

Integrating ActionForms with POJOs

The acronym POJO stands for Plain Old Java Object—in other words, an ordinary JavaBean. POJOs are (among other things) commonly used to transfer data between the various components and architectural layers of a system, for example between the presentation tier and the web tier of a J2EE application, or more fundamentally, between a service and its client. Complex business objects are often represented as a graph of POJOs; for example, an Invoice POJO might contain a Customer POJO, a list of LineItem POJOs, and so forth.

Perhaps the most frequently heard gripe about Struts is that unlike some of the newer web application frameworks (Spring, WebWork, JavaServer Faces, etc.) it can't deal directly with POJOs. As a result, people developing Struts applications often feel forced to spend a considerable amount of time and energy devising solutions to bridge the gap, usually by transferring values between instances of `ActionForm` and POJO graphs. Often there are translations that need to be performed that complicate this process and make for tedious work.

A number of different approaches have been formulated in the past to mitigate this problem (and are still frequent topics of debate on the mailing lists), but in this chapter we'll take a look at a new approach that solves the problem at its root and thereby eliminates the overhead of working with POJOs for Struts developers.



Note

This chapter begins a new section of *Struts Live* that deals with more advanced topics than the earlier chapters. An open source implementation of the solutions presented in these new chapters is available in both binary and source formats from the *strutslive* project hosted on java.net (<http://strutslive.dev.java.net>).

The Trouble With POJOs

Struts developers have nearly all bumped their heads on this at one time or another. Here's the scenario: you need to add a couple of fields to an `ActionForm` to capture numeric values. But because you are new to Struts, or out of haste, forgetfulness, temporary amnesia, or whatever, you type them as `Integer` or `BigDecimal` rather than `String`. The next thing you know, you're looking at an ugly stacktrace, or trying to figure out why invalid input is causing an `Integer` value to change to zero. Let's look at a simple example. The `EmployeeForm` class shown in Listing 10.1 consists of a few harmless `String` properties.

Listing 10.1: A simple subclass of `ActionForm` (`EmployeeForm`) with `String` properties

```
...
public class EmployeeForm extends ActionForm implements Serializable
{
    private static final long serialVersionUID = 1L;

    private String firstName;
    private String lastName;
    private String department;

    public EmployeeForm() { }

    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
    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;
    }
}
```

If we use the JSP shown in Listing 10.2 to render it, everything works fine, and we can enter arbitrary values (or no values) in any of the fields and have the underlying form values updated correctly.

Listing 10.2: EmployeeDetail.jsp

```
<%@ taglib uri="/WEB-INF/lib/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/lib/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/lib/struts-logic.tld" prefix="logic"%>

<html:html locale="true">
<head>
    <link rel="stylesheet" href="employee/stylesheet.css" type="text/css">
    <title><bean:message key="title.example.form"/></title>
</head>
<body>

<h1><bean:message key="heading.employee.modify"/></h1>

<html:form action="/modifyEmployee" method="post">
<table>
    <tr>
        <td class="Label"><bean:message key="label.firstname"/><td>
        <td class="Field"><html:text property="firstName"/></td>
        <td class="ErrorMessage"><html:errors property="firstName"/></td>
    </tr>
    <tr>
        <td class="Label"><bean:message key="label.lastname"/><td>
        <td class="Field"><html:text property="lastName"/></td>
        <td class="ErrorMessage"><html:errors property="firstName"/></td>
    </tr>
    <tr>
        <td class="Label"><bean:message key="label.department"/><td>
        <td class="Field"><html:text property="department"/></td>
        <td class="ErrorMessage"><html:errors property="department"/></td>
    </tr>
    <tr> <td colspan=3><html:submit/></td> </tr>
</table>
</html:form>

</body>
</html:html>
```

The JSP in Listing 10.2 renders the page shown in Figure 10.1, which simply displays three fields and a submit button. The underlying `EmployeeForm` fields are all typed as `String`, so no conversion need be performed when mapping the submitted values to the fields. As a result, any arbitrary values can be entered and successfully processed during the `ActionForm` population performed by Struts.



