Using Struts

Author: Larry Maturo, Athens Group

Athens Group
5608 Parkcrest Drive, Ste. 200
Austin, TX 78731
512.345.0600
FAX: 512.345.4088
http://www.athensgroup.com
info@athensgroup.com

Athens Group is an Austin-based, employee-owned consulting firm helping clients achieve business results through the practical application of technology strategy and software solutions. Founded in June 1998, the company has 46 employees and a diverse client list that includes corporations such as Agilent, DuPont Photomasks and Dell, as well as start-up and mid-growth companies.

Table Of Contents

General

Struts is a fairly complex framework for developing web applications, but one that is fairly painless to use after it is setup. This document does not cover setting up Struts or deploying Struts, as both of these depend on your environment. This document assumes you have Struts set up, but are struggling with using it, especially using some of the Struts tag, which are a powerful, but non-intuitive feature of Struts.

The Important Parts of Struts

You will be modifying struts-config.xml a lot. It contains configuration information, as will be explained below. You will also be creating Action classes and JSPs that work together. Finally, you may be creating Action Form classes to support your Action classes and JSPs.

Basically, for each of your JSP pages there will be an associated Action class. It's possible that the same Action class can be used for multiple JSPs, though this is rare. If you are inputting data on your JSP page, there will also be an associated Action Form class that handles the data for the input. You can also use this Action Form class for storing data to be output if you wish.

One of the most important parts of Struts is the tag library. In the MVC Framework it is part of the view layer. The tag library allows you to do things in your JSP in an html like manner, without embedded scripting. This makes it much easier to maintain the JSPs, and to understand how your application works. As a result, it is a best practice to not use any Java scriptlets (Java code embedded inside JSP) in your JSPs. In the worst case, you can create custom tags to hide the Java scriptlets.

Action Class

Your action class has several jobs. One is to set up the request/session with data for your JSP to display, one is to set up your ActionForm class bean with data, one is to validate and store into a database any data returned from the user in the ActionForm class bean, and one is to determine where to go next based on user input. For this last case, you set up the name/location pairs (local and global forwards) in struts-config.xml and only refer to the name part in your Action Class. Note that you **always** need an Action Class to set up the data for your JSP, so you will never go from JSP to JSP. That's an important point because accessing JSPs directly is dangerous, since it bypasses the controller and thus any access control, etc. provided by the controller would be skipped.

ActionForm Class

The best way to think about the ActionForm class is as a buffer. The problem it solves is that when a user enters bad data, you don't want it going to the database, and you don't want to have to fill in something with the user's old data so he can see where he made a mistake and fix it. All fields of the ActionForm class are normally String. You set initial values here for fields. Struts displays the initial values and allows the user to modify

them. The user submits his form, you validate it, detect errors, give Struts a list of the errors you found, and tell Struts to handle them. Struts redisplays the form to the user, but with your error list at the top of the form, so that he can correct his errors and resubmit the form.

HTML Forms

The html tags in struts usually only work within the context of an html:form tag. For example:

```
<html:form action="/EditDefinition.do"  scope="session">
</html:form>
```

EditDefinition.do is the action class for this jsp, scope is the scope the in which the data will be found. A form is submitted to the server by using an html:submit tag. Below is an example of this.

<html:submit property=”submitButton” value=”Edit”/>

This will set the submitButton string property of the form bean to the value “Edit” and submit the form. This is a key to figuring out what to do in your action class. If this field is blank then you got to your action class from another page and should initialize your action form. If the field is non-blank, then you got here from the user pressing the Edit button. Note that “value” is used both to label the button, and to set the value of the field pointed to by property. You could thus have:

```
<html:submit property=”action” value=”Add”/>
<html:submit property=”action” value=”Edit”/>
<html:submit property=”action” value=”Save”/>
```

and then when the form was submitted you could check the action field to determine if you should add, edit or save.

Handling Input

Most input in struts occurs within the context of an html:form tag (see above.) Most html tags only work in the context of an html:form tag.

