CHAPTER 7

BeanUtils

One of the most popular patterns in Java is the JavaBeans component model (`http://java.sun.com/products/javabeans/`). Originally designed to allow visual design tools to generate AWT user interfaces, the JavaBeans specification provides additional guidelines on top of the basic contracts implied by a given Java class.

The JavaBeans specification provides mechanisms for methods of a Java class to be visible to a builder tool, organized into properties, methods, and events. A property is available via accessor methods, such as `String getFirstName()` and `void setFirstName()`. Methods are ordinary Java methods. Events define standard methods for allowing one component to notify one or more components of an arbitrary event. Visual development tools use these various systems to enable visual construction of user interfaces: a button is dragged onto a panel, the properties are set (such as the label and size), and events are wired (for example, clicking on the button closes the window).

For server-side development, the most popular aspects of the JavaBeans specification are the constructor and property patterns. Specifically, a standard JavaBean must have a no-argument constructor and get/set accessor methods corresponding to the various properties. For example, the simple JavaBean shown in Listing 7-1 has two `String` properties (first name, last name) and a single `int` property (clearance).

Listing 7-1 Simple JavaBean

```
package com.cascadetg.ch07;

public class User
{

    // Private variables
    private String firstName;

    private String lastName;

    private int clearance;
```

(*continues*)

Listing 7-1 (*continued*)

```
    // Accessor methods
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName)
    { this.firstName = firstName; }

    public String getLastName() { return lastName; }
    public void setLastName(String lastName)
    { this.lastName = lastName; }

    public int getClearance() { return clearance; }
    public void setClearance(int clearance)
    { this.clearance = clearance; }
}
```

While the JavaBeans specification describes rules for how components should be written and the expected behavior of the tools, no implementation is provided. Generally speaking, it is assumed that the low-level `java.lang.reflect.*` package will be used to obtain information about the Java classes, and the tool will generate code as needed.

Over time, it has become clear that frameworks, not just visual design tools, can take advantage of the JavaBeans patterns. For example, the object/relational bridge framework Hibernate uses JavaBean patterns to help work with relational databases in a more natural fashion.

Hibernate and FormBean

Hibernate, an object/relational database integration technology, uses the JavaBeans component model as its key foundation. By combining JavaBeans with XML mapping files, a Java developer can work with complex relational databases quickly and easily. The inspiration for this chapter came from the idea that the JavaBeans model can be leveraged not just for persistence, but also for automation of user interface generation.

For more information on the object/relational framework Hibernate, see my book, *Hibernate: A J2EE Developer's Guide to Database Integration* (ISBN: 0321268199, Addison-Wesley Professional).

While it is possible to use the low-level reflection package to deal with JavaBeans, it is easier to use the Jakarta Commons BeanUtils package.

For anyone given to thinking in terms of broader architectural design and framework development, it is easy to think of other areas in which the BeanUtils package may be useful. Obviously, this package would be of interest to anyone interested in building visual development tools. Similarly, dependency injection, dynamic configuration, and runtime binding of application elements such as the user interface and other systems are all potentially of interest.

In this chapter we will look at how to use the Jakarta Commons BeanUtils package to build a simple framework for converting objects based on the JavaBeans standard to HTML forms and back.

UNDERSTANDING BEANUTILS

In many ways, BeanUtils can be considered a metadata wrapper that makes it as easy to work with a JavaBean as a Map. The properties are the keys, and property values can be set by simply setting the property as a value. For example:

```
myUser.setName("Bob");
```

. . . can instead be written:

```
BeanUtils.setProperty(myUser, "name", "Bob");
```

Similarly, an array of the properties available for `myUser` can be simply retrieved:

```
DynaProperty[] properties = WrapDynaClass.
createDynaClass(myUser.class).getDynaProperties();
```

This offers two key advantages: it allows you to decouple components of your application, and it allows you to build frameworks and tools to take advantage of the JavaBeans framework.

As shown in Figure 7-1, the BeanUtils package draws a distinction between a `DynaClass`, which describes a class, and a `DynaBean`, which describes a particular object instance. This notion can actually be extended a bit—a `DynaClass` can be used as a wrapper for a JDBC `ResultSet`, for example (in which the properties correspond to the returned results), and individual records can be returned as `DynaBeans`.