The screenshot shows a web browser window with the title "ActionForm Conversion Example". The main heading is "Modify Employee Info". Below the heading are three text input fields: "First Name" containing "Fred", "Last Name" containing "Jones", and "Department" containing "Accounting". A "Submit" button is positioned below the "Department" field. The browser's address bar and other UI elements are partially visible at the bottom.

Figure 10.1: Modify Employee Info page

However, let's look at what happens if we add a couple of numeric fields to the `EmployeeForm`. Here are the new fields:

```
...
private Integer employeeNumber;
...
private BigDecimal salary;
...
public Integer getEmployeeNumber() {
    return employeeNumber;
}
public void setEmployeeNumber(Integer employeeNumber) {
    this.employeeNumber = employeeNumber;
}
public BigDecimal getSalary() {
    return salary;
}
public void setSalary(BigDecimal salary) {
    this.salary = salary;
}
...
```

We can then add the new fields to the JSP code:

```
...
<tr>
  <td class="Label"><bean:message key="label.employee.number"/></td>
  <td class="Field"><html:text property="employeeNumber"/></td>
  <td class="ErrorMessage"><html:errors property="employeeNumber"/></td>
</tr>
<tr>
  <td class="Label"><bean:message key="label.salary"/></td>
  <td class="Field"><html:text property="salary"/></td>
  <td class="ErrorMessage"><html:errors property="salary"/></td>
</tr>
...
```

We can successfully submit values for these new fields as long as they match the Java types of the underlying `EmployeeForm` properties. But if the user accidentally enters a non-numeric value in either field, weird things happen. For example, if a non-numeric value is entered in the **Employee Number** field, as shown in Figure 10.2, Struts won't be able to correctly populate the `EmployeeForm.employeeNumber` property because it is typed as `Integer`.


The screenshot shows a web browser window with the title "ActionForm Conversion Example". The main heading is "Modify Employee Info". Below the heading are five form fields, each with a label and a text input box:

- First Name:** Fred
- Last Name:** Jones
- Department:** Accounting
- Employee Number:** 123-XYZ (This field is highlighted with a blue border)
- Salary:** 4523.76

At the bottom left of the form is a "Submit" button. At the bottom right of the browser window, there are icons for search, zoom, and print.

Figure 10.2: Entering a non-integer employee number on the Modify Employee Info page

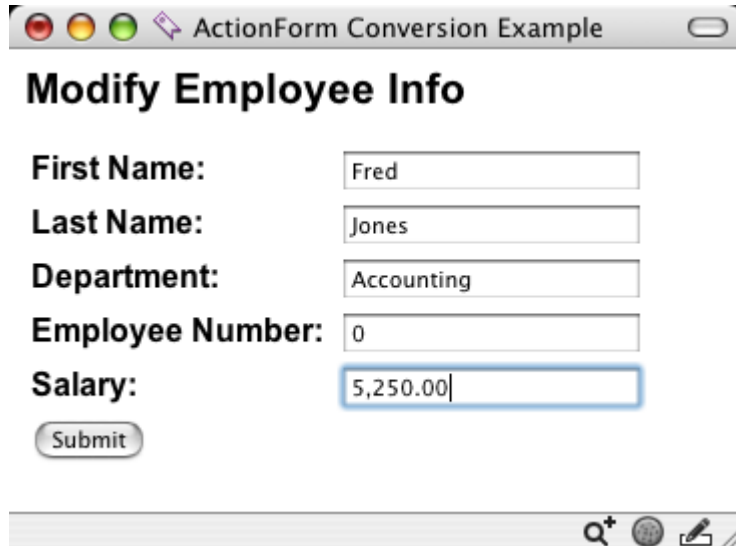
Unfortunately, Struts responds to this situation by setting the `Integer` field's value to `0` (Figure 10.3). This happens before any of your code is called, so no `ActionMessage` is generated to alert the user. If `0` is a valid value for the field (for example the number of dependents to use in calculating an employee's income tax withholding), then the `EmployeeForm`'s `validate()` method would have to be coded to compare the property value against the corresponding request parameter to distinguish this behavior from the user actually entering a `0` in the browser. And then you would still be left with the problem of restoring the property's original value.



