. Almost every developer will need all of these tags anyway; we're not wasting your time here. After that, you don't need to read unless you see a tag you want to use.

We'll start by summarizing the attributes and usage common to all of the UI components.

7.3.1 Common attributes

The list of attributes common to all the Struts 2 UI component tags is large. This is mostly a reflection of the numerous attributes exposed by the underlying HTML elements. If you're like me, a huge table of attributes can be a bit of a brain freeze. We'll focus on the core usage of the Struts 2 tags themselves, and leave the details of various HTML- and JavaScript-related attributes, such as event handlers, to you. While the following tables are long, they aren't exhaustive. If you want the definitive list of all attributes supported by the tags, visit the Struts 2 website. In general, you can assume that all attributes of the underlying HTML elements are supported in at least a pass-through manner.

When browsing the following tables, a few things should be kept in mind. First of all, we need to recall what we learned in the previous chapter about attribute data types and the use of OGNL expressions in attribute values. If the data type is `String`, then the attribute value will be interpreted as a string literal. This means that it won't be evaluated as an OGNL expression unless you force that evaluation with the `%{ expression }` notation. On the other hand, all non-`String` data types will be automatically evaluated as OGNL expressions. Generally, this means that you won't have to use the formal OGNL expression brackets often because you'll typically feed literal strings to the `String` attributes and OGNL expressions to the non-`String` data types.

Table 7.2 shows the common attributes of the UI components.

All of the components that we'll cover in this chapter support the attributes listed in table 7.2. We'll cover the usage of these attributes in more detail as we work our way through the components in the next sections. Most components also use a few specialized attributes. These will be presented with the component itself.

Table 7.2 Common attributes for all UI tags

Attribute	Theme	Data type	Description
`name`	`simple`	`String`	Sets `name` attribute of the `form` input element. Also propagates to the `value` attribute of the component, if that attribute isn't set manually. The name itself is used by the component to target a property on the `ValueStack` as destination for the posted request parameter value.
`value`	`simple`	`Object`	OGNL expression pointing to `ValueStack` property used to set the value of the form input element for pre-population. Defaults to the `name` attribute.
`key`	`simple`	`String`	Pulls localized label from `ResourceBundle`, and can propagate to `name` attribute, and thus to `value` attribute. See chapter 11.
`label`	`XHTML`	`String`	Creates an HTML label for the component. Not needed if setting using the `key` attribute and localized text.
`labelPosition`	`XHTML`	`String`	Location of the element label: left or top.
`required`	`XHTML`	`Boolean`	If `true`, an asterisk appears next to the label indicating the field is required. By default, the value is `true` if a field-level validator is mapped to the field indicated in the `name` attribute.
`id`	`simple`	`String`	HTML `id` attribute. Components will create a unique ID if one isn't specified. IDs are useful for both JavaScript and CSS.
`cssClass`	`simple`	`String`	HTML `class` attribute, for CSS.
`cssStyle`	`simple`	`String`	HTML `style` attribute, for CSS.
`disabled`	`simple`	`Boolean`	HTML `disabled` attribute.
`tabindex`	`simple`	`String`	HTML `tabindex` attribute.
`theme`	N/A	`String`	Theme under which component should be rendered, such as `xhtml`, `css_xhtml`, `ajax`, `simple`. Default value is `xhtml`, set in default.properties.
`templateDir`	N/A	`String`	Used to override the default directory name from which templates will be retrieved.
`template`	N/A	`String`	Template to look up to render the UI tag. All UI tags have a default template (except the component tag), but the template can be overridden.

In addition to these attributes, the components also support the common JavaScript event handler attributes, such as `onclick` and `onchange`. Basically, the components support any HTML attribute you'll want to set. In common tag usage, you'll typically only use a few of these attributes, such as `name`, `key`, `label`, and `value`. Rather than trying to explain these attributes in a vacuum, we'll explain them in the context of actual UI component tag examples.

7.3.2 Simple components

In this section, we'll introduce the most commonly used UI components. We'll start with the infrastructural components, including the important `form` component, which acts as a container for all the other components. With the preliminaries out of the way, we'll meet many of the simpler components such as `textfield`, `password`, and `checkbox`. After that, we'll hit the collection-backed components.

READERS' COURTESY

The organization of this reference section has been designed to require the least amount of reading. Basically, we'll cover all the stuff you'll use first. We'll also use our coverage of these essentials, such as text fields and select boxes, to demonstrate the fundamental patterns of usage common to all components. This structure means you can safely stop reading as soon as you have what you need. Later sections will point you toward richer components, but if you're not interested you won't need to keep reading.

THE HEAD COMPONENT

We will start with the `head` component. This tag doesn't do anything by itself, but it plays an important role in supporting the other tags. The `head` tag must be placed within the HTML `head` element, where it generates information generally found in that location. This information includes HTML link elements that can reference CSS stylesheets, as well as script elements that can define JavaScript functions or reference files of such functions. Since many of the UI component tags come with rich functionality, this `head` tag can link to commonly used JavaScript libraries that help implement that functionality.

Note that if a tag depends upon the resources pulled in by the `head` tag, it'll appear to not work if you omit the `head` tag. This is a common source of "bugs." If you're using the `xhtml` theme, this tag also loads a default CSS stylesheet that defines some basic styles for the form elements rendered under the `xhtml` theme. If your requirements aren't that rigid, this basic styling may be all you need. As you can see from the following snippet from one of our Struts 2 Portfolio application's JSP pages, adding this tag to your page is easy:

```
<head>
  <title>Portfolio Registration</title>
  <s:head/>
</head>
```

No attributes are required. This tag can discover all of the information it needs. What does it create? Here's the markup, assuming it renders under the default `xhtml`

theme. Note that we've abbreviated this a bit. We just want to show you that the `head` tag generates links to stylesheets and JavaScript libraries:

```
<link rel="stylesheet" href=" . . . styles.css" type="text/css"/>

<script language="JavaScript" type="text/javascript" src=". . .dojo.js"/>

<script language="JavaScript" type="text/javascript" src="dojoRequire.js"/>
```

Obviously, this tag doesn't do much by itself. It's more of a helper tag that lays the foundation for other, more concretely productive tags that'll come later in the page. When we cover tags that depend on its presence, we'll explicitly indicate their reliance on the `head` tag.

THE FORM COMPONENT

The `form` component is probably the most important of all. This critical UI component provides the central tie-in point to your Struts 2 application. It's the form, after all, that targets your Struts 2 actions. In addition to the common attributes defined at the beginning of this section, the `form` component also uses the attributes summarized in table 7.3.

