The Supercomputing People

Corporate Profile

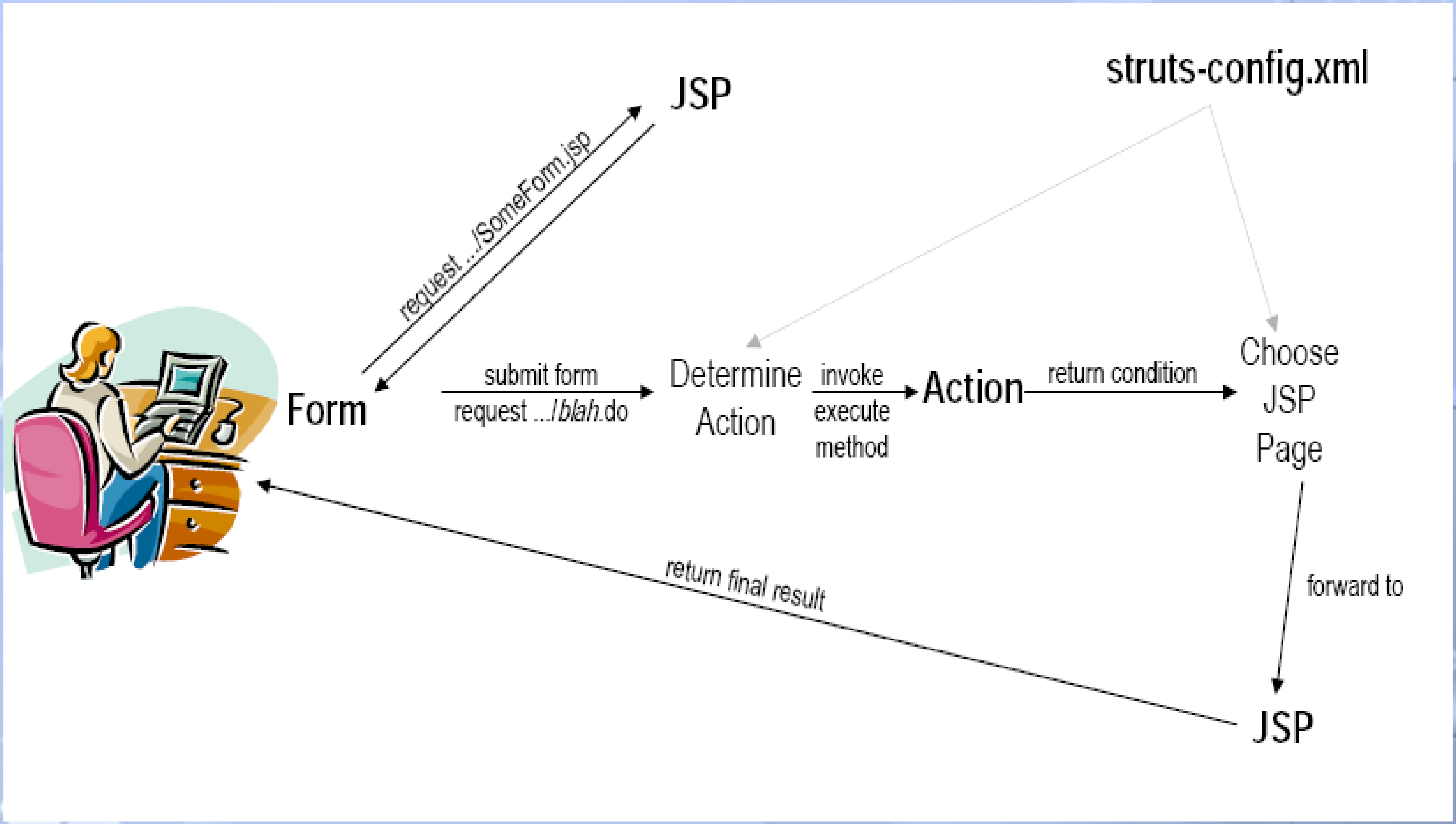
Session-

Struts Actions

Contents

- **Built-in Struct Actions**
  - Action
  - ForwardAction
  - IncludeAction
  - DispatchAction
    - LookupDispatchAction
  - SwitchAction

Struts Control Flow



Creating an Action

To create an action, you must follow these steps:

1. Create an action by subclassing **org.apache.struts.action.Action** as follows:

```
package strutsTutorial;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class DisplayAllUsersAction extends Action {
   ...
}
```

Creating an Action

2. **Override the execute() method.**

- There are three primary things that you want to do with the execute() method:
  - Work with the Model to perform business logic
  - Report any errors to the View
  - Select the next View of the application