The screenshot shows a web browser window with the title "ActionForm Conversion Example". The main content is a form titled "Modify Employee Info". The form consists of five labeled input fields and a "Submit" button. The values in the fields are: First Name: Fred, Last Name: Jones, Department: Accounting, Employee Number: 0, and Salary: 4523.76. The "Submit" button is a rounded rectangle with the text "Submit" inside.

Figure 10.3: Employee number changed to zero as a side effect of a conversion error

The situation with the **Salary** field is even worse. For example, if a user enters a value containing a currency symbol or thousands separators (in the U.S., a dollar sign and commas, respectively) as shown in Figure 10.4, or simply submits a blank value, an exception is thrown by the conversion code.



The screenshot shows a web browser window with the title "ActionForm Conversion Example". The main heading is "Modify Employee Info". Below the heading are five form fields, each with a label and a text input box:

- First Name:** Fred
- Last Name:** Jones
- Department:** Accounting
- Employee Number:** 0
- Salary:** 5,250.00

Below the Salary field is a "Submit" button. The browser's address bar at the bottom shows search, home, and back icons.

Figure 10.4: Entering a salary value containing a comma on the Modify Employee Info page

The Trouble With POJOs

Unfortunately, the conversion exception occurs before the framework calls our code, and there's no way to catch it. The resulting default behavior is to spew a stack trace to the user interface as depicted in Figure 10.5. We could improve upon this by adding a handler for the specific exception type to redirect to a custom error page, but that's almost certainly not the behavior we want.



Figure 10.5: The stack trace that results from a failed BigDecimal conversion. Look familiar?

So what does all this have to do with POJOs? Well, POJOs usually contain properties of various types, so conversion issues such as these (and several others for various other fundamental Java types) would no doubt rear their ugly heads if we were to attempt to integrate a POJO directly with an **ActionForm**.

Where POJOs Fit in the Architecture

In order to provide for clean separation between tiers, web application architectures often include delegates, proxies, or façades that can be called from the web tier to store and retrieve data. This shields the web tier from the underlying persistence implementation. Often, the lower tier uses POJOs as data transfer objects (DTOs), to shuttle data across these interfaces.

Suppose then that we could call an API that would return instances of the `Employee` POJO shown in Listing 10.3, which contains properties identical to those of the `EmployeeForm`.

Listing 10.3: An Employee POJO

```
...
public class Employee implements Serializable
{
    private static final long serialVersionUID = 1L;

    private Integer employeeNumber;
    private String firstName;
    private String lastName;
    private String department;
    private BigDecimal salary;

    public Integer getEmployeeNumber() {
        return employeeNumber;
    }
    public void setEmployeeNumber(Integer employeeNumber) {
        this.employeeNumber = employeeNumber;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
    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 BigDecimal getSalary() {
        return salary;
    }
    public void setSalary(BigDecimal salary) {
        this.salary = salary;
    }
}
```

Ideally, we could make the **Employee** instance (or its clone) a property of the **EmployeeForm**. That would allow us to rewrite the **EmployeeForm** as follows:

```
...
public class EmployeeForm extends ActionForm implements Serializable
{
    private static final long serialVersionUID = 1L;

    private Employee employee;

    public EmployeeForm() { }

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

We could then modify the **property** attributes in the JSP from Listing 10.2 to reference the nested properties in the **Employee** bean, as in the following snippet:

```
...
<html:form action="/modifyEmployee" method="post">
<table>
  <tr>
    <td class="Label"><bean:message key="label.firstname"/></td>
    <td class="Field"><html:text property="employee.firstName"/></td>
    <td class="ErrorMessage"><html:errors property="employee.firstName"/></td>
  </tr>
  ...
  <tr> <td colspan=3><html:submit/></td> </tr>
</table>
</html:form>
...
```

However, because of the conversion issues we observed in the previous section, the Struts documentation discourages following this approach. Instead, developers are advised to mirror POJO properties in **ActionForms**, and to type all ActionForm properties as **String** or **boolean** (or native arrays of those types). As a consequence, Struts applications that make use of POJOs require a layer of code to map values between POJO properties and **ActionForm** fields. Along the way, values must be converted between **String** representations suitable for the user interface and the actual Java types used internally.

The UI layer generally requires this conversion anyway in order to perform calculations required to (among other things) validate submitted values. For example, to check that a number submitted by the user lies in a specified range, the string value that the user submitted must first be converted to a numeric Java type before the comparison is made. Unfortunately, the current approach encourages making these conversions twice—once for the validation code, and again later when transferring values from the `ActionForm` to more strongly typed POJO properties.

Apart from the obvious inefficiency of converting the same value twice, there's an inherent risk that the conversions used in the validation code may get out of sync with the ones used in the bean population code.

How Struts Handles Conversion

Struts relies on an Apache Jakarta library, *commons-beanutils*, to bind values from the HTTP request to **ActionForm** properties during form submission (inbound processing), and from **ActionForm** properties to JSP tags during rendering (outbound processing). The *commons-beanutils* library provides a set of reflection-based services to allow bean properties to be accessed by name. Note that the property name can be an arbitrarily complex keypath, as the *commons-beanutils* library is designed to handle nested properties.

ActionForm Population

The **RequestProcessor** provides the Struts framework's dispatch mechanism. Its **process()** method is invoked by the **ActionServlet**'s **doGet()** or **doPost()** method to process a given request. The **process()** method then makes a series of callbacks to methods that handle discrete steps in processing the request. One of those methods is **processPopulate()**, which is responsible for populating the **ActionForm**'s properties with values from the request, as shown in Figure 10.6.

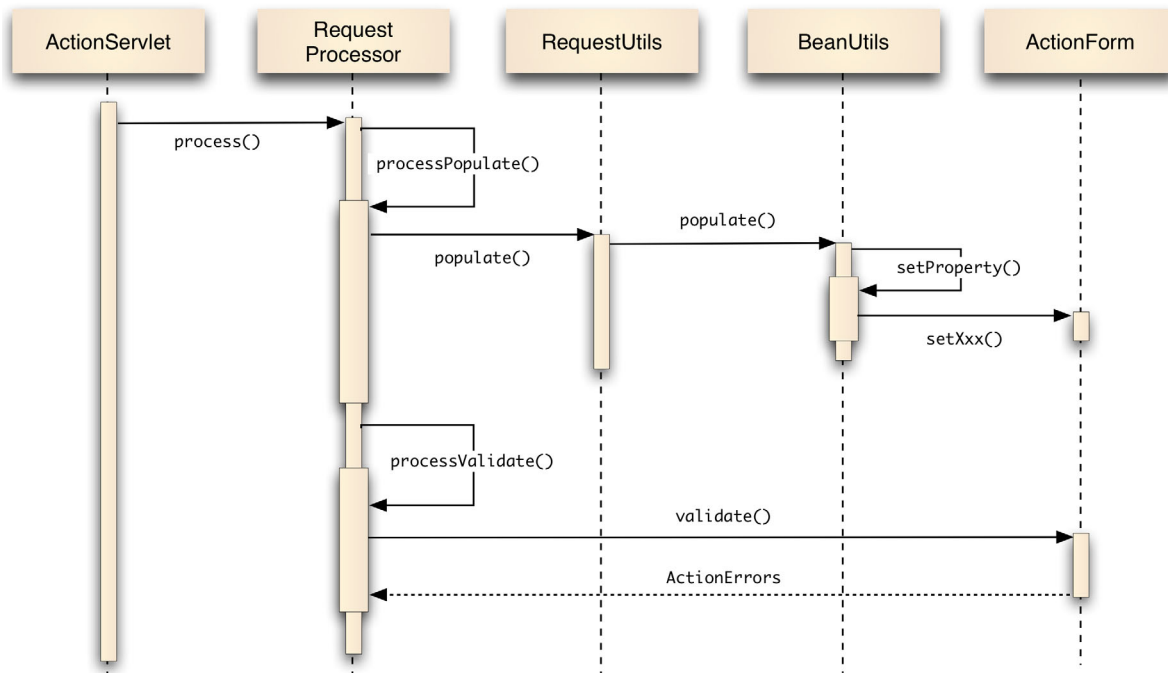


Figure 10.6: Request processing high-level sequence diagram

The **processPopulate()** method delegates the actual population functionality to **BeanUtils.populate()**, handing it the **ActionForm** instance and a **Map** containing the request parameters. The **populate()** method iterates the map's keys and makes callbacks to **BeanUtils.setProperty()** to set individual **ActionForm** properties.

The `setProperty()` method is essentially a wrapper that relies on methods of another class, `PropertyUtils`, to set the property values. (Note: details of the `BeanUtils` class's interactions with other classes in its package, such as `PropertyUtils` and `ConvertUtils`, are not shown in Figure 10.6.) Before invoking the `PropertyUtils` functionality, the `setProperty()` method checks the target property type to determine whether the provided value needs to be converted. (For example, if the request contained a parameter named 'salary' matching a property of type `BigDecimal`, the request value would need to be converted to that type.) If conversion is required, the parameter value and the target type are passed to `ConvertUtils.convert()`, which attempts to perform the conversion.

The `ConvertUtils` class includes a mechanism for registering `Converters` by type, and registers a provided set of default `Converters` in its constructor. `Converters` implement the `Converter` interface, which defines a single method with the following signature:

```
public Object convert(Class type, Object value);
```

Classes that implement the `Converter` interface throw a `ConversionException` to indicate that conversion has failed.

The Conversion Conundrum

It would seem at first glance that `BeanUtils` provides all the functionality that Struts would need to bind values to strongly typed POJO properties. However, there are several problems with the conversion mechanism that present formidable obstacles.

Perhaps the biggest problem is that the `RequestProcessor` calls `populate()` on the `BeanUtils` class instead of directly calling its `setProperty()` method. The `populate()` method simply iterates a `Map` of request values and calls `setProperty()` for each non-null key. The problem is that if a `ConversionException` is thrown during the call to `setProperty()`, the iteration is short-circuited because `populate()` doesn't catch the exception. And even if it did catch `ConversionExceptions`, there would be no way to provide information to the calling method about failed conversions because the return type is `void`.

So in addition to cutting short population when a conversion error occurs, the `populate()` method loses the information that would be needed to gracefully handle the error: the property name and the errant value. These would be needed to create an appropriate error message, and they would also be needed when rendering the input page, so that users could review and correct any invalid values they had entered.

Another problem is that even if the `RequestProcessor` had access to the values that had failed conversion, it would have nowhere to cache them so that they would be available during JSP rendering. For example, if an `ActionForm` had a `Salary` field typed as `String` and the value in the request was the string `foo`, the `Salary` field would be populated with the invalid value, which could then be presented back to the user along with a validation error message. But if the `salary` property was typed as `BigDecimal`, where would the value `foo` be stored?

Another problem is that the design of `ConvertUtils` and the `Converter` classes doesn't adequately support the formatting capabilities that would be needed for strongly typed properties. Since the provided `Converter` implementations simply call `toString()` to format object values, some types, such as `BigDecimal` and `Date`, would require custom implementations. However, because `Converters` can only be registered by property type, there would only be a single format available for all `BigDecimal` properties, all `Date` properties, etc. That's far too limiting for most applications.

As a result of the conversion and formatting issues we've examined in this section, Struts developers are encouraged to design their `ActionForm` classes as simple `String` buffers, effectively providing a caching mechanism that could be—and ought to be—provided by the framework. Developers are then burdened with implementing any conversion, formatting, and population code needed to transfer values to and from the buffer to their destination. They must then write additional validation code to check for potential conversion errors prior to performing the actual conversions.

Solving the POJO Problem

In the previous section, we saw several issues that prevent the use of strongly typed properties in `ActionForms`, which by extension effectively prevents Struts from working with POJOs. We can summarize the issues as follows:

- ▶ Conversion exceptions short-circuit the `BeanUtils.populate()` method, leaving the `ActionForm` in an invalid, partially populated state
- ▶ The framework doesn't cache request values that it can't convert.
- ▶ Even if there was such a cache, there would be no mechanism for Struts JSP tags to access it in order to display the unconverted value for a given property.
- ▶ There's no support for formatting property values when they are accessed by Struts tags.

Fortunately, taking advantage of several convenient extension points in Struts can solve these problems.

Design Strategy

There are actually a number of potential solutions to the POJO problem, but this chapter focuses on a solution that requires relatively little code, and is fairly nonintrusive. Let's take a look at the key design strategies we'll be exploiting.