Table 7.3 provides concise descriptions of the attributes. In practice, it's easy. Let's jump right in by looking at an example. The following snippet shows the form from chapter 7's Login.jsp page:

```
<s:form action="Login">
  <s:textfield name="username" label="Username"/>
  <s:password name="password" label="Password"/>
  <s:submit/>
</s:form>
```

The markup up is simple. The `action` attribute is the most important. We simply specify the action name, sans .action extension, to which we want to submit the form. The name given here is the logical name given to the action in the declarative architecture, in our chapterSeven.xml file in this case. Note that if we specify the action attribute without specifying the `namespace` attribute, it'll assume the current namespace. Since we frequently target actions within the same namespace as the current request, this

Table 7.3 Frequently used `form` tag attributes

Attribute	Data type	Description
`action`	`String`	Target of form submission—can be name of Struts 2 action or a URL.
`namespace`	`String`	Struts 2 namespace under which to search for named action (above), or from which to build the URL. Defaults to current namespace.
`method`	`String`	Same as HTML form attribute. Defaults to POST.
`target`	`String`	Same as HTML form attribute.
`enctype`	`String`	Set to `multipart/form-data` if doing file uploads.
`validate`	`Boolean`	Turns on client-side JavaScript validation, works with Validation Framework.

allows for clean markup. Indeed, we've specified only one attribute. For everything else, we let the framework provide intelligent defaults.

Listing 7.2 shows the output generated by this `form` tag.

Listing 7.2 HTML Output from a form UI component

```
<form id="Login" name="Login" onsubmit="return true;"
  action="/manningSampleApp/chapterSeven/Login.action" method="POST">

  <label for="Login_username" class="label">Username:</label>
  <input type="text" name="username" value="" id="Login_username"/>

  <label for="Login_password" class="label">Password:</label></td>
  <input type="password" name="password" id="Login_password"/>

  <input type="submit" id="Login_0" value="Submit"/>

</form>
```

Remember when we said that the UI components generate layout-related markup? We've suppressed that from this listing; we don't want to confuse things while we're trying to understand the functional parts of the tag. Functionally, the most important attribute is the `action` attribute. In this case, the `action` attribute has become a fully qualified path even though our tag only specified a logical action name. Struts 2 lets us specify a simple logical name for our `action` attribute, and it then generates the full URL for that action.

TIP When your form targets another Struts 2 action, you only need to specify the logical name of the action. You don't need to add the .action extension. And if it's in the same namespace as the current action—the one whose result is rendering the page—you don't even need to specify the namespace. Intelligent defaults like this allow for cleaner, faster coding.

The `form` component tag also creates sensible values for several other attributes, notably the `id` and `method` attributes. The generated ID is unique and built on the name of the action itself. Interior fields of the form will also build upon this naming convention so that you can count on these IDs when applying relevant JavaScript techniques.

NOTE Most of the Struts 2 UI components will automatically generate IDs and names for components even if you don't specify those attributes yourself. This important step lays the foundation for JavaScript and CSS functionality that depends upon being able to specify elements in the HTML DOM by their unique ID. If you don't need these, it's nonintrusive. However, if you ever find yourself needing to go back and use IDs, you can rest assured that all of your pages have been prepared for just such an occasion.

In this example, we followed the most common use case of aiming our form submission at another Struts 2 action. The tag's default behavior supports this, but, as always, the framework is flexible and allows you to easily target any web resource. To be complete, we'll outline the process by which the form tag generates the final action URL for the HTML form. The following list shows the steps followed when determining how to create the URL:

1. If no `action` attribute is specified, the current Struts 2 action is targeted again.
2. If the `action` attribute is specified, it's first interpreted as the name of a Struts 2 action. If no `namespace` attribute is specified, then the action is resolved against the namespace of the current request. If a `namespace` attribute is specified, the action will be searched for in that namespace. Note that actions are specified without the .action extension.
3. If the value set in the `action` attribute doesn't resolve to a Struts 2 action declared in your declarative architecture, then it'll be used to build a URL directly. If the string begins with a slash (`/`), then this is assumed to be relative to the `ServletContext` and a URL is made by appending this to the `ServletContext` path. If the value doesn't start with a slash, then the value is used directly as the URL. Note that the `namespace` attribute, even if specified, is not used in these cases.

The first option is relatively self-explanatory. A URL is generated that'll submit to the same action again. The second option, which involves targeting named Struts 2 actions, is also simple. Mostly you'll do as we did in the example; you'll target an action in the same namespace as the current action. You can also specify an alternate namespace under which the framework should search for the named action. The following snippet shows how to make our chapter 7 version of the login form submit to the chapter 4 `Login` action:

```
<s:form action="Login" namespace="/chapterFour">
```

This tag will generate the following HTML code, building a URL for the `action` attribute that targets the `Login` action in the `chapterFour` namespace. Here's the output:

```
<form id="Login" name="Login" onsubmit="return true;"
action="/manningSampleApp/chapterFour/Login.action" >
```

As you can see, this'll hit the `Login` action in the `chapterFour` namespace. Not exactly what we want, but it serves to demonstrate the syntax for specifying a different namespace.

If the value of the `action` attribute doesn't map to an action name, then the value will be used directly to build a URL. Typically, the process shouldn't get to option three unless you're intentionally specifying a URL rather than an action name. If you want to specify a URL, you have a couple of choices. First, you can specify a path that starts with a slash, such as

```
<s:form action="/chapterSeven/PortfolioHomePage.jsp">
```

Note that we have to specify the .jsp extension. This is because we're specifying an actual URL here. In the previous cases, we were naming an action by its logical name and letting the framework generate the URL for us. The following snippet shows how the framework uses the previous tag to build a URL by appending the action value directly onto the `ServletContext` path:

```
<form id="PortfolioHomePage" onsubmit="return true;"
action="/manningSampleApp/chapterSeven/PortfolioHomePage.jsp">
```

If you want to specify a full URL yourself, to target an external resource, just do so. The framework will know what you're doing because the value doesn't resolve to an action and it doesn't start with a slash. So we can specify a full URL as follows:

```
<s:form action="http://www.google.com">
```

This `form` tag will generate a form that submits to Google. Again, probably not what you want, but this demonstrates how to specify a full URL in case you need to link to target an external resource.

Finally, if you specify a value that doesn't start with a leading slash, doesn't resolve to a action, and doesn't represent a full URL, then the value will be printed as-is into the HTML action attribute. For instance:

```
<s:form action="MyResource">
```

This tag will generate a form like the following:

```
<form id="MyResource" onsubmit="return true;" action="MyResource">
```

When a browser sees a URL like this, it'll interpret it as relative to the URL of the current page. If this form is in our chapterSeven/LoginForm.action page, the browser will start from `chapterSeven` and build a URL like this: http://localhost:8080/manningSampleApp/chapterSeven/MyResource

If you do this, be sure that the resource exists. Typically, relative paths are used to hit static resources, such as images, rather than the application server's dynamic resources. It's unlikely that your form will want to target a static resource. And best practices warn against targeting a dynamic resource, such as a JSP, directly. Accepted best practice is to use a pass-through action to target JSPs and other dynamic resources.