Text Input

To input a text field in the JSP you can use a struts tag such as:

```
<html:text property="username"/>
```

For this to work the ActionForm bean should have a private or protected string variable called username, a setter called setUsername, and a getter called getUsername. Struts will handle fetching the property from the Java Bean and filling it in on the JSP for

display, and fetching the value from the JSP and storing it in the bean on submit. This two-way action is true of most html tags in struts.

Drop-Down Combo Box Input

Drop-Down Combo Boxes are used for letting the user select one item out of many. In Struts this is accomplished using the Select, Option, and Options tags. The idea in a combo box is for the user to select one value out of many, and for struts to stick the selected value into a variable in the form bean. The initial value of this variable is used to pre-select one of the options.

It is also possible to hard code the selection using the value attribute of the select tag. This is only useful in the cases when you always one value to be pre-selected, such as always wanting the United States to be the default selection in a list of countries that are alphabetized by country name. Doing this means that every time the user comes to this screen the United States will be the default value, no matter what he selected the list time he visited the form. Generally, this is a bad idea.

Select can be used with:

a: A collection of strings, such as an arraylist
b: A collection of beans, each of which has string fields
c: Hard coded values
d: A combination of the above

With An ArrayList of Strings

ArrayList name is nameList. This list has been put into
either the request or session.

name of field in form bean to set with the users selection is selectedValue. Set this value in the form to pre-select a value.

```
<html:select property="selectedValue">
      <html:options name="nameList" />
</html:select>
```

How to Hard Code a Selected Value With An ArrayList of Strings

Modify select as follows:

```
<html:select property="selectedValue" value="aName">
      <html:options name="nameList" />
</html:select>
```

where aName is one of the names in the arrayList.

With An ArrayList of Beans

Bean string fields of interest are:

1. displayValue
2. idValue

ArrayList name is beanList. This list has been put into either the request or session.

name of field in form bean to set with users selection is selectedValue. It will be set to the idValue of the selected bean. Set this value in the form to pre-select a value. The user will see the contents of displayValue as the item in the list.

```
<html:select property="selectedValue">
      <html:options collection="nameList" property="idValue"
                                    labelProperty="displayValue"/>
</html:select>
```

How to Hard Code a Selected Value With An ArrayList of Beans

Modify select as follows:

```
<html:select property="selectedValue" value="aProperty">
      <html:options collection="nameList" property="idValue"
                                    labelProperty="displayValue"/>
</html:select>
```

where aProperty is one of the strings in the field of the bean in the list specified by the collection property in the options tag.

With Hard-Coded Strings

name of field in form bean to set with the user's selection is selectedValue. . Set this value in the form to pre-select a value.

```
<html:select property="selectedValue">
      <html:option value="value_1"/>
      <html:option value="value_2"/>
      <html:option value="value_3"/>
</html:select>
```

How to Hard Code a Selected Value With Hard-Coded Strings

Modify select as follows:

```
<html:select property="selectedValue" value="value_2">
      <html:option value="value_1"/>
      <html:option value="value_2"/>
      <html:option value="value_3"/>
</html:select>
```

Dependent Drop-Down Combo Box Lists

In many situations, you will have two combo boxes, with the contents of the second dependent on the value selected in the first. That is, each item in the first combo box is associated with a list. To handle this, you need for the first combo box to do a submit when it changes, so that you have a chance to update the second. You do this with:

```
<html:select property="orgId" onchange='document.forms[0].submit()'>
      <html:options collection="orgList" property="value"
                                          labelProperty="longName"/>
</html:select>

In this example, the property "value", of the beans in orglist are
orgIds and the "labelProperty" property is the name to display.  When
you are in your action class the first time, the orgId string property
of your form bean will be null or blank.  In subsequent submissions the
orgId property of your form bean will contain a valid orgId, so that
you can look up the list associated with that value, and thus populate
the second drop-down combo box propery.
```

Radio Button Input

The idea in radio button input is to present the user with several mutually exclusive choices and allow him to select one. An example is:

```
<html:radio property="selection" value="Required" />
<html:radio property="selection" value="Exempt" />
<html:radio property="selection" value="Done" />
```

Note that because property is the same they will all be in the same radio button group, and they will thus be mutually exclusive. Set the selection attribute in the form bean, in your action class, to match one of the "value" attributes of the tag to pre-select that value.