One aspect of the BeanUtils package worth calling out is the notion of a `Converter`. This provides a generic way to retrieve and set values across a suite of properties using String values, regardless of the type of the property. For example, you may want to set a property with a type of `int` using a `String` such as `"2"`. The Converter package takes care of these details for you.

The following types are supported by built-in converters:

- BigDecimal
- BigInteger
- Boolean
- Byte
- Character
- Class
- Double
- File
- Float
- Integer
- Long
- Short
- SqlDate
- SqlTime
- SqlTimestamp
- String
- URL
- Abstract array
- Boolean array
- Byte array
- Character array
- Double array
- Float array
- Integer array
- Long array
- Short array
- String array

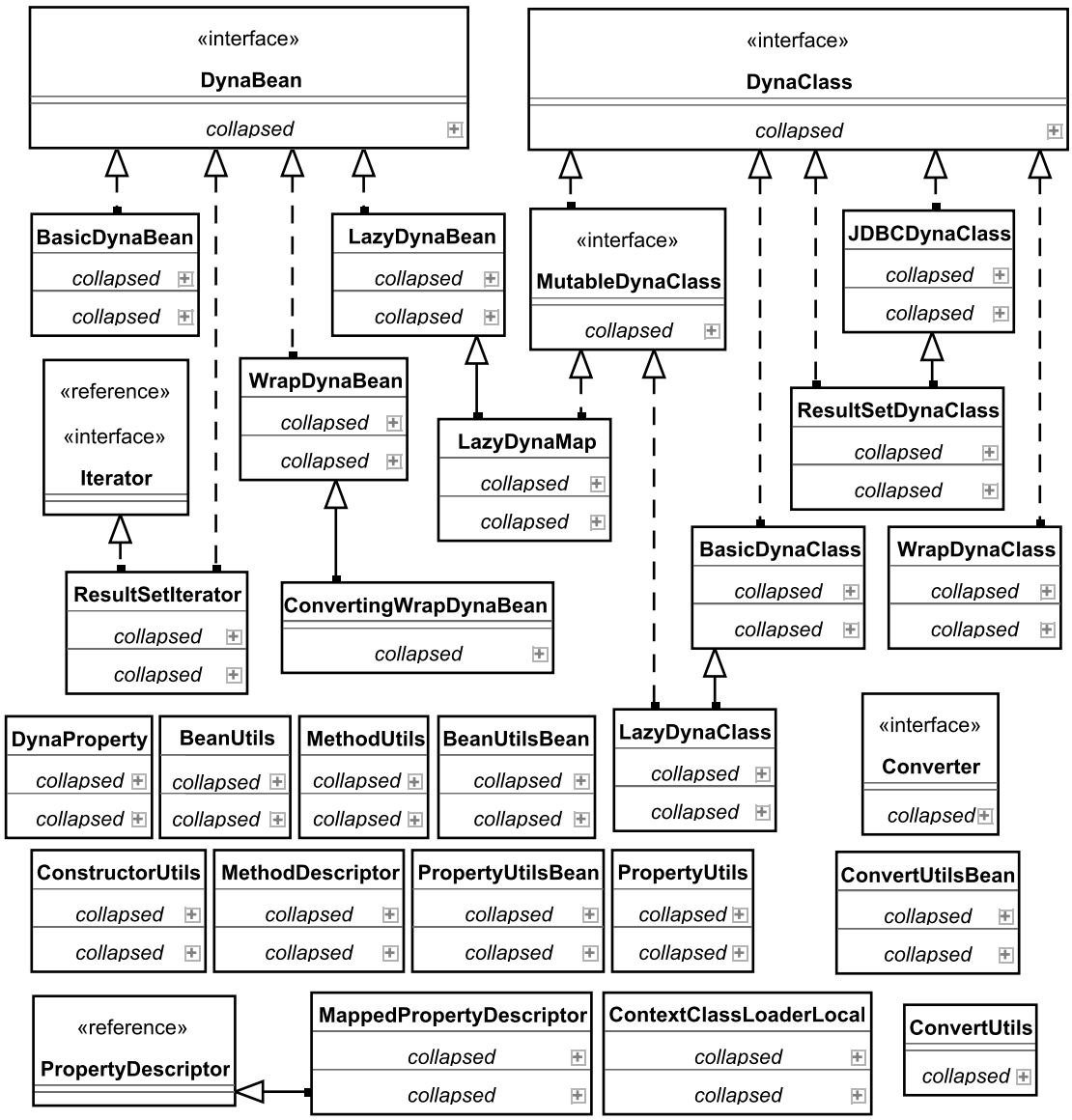

Figure 7-1 BeanUtils class diagram.

