

STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

 HANNING

Chapter 6. Building a view: tags.....	1
Section 6.1. Getting started.....	2
Section 6.2. An overview of Struts tags.....	7
Section 6.3. Data tags.....	12
Section 6.4. Control tags.....	20
Section 6.5. Miscellaneous tags.....	22
Section 6.6. Using JSTL and other native tags.....	26
Section 6.7. A brief primer for the OGNL expression language.....	27
Section 6.8. Summary.....	35

Building a view: tags



This chapter covers

- Working with data tags
- Controlling flow with control tags
- Surveying the miscellaneous tags
- Exploring the OGNL expression language

In this chapter, we'll start looking at the Struts 2 tag library in detail. We'll provide a good reference to the tags and clear examples of their usage. We'll also finish exploring the Object-Graph Navigation Language (OGNL). We've already seen how OGNL is used to bind form fields to Java properties, such as those exposed on our action, to guide the framework's automatic data transfer mechanism. In chapter 5, we learned about the type conversion aspects of OGNL in the context of data entering the framework.

Now, we'll focus on the OGNL expression language (EL) in the context of data exiting the framework through the Struts 2 tag API. While the previous chapter showed us how to map incoming request parameters to Java properties exposed on the `ValueStack`, this chapter will show you how to pull data off of those properties for the rendering of result pages. We'll explore the syntax of the OGNL EL and study the locations from which it can pull data. In particular, we'll look closely at the `ValueStack` and the `ActionContext`. These objects hold all of the data important to

processing a given request, including your action object. While it may be possible to blissfully ignore their existence during much of your development, we think the benefits are too high not to spend a few minutes getting to know them.

But that won't take long, and we'll make the most of the remainder of the chapter. After we demystify these two obscure repositories, we'll provide a reference-style catalog of the general-use tags in the Struts 2 tag API. These powerful new tags allow you to wield the OGNL expression language to feed them with values. But tags are tags, and they also won't take long to cover. So, after covering the tags, we'll provide a concise primer to the advanced features of the OGNL expression language. In the end, you'll wield your OGNL expressions confidently as you navigate through the densest of object graphs to pull data into the dynamic rendering of your view pages. Bon appétit.

First, we need to take a moment to make sure you understand where the framework holds all the data you might want to access from the tags. This usage of globally accessible storage areas may not be familiar to developers coming from other frameworks, including Struts 1. In the next section, we'll take care of these crucial introductions.

6.1 **Getting started**

Before we talk about the details of how Struts 2 tags can help you dynamically pipe data into the rendering of your pages, let's talk about where that data comes from. While we focused on the data moving into the framework in the previous chapter, we'll now focus on the data leaving the framework. When a request hits the framework, one of the first things Struts 2 does is create the objects that'll store all the important data for the request. If we're talking about your application's domain-specific data, which is the data that you'll most frequently access with your tags, it'll be stored in the `ValueStack`. But processing a request requires more than just your application's domain data. Other, more infrastructural, data must be stored also. All of this data, along with the `ValueStack` itself, is stored in something called the `ActionContext`.

6.1.1 **The `ActionContext` and OGNL**

In the previous chapter, we used OGNL expressions to bind form field names to specific property locations on objects such as our action object. We already know that our action object is placed on something called the `ValueStack` and that the OGNL expressions target properties on that stack object. In reality, OGNL expressions can resolve against any of a set of objects. The `ValueStack` is just one of these objects, the default one. This wider set of objects against which OGNL expressions can choose to resolve is called the `ActionContext`. We'll now see how OGNL chooses which object to resolve against, as well as what other objects are available for accessing with OGNL.

TIP The `ActionContext` contains all of the data available to the framework's processing of the request, including things ranging from application data to session- or application-scoped maps. All of your application-specific data, such as properties exposed on your action, will be held in the `ValueStack`, one of the objects in the `ActionContext`.

All OGNL expressions must resolve against one of the objects contained in the `ActionContext`. By default, the `ValueStack` will be the one chosen for OGNL resolution, but you can specifically name one of the others, such as the session map, if you like.

The `ActionContext` is a key behind-the-scenes player in the Struts 2 framework. If you've worked with other web application frameworks, particularly Struts 1, then you might be asking, "Why do I need an `ActionContext`? Why have you made my life more complicated?" We've been trying to emphasize that the Struts 2 framework strives toward a clean MVC implementation. The `ActionContext` helps clean things up by providing the notion of a context for the execution of an action. By *context* we mean a simple container for all the important data and resources that surround the execution of a given action. A good example of the type of data we're talking about is the map of parameters from the request, or a map of session attributes from the Servlet Container. In Struts 1, most of these resources were accessed via the Servlet stuff handed into the execution of every action. We've already seen how clean the Struts 2 action object has become; it has no parameters in its method signature to tie it to any APIs that might have little to do with its task. So, really, your life is much less complicated, though at first it might not seem so.

Before we show you all of the specific things that the `ActionContext` holds, we need to discuss OGNL integration. As we've seen, OGNL expressions target properties on specific objects. The resolution of each OGNL expression requires a root object against which resolution of references will begin. Consider the following OGNL expression:

```
user.account.balance
```

Here we're targeting the `balance` property on the `account` object on the `user` object. But where's the `user` object located? We must define an initial object upon which we'll locate the `user` object itself. Every time you use an OGNL expression, you need to indicate which object the expression should start its resolution against. In Struts 2, each OGNL expression must choose its initial object from those contained in the `ActionContext`. Figure 6.1 shows the `ActionContext` and the objects it contains, any of which you can point your OGNL resolution toward.

As you can see, the `ActionContext` is full of juicy treasures. The most important of these is the `ValueStack`. As we've said, the `ValueStack` holds your application's domain-specific data for a given action invocation. For instance, if you're updating a

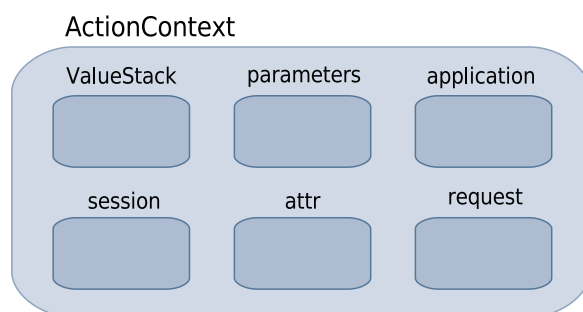


Figure 6.1 The `ActionContext` holds all the important data objects pertaining to a given action invocation; OGNL can target any of them.

student, you'll expect to find that student data on the `ValueStack`. We'll divulge more of the inner workings of the `ValueStack` in a moment. The other objects are all maps of important sets of data. Each of them has a name that indicates its purpose and should be familiar to seasoned Java web application developers, as they correspond to specific concepts from the Servlet API. For more information on where the data in these sets comes from, we recommend the Java Servlet Specification. The contents of each of these objects is summarized in table 6.1.

Table 6.1 The names and contents of the objects and maps in the `ActionContext`

Name	Description
<code>parameters</code>	Map of request parameters for this request
<code>request</code>	Map of request-scoped attributes
<code>session</code>	Map of session-scoped attributes
<code>application</code>	Map of application-scoped attributes
<code>attr</code>	Returns first occurrence of attribute occurring in page, request, session, or application scope, in that order
<code>ValueStack</code>	Contains all the application-domain-specific data for the request

The `parameters` object is a map of the request parameters associated with the request being processed—the parameters submitted by the form, in other words. The `application` object is a map of the application attributes. The `request` and `session` objects are also maps of request and session attributes. By *attribute* we mean the Servlet API concept of an attribute. Attributes allow you to store arbitrary objects, associated with a name, in these respective scopes. Objects stored in application scope are accessible to all requests coming to the application. Objects stored in session scope are accessible to all requests of a particular session, and so forth. Common usage includes storing a user object in the session to indicate a logged-in user across multiple requests. The `attr` object is a special map that looks for attributes in the following locations, in sequence: page, request, session, and application scope.

Now let's look at how we choose which object from the `ActionContext` our OGNL will resolve against.

SELECTING THE ROOT OBJECT FOR OGNL

Up until now, we've hidden the fact that an OGNL expression must choose one of the objects in the `ActionContext` to use as its root object. So how does the framework choose which object to resolve a given OGNL expression against? We did nothing about this while writing our simple input field names in the previous chapter. As with all Struts 2 mysteries, this comes down to a case of intelligent defaults. By default, the `ValueStack` will serve as the root object for resolving all OGNL expressions that don't explicitly name an initial object. You almost don't have to know that the `ValueStack` exists to use Struts 2. But, take our word, that's not a blissful ignorance.

Though you haven't seen it yet, OGNL expressions can start with a special syntax that names the object from the context against which they should resolve. The following OGNL expression demonstrates this syntax:

```
#session['user']
```

This OGNL expression actively names the session map from the `ActionContext` via the `#` operator of the expression language. The `#` operator tells OGNL to use the named object, located in its context, as the initial object for resolving the rest of the expression. With Struts 2, the OGNL context is the `ActionContext` and, thankfully, there's a session object in that context. This expression then points to whatever object has been stored under the key `user` in the session object, which happens to be the session scope from the Servlet API. This could be, for instance, the user object that our Struts 2 Portfolio login action stores in the session.

INSIDER SCOOP

As of version 2.1 of Struts 2, which will most likely be available by the time this book is printed, the expression language used by the framework will become pluggable. While this may sound unsettling, it's actually harmless. You can use OGNL, just as described in this book, or you can insert your own expression language into the framework. Any EL plugged into Struts 2 will have access to the same data and will serve the same purposes as we've described for OGNL. But don't panic: you can just use OGNL! Besides, it might be several versions before switching ELs becomes well established.

