Common-Controls Guided Tour FormTag's

Version 1.0.5 - Last changed: 01. August 2004

Common Controls

Publisher:
SCC Informationssysteme GmbH
64367 Mühltal (Germany)

Tel: +49 (0) 6151 / 13 6 31 0
Internet www.scc-gmbh.com

Product Site:
www.common-controls.com

Inhaltsverzeichnis

1 Guided Tour FormTag's

1.1 Form types

With the Common-Controls, form types are supplied to the page designer, which are required repeatedly when generating the user interface. Using the form types ensures a uniform design within the application. The standard design can also be customized to your own requirements (Corporate Identity).

The following form types are available:

- Input form
- Display form
- Form for error messages and success messages
- Search dialog
- Header

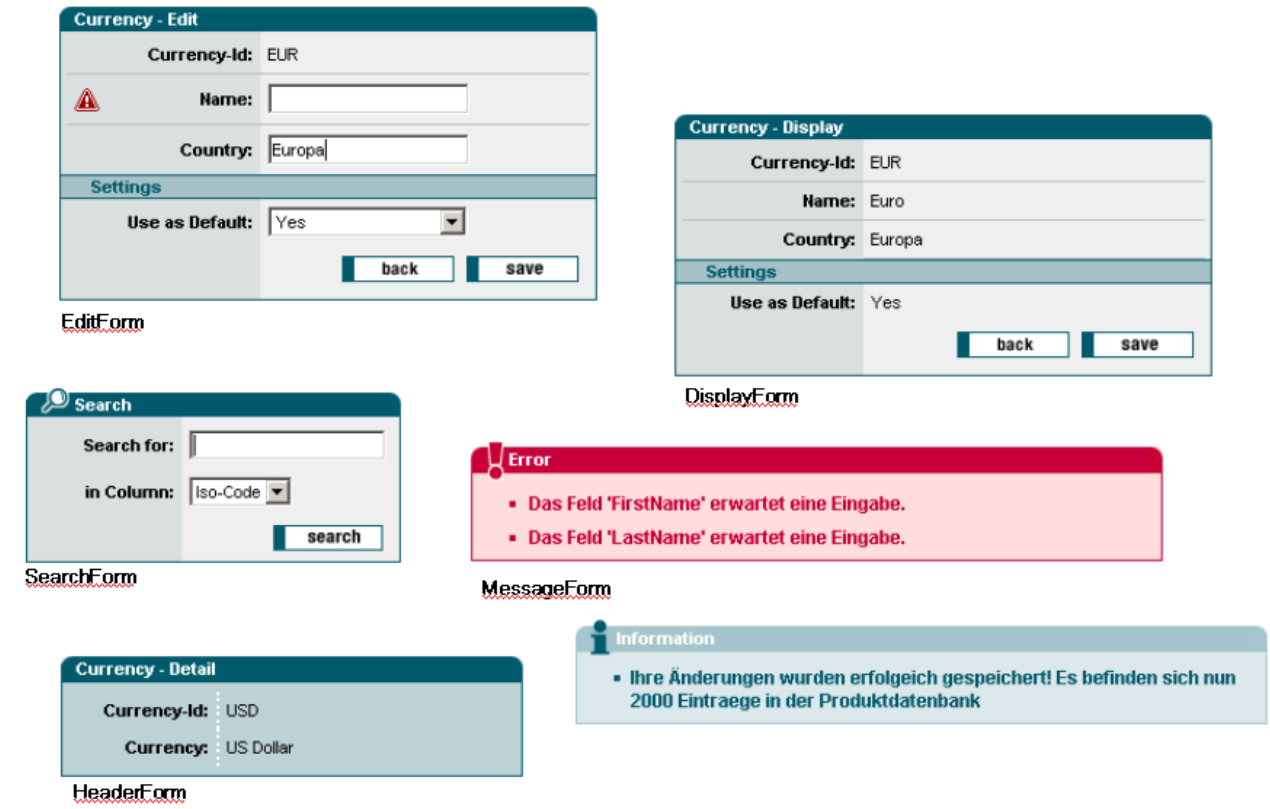
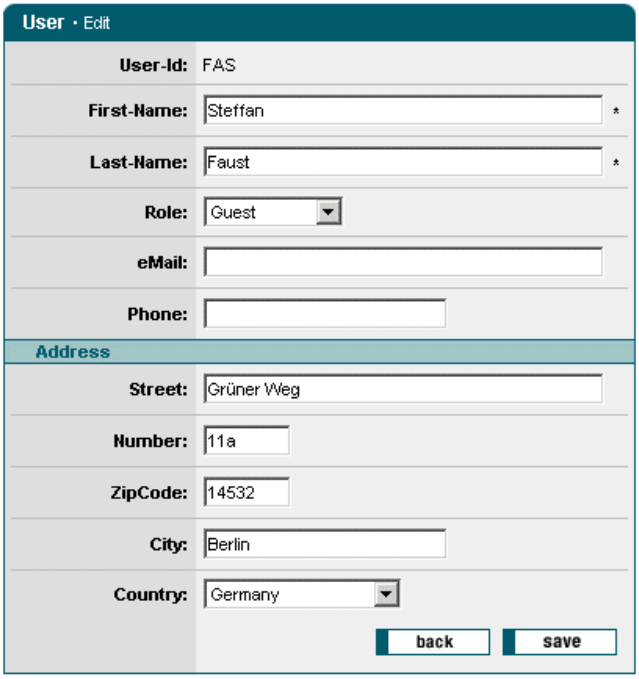