Now that you know how the form sets up the HTML `form` element itself, we should take a look at the layout-related markup that's also generated by the component. As we've indicated, the UI component tags generate additional markup to handle layout. By default, the components render under the `xhmtl` theme. This theme generates an HTML table to format the form elements. Listing 7.3 shows the full markup generated by the `form` tag in the example, rendered under the default `xhtml` theme.

Listing 7.3 The full markup generated by the form component

```
<form id="Login" name="Login" onsubmit="return true;"
  action="/manningSampleApp/chapterSeven/Login.action" method="POST">

  <table class="wwFormTable">
    <tr>
      <td class="tdLabel">
        <label for="Login_username"class="label">Username:</label>
      </td>
      <td>
        <input type="text" name="username" value=""
           id="Login_username"/>
      </td>
    </tr>

    <tr>
      <td class="tdLabel">
```

(1)

(2)

```
        <label for="Login_password" class="label">Password:</label>
      </td>
      <td>
        <input type="password" name="password" id="Login_password"/>
      </td>
    </tr>

    <tr>
      <td colspan="2">
        <div align="right">
          <input type="submit" id="Login_0" value="Submit"/>
        </div>
      </td>
    </tr>
  </table>
</form>
```

We don't need to examine this too much. After all, the whole point of the component is to keep you from having to think about this kind of stuff. We just wanted to show it once so that you know what to expect. The main bit here is that the `xhtml` theme generates an HTML table that handles the layout of the form and the form elements. The `username` text field, for instance, is situated in a single row of the table ❶. You should also note that the elements have `class` attributes defined for CSS tie-in ❷.

Admittedly, this is a big wad of HTML goo. But remember that this markup is all generated from the underlying FreeMarker templates. This means that you can easily alter the template if you need to make some changes to the markup. Again, we'll learn how to modify these templates in chapter 13.

Now that we've seen how to deploy the `form` component, we need to add some input fields to that form.

THE TEXTFIELD COMPONENT

This is one tag you can't avoid using. This component generates the ubiquitous text input field. And, as usual, the most common things are easy in Struts 2. The main thing you need to understand about the `textfield` tag is how the `name` attribute and the `value` attribute tie into the framework. If you don't recall how this works, refer back to our explanation in section 7.1.1. If you're clear on how the `name` and `value` attributes function as OGNL expressions that bind the component to a property on the `ValueStack`, you're good to go.

In addition to the common UI component attributes summarized previously, `textfield` makes frequent use of a few more attributes of its own. Table 7.4 summarizes these attributes.

Table 7.4 Important `textfield` attributes

Attribute	Data type	Description
`maxlength`	`String`	Maximum length for field data.
`readonly`	`Boolean`	If `true`, field is uneditable.
`size`	`String`	Visible size of the `textfield`.

Now, let's a look at an example from the Struts 2 Portfolio application. The following snippet shows the login form from the chapter 7 sample code, Login.jsp:

```
<s:form action="Login">
  <s:textfield key="username"/>
  <s:password name="password" label="Password"/>
  <s:submit/>
</s:form>
```

Since we've already demonstrated, in section 7.1.1, the use of the `name` and `value` attributes to configure a `textfield` component, we'll now show how it works with the `key` attribute. The `key` attribute can be used to pull a localized label value from a `ResoureBundle`; the global-resources.properties file found in the root of the classpath in this case. When you do this, you don't have to set the `label` attribute manually. But, if you're using the `key` attribute, you might as well let the framework handle a couple of other things for you as well.

As we've seen, you can omit setting the `value` attribute by letting the framework use the `name` value as an OGNL expression that points to the property that'll provide the value for the `value` attribute. The framework is inferring the `value` from the `name`. The `key` attribute ups the ante. In addition to allowing you to pull a localized message in for your label, the framework can infer the `name` attribute from the `key` attribute, thereby inferring the `value` attribute as well. The end result, as seen in the previous snippet, is a clean tag that specifies only the `key` attribute.

CONVENTIONAL WISDOM

When the framework does work for you, as when using the `key` attribute to infer the other attributes of a UI component, the magic is all about convention. In general, Struts 2 can save you loads of time if you can follow conventions that it understands. In this case, the framework expects that the value you give for the `key` attribute can locate a localized message in a `ResourceBundle` *and* locate a property on the `ValueStack`. You must follow this synergy when creating your `ResourceBundle` properties files and naming the properties of your data model. If you follow this convention, the framework will do the work for you. If you scoff at convention, no tool is smart enough to divine your logic.

Here's the markup generated by the `textfield` portion of this JSP snippet:

```
<tr>
  <td class="tdLabel">
    <label for="Login_username" class="label">Username:</label>
  </td>
  <td>
    <input type="text" name="username" value="" id="Login_username"/>
  </td>
</tr>
```

Since this is the first form field we've discussed, we decided to show you the layout-related markup generated by the tag as well. We do this just so you see that, under the `xhtml` theme, each input field tag generates its own row and table data markup to

position itself within the table created by the `form` tag. This is true for all UI components that generate a form field under the `xhtml` theme. As you might expect, if you use the `css_xhtml` theme, a CSS-based version of layout markup will be generated instead.

In addition to the table markup, the most striking thing about this example is how specifying a single attribute, `key`, was able to do so much. In addition to getting a localized message for our label, the `key` created the input field's `name` attribute, which of course is an OGNL expression binding the field to a property on the `ValueStack`. Through this binding, the `value` might also be set, but not in this case. In this case, our `value` attribute is empty because the matching property on the `ValueStack` was empty, or nonexistent, when this form rendered; we're not prepopulating the login form.

Using the `key` property is elegant, but you need to be using `ResourceBundle`s to provide the text messages for the label. When the component renders, it'll attempt to find a text message using the `key` attribute value for the `ResourceBundle` lookup key. This is a highly recommended practice, as it allows you to externalize your text messages in a manageable location. For the Struts 2 Portfolio application, we've followed a common web application practice of putting our text resources in properties files.

The framework makes it particularly easy to use properties files. Two convenient options are to externalize your messages in a global properties file or in individual properties files local to a specific class. In this case, we've added a global properties file to the application. This file, global-messages.properties, resides in the classpath at WEB-INF/classes/, and contains the following property:

```
username=Username
```

Again, note that the key of this property matches the OGNL expression that we use to target the `username` property on the `ValueStack`. This works great because the same hierarchical namespace works well in both places. In this example, our key/OGNL is simple, but deeper expressions will work fine.

