Struts Extensions Framework

Author: Phil Zoio, Realsolve Solutions Limited

Date: March 6, 2006

Summary

This document outlines some of the key features provided by the Struts Extension framework, a Java 5 specific set of extensions aimed at greatly increasing productivity and maintainability of Struts applications, while making them consistent with modern software development best practices.

Introduction

In many ways, Struts has been left behind in the last couple of years by other frameworks, which offer a range of productivity features that Struts has increasingly been unable to match, such as data type conversion, data binding and dependency injection.

The extensions discussed in this document take advantage of Java 5 language features, in particular JSR-175 annotations, to greatly enhance the capability of the Struts framework in terms of productivity, readability and maintainability. The extensions affect the following areas of the application:

- Request action workflow
- Form validation
- Form properties to domain model data binding
- Dependency injection
- Page classes

The Struts extensions are generic and are likely to be of value to other projects in the future. For this reason, the intention is to make these available through a separate open source project.

Actions and Control Flow

In Struts, the action is probably the key artifact from the application developer's point of view, because business method invocations are usually initiated from the action. In addition, the action must implement presentation logic. A number of factors make working with Stuts actions quite awkward:

- Struts Actions must be thread safe. A single `Action` instance is can be simultaneously handling multiple requests, making it impossible to use unsynchronized instance fields to hold temporary applications state. This is a real development handicap, because it forces the use of a rather cumbersome programming model when developing Struts actions.
- This state must be held in request, session or application context attributes. The API is verbose, with lots of otherwise unnecessary type casting. Your application code is very dependent of the Servlet API, and applying good OO practice is difficult.
- Struts imposes a fixed inheritance hierarchy, because your actions must subclass Action or some subclass of `Action`.

- Typed request parameters (anything other than String) must be manually converted.
- References to service layer objects (e.g. Spring beans, EJBs, etc.) must be obtained via programmatic hooks. There is no facility for dependency injection.

The Struts extensions address many of these concerns:

- Stateful POJO actions. Actions are created the usual way. However, for each request, a new instance of the action is created. This allows instance fields to be used to hold request-specific state, and opens up the possibility of dependency injection.
- Actions can be implemented as POJOs (plain old Java objects) or ***action beans***. A Struts `Action` is still used, but as an ***action controller***. This allows request workflow which is common to multiple actions to be encapsulated in an action controller. This simplifies the implementation of the actions, and frees up the action bean inheritance hierarchy. Action beans can be registered in *struts-config.xml* instead of (or in addition to) regular Struts `Action` classes.
- In typical Struts applications, a `RequestProcessor` subclass is used for application-wide functions, such as checking a user's authentication status. This is not ideal. `RequestProcessor` is a commonly used extension point, so it should be kept as lightweight as possible. The extension framework supports the use of ***action interceptors*** for these and other types of operations, which can be added in a modular way via configuration.

Action Beans

The example of an action bean shown below, which is used to handle submission of a form, shows some of the advantages of the improvements made to the framework. The example shows the implementation of a form submission action when using the extensions framework.

```
@Controller(name = BasicSubmitController.class)
public class SubmitEditBookingAction implements BasicSubmitAction
{

          private HolidayBookingForm form;

          private HolidayBookingService holidayBookingService;

          private WebHelper webHelper;

          public void preBind()
```

```
        {
        }

        public String cancel()
        {
                webHelper.setRequestAttribute("displayMessage",
                        "Cancelled operation");
                webHelper.removeSessionAttribute("holidayBookingForm");
                return "success";
        }

        public String execute()
        {
                HolidayBooking holidayBooking = form.getBooking();
                holidayBookingService.updateHolidayBooking(holidayBooking);

                webHelper.setRequestAttribute("displayMessage",
                "Successfully updated entry: " + holidayBooking.getTitle());
                webHelper.removeSessionAttribute("holidayBookingForm");

                return "success";
        }

        @InjectActionForm
        public void setForm(HolidayBookingForm form)
        {
                this.form = form;
        }

        @InjectSpringBean(name = "holidayBookingService")
        public void setHolidayBookingService(
                HolidayBookingService holidayBookingService)
        {
                this.holidayBookingService = holidayBookingService;
        }

        @InjectWebHelper
        public void setWebHelper(WebHelper webHelper)
        {
                this.webHelper = webHelper;
        }

}
```

Notice how the action bean implements `BasicSubmitAction`, which implements a basic workflow for form submission: a `preBind()` method to allow for any operations required before the form properties are bound to the domain model, an `execute()` method and a `cancel()` method if the form is cancelled.