USING BEANUTILS TO GENERATE FORMS

Given the information already present in a JavaBean, the sample application will use BeanUtils to create a new library based on a new class, `FormBean`. `FormBean` generates an HTML form directly from the JavaBean information as rendered by BeanUtils. Each property results in a single label and a single

text field. The label is generated automatically from the property name (for example, `getFirstName` is automatically labeled First Name). When the form is submitted, `FormBean` will attempt to set the JavaBean properties using the submitted values.

Although FormBean is not a complete framework (for example, array-based properties are not supported), it does illustrate the power of JavaBeans in conjunction with BeanUtils.

Sample JavaBeans

This example is based on the simple JavaBean shown in Listing 7-1. Note the three properties: two of type `String`, one of type `int`.

In order to demonstrate the lifecycle as handled by `FormBean`, the application needs to have some mechanism for storing a value. Listing 7-2 shows how the application tracks the user.

Listing 7-2 Tracking the User

```
package com.cascadetg.ch07;

public class UserManager
{
    static private User user = new User();

    static public User getUser() { return user; }

    static public void setUser(User in) { user = in; }
}
```

Presenting the Form

Given the `User` JavaBean, `FormBean` generates a form, as shown in Figure 7-2.

If a user enters incorrect data for the required property type, `FormBean` automatically generates and reports the error, as shown in Figure 7-3.

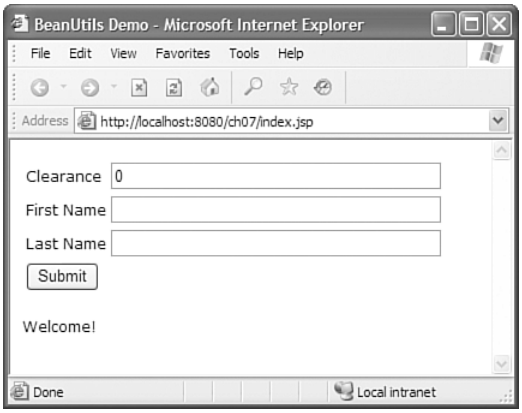

Figure 7-2 Initial `FormBean` form.

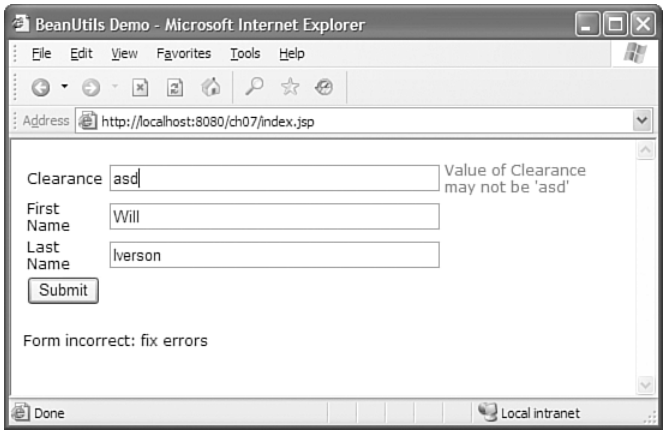

Figure 7-3 Handling a `FormBean` form error.

After the user corrects any errors and resubmits the form, `FormBean` is used to accept the submission, as shown in Figure 7-4.