In order to use global properties files, you need to tell Struts 2 where to find them. As always, such configuration details can be controlled through system properties in the struts.properties file. This file, which the user must create, goes on the classpath, typically at /WEB-INF/classes/struts.properties. Here's the property we set to specify a properties file that should be picked up by the framework:

```
struts.custom.i18n.resources=global-messages
```

The framework's built-in support for managing localized text in `ResourceBundle`s goes quite a bit further than this. For a complete discussion, see chapter 11.

THE PASSWORD COMPONENT

The `password` tag is essentially like the `textfield` tag, but in this case the input value is masked. As with all input fields, the `name`, `value`, and `key` attributes are the most important. Table 7.5 summarizes the additional attributes frequently used with the `password` tag.

Table 7.5 Important `password` attributes

Attribute	Data type	Description
`maxlength`	`String`	Maximum length for field data.
`readonly`	`Boolean`	If `true`, field is uneditable.
`size`	`String`	Visible size of the text field.
`showPassword`	`Boolean`	If set to `true`, the password will be prepopulated from the `ValueStack` if the corresponding property has a value. Defaults to false. Populated value will still be masked in proper password fashion.

While there are no surprises, we'll look at a sample just to make it real. The following snippet, again from this chapter's Login.jsp page, defines a password field:

```
<s:form action="Login">
  <s:textfield key="username"/>
  <s:password name="password" label="Password"/>
  <s:submit/>
</s:form>
```

In this case, we specify the `name` attribute and the `label` attribute. (You could use the `key` attribute if you have set up some `ResourceBundles`.) Again, the `name` value is understood as an OGNL expression that binds the component to a specific property on the `ValueStack`:

```
<label for="Login_password" class="label">Password:</label>
<input type="password" name="password" value="" id="Login_password"/>
```

As you can see, the output is just like `textfield`. The `password` tag generates a label and a password field. Note that, in this example, the tag is actually rendered under the `xhtml` theme, but we've stripped out the table markup to clarify what's going on with the HTML elements.

THE TEXTAREA COMPONENT

The `textarea` tag generates a component built around the HTML `textarea` element. As with all form fields, the `name`, `value`, and `key` attributes are the most important. Table 7.6 summarizes the additional attributes frequently used with the `textarea` tag.

Table 7.6 Important `textarea` attributes

Attribute	Data type	Description
`cols`	`Integer`	Number of columns.
`rows`	`Integer`	Number of rows.
`readonly`	`Boolean`	If `true`, field is uneditable.
`wrap`	`String`	Specifies whether the content in the `textarea` should wrap.

From the development point of view, there's no difference between this and the `textfield`. Because of this, we'll spare you the example.

THE CHECKBOX COMPONENT

The `checkbox` component uses a single HTML checkbox to create a Boolean component. Take heed, this component isn't equivalent to an HTML checkbox. It's specialized for Boolean values only. The property you bind it to on the Java side should be a Boolean property. Don't worry; there's another component, the `checkboxlist`, that solves the other checkbox use case—a list of checkboxes, all with the same name, that allow the user to submit multiple values under that single name. The `checkbox` component is focused on a true or false choice. We'll see the `checkboxlist` component in a few pages when we cover the collection-backed components.

In addition to the commonly used attributes we've already defined, the `checkbox` component also uses the attributes defined in table 7.7.

Table 7.7 Important `checkbox` attributes

Attribute	Data type	Description
`fieldValue`	`String`	The actual value that'll be submitted by the `checkbox`. May be `true` or `false`; `true` by default.
`value`	`String`	In combination with `fieldValue`, determines whether the `checkbox` will be checked. If the `fieldValue = true`, and the `value = true`, then the box will be checked.

To demonstrate the usage of the `checkbox`, we've modified our Struts 2 Portfolio data model to include a Boolean value. We'll now track whether a user wants to receive junk mail. We've added a Boolean field to the `User` object and we'll give the user an opportunity to express their junk mail preference during registration. The `Registration` action is where the `User` object is first created and persisted. Here's the form from Registration.jsp that collects the account information, including the new junk mail preference Boolean value.

```
<s:form action="Register">
  <s:textfield name="username" label="Username"/>
  <s:password name="password" label="Password"/>
  <s:textfield name="portfolioName" label="Enter a portfolio name"/>
  <s:checkbox name="receiveJunkMail" fieldValue="true" label="Check to
    receive junk mail"/>
  <s:submit/>
</s:form>
```

The `checkbox` component is deceptively easy to define. In fact, we didn't even need to set the `fieldValue` because `true` is the default value. We've done this just for clarity. This means that the checkbox, if checked, will submit a true value to the framework. The following code shows the actual HTML `checkbox` element output by this component:

```
<input type="checkbox" name="receiveJunkMail" value="true"
  id="Register_receiveJunkMail"/>
```

As we specified in the `fieldValue` attribute, the `value` attribute of the checkbox is `true`. To receive this Boolean value, we implemented a JavaBeans property on our `Register` action. This property is shown in the following snippet:

```
private boolean receiveJunkMail;

public boolean isReceiveJunkMail() {
  return receiveJunkMail;
}
public void setReceiveJunkMail(boolean receiveJunkMail) {
  this.receiveJunkMail = receiveJunkMail;
}
```

We haven't been showing the Java code for most of the UI components. We only do so here to drive home the point that the `checkbox` component works with Boolean values. If the junk mail checkbox is checked, then its `true` value will be submitted to this Boolean property, in concordance with the OGNL of the `name` attribute. Fairly simple. But what about prepopulation?

Prepopulation is usually accomplished with the `value` attribute of the UI component. It's more complex with the `checkbox`. We now have to specify the value that the field will input as well as the current value of the Java-side property. These aren't equivalent values in the context of the `checkbox`, as they would be with a `textfield`. Reflecting the independence of these two values, the `checkbox` component exposes two attributes, `value` and `fieldValue`. The `fieldValue` attribute determines the actual `value` attribute of the HTML element—the value that'll be submitted if the box is checked. Meanwhile, the `value` attribute, as with the other UI components, points to the actual Java-side property to which the component is bound, a Boolean in this case. The `checkbox` component will handle the translation of the Java-side Boolean into the semantics of whether the `checkbox` should be checked. In our simple example, our `checkbox` `fieldValue` is true. Thus, if our Java-side property, pointed to by the `value` attribute, is false, then the box should be unchecked. If our Java-side property is true, then the box should be checked. Of course, in all of this, recall that our `name` attribute will be reused for the `value` attribute if we don't set it manually.