Figure 1: Form types

1.2 Object

In this exercise, we generate an input form and implement two Callback methods for the Back and Save button. The form contains two mandatory fields, which, in case of a validation failure, should result in a corresponding message.



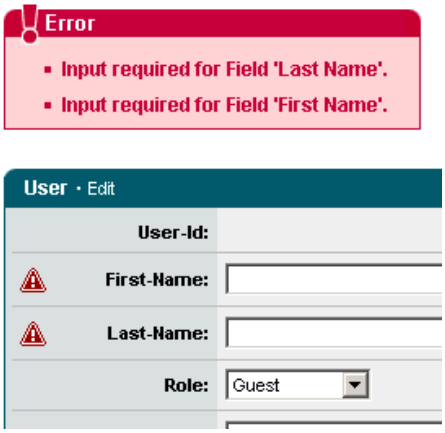


Figure 2: Input form

The following points are handled in this exercise:

1. Selection of the design for the user interface.
2. Generation of the action class.
3. Preparing the form data.
4. Defining the form structure in the JSP-Page.
5. Implementation of the Callback methods.
6. Validation and error presentation.

1.3 Registration of the Painterfactory

The registration of the Painterfactory takes place first. It defines which design the user interface will get. This can be done across the application in the init()-method of the Frontcontroler-Servlet.[1] Here, we select the standard design that the DefaultPainter offers us.[2]

```
import javax.servlet.ServletException

import org.apache.struts.action.ActionServlet;
import com.cc.framework.ui.painter.PainterFactory
import com.cc.framework.ui.painter.def.DefPainterFactory;
import com.cc.framework.ui.painter.html.HtmlPainterFactory;

public class MyFrontController extends ActionServlet {


    public void init() throws ServletException {

       super.init();

       // Register all Painter Factories with the preferred GUI-Layout
       // In this case we only use the Default-Layout.
       PainterFactory.registerApplicationPainter (
              getServletContext(), DefPainterFactory.instance());
       PainterFactory.registerApplicationPainter (
              getServletContext(), HtmlPainterFactory.instance());
    }
}
```

1.4 Derivation of the Action class for the Struts adapter

In our form, we wish to edit the detail information for a user. Therefore, the action-class which takes care of the filling of our form, should have the nomenclature "UserEditAction". The action class is then derived from the class FWAction, which the Struts-action class encapsulates and extends with functionalities of the presentation framework. Instead of the execute()-method, the doExecute()-method is called. [FWAction is derived from org.apache.struts.action.Action]. On calling, it contains the ActionContext, through which the access to additional objects such as the Request- and Response-object is capsulated.

```
import java.io.IOException;
import javax.servlet.ServletException;

import com.cc.framework.adapter.struts.FWAction;
import com.cc.framework.adapter.struts.ActionContext;

public class UserEditAction extends FWAction {

    /**
     * @see com.cc.framework.adapter.struts.FWAction#doExecute(ActionContext)
     */
    public void doExecute(ActionContext ctx)
        throws IOException, ServletException {
        // Code follows in the next chapter
    }
}
```

[1] If it has to be possible for the individual user to choose between different interface designs, then additional PainterFactorys are registered in the user session. This is done mostly in the LoginAction with PainterFactory.registerSessionPainter() in the session Scope.

[2] Additional designs (PainterFactories) are included in the kit of the Professional Edition, or you can develop them yourself.