Check Box Input

The checkbox is slightly different than other input types in Struts. Its type in the form must be boolean. You must also set this value to false in the form bean in the form bean reset method (see below) for it to work. This is because HTML does not send values for check boxes if they are false. An example is:

```
<html:checkbox property="isBillable" value="wasBillable"/>
```

Note that since the isBillable field must be cleared, to have it pre-checked you must use another field in your form to specify the initial value.

Handling Output

Text Output

The bean:write tag is used to output text. An example is:

```
<bean:write property="mnemonic" filter="true"/>
```

outputs the value of the mnemonic field in the form bean. The filter attribute allows the tag to automatically encode any characters that might cause html problems.

Creating a Table in Struts

In Struts you would use the Iterate tag to create a table. A simple example would be:

```
<logic:iterate id="org" name=”organizationList” scope="session"
      type="com.athensgroup.projtrack.controller.org.OrganizationForm">
     <tr>
      <td><bean:write name="org" property="mnemonic"
                                                filter="true"/></td>
      <td><bean:write name="org" property="fullName"
                                                filter="true"/></td>
      <td><bean:write name="org" property="type" filter="true"/></td>
      <td><bean:write name="org" property="parentOrgName”
                                                    filter="true"/></td>
    </tr>
</logic:iterate>
```

Where org is a local variable created by the iterate tag that is of the type specified by the type attribute of the iterate tag. Note that you must use the name attribute of bean:write to tell it to use the bean created by the iterate tag, instead of looking in the form bean for the property.

Using the Indexed Attribute in the Scope of the Iterate Tag

Note that the indexed attribute is only available in nightly builds past 1.02.

Often times, when using iterate, you will use the indexed attribute of many html tags to allow you to do input in a table. A few examples of this will be covered below. It is important to remember, when using this attribute, that the only place you can input data in struts is in the form bean. This means that in many cases you will have to store your list in the form bean to get the indexed attribute to work.

Using Submit Buttons in a Table

When presenting a table of items to be edited or deleted you will often want each row of the table to have a submit button or two. Below is an example of this:

```
<logic:iterate id="org" name=”organizationList” scope="session"
      type="com.athensgroup.projtrack.controller.org.OrganizationForm">
     <tr>
       <td>
         <html:submit indexed="true" property="cmd" value="Edit" />
       </td>
       <td>
         <html:submit indexed="true" property="cmd" value="Delete" />
       </td>
       <td>
```

```
          <bean:write name="org" property="mnemonic" filter="true"/>
        </td>
        <td>
          <bean:write name="org" property="fullName" filter="true"/>
        </td>
        <td>
          <bean:write name="org" property="type" filter="true"/>
        </td>
        <td>
          <bean:write name="org" property="parentOrgName"
                                                    filter="true"/>
        </td>
    </tr>
</logic:iterate>
```

This will cause the cmd string array property to be set to "Edit" for the button pressed, for example cmd[3] == "Edit". This is not that useful, but it can be made very useful by faking an array property, since Struts uses reflection, and reflection determines the type based on the setters and getters. In this example, the form bean parts of interest would look like:

```
private int selectedValue = 0;
private String cmd = "";

public void setCmd(int index, String value) {
        selectedValue = index;
        cmd = value;
}

public String getCmd(int index) {
        return cmd;
}

public String getCmd() {
        return cmd;
}

public int getIndex() {
        return selectedValue;
}
```

Then in your action you can see what button was pressed by checking cmd, and which item by checking index. You will need to retrieve the list you passed in form the session or request, so that you can use index to index into it to get the selected item.

Creating a Table of Radio Buttons

In some situations, you might want to create something like the below table.

Status			
Required	**Exempt**	**Done**	**Gate**
o	*	o	Statement of Work
*	o	o	Kickoff Scheduled

The trick is that unlike the select/options tag, the radio button tag stands alone. In other words, select allows you to tell struts where to put the result, and options allows you to tell struts were to get the list. Because the radio button tag stands alone, the list must come from the same place were the results are stored, and in struts, the only place to store results is in the form bean. Below is an example of how to do this.