Creating an Action

3. Configure the action in the Struts config file.

Now that you have created the action, you need to configure the action in the Struts config file as follows:

```
<action path="/displayAllUsers"
                    type="strutsTutorial.DisplayAllUsersAction">
          <forward name="success“ path="/userRegistrationList.jsp"/>
</action>
```

Action attributes

Attribute	Description
attribute	The attribute specifies the **name of the ActionForm** that is mapped to a given scope. If you do not specify an attribute, the name attribute becomes the value of the attribute. Thus, if the attribute is not specified, the attribute defaults to the name of the ActionForm.
className	The className attribute is used to specify custom config object.
include	You can use the include attribute **to specify a JSP that will handle this request**. The JSP is the only handler of this request. Using this should be the exception, not the norm. This attribute is mutually exclusive with the forward attribute and the type attribute, meaning that only one of the attributes can be set.
input	The input attribute is used to specify the **input View for an action**. The input attribute usually specifies a JSP page that has an HTML form that submits to this action. If there are any problems with form validation, then control of the application will forward back to this input View. This attribute is optional, but if it's not included, an error occurs when Struts attempts to display validation errors.

Action attributes

Attribute	Description
name	The name attribute identifies the **ActionForm** that is related to this action. The html: form tag uses this name to pre-populate the html form. The request processor uses this form name to create and populate an ActionForm to pass to the execute() method of the action handler.
path	The path attribute represents the incoming request path that this action maps to.
parameter	The parameter attribute is similar in concept to servlet init-parameters. Its use is application-specific. You can think of it as a general-purpose init-parameter.
roles	The roles attribute is used to specify J2EE security roles that are allowed to access this action. If this attribute is present, then only those roles specified in the comma delimited list can use this action.
type	The type attribute specifies the action handler for this action. You must specify a fully qualified class name. The type attribute is mutually exclusive with the include and forward attribute.

Action attributes

Attribute	Description
scope	The scope specifies in which scope (request or session) the ActionForm will be placed. The scope attribute defaults to session.
unknown	The unknown attribute is used to specify a default action mapping to handle unknown request paths. Obviously, you can only have one of these for each Struts config file.
validate	The validate attribute is used when you want to validate the ActionForm. This is the case when you are handling form submissions. By specifying validate equals=true, you are causing the request processor to call the ActionForm.validate() method to see if there are any field/form validation errors. The execute() method of the action will not be called if form validation fails and validate equals true. Instead, control of the application would return to the input View. You typically set the validate attribute to false, if you want to pre-populate an HTML form (e.g., an action that loads a form with form data from a database).

ForwordAction

- It is most frequently used built-in Action class.
- ForwardAction allows you to adhere to MVC paradigm when designing JSP navigation.
- Allows you to navigate from one JSP page to another based on the responsibility of the Struts Action Controller.
- The following patterns violates the MVC spirit by directly accessing the JSP PageA to PageB:

  <a href=”PageB.jsp”>Go to Page B</a> (OR)

  <html:link page=”/PageB.jsp”>Go to Page B</html:link>
- Struts provides a built-in Action class called **ForwardAction** to address this issue.

MVC Compliant usgae of LinkTag

- Two steps involved in using the ForwardAction:

1) First, declare the PageA hyperlink that takes you to PageB as follows:

```
<html:link page=”/gotoPageB.do”>Go to Page B</html:link>
```

2) Next, add an ActionMapping in the **Struts-config.xml** file as follows:

```
<action  path=“/gotoPageB”
         parameter=”/PageB.jsp”
         type=”org.apache.struts.actions.ForwardAction” />
```

❖ **All three attributes are mandatory.**

❖ ***Notice: There is no need for ActionForm to this action element.***

Attributes of <html:link>

1) **page** attribute

   e.g.,<html:link **page**=“/gotoPageB.do”>Go to Page B</html:link>

2) **action** attribute

   e.g.,<html:link **action**=“/gotoPageB”>Go to Page B</html:link>

Attributes of <html:link>

3) **forward** attribute

Two steps involved in using the forward attribute:

1) First, declare the PageA hyperlink that takes you to PageB as follows:

```
<html:link forward=”pageBForward”>Go to Page B</html:link>
```

2) Add a Global Forward for “pageBForward” as follows in the globalforwards section:

```
<global-forwards>
        <forward name=”pageBForward” path=”/PageB.jsp” />
</global-forwards>
```

Attributes of <html:link>

- Using two-step approach of forward attribute, the <html:link> gets transformed into the following HTML Link.

  <a href=”App1/PageB.jsp”>Go to Page B</a>

- The HTML Link is now displaying the actual JSP name directly in the browser.
- So, it does not follows MVC pattern.