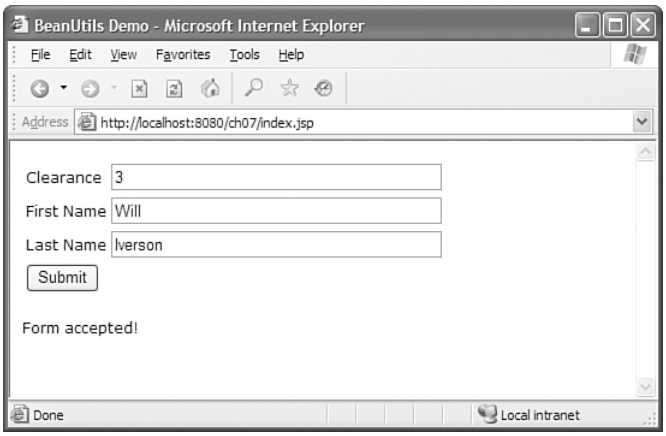

Figure 7-4 Corrected `FormBean` form.

The entire JSP file to generate and handle this form is shown in Listing 7-3. Note the `.error` style defined to highlight the errors.

Listing 7-3 `FormBean` JSP

```
<%@ page language="java" im-
port="com.cascadetg.ch07.*,org.apache.commons.beanutils.*" errorPage="" %>
<%
    FormBean myFormBean = new FormBean(User.class, UserManager.getUser());
    String notification = "Welcome!";
    if(request.getParameter("Submit") != null)
```

(*continues*)

Listing 7-3 (*continued*)

```
    {
        if(myFormBean.updateValue(request))
            notification = "Form accepted!";
        else
            notification = "Form incorrect: fix errors";
    }
%>
<html>
<head><title>BeanUtils Demo</title>
<style type="text/css">
<!--
.error { color: #FF0000; }
-->
</style>
<link href="../ch03/default.css" rel="stylesheet" type="text/css">
</head>
<body>
<table>
<form name="form1" method="post" action="">
<%= myFormBean.toHTMLForm()%>
<tr><td colspan="3"><input type="submit" name="Submit" value="Submit">
</td></tr>
</form>
</table>
<p><%= notification %></p>
</body>
</html>
```

Even a simple form with minimal update and validation logic would be painful to implement by hand with JSP. For a web application with dozens of such forms, a lot of tedious work could be eliminated through the use of a framework like `FormBean`.

FormBean

The initialization of the `FormBean` is shown in Listing 7-4. As you can see, the `FormBean` uses the class and optionally an instance to configure the form. If no instance is passed in, `FormBean` will attempt to instantiate one. Either way, the `FormBean` keeps track of both the original class and instance, and then it wraps the class in a `DynaClass` and the instance in a `DynaBean` (using `WrapDynaClass`).

Listing 7-4 `FormBean` Initialization

```
package com.cascadetg.ch07;

import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import org.apache.commons.beanutils.BeanUtils;

import org.apache.commons.beanutils.DynaBean;
import org.apache.commons.beanutils.DynaClass;
import org.apache.commons.beanutils.DynaProperty;
```

(*continues*)

Listing 7-4 (*continued*)

```
import org.apache.commons.beanutils.WrapDynaBean;
import org.apache.commons.beanutils.WrapDynaClass;

public class FormBean
{
    private HashMap errors = new HashMap();

    // The class refers to the compiled version of the class
    private Class baseClass;
    private DynaClass dynaClass;

    // The object refers to the runtime (in-memory) version
    private Object baseObject;
    private DynaBean dynaObject;

    private DynaProperty[] properties;

    // Used to help format the resulting text boxes
    private int displayStringLength = 40;
    private int maxStringLength = 100;

    /** For creation forms */
    public FormBean(Class myClass)
    {
        baseClass = myClass;
        dynaClass = WrapDynaClass.createDynaClass(baseClass);
        properties = dynaClass.getDynaProperties();
        try
        {
            baseObject = myClass.newInstance();
            dynaObject = new WrapDynaBean(baseObject);
        } catch (Exception e)
        {
            System.err
                    .println("FATAL ERROR: Unable to instantiate "
                            + dynaClass.getName());
            e.printStackTrace();
        }
    }

    /** For update forms */
    public FormBean(Class myClass, Object myObject)
    {
        baseObject = myObject;
        dynaObject = new WrapDynaBean(baseObject);
        baseClass = myClass;
        dynaClass = WrapDynaClass.createDynaClass(baseClass);
        properties = dynaClass.getDynaProperties();
    }
```

Given the class and an instance, `FormBean` has the information it needs to create a form. As shown in Listing 7-5, `FormBean` loops through the properties to generate the label and the input form and to set the default values for the form based on the object instance. Note that errors are collected in a `java.util.Map`, with the key being the property and the value being the error message.