Request workflow common to form handling has been moved into a `Action` controller implementation. In Vanilla Struts applications, this logic would need to go in the `Action` implementation.

The implementation of the action controller, `BasicSubmitController`, is shown below:

```
@ActionInterface(name = BasicSubmitAction.class)
public class BasicSubmitController extends BaseControllerAction
{

	@Override
	protected ActionForward executeAction(Object actionBean,
			ActionContext context)
	{

			BasicSubmitAction action = (BasicSubmitAction) actionBean;
			ActionForm form = context.getForm();
			HttpServletRequest request = context.getRequest();

			boolean cancelled = false;
			if (request.getAttribute(Globals.CANCEL_KEY) != null)
			{
				cancelled = true;
			}

			if (form instanceof BindingForm && !cancelled)
			{
				action.preBind();
				BindingForm validBindingForm = (BindingForm) form;
				validBindingForm.bindInwards();
			}

			String result = cancelled ? action.cancel() : action.execute();
			return context.getMapping().findForward(result);

	}

}
```

The `@ActionInterface` annotation enforces the contract which determines which interface action beans should implement. Notice how the decisions on whether to call `cancel()` and `execute()`, and whether to perform data binding, are made in the action controller.

Form Validation

Vanilla Struts currently offers two mechanisms for validation – either manual validation or use of the Commons Validator framework. Manual validation requires writing lots of verbose and tedious code, while the Validator framework itself has problems, in particular a large validation file with a verbose format, in short "XML hell".

The extension form validation mechanism uses annotations as an alternative mechanism. The mechanism is fully extensible in that no configuration file changes need to be made to introduce new validators. Under the covers, Commons Validator methods are generally used. However, the validation handlers all implement a `Validator` interface, with no restriction on how the interface is actually implemented.

The form validation allows for much more concise validation. Instead of

```
if (days == null)
{
    ActionMessage error = new ActionMessage("holidaybookingform.days.null");
    errors.add("days", error);
    hasError = true;
}
else
{
    if (!GenericValidator.isInt(days))
    {
        ActionMessage error =
          new ActionMessage("holidaybookingform.days.number”);
        errors.add("days", error);
        hasError = true;
    }
}
```

validation is simply a case of adding annotations to the setter methods of the relevant form property. The only additional restriction is that the form class must extend `ValidBindingForm`. Here's an example:

```
@ValidateRequired(key = "holidaybookingform.days.null")
@ValidateInteger(key = "holidaybookingform.days.number")
public void setDays(String days)
{
    this.days = days;
}
```

Multiple annotations can be added in this way. Where custom programmatic validation is required, this is implemented in the `validate()` method of the `ValidBindingForm` subclass.

Supported annotations include:

- `ValidateBlankOrNull`
- `ValidateDate`
- `ValidateDouble`
- `ValidateInteger`
- `ValidateIntegerRange`
- `ValidateLong`
- `ValidateLongRange`
- `ValidateMaxLength`
- `ValidateRequired`

Forms which use the validation mechanism are easily unit testable.

Data Binding and Conversion

As with validation, code for data binding of form properties to domain model properties is generally tedious and error prone. There are a few reasons for this:

- although Struts `ActionForms` theoretically allow typed form properties (as against simple `Strings`), in practice this is not possible. This is because if the user enters data of the incorrect type, it cannot be type converted, which means that the form cannot display the data that the user entered when the form is re-rendered. For example, if the user enters "one" into a field which holds a numerical value, the form will be re-rendered with the value "0" when data type conversion fails. From a usability point of view, this is not acceptable. In practice, this means that type conversion must be manual
- manual code needs to implement necessary null checks before attempting to convert data types
- when automatic type conversion is possible, Struts does not allow much flexibility in how data is type converted. It is not possible to specify how type conversion takes place on a per-property basis