In this example the data structure is an arraylist of beans, where each bean has a gateName property and a status property. If the status property of the bean matches the value specified in the radio button tag, then that radio button will be checked when the table displays. After the user submits the form, you can fetch the arraylist and get the current value of the status field of each bean.

Note that the real trick here is making sure the name you use for id in the iterate tag exactly matches the name of the property in your form bean, because it used not only to create a local variable, but also to specify where the results goes. It's this double duty for id that makes everything work.

In the form bean:

```
private ArrayList processGates = new ArrayList();

public java.util.ArrayList getProcessGates() {
      return processGates;
}

public void setProcessGates(java.util.ArrayList processGates) {
     this.processGates = processGates;
}

public ProjectProcessGate getProcessGates(int index) {
    return (ProjectProcessGate) processGates.get(index);
}

public void setProcessGates(int index, ProjectProcessGate gate) {
      processGates.set(index,gate);
}
```

In the action class on setting up the form bean:

```
ProjectForm pForm = new ProjectForm();
ArrayList ppgList = new ArrayList();
// Fill in the list here
pForm.setProcessGates(ppgList);
```

In the jsp:

```
<table border="">
  <tr>
    <td colspan="3"><b><big><center>Status</center></big></b></td>
    <td></td>
  </tr>
  <tr>
    <td align="center" width="80"><b>Required</b></td>
    <td align="center" width="80"><b>Exempt</b></td>
    <td align="center" width="80"><b>Done</b></td>
    <td align="center" width="150"><b><big>Gate</big></b></td>
  </tr>
    <logic:iterate id="processGates" name="projForm"
          property="processGates">
  <tr>
    <td>
      <center>
          <html:radio name="processGates" property=”status"
          value="<%= ProcessStatus.REQUIRED.getMnemonic() %>"
                                                  indexed="true" />
      </center>
    </td>
    <td>
      <center>
          <html:radio  name="processGates" property”status"
          value="<%= ProcessStatus.EXEMPT.getMnemonic() %>"
                                                  indexed="true" />
      </center>
    </td>
    <td>
      <center>
          <html:radio  name="processGates" property=”status"
          value="<%= ProcessStatus.DONE.getMnemonic() %>"
                                                  indexed="true" />
      </center>
    </td>
    <td>
      <center>
          <bean:write name="processGates" property=”gate" />
      </center>
    </td>
  </tr>
  </logic:iterate>
</table>
```

In the action class after submitting the form:

```
ProjectForm pForm = (ProjectForm) form;
ArrayList ppgList = pForm.getProcessGates();
```

Error Messages

Struts provides a tag for showing error messages: <html:errors/>. This struts tag will cause a list of error messages to be inserted into the generated html at the point it occurs within the jsp. In addition, this tag can have a property attribute, in which case it is used thus: <html:errors property="key"/>, and then the tag will only cause only the error messages that match the key to be displayed.

In your action class, when you catch an error, you will have something like:

```
ActionErrors errors = new ActionErrors( );
errors.add(Param1,new ActionError(Params));
saveErrors(request, errors);
return (new ActionForward(mapping.getInput()));
```

The error message list is created with "ActionErrors errors = new ActionErrors();", as shown, and an individual ActionError is added to it using the "add" method, as shown. The add method takes two parameters, a key, for use within the html:errors tag, and an Action Error which encapsulates the error message. (The parameter *property* in errors.add(String property, ActionError error) can be thought of as defining the category of error.)

The ActionError object constructor takes one to five parameters. The first parameter is a key into the ApplicationResources.properties file. This file contains the text of the error message, which can contain up to four embedded parameter "place-holders". The rest of the parameters, which are optional, are the values to be substituted into the "place-holders" in the error message. For a complete discussion on how to create parameterized message strings, consult the API documentation for "java.text.MessageFormat".