As far as the full syntax of OGNL goes, we'll wait a bit on that. In the previous chapter, we saw as much of the OGNL syntax as we needed for writing our input field names that would target our properties. At the most complex, this included expressions that could reference map entries via string and object keys. But OGNL is much more. The OGNL expression language contains many powerful features, including the ability to call methods on the objects you reference. While these advanced features give you a set of flexible and powerful tools with which to solve the thorn that you inevitably find stuck in your side late on a Friday afternoon, they aren't necessary in the normal course of things. We'll continue to delay full coverage of the OGNL expression language until the end of this chapter, preferring instead to only introduce as much as we need while demonstrating the tags. The main idea here is to understand the role that the expression language plays in the framework.

We'll start exploring that role by taking a long-delayed look at the default root object for all OGNL resolution, the `ValueStack`.

6.1.2 The ValueStack: a virtual object

Back to that default root object of the `ActionContext`. Understanding the `ValueStack` is critical to understanding the way data moves through the Struts 2 framework. By now, you've got most of what you need. We've seen the `ValueStack` in action. When Struts 2 receives a request, it immediately creates an `ActionContext`, a `ValueStack`,

and an action object. As a carrier of application data, the action object is quickly placed on the `ValueStack` so that its properties will be accessible, via OGNL, to the far reaches of the framework.

First, these will receive the automatic data transfer from the incoming request parameters. As we saw in chapter 4, this occurs because the `params` interceptor sets those parameters on properties exposed on the `ValueStack`, upon which the action object sits. While other things, such as the model of the `ModelDriven` interface, may also be placed on the stack, what all data on the `ValueStack` has in common is that it's all specific to the application's domain. In MVC terms, the `ValueStack` is the request's view of the application's model data. There are no infrastructural objects, such as Servlet API or Struts 2 objects, on the `ValueStack`. The action is only there because of its role as domain data carrier; it's not there because of its action logic.

There's only one tricky bit about the `ValueStack`. The `ValueStack` pretends to be a single object when OGNL expressions are resolved against it. This virtual object contains all the properties of all the objects that've been placed on the stack. If multiple occurrences of the same property exist, those lowest down in the stack are hidden by the uppermost occurrence of a similarly named property. Figure 6.2 shows a `ValueStack` with several objects on it.

As you can see in figure 6.2, references to a given property resolve to the highest occurrence of that property in the stack. While this may seem complicated, it's actually not. As with most Struts 2 features, the flexibility and power to address complex use cases is there, but the common user can remain ignorant of such details.

Let's examine figure 6.2. As usual, the action object itself has been placed on the stack first. Then, a model object was added to the stack. This most likely has occurred because the action implements `ModelDriven`. Sometime after that, another object, apparently some sort of random-number-making bean, was added to the stack. By *bean* we simply mean a Java object that either serves as a data carrier or as a utility-providing

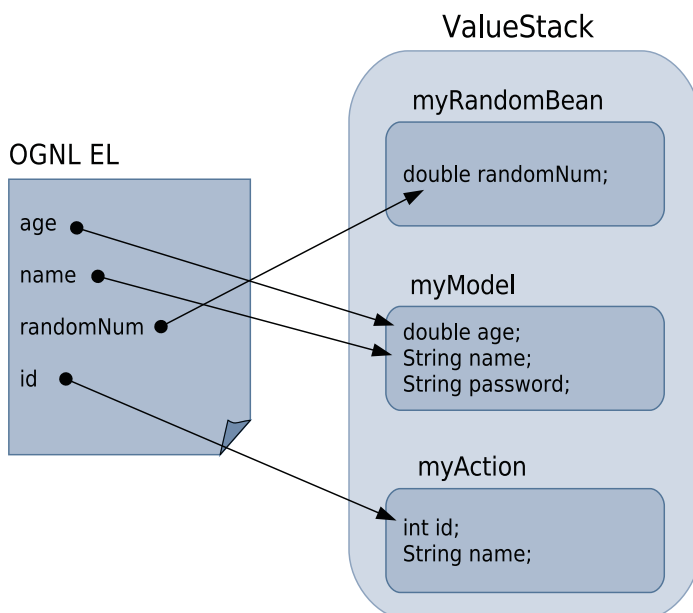


Figure 6.2 The `ValueStack` is the default object against which all OGNL expressions are resolved.

object. In other words, it's usually just some object whose properties you might want to access from your tags with OGNL expressions.

At present, we just want to see how the `ValueStack` appears as a single virtual object to the OGNL expressions resolving against it. In figure 6.2, we have four simple expressions. Each targets a top-level property. Behind the scenes, OGNL will resolve each of these expressions by asking the `ValueStack` if it has a property such as, for instance, `name`. The `ValueStack` will always return the highest-level occurrence of the `name` property in its stack of objects. In this case, the action object has a `name` property, but that property will never be accessed as long as the model object's `name` property sits on top of it.

Definition

When Java developers talk about beans in the context of view technologies, such as JSPs, they frequently mean something different than just a Java object that meets the JavaBeans standard. While these beans are most likely good JavaBeans as well, they don't have to be. The usage in this context more directly refers to the fact that the bean is a Java object that exposes data and/or utility methods for use in JSP tags and the like. Many developers call any object exposed like this a "bean."

This nomenclature is a historical artifact. In the past, expression languages used in tags couldn't call methods. Thus, they could only retrieve data from an object if it were exposed as a JavaBean property. Since the OGNL expression language allows you to call methods directly, you could completely ignore JavaBeans conventions and still have data and utility methods exposed to your tags for use while rendering the page. However, in order to keep your JSP pages free of complexity, we strongly recommend following JavaBeans conventions and avoiding expression language method invocation as long as possible.

Just so you don't worry about it, we might as well discuss how that bean showed up on top of the stack. Prior to this point, we've just had stuff automatically placed on the `ValueStack` by the framework. So how did the bean get there? There are many ways to add a bean to the stack. Many of the most common ways occur within the tags that we'll soon cover. For instance, the `push` tag lets you push any bean you like onto the stack. You might do such a thing just before you wanted to reference that bean's data or methods from later tags. We'll demonstrate this with sample code when we cover those tags.

With a clear view of where the data is and how to get to it, it's time to get back to the Struts 2 tags that are the focus of this chapter, and are the means of pulling data from the `ActionContext` and `ValueStack` into the dynamic rendering of your view layer pages.

6.2 An overview of Struts tags

The Struts 2 tag API provides the functionality to dynamically create robust web pages by leveraging conditional rendering and integration of data from your application's domain model found on the `ValueStack`. Struts 2 comes with many different types of

tags. For organizational purposes, they can be broken into four categories: *data tags*, *control-flow tags*, *UI tags*, and *miscellaneous tags*. Since they are a complex topic all to themselves, we'll leave the UI tags for chapter 7. This chapter examines the other three categories.

Data tags focus on ways to extract data from the `ValueStack` and/or set values in the `ValueStack`. Control-flow tags give you the tools to conditionally alter the flow of the rendering process; they can even test values on the `ValueStack`. The miscellaneous tags are a set of tags that don't quite fit into the other categories. These leftover tags include such useful functionality as managing URL rendering and internationalization of text. Before we get started, we need to make some general remarks about the conventions that are applied across the usage of all Struts 2 tag APIs.

6.2.1 *The Struts 2 tag API syntax*

The first issue to address is the multiple faces of the Struts 2 tag API. As we've mentioned earlier, Struts 2 tags are defined at a higher level than any specific view-layer technology. Using the tags is as simple as consulting the API. The tag API specifies the attributes and parameters exposed by the tag. Once you identify a tag that you want to use, you simply move on to your view technology of choice—JSP, Velocity, or FreeMarker. Interfaces to the tag API have been implemented in all three technologies. The differences in usage among the three are so trivial that we'll be able to cover them in the rest of this short subsection. After that, we present our functional reference of the tags, including a summary of their attributes and parameters. We also include examples of the tags in action. These examples are done in JSP, but we think you'll soon see that taking your knowledge of the Struts 2 tags API to one of the other technologies will take approximately zero effort. Let's start with JSPs.

JSP SYNTAX

The JSP versions of the Struts 2 tags are just like any other JSP tags. The following property tag demonstrates the simple syntax:

```
<s:property value="name"/>
```

The only other thing to note is that you must have the `property taglib` declaration at the top of your page before using the Struts 2 tags. This is standard JSP stuff, and the following snippet from one of our Struts 2 Portfolio application's JSPs should show you what you need:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
```

The second line, the `taglib` directive, declares the Struts 2 tag library and assigns them the "s" prefix by which they'll be identified.

VELOCITY SYNTAX

You can also use Velocity templates for your view technology. All you need to do is specify your result type to the built-in `velocity` result type. We'll see the details of declaring a result to use a Velocity result type in chapter 8. For now, rest assured that the framework supports using Velocity out of the box. Let's see how the Struts 2 tags

are accessed from Velocity. In Velocity, the Struts 2 tag API is implemented as Velocity macros. This doesn't matter though; the API is the same. You just need to learn the macro syntax that specifies the same information. Here's the Velocity version of the property tag:

```
#sproperty( "value=name" )
```

Struts 2 tags that would require an end tag may require an #end statement in Velocity. Here's a JSP form tag from the Struts 2 Portfolio application that uses a closing tag:

```
<s:form action="Register">
  <s:textfield name="username" label="Username"/>
  <s:password name="password" label="Password"/>
  <s:textfield name="portfolioName" label="Enter a name"/>
  <s:submit value="Submit"/>
</s:form>
```

And here's the same tag as a Velocity macro:

```
#sform ("action=Register")
  #stextfield ("label=Username" "name=username")
  #spassword ("label=Password" "name=password")
  #stextfield ("label=Enter a name" "name=portfolioName")