All of this matters most when prepopulating the form. The registration form is not prepopulated because there is no existing user data at that point. If you want to see how the `checkbox` prepopulates, check out the account update process available when you log into an existing account. We won't detail this update action here, as the prepopulation process was covered earlier in the chapter.

We've now covered the simple UI components. The next set of UI components are those that present a set of options to the user. These options are typically backed by collections of data on the Java side. We'll explore these collection-backed components in the next section.

7.3.3 Collection-backed components

This section will introduce a set of components that allow a user to select a choice from a set of options. In some ways, this is a simple and familiar task to most web

developers. One of the most common scenarios involves a user being presented with a list of states or countries. The user selects one of the options and that value is sent under the name of the select box itself.

As this plays out in a Java web application, we typically have a Java-side property that is some sort of collection of data. This might be an array, a `Map`, or a `List`. A full complement of types is supported by the Struts 2 tags. The basic logic of the collection-backed components is that the Java-side data set is presented to the user as a set of options. The user then selects one of the options, such as `Colorado`, and that value is submitted with the request.

Some of the complications encountered with these components involve whether the Java-side data set consists of simple types, such as `String`s or `int`s, that can be themselves used as the option values, or of complex types, such as our `User` object, which don't map so easily to an option value. As you'll see, the Struts 2 collection-backed components provide a mechanism for indicating which of the `User` object's properties should be used as the option value. We'll start by demonstrating the simpler case of using a data set of simple types, then move on to the more complex use case of complex types.

We'll explain all of this in the context of our first, and most ubiquitous, component—the `select` component.

THE SELECT COMPONENT

The `select` component is perhaps the most common collection-based UI component. This component is built on the HTML select box, which allows the user to select a value from a list of options. In a Java web application, it's common to build these lists of options from `Collections`, `Maps`, or arrays of data. The `select` component offers a rich, flexible interface for generating select boxes from a wide assortment of back-end data sets. Just to make this easy, we'll start with a trivial but illustrative sample of using a `List` to build a `select` component. In this first example, the `List` will be of `String`s; we'll show how to use sets of complex types in a minute. The following snippet shows the simplest use case of the `select` UI component:

```
<s:select name="user.name" list="{'Mike','Payal','Silas'}" />
```

The `list` attribute of the `select` component points to the data set that will back the component. We've supplied an OGNL list literal for this value. You'll typically be using an OGNL expression to point to a list of data on the `ValueStack` rather than generate a list literally. Here, we're striving to simplify the usage while introducing the tag. As usual, the `name` attribute is an OGNL expression that'll target a destination property on the `ValueStack`. One of the strings in the list will be selected, submitted, and transferred by the framework onto the property referenced by our `name` attribute. We've seen this many times by now.

Here's how the preceding tag renders into HTML to present the user with the choice:

```
<select name="user.name" id="ViewPortfolio_user_name">
  <option value="Mike">Mike</option>
  <option value="Payal">Payal</option>
  <option value="Silas">Silas</option>
</select>
```

Again, we've removed the layout-related markup from the HTML output of this component. As you can see, each of the values in the list was used to create an `option` element. But what about prepopulation? Since we didn't specify a value attribute ourselves, the `name` attribute will be used to infer the value. If the `ValueStack` had contained a value in the property `user.name` when this tag rendered, that value would've been matched against the values of the option elements to preselect one of them. In this case, none have been preselected; the `ValueStack` must not have contained a value for the `user.name` property. That's about as simple as it gets. But if you understand the principle, the rest should make sense. Let's move on to the richer use cases of the `select` component.

Now we'll see how the `select` component supports using a wide range of data sets and offers flexible control over the various attributes of the generated HTML select box. Table 7.8 summarizes the attributes specific to using the `select` UI component.

Table 7.8 Important `select` attributes

Attribute	Data type	Description
`list`	`Collection`, `Map`, `Array`, or `Iterator`	A set of data used to generate the options for the select box.
`listKey`	`String`	The property of the `List`'s elements to be used for the value submitted when those elements are complex types; `key` by default.
`listValue`	`String`	The property of the `List`'s elements to be used for the content of the option when those elements are complex types—in other words, the string seen by the user; `value` by default.
`headerKey`	`String`	Used with the header. Specifies the value to submit if the user selects the header.
`headerValue`	`String`	Shown to the user as a header for the `List`, for example `"States"`, `"Countries"`.
`emptyOption`	`Boolean`	Used with the header. Places an empty spacer option between the header and the real options.
`multiple`	`Boolean`	User can select more than one value.
`size`	`String`	Number of choices shown at one time.

We'll now demonstrate a `Collection`-backed `select` component with an example from the Struts 2 Portfolio application. We've added a couple of functions to our home page that allow a user to select a portfolio for viewing. At this point, our sample application doesn't have a database and, hence, doesn't use numeric keys to identify such things as portfolios or artists. At this point, artist usernames must be unique. Furthermore, a portfolio name must be unique in the context of the artist who owns it. We can then assume that a username and portfolio name pair serves as a unique key for retrieving a given portfolio. The following examples use these requirements.

If you check the chapter 7 home page, you'll see some additions to our user interface. The first of these is a `select` component that asks the user to choose an artist. A screen capture of this component is shown in figure 7.3.

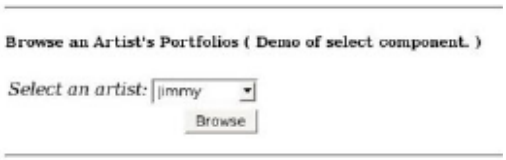

Figure 7.3 The `select` component presents a `Collection` of data as a set of choices.

The user can now select an artist and browse that artist's work. After the artist has been selected, the user will be taken to a page that presents her with a similar box presenting a selection of the portfolios associated with the chosen artist. Feel free to explore the full workflow of the application at your convenience. For now, we'll focus on how selecting an artist works. The following snippet from this chapter's PortfolioHomePage.jsp file shows how the select box in figure 7.3 is created:

```
<h5>Browse an Artist's Portfolios ( Demo of select component. )</h5>
  <s:form action="SelectPortfolio" >
  <s:select name="username" list='users' listKey="username"
     listValue="username" label="Select an artist" />
  <s:submit value="Browse"/>
</s:form>
```

This example seems more complicated than the first example we gave. It's not really though. First, we point the list attribute at a `Collection` of `User` objects exposed by our action class; check out `manning.chapterSeven.PortfoioHomePage` if you want to see this property. Next, we have two new attributes: `listKey` and `listValue`. These attributes are required, since the collection backing this `select`, unlike the first example we showed, holds complex types, `Users`, rather than simple strings. These new attributes allow us to select the specific properties from our `User` objects to use in building the component. The tag will iterate over the collection of `Users` and build an `option` element from each one. There will be one `option` element for each artist in the system then.