As with form validation, extensions were added which use annotations to facilitate type conversion. This allows specially written data binding code such as this:

```
public void readFrom(HolidayBooking booking)
{
    if (booking != null)
    {
        if (booking.getStartDate() != null)
          {
             this.startDate = new
java.sql.Date(booking.getStartDate().getTime()).toString();
          }
```

```
    }
}

public void writeTo(HolidayBooking booking)
{
    if (this.startDate != null && this.startDate.trim().length() > 0)
        booking.setStartDate(java.sql.Date.valueOf(startDate));
}
```

to be replaced with code like this:

```
private HolidayBooking booking;

@BindSimple(expression = "booking.startDate", converter = SQLDateConverter.class)
public String getStartDate()
{
          return startDate;
}

/**
* Setter method for the domain object
*/
public void setBooking(HolidayBooking booking)
{
          this.booking = booking;
}
```

Once again, the code is easily unit tested. The converter can be specified on a per-field basis. Any custom converter can be used, as required. Binding can be to any object accessible from within the form. As with the validation, the data conversion by default uses the Commons BeanUtils library, which Struts uses internally.

Dependency Injection

Dependency injection is supported through the use of Java 5 annotations. The dependency injection allows for a much simpler and elegant programming style. Consider a simple Struts action which retrieves a numerical ID from the request, then uses this to look up some data using a Spring service bean. The code is shown below:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request,
        HttpServletResponse response) throws Exception
{
    HolidayBookingService service = (HolidayBookingService)
          getWebApplicationContext().getBean("holidayBookingService");
```

```
    long id = Long.parseLong(request.getParameter("holidayBookingId"));
    HolidayBooking holidayBookings = service.getHolidayBooking(id);
    request.setAttribute("holidayBooking", holidayBookings);

    return mapping.findForward("success");
}
```

A programmatic hook is required to get a reference to `HolidayBookingService`. The ID needs to be type converted. In this case, if the request is executed without the parameter `holidayBookingId`, the error will not be particularly meaningful.

This code can effectively be replaced by the following:

```
public String execute()
{
    HolidayBooking holidayBookings = service.getHolidayBooking(holidayBookingId);
    webHelper.setRequestAttribute("holidayBooking", holidayBookings);
    return "success";
}

@InjectSpringBean(name = "holidayBookingService")
public void setService(HolidayBookingService service) {
    this.service = service;
}

@InjectRequestParameter(required = true)
public void setHolidayBookingId(long holidayBookingId){
    this.holidayBookingId = holidayBookingId;
}

@InjectWebHelper
public void setWebHelper(WebHelper helper){
    this.webHelper = helper;
}
```

Dependencies resolution is reduced to getter methods. The effective lines of code have reduced substantially, and the interface is substantially simpler.

Page Classes

The extension framework supports the use of page classes. Page classes are classes which are directly coupled to a JSP implementation. The purpose of the page class is three-fold:

- Page classes provide a convenient point of access for data retrieved or created in the action which needs to be accessible to the JSP. This reduces the number of request attributes that the application

needs to use. In other words, it provides a tighter binding between the JSP and Java within the *view layer* of the web tier, making the application easier to understand and maintain. It also eliminates the need to use `ActionForms` for anything other than handling forms.

- Page classes allow formatting logic to be implemented in Java. Page classes can be used for markup generation where this makes sense, or for handling conditional logic.
- Data which is required by all the pages can be expressed in a common interface which all the page classes must implement. For example, all of the pages may need to determine which links should be available in the header include page, and provide additional application status information. An interface can encapsulate this requirement, which all page classes can implement.

The use of page classes simplifies the *struts-config.xml* mapping file, because the target JSPs do not need to be specified there. The cost, however, is that the action implementations need to be aware of the identity of the target page for each request. In practice, this is desirable, or at worst, not a problem.

Other Features

The improved version also adds various other features which make it a more compelling framework:

- support for Struts-style **dispatch actions**
- **full Spring integration**: action beans can be Spring managed objects, offering the power of Spring AOP
- **pluggable action bean annotations**: you can define your own annotations and annotations handlers for your action beans and implement a controller which
- **pluggable navigation**: navigation is no longer directly coupled to the ActionForward class. It would be straightforward, for example, to plug in the Spring MVC view layer into the controller framework, offering a great deal more choice and flexibility in using different types of view technologies