MVC Compliant usgae of **forward** Attribute

1) First, declare the PageA hyperlink that takes you to PageB as follows:

```
<html:link forward=”pageBForward”>Go to Page B</html:link>
```

2) Define an ActionMapping as follows:

```
<action path=”/gotoPageB”
            parameter=”/PageB.jsp”
            type=”org.apache.struts.actions.ForwardAction” />
```

3) Next, modify the global forward itself to point to the above ActionMapping.

```
<global-forwards>
            <forward name=”pageBForward” path=”/gotoPageB.do” />
</global-forwards>
```

ForwardAction : **parameter** Attribute

- specifies the resource to be forwarded to.
- It can be the physical page like *PageB.jsp* or it can be a URL pattern handled by another controller, maybe somewhere outside Struts.
- E.g.,

```
<action path=”/gotoPageB”
        parameter=”/xoom/AppB”
        type=”org.apache.struts.actions.ForwardAction” />
```

IncludeAction

- IncludeAction is much like ForwardAction except that the resulting resource is included in the HTTP response instead of being forwarded to.
- It is rarely used.
- Its only significant use is to integrate legacy applications with Struts transparently.

IncludeAction

- To access non-Struts resource,

  <jsp:include page=”/xoom/LegacyServletA” />

- To access a Struts resource,

1) <jsp:include page=”/App1/legacyA.do” />

2) <action path=”/legacyA”
parameter=”/xoom/LegacyServletA”
type=”org.apache.struts.actions.IncludeAction” />

DispatchAction

- **DispatchAction** is another useful built-in Struts Action. However you cannot use it as is.
- You will have to extend it to provide your own implementation.
- It provides the implementation for the **execute() method** (no need to override in your class)**,** but declared as **abstract**.
- You can directly implement desired methods.
- They have same signature as execute(….) method.

Package ➔ **org.apache.http.actions.DispatchAction**

steps to setup the **DispatchAction**

1. Create a **subclass** of DispatchAction.

2. Identify the related actions and **create a method for each of the logical actions**. Verify that the methods have the fixed method signature as **execute**.

3. **Identify the request parameter** that will uniquely identify all actions.

4. Define an **ActionMapping** for this subclass of DispatchAction and **assign** the previously **identified request parameter** as the value of the ***parameter*** attribute.

5. Set your **JSP** so that the previously identified request parameter (Step 3) takes on DispatchAction subclass **method names as its values**.

Example : A subclass of DispatchAction

```
public class CreditAppAction extends DispatchAction {
    public ActionForward reject(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) throws Exception
    {
        String id = request.getParameter("id");
        // Logic to reject the application with the above id
                ... ... …
        mapping.findForward("reject-success");
    }
    public ActionForward approve(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) throws Exception
    {
        String id = request.getParameter("id");
        // Logic to approve the application with the above id
```

Example : A subclass of DispatchAction

```
                ... ... ...
        mapping.findForward(“approve-success”);
    }
    public ActionForward **addComment**(ActionMapping mapping, ActionForm
    form, HttpServletRequest request, HttpServletResponse response) throws
    Exception
    {
        String id = request.getParameter(“id”);
        // Logic to view application details for the above id
                ... ... ...
        mapping.findForward(“viewDetails”);
    }
    …
}
```

ActionMapping for DispatchAction

```
<action   path="/screen-credit-app"
          input="/ListCreditApplications.jsp"
          type="mybank.example.list.CreditAppAction"
          parameter="step"
          scope="request"
          validate="false">
          <forward name=“reject-success”  path=“RejectAppSuccess.jsp”
          redirect="true"/>
          …………
</action>
```

Example **JSP**

```
<html:form action=“/ screen-credit-app ">
    //put UI elements
            ……………….
    <html:submit property="step" value="approve" ></html:submit>
    <html:submit property="step" value="reject"></html:submit>
    <html:submit property="step" value="viewDetails"></html:submit>
</html:form>
```

Notes:

- **DispatchAction** knows what parameter to look for in the incoming URL request through this **attribute** named **parameter** in *struts-config.xml.*

- From the value of parameter attribute, it knows the method to be invoked on the subclass.

- Use DispatchAction when a set of **actions is closely related** and separating them into multiple Actions would result in duplication of code or usage of external helper classes to refactor the duplicated code.

- DispatchAction is a good choice when there are **form submissions using the regular buttons** (not the image buttons).

- Just **name all the buttons same**.

Notes:

- If the ActionMapping does not specify a form bean, then the ActionForm argument has a null value.
- In the above example, all the methods got a null ActionForm.
- But that did not matter since the HTTP request parameters were used directly in the Action.
- You can have a Form bean if there are a lot of HTTP parameters (and perhaps also require validation). The HTTP parameters can then be accessed through the Form bean.