Two questions remain. First, how should we derive the value of each `option` element, the value that'll be submitted when that option is selected? Easy. The `listKey` attribute binds one of the properties of your object to the `value` attribute of the generated `option` element, which determines the request parameter value that'll be submitted if that option is selected. This `key` nomenclature comes from the idea that the `value` attribute of the select box is frequently set to the `key` value of the data object. Sometimes this is a numeric key from the database, but it doesn't have to be. It just needs to be a unique identifier. As we've said, usernames work for us at this point. Thus, we've set the `listKey` attribute to the `username` property.

Next question. How should we represent that choice to the user—what'll the body of the `option` element be? Again, an easy answer. The `listValue` attribute determines what the user will see in the UI select box. In other words, this determines the text visually displayed by each option. Don't be confused. This isn't the HTML option element's `value` attribute; that attribute is set by the previously explained `listKey`. In this case, we've set both `listKey` and `listValue` to the same property, `username`. This is

only because the username is both a unique identifier and a good string for visually representing the options to the user.

Now, let's see how the `select` tag renders. The following snippet shows the markup generated by that tag as it iteratively renders over the set of `Users` in the list:

```
<form id="SelectPortfolio" name="SelectPortfolio"
    action="/manningSampleApp/chapterSeven/SelectPortfolio.action" >
  <select name="username" id="SelectPortfolio_username">
    <option value="Jimmy">Jimmy</option>
    <option value="Chad">Chad</option>
    <option value="Mary">Mary</option>
  </select>
  <input type="submit" id="SelectPortfolio_0" value="Browse"/>
</form>
```

As you can see, there is an `option` element for each of the elements in the collection of `Users`. Each option uses the username both to depict the option to the user and to set the `value` attribute. Remember that the `value` attribute of the `option` element is set by the `listKey`, whereas the display string is set by the `listValue`. And that's how you wire a `select` component to a Java-side `Collection` property. As we've said, you can use a variety of Java-side types to back your components. Let's do the same thing again, with a `Map` on the Java side.

The following snippet shows the same component from the home page implemented with a `Map` of users instead of a `Collection`.:

```
<h5>Browse an Artist's Portfolios ( Demo of select component. )</h5>
<s:form action="SelectPortfolio" >
  <s:select name="username" list='users'  listValue="value.username"
    label="Select an artist" />
  <s:submit value="Browse"/>
</s:form>
```

As the `Map` that backs this holds the same `Users` as the previous example's `Collection`, this tag will generate exactly the same output as before. There are some slight differences in how the data is accessed, though. When the tag iterates over the `Map` of users, it doesn't iterate directly over the `User` objects themselves. This is just a detail of the Java `Map` API. Instead, it iterates over `Entry` objects. The `Entry` object has two properties, the `key` and the `value`. In our case, the `key` is the `username` and the `value` is the `User` object itself.

By default, the `select` component's `listKey` attribute will be set to `key` and, thus, point to the `username`. Also by default, the `listValue` attribute will be set to `value` and, thus, point to the `User` object. If you're using `Maps`, you can sometimes accept these defaults. In this example, we use the default `listKey` because our keys are our `usernames`, which is what we want. As for the `listValue`, we can't just use the entire value from the entry, because that would be the entire `User` object. If our `User` object implemented a suitable `toString()` method, we could allow the `select` component to use it for representing the choices to the user. But it doesn't. So, as you can see, we use a concise OGNL expression to target the `portfolioName` as the `listValue`. The bottom-line difference between using maps and using collections is that maps may

require a longer OGNL expression, such as `value.username` instead of just `username`. Here's the markup generated from rendering our `select` component tag:

```
<form id="SelectPortfolio" name="SelectPortfolio"
    action="/manningSampleApp/chapterSeven/SelectPortfolio.action" >
  <select name="username" id="SelectPortfolio_username">
    <option value="Jimmy">Jimmy</option>
    <option value="Chad">Chad</option>
    <option value="Mary">Mary</option>
  </select>
  <input type="submit" id="SelectPortfolio_0" value="Browse"/>
</form>
```

As you can see, everything is the same as when we rendered the tag using a `Collection` of users. The `Map` we use for this example contains the same set of `User` objects as the `Collection`. The notation for reaching the data is different, but functionally it doesn't matter whether you use `Map`s or `Collection`s.

The bottom line is that you can easily build a `select` component from any group of data. You can even pass arrays and iterators to the tag. We won't detail these uses, but they work pretty much as you would expect.

Tip

If you understand the `ValueStack` and the use of OGNL expressions in the tag attributes, you'll have no trouble exploring all the rich functionality offered by the Struts2 UI component tags. We can't begin to cover every bit of functionality they offer in the space of this book. Even if we could, our laborious efforts would be undermined by the constant flow of new components. A rich set of Ajax tags is in the foundry as we write.

Just keep in mind that all of the tags use the `ValueStack`. Some even push objects temporarily onto the `ValueStack` so that they can conveniently access the properties of those tags during their brief rendering cycle. For instance, when the select tag iterates over the set of objects handed to its list attribute, it pushes each object, temporarily, onto the `ValueStack`. The OGNL expressions in the tag's attributes can then resolve against this state of the `ValueStack`. When the iteration cycle ends, the object is popped and a new one is pushed when the next iteration cycle begins. This is the power of the `ValueStack`.

THE RADIO COMPONENT

The `radio` component offers much the same functionality as the `select` component, but presented in a different manner. Figure 7.4 shows what the `radio` component looks like on the page.

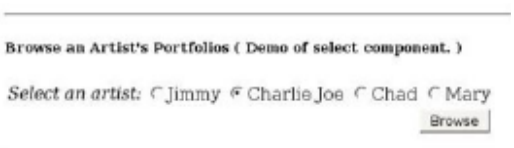

Figure 7.4 The `radio` component presents a collection of data as a set of choices.

The usage of the `radio` component is the same as the `select` component, except

that it has a few less attributes. For instance, the `radio` component doesn't allow multiple selections. You can use `Collections`, `Maps`, arrays, and `Iterators`, just as before. Again, the `radio` component tag uses the common UI component attributes. In addition to the common attributes, the `radio` component uses the attributes summarized in table 7.9.

Table 7.9 Important `radio` attributes

Attribute	Data type	Description
`list`	`Collection`, `Map`, `Array`, or `Iterator`	A set of data used to generate the radio selections.
`listKey`	`String`	The property of the collection's elements to be used for the value submitted; `key` by default.
`listValue`	`String`	The property of the collection's elements to be used for the content of the option; in other words, the string seen by the user; `value` by default.