  #ssubmit ("value=Submit")
#end
```

Again, it's the same tag, different syntax. Everything, as pertains to the API, is still the same.

FREEMARKER SYNTAX

The framework also provides out-of-the-box support for using FreeMarker templates as the view-layer technology. We'll see how to declare results that use FreeMarker in chapter 8. For now, here's the same property tag as it would appear in FreeMarker:

```
<@s.property value="name"/>
```

As you can see, it's more like the JSP tag syntax. In the end, it won't matter what view-layer technology you choose. You can easily access all the same Struts 2 functionality from each technology. While we'll demonstrate the tag API using JSP tags, we trust that you'll be able to painlessly migrate that knowledge to Velocity or FreeMarker according to the syntax conventions we've just covered.

Now we'll outline some important conventions regarding the values you pass into the attributes of your Struts 2 tags, regardless of which view technology you use.

6.2.2 Using OGNL to set attributes on tags

There are a couple of things you have to understand when setting values in tag attributes. The basic issue is whether the attribute expects a string literal value or some OGNL that'll point to a typed value on the ValueStack. We'll introduce this issue now, and revisit it as we cover the tags themselves.

First, we need to consider that an attribute on a tag is eventually going to be processed by the Java code that backs the tag implementation. But in the JSP page, for

instance, everything is a string. Working with other technologies, you've become familiar with using some sort of escape sequence to force the attribute value to be parsed as an expression versus being interpreted as a string literal; this frequently leads to markup that borders on being hard to read, with its proliferation of `${expression}` notation. In an effort to make the use of OGNL in tags more intuitive and readable, Struts 2 makes assumptions about what kind of data it expects for each attribute. In particular, a distinction is made in the handling of attributes whose type, in the underlying execution, will be `String` and those whose type will be non-`String`.

STRING AND NON-STRING ATTRIBUTES

If an attribute is of type `String`, then the value written into the attribute, in the actual JSP or Velocity page, is interpreted as a string literal. If an attribute is some non-`String` type, then the value written into the attribute is interpreted as an OGNL expression. This makes sense because the OGNL expressions can point to typed Java properties on the `ValueStack`, thus making a perfect tool for passing in typed parameters. The following property tags demonstrate the difference between `String` and non-`String` attribute types:

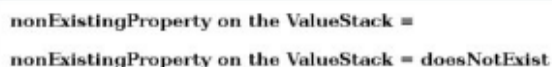
```
nonExistingProperty on the ValueStack = <s:property
  value="nonExistingProperty" />

nonExistingProperty on the ValueStack = <s:property
  value="nonExistingProperty" default="doesNotExist" />
```

Here, we have two somewhat identical uses of the Struts 2 property tag. First of all, note that the property tag's value attribute is typed as `Object`, a non-`String` attribute. The use case of the property tag is to take a Java property, typically from the `ValueStack`, and write it as a string into the rendering page. As we know, the properties on the `ValueStack` might be of any Java type; the conversion to a string will be handled automatically by the OGNL type converters. The property tag's value attribute tells it the property to render to the page. In the case of these examples, both tags are looking for a property called `nonExistingProperty`.

The property tag will try to locate this property by resolving the value attribute's value as an OGNL expression. Since no specific object from the `ActionContext` is named with the `#` operator, it'll look on the `ValueStack`. As it turns out, `nonExistingProperty` doesn't exist on the `ValueStack`. What then? A null value will be converted to an empty string. In the case of the first tag, as you can see in figure 6.3, nothing will render.

But the second tag does write something: the string `doesNotExist`. The second property tag still tries to pull the `nonExistingProperty` from the `ValueStack`, which again comes up empty. However, it also specifies a default attribute that gives a value to use if the property doesn't exist. Since the purpose of the default attribute is to provide a default string for the tag to put in the page, its type is `String`. When the value given to an attribute is ultimately to be used as a string, it makes sense that the



```
nonExistingProperty on the ValueStack =
nonExistingProperty on the ValueStack = doesNotExist
```

Figure 6.3 Output from property tags with no value: one specifies a default value while the other doesn't

default behavior is to interpret the attribute value as a string literal. Thus, the default value of `doesNotExist` isn't used as an OGNL expression. As you can see in figure 6.3, which shows the output of these two tags, the second tag uses `doesNotExist` as a string literal in its rendering.

HEADS-UP Attributes passed to Struts 2 tags are divided into two categories. Attributes that'll be used by the tag as String values are referred to as *string attributes*. Attributes that point to some property on the `ValueStack`, or in the `ActionContext` are referred to as *nonstring attributes*. All nonstring attributes will be interpreted as OGNL expressions and used to locate the property that'll contain the value to be used in the tag processing. All string attributes will be taken literally as Strings and used as such in the tag processing. You can force a string attribute to be interpreted as an OGNL expression by using the `%{expression}` syntax.

If you've worked with other expression languages embedded into tags, you're probably wondering when and how you can use some sort of expression language escape sequence. Many of you are familiar with various markers, such as `${expression}`, that indicate which text should be regarded as an expression and which as an actual string. The convention we've just described, where Struts 2 assumes that some attributes will be expressions and others will be strings, avoids many of the scenarios in which such escape sequences would be called for. However, we'll sometimes want to use an OGNL expression with a String attribute. What then?

FORCING OGNL RESOLUTION

Let's say, assuming the previous example of the `nonExistingProperty`, that you wanted to use a String property from the `ValueStack` as your default attribute value. In this case, you wouldn't want the default attribute to be interpreted as a string literal. You'd want it to be interpreted as an OGNL expression pointing to your String property on the `ValueStack`. If you want to force OGNL resolution of a String attribute, such as the default attribute of the `property` tag, then you need to use an OGNL escape sequence. This escape sequence is `%{expression}`. The following snippet revisits the scenario from the previous `property` tag example using the escape sequence to force the default attribute value to be interpreted as an OGNL expression:

```
nonExistingProperty on the ValueStack =
<s:property value="nonExistingProperty" default="%{myDefaultString}" />
```

Now the value of the default attribute will be interpreted as OGNL, and the actual string used as the default string will be pulled from the `myDefaultString` property on the `ValueStack`.

Note the similarity between the bracket syntax used to force the String attribute to evaluate as an OGNL expression and the JSTL Expression Language syntax.

Struts 2 OGNL Syntax	JSTL Expression Language
<code>%{ expression }</code>	<code>\${ expression }</code>

OGNL uses % instead of \$. While this may seem confusing to some JSP veterans, in reality you don't use the OGNL escape sequence very often. Due to the intelligent default behavior of the tags, you can almost always let the tags decide when to interpret your attributes as OGNL expressions and when to interpret them as string literals. This is another way that the framework eases the common tasks.

We recognize that some of this may be abstract at this point. Believe us, knowing string and nonstring attributes will make learning the tags much, much easier. Besides, we're now ready to look at the tags themselves, complete with plenty of sample code.

6.3 *Data tags*

The first tags we'll look at are the data tags. Data tags let you get data out of the ValueStack or place variables and objects onto the ValueStack. In this section, we'll discuss the property, set, push, bean, and action tags. In this reference, our goal is to demonstrate the common uses of these tags. Many of the tags have further functionality for special cases. To find out everything there is to know, consult the primary documentation on the Struts 2 website at <http://struts.apache.org/2.x/>.

ALERT All of the tag usage examples found in the following reference section can be found in the chapter 6 version of the Struts 2 Portfolio sample application.

Most of these examples use the same action class implementation, `manning.chapter-Six.TagDemo`. This simple action conducts no real business logic. It merely exposes and populates a couple of properties with data so the tags have something with which to work. In particular, two properties, `users` and `user`, are exposed; the first exposes a collection of all users in the system and the second exposes one user of your choice. For the purposes of a tag reference, we won't try to integrate these tag demonstrations into the core functionality of the Struts 2 Portfolio. We'll focus on making the usage clear.

6.3.1 *The property tag*

The property tag provides a quick, convenient way of writing a property into the rendering HTML. Typically, these properties will be on the ValueStack or on some other object in the ActionContext. As these properties can be of any Java type, they must be converted to strings for rendering in the result page. This conversion is handled by the framework's type converters, which we covered in chapter 5. If a specific type has no converter, it'll typically be treated as a string. In these cases, a sensible `toString()` method should be exposed on the class. Table 6.2 summarizes the most important attributes.

Table 6.2 property tag attributes

Attribute	Required	Default	Type	Description
value	No	<top of stack>	Object	Value to be displayed
default	No	null	String	Default value to be used if value is null
escape	No	True	Boolean	Whether to escape HTML

Now, let's look at a property tag in action. As with all the examples in this chapter, you can see this in action by clicking the property tag link on the chapter 6 home page of the sample application. The following tag accesses the user property exposed on the ValueStack via our TagDemo action:

```
<h4>Property Tag</h4>
The current user is <s:property value="user.username"/>.
```

The output is as you'd expect and is shown in figure 6.4.

In this case, the user property holds a user with the username of mary. When the property tag pulls the property out to render, it'll be converted to a string based on the appropriate type converters. While this property was a Java String, it still must be formally converted to a text string in the rendering page. See chapter 5 for more on the type conversion process.



Figure 6.4 Pulling the username into the page with the property tag

6.3.2 The set tag

In the context of this tag, *setting* means assigning a property to another name. Various reasons for doing this exist. An obvious use case would be taking a property that needs a deep, complicated OGNL expression to reference it, and reassigning, or setting, it to a top-level name for easier, faster access. This can make your JSPs faster and easier to read.

You can also specify the location of the new reference. By default, the property becomes a named object in the ActionContext, alongside the ValueStack, session map, and company. This means you can then reference it as a top-level named object with an OGNL expression such as `#myObject`. However, you can also specify that the new reference be kept in one of the scoped maps that are kept in the ActionContext. Table 6.3 provides the attributes for the set tag.

Table 6.3 set tag attributes

Attribute	Required	Type	Description
name	Yes	String	Reference name of the variable to be set in the specified scope.
scope	No	String	application, session, request, page, or action. Defaults to action.
value	No	Object	Expression of the value you wish to set.

Here's an example from the chapter 6 sample code:

```
<s:set name="username" value="user.username"/>
Hello, <s:property value="#username"/>. How are you?
```

In this case, we aren't saving much by making a new reference to the username property. However, it illustrates the point. In this sample, the set tag sets the value from the `user.username` expression to the new reference specified by the name property. Since

we don't specify a scope, this new `username` reference exists in the default "action" scope—the `ActionContext`. As you can see, we then reference it with the `#` operator. Figure 6.5 shows the output.

In case you're wondering what it looks like to set the new reference to a different scope, the following sets the new reference as an entry in the application scope map that is found in the `ActionContext`:

```
<s:set name="username" scope="application" value="user.username"/>
Hello, <s:property value="#application['username']"/>. How are you?
```

Note that we have to use the OGNL map syntax to get at the property in this case. We can't say we've made any readability gains here, but we have managed to persist the data across the lifetime of the application by moving it to this map. It's probably not a good idea to persist a user's username to the application scope, but it does serve to demonstrate the tag functionality.

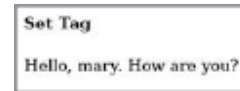


Figure 6.5 Using the `set` tag to make data available throughout the page

6.3.3 The push tag

Whereas the `set` tag allows you to create new references to values, the `push` tag allows you to push properties onto the `ValueStack`. This is useful when you want to do a lot of work revolving around a single object. With the object on the top of the `ValueStack`, its properties become accessible with first-level OGNL expressions. Any time you access properties of an object more than a time or two, it'll probably save a lot of work if you push that object onto the stack. Table 6.4 provides the attribute for the `push` tag.

Table 6.4 `push` tag attributes

Attribute	Required	Type	Description
value	Yes	Object	Value to push onto the stack

Here's an example of the usage:

```
<s:push value="user">
  This is the "<s:property value="portfolioName"/>" portfolio,
  created by none other than <s:property value="username"/>
</s:push>
```

This `push` tag pushes a property named `user`, which is exposed by the `TagDemo` action as a `JavaBeans` property, onto the top of the `ValueStack`. With the `user` on top of the stack, we can access its properties as top-level properties of the `ValueStack` virtual object, thus making the OGNL much simpler. As you can see, the `push` tag has a start tag and close tag. Inside the body of the tag, we reference the properties of the `user` object as top-level properties on the `ValueStack`. The closing tag removes the `user` from the top of the `ValueStack`.

Tip

The `push` tag, and even the `set` tag to a limited extent, can be powerful when trying to reuse view-layer markup. Imagine you have a JSP template that you'd like to reuse across several JSP result pages. Consider the namespace of the OGNL references in that JSP template. For instance, maybe the template's tags use OGNL references that assume the existence of a `User` object exposed as a model object, as in `ModelDriven` actions. In this case, the template's tags would omit the `user` property and refer directly to properties of the user, for example `<s:property value="username"/>`.

If you try to include this template in the rendering of a result whose action exposes a `User` object as a JavaBeans property, rather than a model object, then this reference would be invalid. It would need to be `<s: value="user.username"/>`. Luckily, the `push` tag gives us the ability to push the user object itself to the top of the `ValueStack`, thus making the top-level references of the template valid in the current action. In general, the `push` tag and the `set` tag can be used in this fashion.

If you want to see how this example actually renders, you can check out the sample code. But we think you'll find no surprises.

6.3.4 The bean tag

The bean tag is like a hybrid of the `set` and `push` tags. The main difference is that you don't need to work with an existing object. You can create an instance of an object and either push it onto the `ValueStack` or set a top-level reference to it in the `ActionContext`. By default, the object will be pushed onto the `ValueStack` and will remain there for the duration of the tag. In other words, the bean will be on the `ValueStack` for the execution of all tags that occur in between the opening and closing tags of the bean tag. If you want to persist the bean longer than the body of the tag, you can specify a reference name for the bean with the `var` attribute. This reference will then exist in the `ActionContext` as a named parameter accessible with the `#` operator for the duration of the request.

There are a few requirements on the object that can be used as a bean. As you might expect, the object must conform to JavaBeans standards by having a zero-argument constructor and JavaBeans properties for any instance fields that you intend to initialize with `param` tags. We'll demonstrate all of this, including the use of `param` tags, shortly. First, table 6.5 details the attributes for the bean tag.

Table 6.5 bean tag attributes

Attribute	Required	Type	Description
name	Yes	String	Package and class name of the bean that is to be created
var	No	String	Variable name used if you want to reference the bean outside the scope of the closing bean tag

Our first example demonstrates how to create and store the bean as a named parameter in the `ActionContext`. In this case, we'll create an instance of a utility bean that helps us simulate a for loop. This counter bean comes with Struts 2. For this example, we'll create the bean and use the `var` attribute to store it in the `ActionContext` as a named parameter. The following markup shows how this is done:

```
<s:bean name="org.apache.struts2.util.Counter" var="counter">
  <s:param name="last" value="7"/>
</s:bean>

<s:iterator value="#counter">
  <li><s:property/></li>
</s:iterator>
```

Figure 6.6 how this markup will render in the result page.

Now let's look at how it works. The bean tag's name attribute points to the class that should be instantiated. The `var` attribute, repeating a common Struts 2 tag API pattern, specifies the reference name under which the bean will be stored in the `ActionContext`. In this case, we call the bean counter and then refer to that bean instance in the iterator tag's value attribute with the appropriate OGNL. Since the bean is in the `ActionContext`, rather than on the `ValueStack`, we need to use the `#` operator to name it, resulting in the OGNL expression `#counter`. The bean tag is the first of a few tags we'll explore that are parameterized. In the case of the counter, we can pass in a parameter that sets the number of elements it will contain, in effect setting the number of times the iterator tag will execute its body markup.

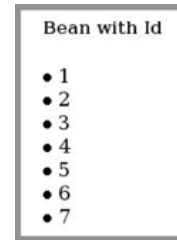


Figure 6.6 Declaring a bean that you can use throughout a page

TIP The `var` attribute occurs in the usage of many Struts 2 tags. Any tag that creates an object, the bean tag being a good example, offers the `var` attribute as a way to store the object under a name in the `ActionContext`. The name comes from the value given to the `var` attribute. Most tags that offer the `var` attribute make it optional; if you don't want to store the created object in the `ActionContext`, it will simply be placed on the top of the `ValueStack`, where it'll remain during the body of the tag.

WARNING! If you're using a version of Struts 2 that is older than 2.1, the `var` attribute should be replaced with the `id` attribute.

Now that the counter bean has been created and stored, we can use it from the Struts 2 iterator tag to create a simulation of for loop-style logic. The bean tag doesn't have to be used with the iterator tag; it's in this example because the counter bean is meant to be used with the iterator tag. The counter bean works in combination with the iterator tag, which we'll cover shortly, to provide a pseudo for loop functionality. Generally, the iterator tag iterates over a `Collection`, thus its number of iterations is based upon the number of elements in the `Collection`. For a for loop, we want to specify a number of iterations without necessarily providing a set of objects. We just

want to iterate over our tag's body a certain number of times. The counter bean serves as a fake `Collection` of a specified number of dummy objects that allows us to control the number of iterations. In the case of our example, we do nothing more than print a number to the result stream during each iteration. Note that we use the property tag without any attributes; this idiom will simply write the top property on the `ValueStack` to the output.

NOTE The bean tag allows you to create any bean object that you might want to use in the page. If you want to make your own bean to use with this tag, just remember that it needs to follow JavaBeans conventions on several important points. It has to have no-argument constructor, and it must expose JavaBeans properties for any parameters it'll receive with the param tag, such as the counter bean's last parameter.

Now, let's look at how to use the bean tag to push a newly created bean onto the `ValueStack` rather than store it in the `ActionContext`. While we're at it, we'll further demonstrate the use of the param tag to pump parameters into our home-roasted bean. This is all simple. To use the `ValueStack` as the temporary storage location for our bean, we just use an opening-and-closing-style tag configuration. All tags inside the body of the bean tag will resolve against a `ValueStack` that has an instance of our `JokeBean` on top. Here's the example:

```
<s:bean name="manning.utils.JokeBean" >
  <s:param name="jokeType">knockknock</s:param>
  <s:property value="startAJoke()" />
</s:bean>
```

In this example, also from the chapter 6 sample code, we create an instance of a utility bean that helps us create jokes. If you look at the sample application, you'll see that this outputs the first line of a joke—"knock knock." Though inane, this bean does demonstrate the sense of utility beans. If you want to provide a canned joke component to drop into numerous pages, something that unfortunately does exist in the real world, you could embed that functionality into a utility bean and grab it with the bean tag whenever you liked. This keeps the joke logic out of the core logic of the action logic.

This markup demonstrates using the bean tag to push the bean onto the `ValueStack` rather than place it as a named reference in the `ActionContext`. Note that we no longer need to use the `var` attribute to specify the reference under which the bean will be stored. When it's on top of the `ValueStack`, we can just refer to its properties and methods directly. This makes our code concise. The bean is automatically popped from the stack at the close tag.

Using the bean is easy. In this case, we use the OGNL method invocation syntax, `startAJoke()`. We do this just to demonstrate that the bean tag doesn't have to completely conform to JavaBeans standards—`startAJoke()` is clearly not a proper getter. Nonetheless, OGNL has the power to use it; we'll cover this and more in the primer at the end of this chapter.

Finally, note that we pass a parameter into our `JokeBean` that controls the type of joke told by the bean. This parameter is automatically received by our bean as long as the bean implements a `JavaBeans` property that matches the name of the parameter. If you look at the source code, you can see that we've done this. FYI: this joke bean also supports an "adult" joke mode, but you'll probably be disappointed; it's quite innocuous.

The bean tag is ultimately straightforward. What you want to be clear about is the difference between the use of the `var` attribute to create a named reference in the `ActionContext`, and the use of the opening and closing tags to work with the bean on the `ValueStack`. The real trick here is in understanding the `ValueStack`, `ActionContext`, and how `OGNL` gets to them. That's why we began this chapter with a thorough introduction to these concepts. If you're confused, you might want to refer back to those earlier sections.

With those fundamental concepts in place, the conventions of the bean tag, and other similar tags, should be straightforward enough. If you have all this straight, congratulate yourself on a mastery of what some consider to be the most Byzantine aspect of Struts 2.

6.3.5 The action tag

This tag allows us to invoke another action from our view layer. Use cases for this might not be obvious at first, but you'll probably find yourself wanting to invoke secondary actions from the result at some point. Such scenarios might range from integrating existing action components to wisely refactoring some action logic. The practical application of the action tag is simple: you specify another action that should be invoked. Some of the most important attributes of this tag include the `executeResult` attribute, which allows you to indicate whether the result for the secondary action should be written into the currently rendering page, and the name and namespace attributes, by which you identify the secondary action that should fire. By default, the namespace of the current action is used. Table 6.6 contains the details of the important attributes.

Table 6.6 action tag attributes

Attribute	Required	Type	Description
name	Yes	String	The action name
namespace	No	String	The action namespace; defaults to the current page namespace
var	No	String	Reference name of the action bean for use later in the page
executeResult	No	Boolean	When set to true, executes the result of the action (default value: false)

Table 6.6 action tag attributes (*continued*)

Attribute	Required	Type	Description
flush	No	Boolean	When set to true, the writer will be flushed upon end of action component tag (default value: true)
ignoreContextParams	No	Boolean	When set to true, the request parameters are not included when the action is invoked (default value: false)

Here's an example that chooses to include the secondary action's result:

```
<h3>Action Tag</h3>
<h4>This line is from the ActionTag action's result.</h4>
<s:action name="TargetAction" executeResult="true"/>
```

Note that the default is to not include the result, so we have to change this by setting the `executeResult` attribute to true. The output looks like figure 6.7.

One thing to note is that the result of the secondary action should probably be an HTML fragment if you want it to fit into the primary page.

Often, you might want the secondary action to fire, but not write a result. One common scenario is that the secondary action, instead of writing to the page, will produce side effects by stashing domain data somewhere in the `ActionContext`. After control returns, the primary action can access that data. The following markup shows how to target an action in this fashion:

```
<h4>This line is before the ActionTag invokes the secondary action.</h4>
<s:action name="TargetAction"/>
<h4>Secondary action has fired now.</h4>
<h5>Request attribute set by secondary action = </h5>
<pre> <s:property value="#request.dataFromSecondAction"/></pre>
```

The execution of the secondary action is a bit of a side effect unless we reach back to get something that it produced. We retrieve a property that was set by the secondary action into the request map, just to prove that the secondary action fired. You can check the output yourself by visiting the chapter 6 sample code. Many times, however, a side effect may be just what you want. Note also that the secondary action can receive, or not receive, the request parameters from the primary request, according to the `ignoreContextParams` attribute.

That finishes up our coverage of data tags. In the next section, we'll show how to introduce conditional logic to your page rendering with the control tags.

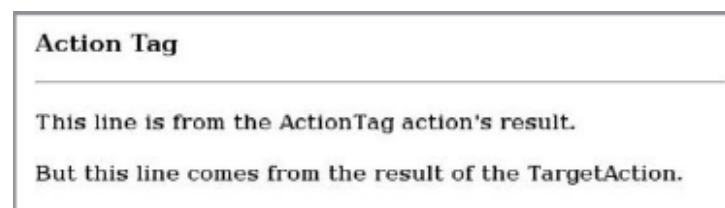


Figure 6.7 Using the action tag to invoke another action from the context of a rendering page

6.4 Control tags

Since most web pages are built on the fly, it's going to be valuable to learn how to manipulate, navigate over, and display data. Struts 2 has a set of tags that make it easy to control the flow of page execution. Using the iterator tag to loop over data and the if/else/elseif tags to make decisions, you can leverage the power of conditional rendering in your pages.

6.4.1 The iterator tag

Other than the property tag, the other most commonly used tag in Struts 2 is the iterator tag. The iterator tag allows you to loop over collections of objects easily. It's designed to know how to loop over any Collection, Map, Enumeration, Iterator, or array. It also provides the ability to define a variable in the ActionContext, the iterator status, that lets you determine certain basic information about the current loop state, such as whether you're looping over an odd or even row. Table 6.7 provides the attributes for the iterator tag.

Table 6.7 iterator tag attributes

Attribute	Required	Type	Description
value	Yes	Object	The object to be looped over.
status	No	String	If specified, an IteratorStatus object is placed in the action context under the name given as the value of this attribute.

We already saw the iterator tag in action when we looked at the bean tag. Now we'll take a closer look. The chapter 6 sample application includes an example that loops over a set of the Users of the Struts 2 Portfolio. Here's the markup from the result page:

```
<s:iterator value="users" status="itStatus">
  <li>
    <s:property value="#itStatus.count" />
    <s:property value="portfolioName"/>
  </li>
</s:iterator>
```

As you can see, it's straightforward. The action object exposes a set of users and the iterator tag iterates over those users. During the body of the tag, each user is in turn placed on the top of the ValueStack, thus allowing for convenient access to the user's properties. Note that our iterator also declares an IteratorStatus object by specifying the status attribute. Whatever name you give this attribute will be the key for retrieving the iterator status object from the ActionContext, with an OGNL expression such as #itStatus. In this example, we use the iterator status's count property to get a sequential list of our users. The output is shown in figure 6.8.

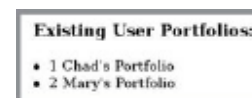


Figure 6.8 Iterating over a set of data during page rendering

We should probably take a minute to see what else the `IteratorStatus` can provide for us.

USING ITERATORSTATUS

Sometimes it's desirable to know status information about the iteration that's taking place. This is where the status attribute steps in. The status attribute, when defined, provides an `IteratorStatus` object available in the `ActionContext` that can provide simple information such as the size, current index, and whether the current object is in the even or odd index in the list. The `IteratorStatus` object can be accessed through the name given to the status attribute. Table 6.8 summarizes the information that can be obtained from the `IteratorStatus` object.

As you can see, this list provides just the kind of data that can sometimes be hard to come by when trying to produce various effects within JSP page iterations. Happy iterating!

Table 6.8 Public methods of `IteratorStatus`

Method name	Return type
<code>getCount</code>	<code>int</code>
<code>getIndex</code>	<code>int</code>
<code>isEven</code>	<code>boolean</code>
<code>isFirst</code>	<code>boolean</code>
<code>isLast</code>	<code>boolean</code>
<code>isOdd</code>	<code>boolean</code>
<code>modulus(int operand)</code>	<code>int</code>

6.4.2 The `if` and `else` tags

Not many languages, of any sort, fail to provide the familiar `if` and `else` control logic. The `if` and `else` tags provide these familiar friends for the Struts 2 developer. Using them is as easy as you might suspect. As you can see in table 6.9, there's just one attribute, a Boolean test.

Table 6.9 `if` and `elseif` tag attribute

Attribute	Required	Type	Description
<code>test</code>	Yes	Boolean	Boolean expression that is evaluated and tested for true or false

Here's an example of using them. You can put any OGNL expression you like in the test.