Simple Example

In the ApplicationResources.properties file:

```
DBError=<li>Error: {0} occurred trying to {1}
```

In the jsp file:

```
<html:errors/>
```

In the Action Class File

```
errors.add("parentOrgDBError",
        new ActionError("DBError",
                        "field not found" ,
                        "get a list of parent organizations"));
```

Actual Error Message Displayed in JSP:

Error: field not found occurred trying to get a list of parent organizations

Use of Constants in Error Messages

Since strings do not get error checked at compile time, it is a standard to use constants wherever possible to allow the compiler to help find potential run-time errors. In the case of error messages, there are two places where it is appropriate to use constants.

The first is to make the message key within the ApplicationResources.properties files a constant in some Constants file. This assures that all references to error message keys actually match to an error message in the ApplicationResources.properties file.

The second is to create local constants in Action subclasses for the keys that identify individual ActionError objects put into the ActionErrors class. This ensures that the key that matches up with the property attribute of the html:errors tag will match something in the list that's being returned.

As it turns out, since each error has a type (specified by the key into ApplicationResources.properties file), the key used in the second case should reference the message key in the first case. The way to do this is to make the second key compound. This guarantees uniqueness of the keys. This is important if you want to display errors by each field, and you only want one error message shown. Without this you could end up with a list.

Example with Constants

In the com.athensgroup.Constants.java File:

```
public static final String DB_ERROR = "DBError";
```

In the ApplicationResources.properties file:

```
DBError=<li>Error: {0} occurred trying to {1}
```

In the jsp file:

```
<html:errors />
```

In the Action Class File

```
public final String PARENT_ORGS = "ParentOrgs";
public final String PARENT_ORGS_DB_ERROR =
                                        Constants.DB_ERROR + "." + PARENT_ORGS;

errors.add(PARENT_ORGS_DB_ERROR,
            new ActionError (Constants.DB_ERROR
                                ”field not found” ,
                                "get a list of parent organizations"));
```

Actual Error Message:

Error: field not found occurred trying to get a list of parent organizations

Simple Example that Actually Uses the Constant Within the JSP File

In the Constants File:

```
public static final String DB_ERROR = "DBError";
```

In the ApplicationResources.properties file:

```
DBError=<li>Error: {0} occurred trying to {1}
```

In the jsp file:

```
<%@ page
        import="com.athensgroup.projtrack.controller.org.AddOrganizationAction" %>
<html:errors
        property= "<%= AddOrganizationAction. PARENT_ORGS_DB_ERROR %>" />
```

In the Action Class File

```
public final String PARENT_ORGS = “ParentOrgs”;
public final String PARENT_ORGS_DB_ERROR =
                                        Constants.DB_ERROR + "." + PARENT_ORGS;

errors.add(PARENT_ORGS_DB_ERROR,
            new ActionError (Constants.DB_ERROR
                                ”field not found” ,
                                "get a list of parent organizations"));
```

Actual Error Message:

Error: field not found occurred trying to get a list of parent organizations

Comments

1. Although one could specify the error message parameters in constants, there is no real need for this, especially since many times the string parameter data will be coming from the message text of an exception.
2. It is possible to associate an error message with a particular field. To do this one would want to encapsulate each “field set” in its own try/catch, so that every error message was very specific. When the html:errors tag is used without further parameters, it causes a line to be drawn after the error message list, dividing the page in half. This turns out to be both good-looking and very readable.

Specifying a Header and Footer for the Error Messages

Not only are the error messages stored in the ApplicationResources.properties file, but the error header and footer are also stored there. An example would be:

```
errors.header=<hr><p><b><center>Errors for Page</center></b></p>
errors.footer=<hr>
```

This would look something like:

Errors for Page

- Error: This is an error
- Error: This is another error

The rest of your jsp

Note that if you want to display the errors next to individual fields you need to make sure you have your header and footer set to nothing.

ActionForm Validate and Reset

You can override, in the Java Bean based on ActionForm, a validate method declared as:

public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)

that can be called to validate the user's input. The Java Bean should also specify a reset method declared as:

public void reset(ActionMapping mapping, HttpServletRequest request)

to reset the variables in the Java Bean to their default.