As we said, the usage is nearly identical to the `select` component. In fact, to generate the screen shot of the `radio` component, we merely changed the name of the tag in PortfolioHomePage.jsp from `select` to `radio`. That's it. Because of this, we won't go into details on this component.

THE CHECKBOXLIST COMPONENT

The `checkboxlist` component is also similar to the `select` component. As you can see in figure 7.5, it presents the same selection choices, but using checkboxes, so it allows for multiple selections.

Figure 7.5 The `checkboxlist` component presents a collection of data as a set of choices from which the user can select several.

Using the `checkboxlist` is just like using the `select` component with the multiple selection option chosen. Again, for our screenshot, we just changed the name of the tag in PortofolioHomePage.jsp from `select` to `checkboxlist`. In addition to the common UI component attributes, the `checkboxlist` also frequently uses the attributes summarized in table 7.10.

Table 7.10 Important `checkboxlist` attributes

Attribute	Data type	Description
`list`	`Collection`, `Map`, `Array`, or `Iterator`	A set of data used to generate the `checkboxlist` selections.
`listKey`	`String`	The property of the collection's elements to be used for the value submitted; `key` by default.
`listValue`	`String`	The property of the collection's elements to be used for the content of the option, in other words, the string seen by the user; `value` by default.

Note that these attributes are used in the same manner as for the other collection-driven UI components, such as the `select` component.

PREPOPULATION WITH COLLECTION-BACKED COMPONENTS

Prepopulation of the collection-backed components may not seem straightforward at first glance. For starters, we'll call it preselection, as that more accurately describes what'll happen. It works just as the prepopulation of the simple component tags; the `value` attribute points to a property on the `ValueStack` that'll be used as the current value when preselecting one of the options. Remember, you'll frequently leave the `value` attribute unset, allowing the framework to infer it from the `name` attribute, as we've seen.

With simpler components, like the `textfield`, the value will directly populate the input field. With collection-backed components, we don't have a simple input field; we have a selection of options. Each of these HTML options has a `value` attribute that represents the value of the component if that option is selected. The trick, then, is to use the `value` attribute of the Struts 2 tag to match one of those option values. When a match occurs, that option is preselected. Let's demonstrate how this preselection works.

To do so, we'll modify the `PortfolioHomePage` to automatically select one of the artists as the default choice. Let's imagine that, each week, the Struts 2 Portfolio will feature the work of one artist by making that artist the default choice. In order to have one of the options preselected, we must provide a property on the `ValueStack` that holds the `username`, our key, of the default artist. We've implemented a method on our `PortfolioService` that returns the `username` of the currently featured artist. We'll retrieve this and set it on a `defaultUsername` property on the action. To make things clearer, we'll specify the component's `value` attribute separately from the `name` attribute, rather than letting the framework infer it. The following snippet shows the code from the `PortfolioHomePage` action's `execute()` method, which does the necessary work:

```
public String execute(){
  Collection users =  getPortfolioService().getUsers();          ❶
  setUsers( users );

  String selectedUsername = getPortfolioService().getDefaultUser();   ❷
  setDefaultUsername( selectedUsername );

  return SUCCESS;
}
```

First, we set the collection of users that'll be used to create the collection-backed component ❶. Then, we retrieve the username of the featured artist, and we set this on the `defaultUsername` JavaBeans property ❷ on the action itself. This'll make it available on the `ValueStack`. Now, let's see how this works with our radio component.

```
<s:form action="SelectPortfolio" >
  <s:radio name="username" list='users' value="defaultUsername"
    listKey="username" listValue="username" label="Select an artist" />
  <s:submit value="Browse"/>
</s:form>
```

Just as in the previous examples, we point the component to our collection of `Users` and tell it to use each `User`'s username as both the `listKey` and `listValue`. Then, we preselect our artist of the week by pointing the `value` attribute at our `defaultUsername` property. At the moment, our featured artist is the user `Chad`. So the following markup is generated by our tag and action logic:

```
<input type="radio" name="username"  value="Jimmy"/>
<input type="radio" name="username"  value="Charlie Joe"/>
<input type="radio" name="username"  checked="checked" value="Chad"/>
<input type="radio" name="username"  value="Mary"/>
```

As you can see, the username of the featured user was `Chad`, so the radio button with that value was checked. Note that you can easily go back and reimplement this to stash the featured artist's username in a property on the action called `username`. Then you wouldn't have to manually specify the `value` attribute separately from the `name` attribute. This would've been confusing in our example, since our `listKey` and `listValue` attributes are also `usernames`. How can this be? Consider that the `listKey` and `listValue` are used during the iterative cycle of the collection-backed components. During this cycle, each of the `Users` in the collection is pushed onto the `ValueStack`. Thus, the `listKey` and `listValue` hit the property as found on the current `User` object. But when the `name` attribute resolves, the iteration hasn't started; the top object on the `ValueStack` is the action, not one of the `Users`. This subtlety explains how the same OGNL, `username`, can mean different things depending upon the state of the `ValueStack`. This is admittedly tricky. But once you master the `ValueStack`, it'll all seem elegant and powerful.

This same preselection process works for all of the collection-backed components. We should make a point about multiple selection before moving on. If you're working with a multiple-selection component, such as the `select` component with the `multiple` attribute set to `true`, your value attribute can point to a property that contains more than one choice, such as an array of usernames in our case. The component will select each of those values.

That does it for the components that we expect all developers will need to use on a regular basis. We'll round out the chapter by pointing you toward some rich components that would enhance any page.

7.3.4 Bonus components

We call this the bonus components section because these components are useful but, for one reason or another, not quite as much as the previous components. For this reason, we'll be slightly less exhaustive in our explanations. We'll provide enough to make sure you can use them, but we won't explore advanced cases. As always, we recommend visiting the Struts 2 site to get the details on the full set and functionality of UI component tags.

THE LABEL COMPONENT

The `label` component shouldn't be confused with the label generated by the other UI components we've covered previously. The `label` component has a special and simple use case. Figure 7.6 shows how a `label` component is used.

In the form shown in the figure, the `username` property of a `User` is written onto the form in a read-only form. This is the purpose of a `label` component. Usage is straightforward, as you can see from the following snippet, which shows the tag that created the label shown in figure 7.6:

Figure 7.6 A `label` component can be used to display read-only data on a form.

```
<s:label name="username" label="Username" />
```

Basically, it's just like a read-only `textfield`.