LookupDispatchAction

- One way of addressing the problem when regular buttons are used.
- **LookupDispatchAction** is a subclass of **DispatchAction**.
- Package ➔ **org.apache.http.actions.DispatchAction**

steps to setup the **LookupDispatchAction**

1. Create a subclass of **LookupDispatchAction**.

2. Identify the related actions and create a method for each of the logical actions. Verify that the methods have the fixed method signature as similar to execute(…).

3. Identify the request parameter that will uniquely identify all actions.

steps to setup the **LookupDispatchAction**

4. Define an **ActionMapping** in *struts-config.xml in the same way as* DispatchAction. Assign the previously identified request parameter as the value of the **parameter attribute** in the ActionMapping.

5. Implement a method named **getKeyMethodMap()** in the subclass of the LookupDispatchAction. The method returns a **java.util.Map**. The keys used in the Map should be also used as keys in Message Resource Bundle. The values of the **keys in the Resource Bundle** should be the **method names** from the step 2 above.

6. Next, create the buttons in the JSP by using **<bean:message>** or their names. This is very important. If you hardcode the button names you will not get benefit of the LookupDispatchAction.

Example : A subclass of LookupDispatchAction

```
public class CreditAppAction extends LookupDispatchAction {
    public ActionForward reject(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) throws Exception
    {
        String id = request.getParameter(“id”);
        // Logic to reject the application with the above id
                ... ... …
        mapping.findForward(“reject-success”);
    }
    public ActionForward approve(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) throws Exception
    {
        String id = request.getParameter(“id”);
        // Logic to approve the application with the above id
```

Example : A subclass of LookupDispatchAction

```
                  … … …
        mapping.findForward(“approve-success”);
    }
    public ActionForward addComment(ActionMapping mapping, ActionForm
    form, HttpServletRequest request, HttpServletResponse response) throws
    Exception
  {
        String id = request.getParameter(“id”);
        // Logic to view application details for the above id
                  … … …
        mapping.findForward(“viewDetails”);
    }
```

Example : A subclass of LookupDispatchAction

```
    public  Map getKeyMethodMap()
    {
        Map map=new HashMap();
        map.put("button.approve","approve");
        map.put("button.reject","reject");
        map.put("button.addComment","addComment");
    }


}
```

ActionMapping for DispatchAction

```
<action  path="/screen-credit-app"
         input="/ListCreditApplications.jsp"
         type="mybank.example.list.CreditAppAction"
         parameter="step"
         scope="request"
         validate="false">
         <forward name=“reject-success”  path=“RejectAppSuccess.jsp”
         redirect="true"/>
         …………
</action>
```

Example **JSP**

```
<html:form action=“/ screen-credit-app ″>
     //put UI elements
               ……………….
        <html:submit property=”step”> <bean:message key=”button.approve”/>
       </html:submit>
       <html:submit property=”step”><bean:message key=”button.reject”/>
       </html:submit>
       <html:submit property=”step”><bean:message key=”button.comment”/>
       </html:submit>

</html:form>
```

in the Resource Bundle

- button.approve=Approve
- button.reject=Reject
- button.comment=Add Comment

Notes:

- In summary, for every form submission, LookupDispatchAction does the reverse lookup on the resource bundle to get the key and then gets the method whose name is associated with the key into the Resource Bundle (from getKeyMethodmap()).

- That was quite a bit of work just to execute the method.

- DispatchAction was much easier!

- The method name in the Action is not driven by its name in the front end, but by the Locale independent key into the resource bundle.

- Since the key is always the same, the LookupDispatchAction shields your application from the side effects of I18N.

SwitchAction

The SwitchAction class is used to support switching from module to module.

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>config/admin</param-name>
        <param-value>/WEB-INF/struts-config-admin.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

SwitchAction

Perform the following steps:

1. First, map a SwitchAction into the default module as shown here:

```
<action         path="/switch"
                type="org.apache.struts.actions.SwitchAction”>
</action>
```

2. Now, you can set up a forward in the action that edits the users as follows:

```
<action         path="/userSubmit"
                attribute="userForm"
                input="/form/userForm.jsp"
                name="userForm”
                scope="request"
                type="action.UserAction">
        <forward  name="success"
                path="/switch.do?page=/listUsers.do&amp;prefix=/admin”/>
</action>
```

Action Helper Methods

- The **Action** class contains many helper methods, which enable you to add advanced functionality to your Struts applications.

- Some of the functionality includes:
  - Make sure that a form is not submitted twice using **saveToken()** and **isTokenValid().**
  - Display dynamic messages that are i18n-enabled using **saveMessages()** and **getResources()**.
  - Allow users to cancel an operation by using **isCancelled()**.
  - Allow users to change their locale by using **getLocale()** and **setLocale()**.

Thank you!