The first of these strategies is to shift responsibility for `ActionForm` population to an abstract subclass of `ActionForm`. Managing its own population will give the subclass improved error handling capabilities, which will provide the following advantages over the current `RequestProcessor/BeanUtils` implementation:

- ▶ Conversion exceptions will no longer abort `ActionForm` population. This will allow comprehensive reporting of errors to the user.
- ▶ The `ActionForm` will be able to automatically post an appropriate error message keyed to the corresponding property whenever a conversion error occurs.
- ▶ Unconverted values will be cached in the `ActionForm` so that they can be presented back to the user.

The second design strategy is to provide a more flexible and comprehensive mechanism to handle conversion and formatting by creating a set of **Formatter** classes. The **Formatters** will have several advantages over **Converters**:

- ▶ **Formatters** will have separate methods for inbound and outbound processing to avoid ambiguity.
- ▶ Developers will be able to register a specific **Formatter** class to be used for a given property in addition to being able to register them by type. This will allow values to be automatically formatted for display, relieving the **ActionForm** of this responsibility
- ▶ Developers will be able to provide optional settings that provide finer control over formatting.
- ▶ **Formatters** will provide keys that can be used to look up canonical conversion error messages

The third and final element of the design strategy is to give Struts tags automatic access to unconverted request values and properly formatted values bean properties by customizing the way that **PropertyUtils** accesses **ActionForm** values. This will provide the following advantages:

- ▶ If the request value for a given property can't be converted, the request value will automatically be substituted for the property value for all tag references.
- ▶ Property values will be automatically formatted for display based on their registered **Formatters**.

Let's look at an overview of the steps we'll need to perform to implement the solution to the POJO problem.

Solution Steps

Here are the steps we'll follow to implement the design strategy outlined in the previous section:

1. **Create Formatters.** Create a `Formatter` base class that provides default functionality, and allows registration of `Formatter` subclasses by type as well as by property. Create an initial set of subclasses for testing purposes.
2. **Implement the PojoForm interface.** Create a `PojoForm` interface that defines methods for accessing unconverted values, accessing formatted versions of bean properties, and populating the bean. Create an `ActionForm` subclass (`PojoActionForm`) that implements the methods to provide the needed support for improved conversion error handling and the new formatting capabilities.
3. **Customize PropertyUtils.** Create a `PropertyUtilsBean` subclass that overrides `getProperty()` to provide special handling for `PojoForm` instances, returning an unconverted value for the property if one is present, or else calling the `PojoForm`'s formatting code to format the property value.
4. **Test PojoActionForm Functionality.** Write `MockStrutsTestCase` tests to verify that the new `PojoActionForm` functionality works correctly.
5. **Customize the RequestProcessor.** Create a `RequestProcessor` subclass that will invoke `populate()` on `PojoForm` instances, and that will handle the returned `ActionErrors` the same way that the base class handles the `ActionErrors` returned by the call to `ActionForm.validate()`.
6. **Integrate with web app.** Configure the `RequestProcessor` and `ActionForm` subclasses in `struts-config.xml`, and create a Struts `Plugin` that sets up the custom `PropertyUtilsBean` at load time.

The remainder of this chapter provides detailed explanations and code examples for each of the steps of the proposed solution. When complete, the solution will provide automatic formatting, type conversion, validation (partially—we can extend the validation to be more comprehensive later), population, and error messaging for subclasses of `PojoActionForm`.