Listing 7-5 `FormBean` Form Generation

```
/** Converts the object into a simple HTML form. */
public String toHTMLForm()
{
    StringBuffer output = new StringBuffer();

    for (int i = 0; i < properties.length; i++)
    {
        String currentProperty = properties[i].getName();
        if (currentProperty.compareTo("class") != 0)
        {

            // Start the row
            output.append("<tr>");

            // The cell for the label
            output.append("<td>");
            output.append(FormBeanUtils
                    .formatName(currentProperty));
            output.append("</td>");

            // The cell for the input form element
            output.append("<td>");
            output.append("<input ");
            FormBeanUtils.appendAttribute(output, "name",
                    currentProperty);

            // The cell for the current value, if there is
            // one
            if (this.dynaObject.get(currentProperty) != null)
            {
                FormBeanUtils.appendAttribute(output,
                        "value", this.dynaObject.get(
                                currentProperty).toString());
            }

            // Finish the input cell
            FormBeanUtils.appendAttribute(output, "size",
                    displayStringLength + "");
            FormBeanUtils.appendAttribute(output,
                    "maxlength", maxStringLength + "");
            output.append(">");
            output.append("</td>");

            // This cell displays any errors for this
            // property
            output.append("<td class='error'>");
            if (errors.containsKey(currentProperty))
            {
                output.append(errors.get(currentProperty)
                        .toString());
            }
            output.append(" </td>");

            // Finish up this row
            output.append("</tr>");
        }
    }

    return output.toString();
}
```

Listing 7-6 shows how a submitted form is handled. Note that the logic is expressed in terms of a `Map` (not explicitly tied in to the servlet model), allowing the `FormBean` to be tested outside of the context of a container. The `BeanUtils` class is used to attempt to set the values of the bean using the string submitted by the user. A `try/catch` block wraps the conversion attempt, and failures are logged to a `Map` for later display to the user.

Listing 7-6 `FormBean` Update Request

```
/**
 * Returns true if all of the values pass validation.
 * Otherwise, returns false (the user should therefore be
 * prompted to correct the errors).
 *
 * The incoming Map should contain a set of values, where the
 * incoming values are a single key String and the values are
 * String[] objects.
 */
public boolean updateValue(Map in)
{
    // Initialize the converters - we want format exceptions.
    FormBeanUtils.initConverters();

    boolean isGoodUpdate = true;

    for (int i = 0; i < properties.length; i++)
    {
        String key = properties[i].getName();
        Object value = in.get(key);
        try
        {
            BeanUtils.setProperty(baseObject, key, value);
        } catch (Exception e)
        {
            if (value != null)
            {
                errors.put(key, "Value of "
                        + FormBeanUtils.formatName(key)
                        + " may not be '"
                        + ((String[]) value)[0].toString()
                        + "'");
            } else
                errors.put(key, "Value may not be null");
            isGoodUpdate = false;
        }
    }

    return isGoodUpdate;
}

/**
 * Returns true if all of the values pass validation.
 * Otherwise, returns false (the user should therefore be
 * prompted to correct the errors).
 */
public boolean updateValue(HttpServletRequest request)
{
    Map in = request.getParameterMap();
    return this.updateValue(in);
}
```

Listing 7-7 demonstrates how the `FormBean` was developed—outside of a servlet container.

Listing 7-7 `FormBean`

```
    /**
     * Note that this particular design allows you to test your
     * bean programmatically, outside of the context of a web
     * application server.
     */
    public static void main(String[] args)
    {
        FormBean myFormBean = new FormBean(User.class);
        System.out.println(myFormBean.toHTMLForm());

        User myUser = new User();
        myUser.setClearance(5);
        myUser.setFirstName("Bob");
        myUser.setLastName("Smith");

        myFormBean = new FormBean(User.class, new WrapDynaBean(myUser));
        System.out.println(myFormBean.toHTMLForm());

        Map myMap = new HashMap();
        myMap.put("firstName", new String[] { "Ralph"});
        myMap.put("lastName", new String[] { "Bingo"});
        myMap.put("clearance", new String[] { "5"});
        myFormBean.updateValue(myMap);
        System.out.println(myFormBean.toHTMLForm());

        myMap.remove("clearance");
        myMap.put("clearance", new String[] { "invalid"});
        myFormBean.updateValue(myMap);
        System.out.println(myFormBean.toHTMLForm());
    }
}
```