```
<s:if test="user.age > 35">This user is too old.</s:if>
<s:elseif test="user.age < 35">This user is too young</s:elseif>
<s:else>This user is just right</s:else>
```

Here we conduct a couple of tests on a user object exposed by our action and, ultimately, found on the `ValueStack`. The tests are simple Boolean expressions; you can chain as many of the tests as you like.

That was easy enough. Indeed, the `if` and `else` tags are as simple as they seem, and remain that way in use. We still have a few useful tags to cover, and we'll hit them in the next section, which covers miscellaneous tags.

6.5 *Miscellaneous tags*

As we mentioned at the start of this chapter, Struts 2 includes a few different types of tags. You’ve already seen how the data tags and control tags work. Let’s now look at the miscellaneous tags that, although useful, can’t be easily classified. In this section, we’ll discuss the Struts 2 `include` tag (a slight variation on the `<jsp:include>` tag), the `URL` tag, and the `i18n` and `text` tags (both used for internationalization). Finally, we’ll take another look at the `param` tag, which you’ve already seen in the context of the `bean` tag, and show how it can be used to its full power.

6.5.1 *The include tag*

Whereas JSP has its own `include` tag, `<jsp:include>`, Struts 2 provides a version that integrates with Struts 2 better and provides more advanced features. In short, this tag allows you to execute a Servlet API–style include. This means you can include the output of another web resource in the currently rendering page. One good thing about the Struts 2 `include` tag is that it allows you to pass along request parameters to the included resource.

This differs from the previously seen `action` tag, in that the `include` tag can reference any servlet resource, while the `action` tag can include only another Struts 2 action within the same Struts 2 application. This inclusion of an action stays completely within the Struts 2 architecture. The `include` tag can go outside of the Struts 2 architecture to retrieve any resource available to the web application in which the Struts 2 application is deployed. This generally means grabbing other servlets or JSPs. The `include` tag may not make a lot of sense unless you’re pretty familiar with the Servlet API. Again, the Servlet Specification is recommended reading: <http://java.sun.com/products/servlet/download.html>.

Table 6.10 lists the sole attribute for the `include` tag.

Table 6.10 `include` tag attribute

Attribute	Required	Type	Description
value	Yes	String	Name of the page, action, servlet, or any referenceable URL

We won’t show a specific example of the `include` tag, as its use is straightforward. When using the `include` tag, you should keep in mind that you’re including a JSP, servlet, or other web resource directly. The semantics of including another web resource come from the Servlet API. The `include` tag behaves similarly to the JSP `include` tag. However, it’s more useful when you’re developing with Struts 2, for two reasons: it integrates better with the framework, and it provides native access to the `ValueStack` and a more extensible parameter model. What does all of this mean?

Let’s start with the framework integration. For example, your tag can dynamically define the resource to be included by pulling a value from the `ValueStack` using the `%{ ... }` notation. (You have to force OGNL evaluation here, as the `value` attribute is of type `String` and would normally be interpreted as a string literal.) Similarly, you

can pass in querystring parameters to the included page with the `<s:param>` tag (discussed in a moment). This tag can also pull values from the `ValueStack`. This tight integration with the framework makes the Struts 2 include tag a powerful choice.

Another advantage to choosing the Struts 2 include tag over the native JSP version is plain-old user-friendliness. For example, it'll automatically rewrite relative URLs for you. If you want to include the URL `../index.jsp`, you're free to do so even though some application servers don't support that type of URL when using the JSP include tag. The Struts 2 include tag will rewrite `../index.jsp` as an absolute URL based on the current URL where the JSP is located.

6.5.2 The URL tag

When you're building web applications, URL management is a central task. Struts 2 provides a URL tag to help you do this. The tag supports everything you might want to do with a URL, from controlling parameters to automatically persisting sessions in the absence of cookies. Table 6.11 lists its attributes.

Table 6.11 URL tag attributes

Attribute	Required	Type	Description
Value	No	String	The base URL; defaults to the current URL the page is rendering from.
action	No	String	The name of an action to target with the generated URL; use the action name as configured in the declarative architecture (without the <code>.action</code> extension).
var	No	String	If specified, the URL isn't written out but rather is saved in the action context for future use.
includeParams	No	String	Selects parameters from all, get, or none; default is <code>get</code> .
includeContext	No	Boolean	If true, then the URL that is generated will be prepended with the application's context; default is <code>true</code> .
encode	No	Boolean	Adds the session ID to the URL if cookies aren't enabled for the visitor.
scheme	No	String	Allows you to specify the protocol; defaults to the current scheme (HTTP or HTTPS).

Here are a couple of examples. First we look at a simple case:

```
URL = <s:url value="IteratorTag.action"/>
<a href='<s:url value="IteratorTag.action" />'> Click Me </a>
```

And here's the output markup:

```
URL = IteratorTag.action
<a href='IteratorTag.action'> Click Me </a>
```

The `URL` tag just outputs the generated URL as a string. First we display it for reference. Then we use the same markup to generate the `href` attribute of a standard anchor tag. Note that we set the target of the URL with the `value` attribute. This means we must include the `.action` extension ourselves. If we want to target an action, we should probably use the `action` attribute, as seen in the next example:

```
URL =      <s:url action="IteratorTag" var="myUrl">
           <s:param name="id" value="2"/>
           </s:url>
           <a href='<s:property value="#myUrl" />'> Click Me </a>
```

Now, let's see the markup generated by these tags:

```
URL =
<a href='/manningHelloWorld/chapterSix/IteratorTag.action?id=2'>
  Click Me
</a>
```

As you can see, the `URL` tag didn't generate any output in this example. This happened because we used the `var` attribute to assign the generated URL string to a reference in the `ActionContext`. This helps improve the readability of the code. In this example, our `URL` tag, with its `param` tags, has become unwieldy to embed directly in the anchor tag. Now we can just pull the URL from the `ActionContext` with a `property` tag and some `OGNL`. This is also useful when we need to put the URL in more than one place on the page.

The `param` tag used in this example specifies `querystring` parameters to be added to the generated URL. You can see generated `querystring` in the output. Note that you can use the `includeParams` attribute to specify whether parameters from the current request are carried over into the new URL. By default this attribute is set to `get`, which means only `querystring` params are carried over. You can also set it to `post`, which causes the posted form parameters to also be carried over. Or you can specify `none`.

6.5.3 *The `i18n` and text tags*

Many applications need to work in multiple languages. The process of making this happen is called *internationalization*, or *i18n* for short. (There are 18 letters between the I and the N in the word internationalization.) Chapter 11 discusses Struts 2's internationalization support in detail, but we'd like to take a moment to detail the two tags that are central to this functionality: the `i18n` tag and the `text` tag.

The `text` tag is used to display language-specific text, such as English or Spanish, based on a key lookup into a set of text resources. This tag retrieves a message value from the `ResourceBundles` exposed through the framework's own internationalization mechanisms. We'll explain this in greater detail in chapter 11. For now, we'll just note the usage of the tag; it takes a `name` attribute that specifies the key under which the message retrieval should occur. The framework's default `Locale` determination will determine the `Locale` under which the key will be resolved. Consult chapter 11 for full details on where to place properties files to make their resources available to this tag.

Table 6.12 lists the attributes that the text tag supports.

Table 6.12 text tag attributes

Attribute	Required	Type	Description
name	Yes	String	The key to look up in the ResourceBundle(s).
var	No	String	If specified, the text is stored in the action context under this name.

You can also name an ad hoc ResourceBundle for resolving your text tags. If you want to manually specify the ResourceBundle that should be used, you can use the `i18n` tag. Table 6.13 lists the attributes of the `i18n` tag.

Table 6.13 i18n tag attribute

Attribute	Required	Type	Description
name	Yes	String	The name of the ResourceBundle

Here’s a quick example that shows how to set the ResourceBundle with the `i18n` tag and then extract a message text from it with the text tag:

```
<s:i18n name="manning.chapterSix.myResourceBundle_tr">
  In <s:text name="language"/>,
  <s:text name="girl" var="foreignWord"/>
</s:i18n>

"<s:property value="#foreignWord"/>" means girl.
```

Figure 6.9 shows the output.

The `i18n` tag simply specifies a resource bundle to use. The bundle is only used during the body of the tag. However, as our example demonstrates, you can “persist” a message from the bundle using the text tag’s `var` attribute to set the message to the `ActionContext` as a named reference. This usage of the `var` attribute should be more familiar by now. The first text tag writes the message associated with the key “language” directly to the output. The second text tag stores the message associated with the key “girl” under the reference name “foreignWord”.

These tags are simple, but seem out of context without full knowledge of the built-in internationalization features of Struts 2. In particular, you’ll need to know how the framework locates, loads, and uses properties file ResourceBundles. We’ll cover this in detail in chapter 11.

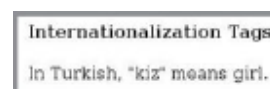


Figure 6.9 Using internationalization tags to pull text from ResourceBundles

6.5.4 The param tag

The last tag we'll discuss has already been used throughout this chapter. The `param` tag does nothing by itself, but at the same time it's one of the more important tags. It not only serves an important role in using many of the tags covered in this chapter, it'll also play a role in many of the UI component tags, as you'll see in chapter 7. Table 6.14 lists the attributes you're now familiar with.

Table 6.14 `param` tag attributes

Attribute	Required	Type	Description
name	No	String	The name of the parameter
value	No	Object	The value of the parameter

Using the `param` tag has already been established in this chapter. In particular, our coverage of the `bean` tag showed a couple of use cases for the `param` tag, including as a means for passing parameters into your own custom utility objects. As long as you have the general idea, it's just a matter of perusing the APIs to see which tags can take parameters. Toward this end, it's always a good idea to consult the online documentation of the Struts 2 tags to see if a given tag can take a parameter of some sort. For one, this book doesn't attempt to be exhaustive in its coverage of the tags. Additionally, the tag API is always being improved and expanded. Struts 2 is a new framework and growth is rapid.

That's all of the general-use Struts 2 tags that we'll cover. Chapter 7 will cover the UI component tags, which allow you to quickly develop rich user interfaces in your pages. Before that, we'll touch on a couple of advanced topics that pertain to tag usage. First, we'll comment on the use of native tags from the various view technologies. After that, we'll provide a short OGNL expression language primer to empower your tag usage.

6.6 Using JSTL and other native tags

What if you want to use the native tags and expression languages of your chosen view-layer technology? That's fine, too. While the Struts 2 tags provide a high-level tag API that can be used across all three view technologies that the framework supports out of the box, you can use the tags and macros provided natively by each of those technologies if you wish. The JSTL, for instance, is still available in your JSP pages. We don't cover the JSTL in this book and assume that, if you have enough reason to use JSTL instead of the built-in Struts 2 tags, you probably already know quite a bit about the JSTL. We'll just note that the result types that prepare the environment for JSP, Velocity, and FreeMarker rendering do make the `ValueStack` and other key Struts 2 data objects available to the native tags of each technology. Bear in mind that the exposure of the Struts 2 objects to those native tags and ELs may not be consistent, due to the differences between those technologies.

Next up, the OGNL expression language details we've been promising for oh, so long.

6.7 A brief primer for the OGNL expression language

Throughout Struts 2 web applications, a need exists to link the Java runtime with the text-based world of HTML, HTTP, JSP, and other text-based view-rendering technologies. There must be a way for these text-based documents to reference runtime data objects in the Java environment. A common solution to this problem is the use of expression languages. As we've seen, Struts 2 uses OGNL for this purpose. We'll now take the opportunity to cover the features of this expression language that you'll most likely need to use in Struts 2 development.

Before we get too far, we should point out that this section could easily be skipped. For most use cases, you've probably already learned enough OGNL expression language to get by. You could treat this section as a rainy-day reference if you like. On the other hand, you'll probably find some stuff you can use immediately. It's your choice.

6.7.1 What is OGNL?

The Object-Graph Navigation Language exists as a mature technology completely distinct from Struts 2. As such, it has purposes and features much larger than its use within Struts 2. OGNL is an expression and binding language. In Struts 2, we use the OGNL expression language to reference data properties in the Java environment, and we use OGNL type converters to manage the type conversion between HTTP string values and the typed Java values.

In this last section, we'll try to summarize the syntax and some of the more useful features of the OGNL expression language. First we'll cover the syntax and features most commonly used in Struts 2 development. Then we'll cover some of the other OGNL features that you might find handy. OGNL has many of the features of a full programming language, so you'll find that most everything is possible. Also note that this section doesn't try to be a complete reference to OGNL. If you want more OGNL power, visit the website at www.ognl.org for more information.

WARNING Keep those JSPs clean! While OGNL has much of the power of a full-featured language, you might want to think twice before squeezing the trigger. It's a well-established best practice that you should keep business logic out of your pages. If you find yourself reaching for the OGNL power tools, you might well be pulling business logic into your view layer. We're not saying you can't do it, but we recommend giving it a moment's thought before complicating your view pages with too much code-style logic. If you're getting complex in your OGNL, ask yourself if what you're doing should be done in the action or, at least, encapsulated in a helper bean that you can use in your page.

6.7.2 Expression language features commonly used in Struts 2

First, we should review the most common uses of the OGNL expression language in Struts 2 development. In this section, we'll look at how the expression language serves its purpose for most daily development. Basically, we use it to map the incoming data

onto your `ValueStack` objects, and we use it in tags to pull the data off of the `ValueStack` while rendering the view. Let's look at the expression language features most commonly used in this work.

REFERENCING BEAN PROPERTIES

First of all, we need to define what makes an expression. The OGNL expression language refers to something called a *chain of properties*. This concept is simple. Take the following expression:

```
person.father.father.firstName
```

This property chain consists of a chain of four properties. We can say that this chain references, or targets, the `firstName` property of `person`'s grandfather. You can use this same reference both for setting and getting the value of this property, depending on your context.

SETTING OR GETTING?

When we use OGNL expressions to name our form input parameters, we're referring to a property that we'd like to have set for us. The following code snippet shows the form from our Struts 2 Portfolio application's `Registration.jsp` page:

```
<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:submit/>
</s:form>
```

The name of each input field is an OGNL expression. These expressions refer to, for example, the `username` property exposed on the root OGNL object. As we've just learned, the root object is our `ValueStack`, which probably contains our action object and perhaps a model object. When the `params` interceptor fires, it'll take this expression and use it to locate the property onto which it should set the value associated with this name. It'll also use the OGNL type converters to convert the value from a string to the native type of the target property.

There is one common complication that arises when the framework moves data onto the properties targeted by the OGNL expressions. Take the deeper expression:

```
user.portfolio.name
```

If a request parameter targets this property, its value will be moved onto the `name` property of the `portfolio` object. One problem that can occur during runtime is a null value for one of the intermediate properties in the expression chain. For instance, what if the user hasn't been created yet? If you recall, we've been omitting initialization for many of our properties in our sample code. Luckily, the framework handles this.

When the framework finds a null property in a chain that it needs to navigate, it'll attempt to create a new instance of the appropriate type and set it onto the property. However, this requires two things on the developer's part. First, the type of the

property must be a class that conforms to the JavaBeans specification, in that it provides a no-argument constructor. Without this, the framework can't instantiate an object of the type. Next, the property must also conform to the JavaBeans specification by providing a setter method. Without this setter, the framework would have no way of injecting the new object into the property. Keep these two points in mind and you'll be good to go.

In addition to targeting properties onto which the framework should move incoming data, we also use OGNL when the data leaves the framework. After the request is processed, we use the same OGNL expression to target the same property from a Struts 2 tag. Recall that the domain model data stays on the ValueStack from start to finish. Thus, tags can read from the same location that the interceptors write. The following snippet from the RegistrationSuccess.jsp page shows the property tag doing just this:

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="portfolioName" /> Portfolio</h3>
```

In this snippet, we see that the Struts 2 property tag takes an OGNL expression as its value attribute. This expression targets the property from which the property tag will pull the data for its rendering process, a simple process where it merely converts the property to a string and writes it into the page.

As you can see, OGNL expressions, as commonly used in Struts 2, serve as pointers to properties. Whether the use case is writing to or reading from that property is up to the context. Though not nearly as common, you can also use the fuller features of the OGNL expression language, operators in particular, to write self-contained expressions that, for instance, set the data on a property themselves. But, as this is outside of the normal Struts 2 use case, we'll only discuss such features in the advanced section.

WORKING WITH JAVA COLLECTIONS

Java Collections are a mainstay of the Java web developer's daily workload. While the JavaBeans specification has always supported indexed properties, working with actual Java Collections, while convenient in the Java side, has always been a hassle in contexts such as JSP tags. One of the great things about the OGNL expression language is its simplified handling of Collections. We've already seen this in action when we demonstrated the automatic type conversion to and from complex Collection properties in the previous chapter. We'll now summarize the OGNL syntax used to reference these properties.

WORKING WITH LISTS AND ARRAYS

References to lists and arrays share the same syntax in OGNL. Table 6.15 summarizes the basic syntax to access list or array properties. To see these in action, refer back to the code samples from chapter 5 that demonstrated type conversion to and from lists and arrays.

As table 6.15 demonstrates, the syntax for referencing elements or properties of lists and arrays is intuitive. Basically, OGNL uses array index syntax for both. This makes perfect sense, due to the ordered, indexed nature of lists.

Table 6.15 OGNL expression language syntax for referencing elements of array and list properties

Java code	OGNL expression
<code>list.get(0)</code>	<code>list[0]</code>
<code>array[0]</code>	<code>array[0]</code>
<code>((User) list.get(0)).getName()</code>	<code>list[0].name</code>
<code>array.length</code>	<code>array.length</code>
<code>list.size()</code>	<code>list.size</code>
<code>list.isEmpty()</code>	<code>list.isEmpty</code>

A couple of things warrant remarks. First, the reference to the name property of a list element assumes something important. As we know, Java Lists are type-agnostic. In Java, we always have to cast the element to the appropriate type, in this case `User`, before we try to reference the name property. We can omit this in OGNL if we take the time to specify the `Collection` element type as we learned earlier in this chapter. This syntax assumes that has been done. We should also note that you can reference other properties, such as `length` and `size`, of arrays and lists. In particular, note that OGNL makes the `List` class's non-JavaBeans-conformant `size` method answer to a simple property reference. This is something nice that OGNL provides as free service to its valued customers!

OGNL also allows you to create `List` literals. This can be useful if you want to directly create a set of values to feed to something like a select box. Table 6.16 shows the syntax for creating these literals.

Table 6.16 Creating a list dynamically in OGNL

Java code	OGNL expression
<pre>List list = new ArrayList(3); list.add(new Integer(1)); list.add(new Integer(3)); list.add(new Integer(5)); return list;</pre>	<code>{1,3,5}</code>

You probably only want to do this with trivial data, since creating complex data in the view layer would make a mess. Nonetheless, sometimes this will be the perfect tool for the job.

WORKING WITH MAPS

OGNL also makes referencing properties and elements of maps delightfully simple. Table 6.17 shows a variety of syntax idioms for referencing Map elements and properties.

Table 6.17 OGNL expression language syntax for referencing map properties

Java code	OGNL expression
<code>map.get("foo")</code>	<code>map['foo']</code>
<code>map.get(new Integer(1))</code>	<code>map[1]</code>
<code>User user = (User)map.get("userA"); return user.getName();</code>	<code>map['userA'].name</code>
<code>map.size()</code>	<code>map.size</code>
<code>map.isEmpty()</code>	<code>map.isEmpty</code>
<code>map.get("foo")</code>	<code>map.foo</code>

As you can see, you can do a lot with maps. The main difference here is that, unlike Lists, the value in the index box must be an object. If the value in the box is some sort of numeric data that would map to a Java primitive, such as an `int`, then OGNL automatically converts that to an appropriate wrapper type object, such as an `Integer`, to use as the key. If a string literal is placed in the box, that becomes a string object which will be used for the key. The last row in the table shows a special syntax for maps with strings as keys. If the key is a string, you may use this simpler, JavaBeans-style property notation.

As for other object types that you might use as a key, you ultimately have the full power of OGNL to reference objects that might serve as the key. The possibilities are beyond the capacity of the table format. Note that, as with the List syntax, the direct reference to the name property on the uncast map element depends on the configuration of the OGNL type conversion to know the specific element type of the map, as we learned in chapter 6.

You can also create Maps on the fly with the OGNL literal syntax. Table 6.18 demonstrates this flexible feature.

Table 6.18 Creating maps dynamically in OGNL

Java code	OGNL expression
<code>Map map = new HashMap(2); map.put("foo", "bar"); map.put("baz", "whazzit"); return map;</code>	<code>#{"foo": "bar", "baz": "whazzit" }</code>
<code>Map map = new HashMap(3); map.put(new Integer(1), "one"); map.put(new Integer(2), "two"); map.put(new Integer(3), "three"); return map;</code>	<code>#{ 1 : "one", 2 : "two", 3 : "three" }</code>

As you can see, the syntax for creating a Map literal is similar to that for creating a List literal. The main difference is the use of the # sign before the leading brace.