The validate() method is called by the controller servlet after the Java Bean properties have been populated, from the JSP, but before the corresponding action class's perform() method is invoked.

The validate() method has the following options:

1. Perform the appropriate validations and find no problems -- Return either null or a zero-length ActionErrors instance, and the controller servlet will proceed to call the perform() method of the appropriate Action class.
2. Perform the appropriate validations and find problems -- Return an ActionErrors instance containing ActionError's, which are classes that contain the error message keys (into the application's MessageResources bundle) that should be displayed. The controller servlet will store this array as a request attribute suitable for use by the <html:errors> tag, and will forward control back to the input form (identified by the inputForm property for this ActionMapping).

The default implementation of the validate() method returns null, and the controller servlet will assume that any required validation is done by the action class. This is the normal case, since most validation will require validating against business logic, and the ActionForm bean has no business logic associated with it.

Note that reset is normally not used in your form bean unless you are using checkboxes, in which case it mandatory.

In addition to the Java Bean associated with your JSP you will normally have a set of classes associated with your business logic, and other classes associated with storing into and fetching data from a database.

What's In struts-config.xml?

Struts-config.xml contains 3 types of information:

Global Forwards

These are of the form: <forward name="Name for JSP" path="/JSP Name.jsp"/>

An example of this is: <forward name="Home" path="/Home.jsp"/>

These can be used in your JSP's, for links to other pages, by using the name. Thus if the path changes, you can just change the struts-config.xml file without having to edit your JSPs. To do this use the form:

```
<html:link forward="value defined in global forward">
      link name to display
</html:link>
```

An example of this is:

```
<html:link forward="viewEvaluationResults">
      View Evaluation Results
</html:link>
```

Bean Declarations

These are of the form: <form-bean name="FormName" type="Path to Bean"/>

An example of this is:

```
<form-bean
        name="logon"
        type="com.athensgroup.eval.ActionForms.LogonForm" />
```

There will be one of these for each of your JSPs that accepts user input.

Action Mappings

There are two forms of these, one for JSPs that don't accept input, and one for JSPs that do accept input. The form for JSPs that don't accept input, and thus does not have an associated ActionForm, is:

```
<action  path="/Path to invoke this page"
       type="path to action class"
       scope="request"
       input="Path to associated JSP">
       <forward name="some name" path="/Path to JSP"/>
</action>
```

An example of this is:

```
<action path="/EvaluationsHome"
      type="com.athensgroup.eval.Action.EvaluationsHomeAction"
      scope="request"
      input="/jsp/EvaluationsHome.jsp">
      <forward name="success" path="/jsp/EvaluationsHome.jsp"/>
</action>
```

The path attribute gives the path to this page.
The type attribute gives the path to the Action class for this Action.
The input attribute gives the path to the JSP for this Action.
The forward attribute gives names to use in your Action class, for example, in your Action class you might have a need to go to different pages based on user input. If you do, you can do it by using the names defined here. You would define one name per page you might want to go to. This allows you to just edit the struts-config.xml file if the target page changes. In this case, since this is the case of no user input, we will just go back to our own page on success. In your Action class you refer to this name in the return of your Action Class thusly:

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

The form for the Action Mapping for JSPs that do accept input is:

```
          <action  path="/Path to invoke this page"
                 type="path to action class"
                 scope="request"
                 input="Path to associated JSP"
                 name="Name for Action Form" >
                      <forward name="some name" path="/Path to JSP"/>
          </action>

An example of this is:
      <action path="/EvaluationsHome"
            type="com.athensgroup.eval.Action.EvaluationsHomeAction"
            scope="request"
            input="/jsp/EvaluationsHome.jsp"
            name="EvaluationsHomeForm">
            <forward name="success" path="/jsp/EvaluationsHome.jsp"/>
      </action>
```

In this case the name attribute specifies the name EvaluationsHomeForm, which must have been previously defined in a Form Beans declaration.

Note that recommend usage of Struts is to have one ActionForm bean for each major section of your application.

Acknowledgements

- Gabriel Sidler [sidler@teamup.ch] made many good suggestions for improving this document, which I gladly incorporated.