THE HIDDEN COMPONENT

The `hidden` component also satisfies a specific use case. Frequently, we need to embed hidden request parameters into a form without showing them to the user. Sometimes you set the values of these hidden fields with values from the server. Sometimes you use JavaScript functionality to calculate the values for these hidden fields. This book won't show you how to write JavaScript, but it'll show you how to use the `hidden` component. Here's the markup:

```
<s:hidden name="username" />
```

And here's the hidden input field as written into the HTML:

```
<input type="hidden" name="username" value="Chad"
  id="UpdateAccount_username"/>
```

Note that this example obviously rendered with a `username` property on the `ValueStack` that contained the value of `Chad`. As you might guess, this can't be seen on the page in the browser, but it'll be submitted with the other request parameters.

Now, we'll move on to a more complex component of the bells-and-whistles class.

THE DOUBLESELECT COMPONENT

The `doubleselect` component addresses the common need to link two select boxes together so that selecting one value from the first box changes the set of choices available in the second box. Such a component might link a first select box of state names to a second box of city names. If you select `California` in the first box, you get a list of California cities in the second box. When you change to another state, the second box automatically changes to reflect this choice.

To use the `doubleselect` component, you first have to tell the component which data set it should use to generate the first select box. This works just like setting up a normal `select` component. The same attributes are used and they function in the same fashion as before. Next, you have to specify a property that'll refer to another data set that can be used as the second list. This property will effectively need to refer to multiple sets of data, most likely one for each item in the first list—for example, a set of cities for each state in the first list. While this can begin to sound tricky, you need to recall that all these things resolve against the `ValueStack`. With this in mind, everything should make perfect sense. Well, let's hope so anyway.

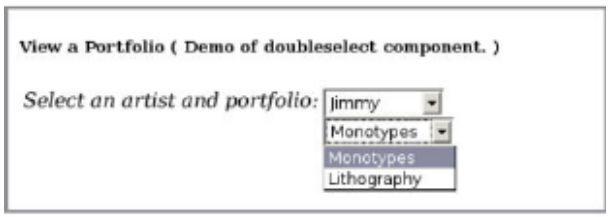

Figure 7.7 The `doubleselect` component dynamically populates a secondary select box based on the primary select box's choice.

Let's start by looking at a screen shot to see what we're doing. Figure 7.7 shows a `doubleselect` component that we've put on our `PortfolioHomePage`.

To select a portfolio to view, you must first select an artist from the primary select box. The `doubleselect` component will then dynamically populate the second box with the names of all of that artist's portfolios. After browsing for a few minutes, the user will select one of the portfolios and that portfolio will be displayed.

The following snippet shows how this `doubleselect` was set up:

```
<h4>Select a portfolio to view.</h4>
  <s:form action="ViewPortfolio">
  <s:doubleselect name="username" list='users' listKey="username"
    listValue="username" doubleName="portfolioName"
    doubleList="portfolios" doubleListValue="name" />
  <s:submit value="View"/>
</s:form>
```

The attributes are just like the normal select box, except there's a second set of attributes that refers to the secondary select box with names such as `doubleXXX`. As we indicated before, the first set of attributes works just like the standard `select` component.

Notably, the `list` attribute refers to a property that holds a set of data from which the first select box will be generated. In this case, it's a set of `User` objects exposed as the `users` property on our `PortfolioHomePage` action. While this tag renders, it'll iterate over each user in the collection, pushing that user onto the `ValueStack` while it renders the markup related to that particular user. In the case of the `doubleselect` component, the markup related to each user will be some JavaScript for dynamically repopulating the secondary select box with the portfolios for that user.

The `doubleList` attribute, like the `list` attribute, is an OGNL expression that points to a property on the `ValueStack`. In logical terms, it points to a set of data that'll be used to build the secondary select box. As we've said, while the `doubleselect` component iterates over the `users`, it pushes each user onto the `ValueStack`. We've implemented our `User` class so that it exposes each user's set of portfolios as a property named `portfolios`. Thus, when a specific user object is on the `ValueStack`, the OGNL expression `portfolios` will evaluate to that user's collection of portfolios. This use of the `ValueStack` as a mini-execution context for each iterative cycle of a tag's rendering is powerful. Try to keep this in mind when you get ready to build your own components, which we'll learn how to do in chapter 13.

As this is a bonus component, and as JavaScript can eat up some book space in a hurry, we won't show you the HTML source generated by this `doubleselect`. But you can always look at it yourself. Ultimately, the `doubleselect` component is a convenient

and powerful component that also, incidentally, illustrates elegant use of the `ValueStack` that drives dynamic generation of HTML with minimal lines of code count.

7.4 Summary

This chapter introduced the powerful Struts 2 UI component tags. With these components in hand, you should be able to quickly assemble rich interfaces that easily wire all your application domain data to the keystrokes and mouse clicks of your end users. If we've done our job, you now understand that the component tags are much more than just tags. In reality, the Struts 2 UI component tags are a mini-MVC framework unto themselves. Let's recap the highlights of this lengthy chapter.

We started the chapter off by detailing how these component tags differ from plain, ordinary tags. First of all, these components do a whole lot more than just render an HTML element. Based on which theme you choose, they can render additional markup to support everything from rich layout to the structural foundations for advanced JavaScript and Ajax support. Perhaps more importantly, the UI components, with a helping hand from OGNL, bind your user interface to all the framework's internal components, including the Validation Framework and the `ValueStack`. This powerful binding allows you to accomplish such things as automatic validation error reporting and form prepopulation with minimum coding.

We also took pains to assure you that the autogeneration of markup wasn't going to tie your hands as a developer. In fact, the opposite is true. Instead of making it harder to handle special cases and quirky requirements, the UI component architecture has been carefully designed to allow the developer to modify the underlying templates as little or as much as necessary to bend the components to their own needs. Most importantly, your modifications will still fit into the component structure, so you'll be able to reuse and manage your new components as elegantly as the next Struts 2 guru. We'll learn all about modifying the components in chapter 13.

While we don't need to take time to rehash the specifics of the various tags we introduced in this chapter, we want to make a couple of points about them. First of all, we've tried to provide you with a solid understanding of how the UI component tags work. We've also tried to provide you with examples and demonstrations of the most commonly used components and their most common use cases. As always, the Struts 2 framework provides easy paths to the most common tasks, and then leaves the door open for nearly anything else. With this in mind, there's much we left unsaid about even the most common tags. If you need something that doesn't seem to be here, please refer to the Struts 2 website for a comprehensive listing of all the details of each tag. There are even more components than we've shown. In particular, there are additional tags of the rich functionality variety. The website is your best resource for an up-to-date list of the full set of tags.

Now it's time to wrap up the view part of the book. We noted in the introductory section of the book that the view component of the framework was something called the *result*. Working with results is so easy that we haven't said much about them. But we will now. The next chapter discusses the result in detail.