WARNING OGNL uses the # sign in a few different ways. Each is distinct. The uses are completely orthogonal, so you shouldn't be confused as long as you're alert to the fact that they're different use cases. In particular, this isn't the same use of the # sign as we saw when specifying a nonroot object from the ActionContext for an expression to resolve against. We'll also see another use of the # sign in a few moments.

Dynamic maps are especially useful for radio groups and select tags. The Struts 2 tag libraries come with special tags for creating user interface components. These will be explored in chapter 7. For now, just note that you can use literal maps to feed values into some of the UI components. If you wanted to offer a true/false selection that displays as a Yes/No choice, `#{true : 'Yes', false : 'No'}` would be the value for the `list` attribute. The value for the `value` attribute would evaluate to either true or false.

FILTERING AND PROJECTING COLLECTIONS

OGNL supports a couple of special operations that you can conduct on your collections. Filtering allows you to take a collection of objects and filter them according to some rule. For instance, you could take a set of users and filter them down to only those who're more than 20 years old. Projection, on the other hand, allows you to transform a collection of objects according to some rule. For instance, you could take a set of user objects, having both first and last name properties, and transform it into a set of String objects that combines the first and last name of each user into a single string. To clarify, filtering takes a Collection of size N and produces a new collection containing a subset of those elements ranging from size 0 to size N . On the other hand, projecting always produces a Collection with exactly the same number of elements as the original Collection; projecting produces a one-for-one result set.

The syntax for filtering is as follows:

```
collectionName.{? expression }
```

In the expression, you can use `#this` to refer to the object from the collection being evaluated. This is another distinct use of the # sign. The syntax for projection is as follows:

```
collectionName.{ expression }
```

Table 6.19 shows some examples of both of these useful operations in action.

Table 6.19 Producing new collections by filtering or projecting existing collections

OGNL expression	Description
<code>users.{?#this.age > 30}</code>	A filtering process that returns a new collection with only users who are older than 30
<code>users.{username}</code>	A projection process that returns a new collection of username strings, one for each user

Table 6.19 Producing new collections by filtering or projecting existing collections (*continued*)

OGNL expression	Description
<code>users.{firstName + ' ' + lastName}</code>	A projection process that returns a new collection of strings that represent the full name of each user
<code>users.{?#this.age > 30}. {username}</code>	A projection process that returns the usernames of a filtered collection of users older than 30

As you can see, each filtering or projection simply returns a new collection for your use. This convenient notation can be used to get the most out of a single set of data. Note that you can combine filtering and projection operations. That about covers it for aspects of OGNL that are commonly used in Struts 2. In the next section, we'll cover some of the advanced features that might help you out in a pinch, but, still, we recommend keeping it simple unless you have no choice.

6.7.3 Advanced expression language features

As we've indicated, OGNL is a full-featured expression language. In fact, its features rival that of some full-fledged programming languages. In this section, we give a brief summary of some of the advanced features that you might use in a pinch. Some of these things are basic features of OGNL, but advanced in the context of Struts 2 usage. Take our terminology with a grain of salt. Also, we'll make little effort to introduce use cases for these features. We consider their usage to be nonstandard practice. With that said, we also know that these power tools can save the day on those certain occasions that always seem to occur.

LITERALS AND OPERATORS

Like most languages, the OGNL expression language supports a wide array of literals. Table 6.20 summarizes these literals.

Table 6.20 Literals of the OGNL expression language

Literal type	Example
Char	'a'
String	'hello'
"hello"	Boolean
True	False
int	123
double	123.5
BigDecimal	123b
BigInteger	123h

The only thing out of the ordinary would be the usage of both single and double quotes for string literals. Note, however, that a string literal of a single character must use double quotes, or it'll be interpreted as a char literal. Table 6.21 shows the operators.

Table 6.21 Operators of the OGNL expression language

Operation	Example
add (+)	2 + 4 'hello' + 'world'
subtract (-)	5 - 3

Table 6.21 Operators of the OGNL expression language (*continued*)

Operation	Example
multiply (*)	8 * 2
divide (/)	9/3
modulus (%)	9 % 3
increment (++)	++foo foo++
decrement (--)	bar-- --bar
equality (==)	foo == bar
less than (<)	1 < 2
greater than (>)	2 > 1

As you can see, all the usual suspects are here. This would probably be a good time to note that the OGNL expression language also allows multiple comma-separated expressions to be linked in a single expression. The following snippet demonstrates this process:

```
user.age = 10, user.name = "chad", user.username
```

This relatively meaningless example demonstrates an expression that links three sub-expressions. As with many languages, each of the first two expressions executes and passes control on to the next expression. The value returned by the last expression is the value returned for the entire expression. Now we'll see how to invoke methods with OGNL.

CALLING METHODS

One power that many a JSP developer has wished for is the ability to call methods from the expression language. Until recently, this was rare. Actually, even the simplest property reference involves a method call. But those simple property references can invoke methods based on JavaBeans conventions. If the method you want to invoke doesn't conform to JavaBeans conventions, you'll probably need the OGNL method invocation syntax to get to it. This can sometimes get you out of a jam. It can also be useful in calling utility methods on helper beans. Table 6.22 shows how it works.

Table 6.22 Calling methods from the OGNL expression language

Java code	OGNL expression
utilityBean.makeRandomNumber()	makeRandomNumber()
utilityBean.getRandomNumberSeed()	getRandomNumberSeed() randomNumberSeed

Note that in this table we assume that a random number generator bean, named `utilityBean`, has been pushed onto the `ValueStack` prior to the evaluation of these OGNL expressions. With this bean in place, you can omit the object name in the OGNL expression, because it resolves to the `ValueStack` by default. First, we invoke the `makeRandomNumber()` method as you might expect. In the second example, we show that you can even use a full method invocation syntax to access JavaBeans properties, though you don't have to. The result is no different than when using the simpler property notation.

We should note that these method invocation features of the OGNL expression language are turned off during the incoming phase of Struts 2 data transfer. In other words, when the form input field names are evaluated by the `params` interceptor, method invocations, as well as some other security-compromising features of the expression language, are completely ignored. Basically, when the `params` interceptor evaluates OGNL expressions, it'll only allow them to point to properties onto which it should inject the parameter values. Nothing else is permitted.

ACCESSING STATIC METHODS AND FIELDS

In addition to accessing instance methods and properties, you can also access static methods and fields with the OGNL expression language. There are two ways of doing this. One requires specifying the fully qualified class name, while the other method resolves against the `ValueStack`. The syntax that takes the full class name is `@[fullClassName]@[property or methodCall]`. Here are examples of using full class names to access both a static property and a static method:

```
@manning.utils.Struts2PortfolioConstants@USER
@manning.utils.PortfolioUtilityBean@startImageWrapper()
```

Besides the `@` signs, these are no different than normal property specification or method invocation. As we said, you can forgo specifying the class name if and only if your property or method will resolve on the `ValueStack`. Here we have the same two examples, but they assume that some object on the `ValueStack` exposes what they need. The syntax replaces the class name with the `vs` symbol, which stands for `ValueStack`:

```
@vs@USER
@vs@startImageWrapper()
```

That wraps up our coverage of some of the advanced features of OGNL. You'll probably find yourself coming back to this quick reference in the future as you butt heads with some odd wall or two. Again, we recommend taking it easy on the OGNL power tools. However, we're compelled to tell you that OGNL contains even more powerful features than we've felt comfortable divulging. For the full details, we refer you directly to the primary OGNL documentation found at [www.ognl.org](http://www ognl.org).

6.8 Summary

Well, that was a long chapter. That should be about as long as they'll get in this book. I'm worn out from writing it. To be fair, a large portion of the chapter was filled up

with reference material, screen shots, tables, and the like. Let's take a moment to consider the range of information that this chapter covered.

This chapter started by trying to clarify where data is kept during request processing. This key concept may be one of the most challenging parts of learning Struts 2. It's not really that complicated; it's just different than some frameworks you might've worked with in the past. As we noted, with the cleaned-up action component—it has no heavy parameter list on the `execute()` method signature—there's a strong need for a location where we can centralize all the data important to the execution of the action. This data makes up the context in which the action executes. Thus, the location where most of the important data resides is known as the `ActionContext`.

The `ActionContext` contains all kinds of important data, ranging from request-, session-, and application-scoped maps to the all-important `ValueStack` itself. We saw that we can access all these data items via the OGNL expression language. In particular, we learned that, by default, OGNL will resolve against the `ValueStack`, but we can also specify a different object from the `ActionContext` for our OGNL expressions by using the `#` operator to name our initial object. The `ValueStack`, in addition to being the default object for OGNL, is also distinguished by its special qualities. The most important quality of the `ValueStack` is that it presents a synthesis of the properties in the stack as if they were all properties on a single virtual object, with duplicate properties resolving to the instance of the property highest in the stack.

This section has introduced brand-new concepts for our Struts 1 friends. These new capabilities allow complex websites to be built easily. This might be a good time to reward you with a break. For many, understanding these data repositories can be the biggest hurdle in learning Struts 2.

With all that out of the way, we ran through the Struts 2 tag API at a gallop. The most important things to remember about the tag API are that it's implemented at a layer higher than the specifics of a given view-layer technology. Everything we've learned about using the specific tags, though we demoed them in JSPs, can easily be transferred to Velocity or FreeMarker. Just consult the syntactical changes we specified in this chapter and go. The APIs are all the same.

Actually, we've just started our tour of the Struts 2 tag API. This chapter covered the general-use tags. In chapter 7, we'll look at the UI component tags. These powerful tags will help us build rich user interfaces for our view layer. We're now deep into the view layer of the Struts 2 framework. In many ways, it's just getting interesting. Wait till you see how easy it is to make powerful forms with the Struts 2 UI tags.