FormBeanUtils

A few utility methods are needed to support the `FormBean`, as shown in Listing 7-8. The `formatName` method is used to generate proper English labels from JavaBean properties. The `appendAttribute` method is used to ease the generation of HTML-style attribute values. Most importantly, the `initConverters` method is used to cause the `BeanUtils` property setter to throw an exception in the event of a failed conversion attempt. By default, `BeanUtils` will silently fail if a conversion attempt fails. By installing the converter as shown, failed attempts to convert a `String` to an `int` or `Integer` value will generate an exception. If you wish to support additional property types and generate errors for failed conversions, you can install additional converters. The BeanUtils package includes converters for all core Java types (as listed earlier in this chapter), and you can create your own customer converters as well.

Listing 7-8 `FormBean` Utilities

```
package com.cascadetg.ch07;

import org.apache.commons.beanutils.ConvertUtils;
import org.apache.commons.beanutils.Converter;
import org.apache.commons.beanutils.converters.IntegerConverter;

public class FormBeanUtils
{
    /**
     * A utility function, takes a standard JavaBean property
     * name and converts it to a nice US English spacing.
     *
     * For example, firstName = First Name   */
    public static String formatName(String in)
    {
        String result = new String();

        for (int i = 0; i < in.length(); i++)
        {
            if (Character.isUpperCase(in.charAt(i)))
            { result = result + (" "); }
            result = result + (in.charAt(i) + "");
        }

        String result2 = new String();

        for (int i = 0; i < result.length(); i++)
        {
            if (Character.isDigit(result.charAt(i)))
            { result2 = result2 + (" "); }
            result2 = result2 + (result.charAt(i) + "");
        }

        char titleChar = result2.charAt(0);
        String result3 = Character.toUpperCase(titleChar) + "";

        result3 = result3
                + (result2.substring(1, result2.length()));

        return result3;
    }

    /**
     * A utility method, used to add an attribute in the form
     * attribute='value' with a space afterward.
     */
    public static void appendAttribute(StringBuffer in,
            String attribute, String value)
    {
        in.append(attribute);
        in.append("='");
        in.append(value);
        in.append("' ");
    }

    static private boolean convertersInitialized = false;

    static public void initConverters()
```

(*continues*)

Listing 7-8 (*continued*)

```
    {
        if (!convertersInitialized)
        {
            // No-args constructor gets the version that throws
            // exceptions
            Converter myConverter = new IntegerConverter();

            // Convert the primitive values
            ConvertUtils.register(myConverter, Integer.TYPE);

            // Convert the object version
            ConvertUtils.register(myConverter, Integer.class);
            convertersInitialized = true;
        }
    }
}
```

SUMMARY

This chapter shows how to leverage the JavaBeans pattern to provide richer application frameworks. This allows you to think of your application in terms of components and reusable frameworks, instead of a hard-coded monolith. By decomposing your application, it becomes easier to build, reuse, and test individual components.

It's easy to imagine building complex graphs of JavaBean objects, with complex access code required in certain situations involving collections. In the next chapter, JXPath is shown as a tool to assist in dealing with complex object graphs.

Project Ideas

Build a framework to test JavaBeans by inspecting the various properties, setting the properties, and calling a method on the JavaBeans. How much configuration is required beyond that which can be detected from the JavaBean type information?

Write an application to compare the performance of BeanUtils and Java's built-in reflection capabilities. Does the performance of one approach or another vary if different bits of information are cached?

If you find the idea of `FormBean` intriguing but are interested in a more complete solution, check out BeanView (`http://beanview.attainware.com/`). As of this writing, I have posted the source for a more complete framework, developed as an outgrowth of writing this chapter. My expectation is to remove the form generation and rendering components and instead rely on JavaServer Faces (`http://java.sun.com/j2ee/javaserverfaces/`) as a more robust framework. Feedback and comments are appreciated.