1.5 Preparing the form data

For filling our form, we make use of a FormBean, the UserEditForm, which can be derived directly from org.apache.struts.action.ActionForm. The FormBean is, in our example, initialized in a simplified manner with our UserObject, which has previously loaded the data for the key transferred in the Request from a database.

```
import java.io.IOException;
import javax.servlet.ServletException;

import com.cc.framework.adapter.struts.ActionContext;
import com.cc.framework.adapter.struts.FWAction;

public class UserEditAction extends FWAction {

    /**
     * @see com.cc.framework.adapter.struts.FWAction#doExecute(ActionContext)
     */
    public void doExecute(ActionContext ctx)
        throws IOException, ServletException {

        String userId = ctx.request().getParameter("userid");

        try {
            // Load the User
            User user = new User(userId);
            user.load();

            // Initialice the Form with the User-Data
            UserEditForm form = (UserEditForm) ctx.form();
            form.setUser(user);

            // In our Example we store the UserObject in our Session
            ctx.session().setAttribute("userobj", user);
        }
        catch (Throwable t) {
            ctx.addGlobalError("Error: ", t);
            ctx.forwardByName(Forwards.BACK);
        }

        // Call the JSP-Page with the Form
        ctx.forwardToInput();
    }

}
```

1.6 Defining the form structure in the JSP-Page

To use the Form Tags on a einer JSP Page, the corresponding tag library has to be declared at the start of the page. Then, the form elements can be referenced with the prefix <forms:*tagname* />. [In addition, the incorporation of the tag libraries must be included in the Deployment descriptor, the WEB-INF/web.xml file]

Our form also contains, apart from input fields and selection boxes, a section for the grouping of information and a button bar. The required elements are defined with the following tags:

Tag	Description
<forms:form/>	Defines the form.
<forms:plaintext/>	Serves for editing text
<forms:text/>	Generates an input field. The Required attribute serves to mark the mandatory fields
<forms:select/>	Defines a selection box
<base:options/>	Defines an option list for a selection box
<forms:section/>	Defines and draws a section
<forms:buttonsection/>	Defines a button bar and specifies a default button
<forms:button/>	Defines and draws a button
<forms:message/>	Draws a message dialog. The dialog is classified as an error dialog using the attribute severity="error"

```
<%@ taglib uri="/WEB-INF/struts-html.tld"      prefix="html" %>
<%@ taglib uri="/WEB-INF/tlds/cc-forms.tld"    prefix="forms" %>
<%@ taglib uri="/WEB-INF/tlds/cc-controls.tld" prefix="ctrl" %>


<forms:message severity="error" caption="Error"/>

<html:form action="/sample101/userEdit">
    <forms:form type="edit" caption="User - Edit" formid="frmEdit">
        <forms:plaintext
            label="User-Id"
            property="userId"/>

        <forms:text
            label="First-Name"
            property="lastName"
            size="45"
            required="true"/>

        <forms:text
            label="Last-Name"
            property="firstName"
            size="45"
            required="true"/>

        <forms:select   label="Role"    property="rolekey">
            <ctrl:base property="roleOptions"/>
        </forms:select>

        <forms:text
            label="eMail"
            property="email"
            size="45"/>

        <forms:text
            label="Phone"
            property="phone"
            size="25" />

        <forms:section title="Address">
            <forms:text
                label="Street"
                property="street"
                size="45"/>
```

```
            <forms:text
                label="Number"
                property="streetnumber"
                size="5"/>
            <forms:text
                label="ZipCode"
                property="zipcode"
                size="5"/>
            <forms:text
                label="City"
                property="city"
                size="25"/>

            <forms:select   label="Country"     property="countrycode">
                <ctrl:options
                    property="countryOptions"
                    labelProperty="country"/>
            </forms:select>
        </forms:section>

        <forms:buttonsection default="btnSave">
            <forms:button name="btnBack" src="btnBack1.gif"/>
            <forms:button name="btnSave" src="btnSave1.gif"/>
        </forms:buttonsection>
    </forms:form>
</html:form>
```

The <forms>-tag in our example is embeded in a Struts <html:form>-tag. Thus, with the specified action (/sample101/userEdit) it gets access to the Form-Bean, which provides the display data. The <forms>-tag can also be used alone without Struts <html:form> tag; however, then, the action attribute must be given additionally.

All tags of the Common Controls library work, if required, in conjunction with the Struts tags!

1.7 Implementation of the Callback methods

The Back- and Save buttons in our form each generate a click event, to which we can react within our action by incorporating two Callback methods. The name of the method is composed of the name of the button and the suffix **onClick**. Form buttons must then be named with the prefix **"btn"**. Otherwise, no Callback method is invoked.

The button **btnBack** thus results in a call to the method **back_onClick**. The method is then passed on to the FormActionContext, which capsulates the access to the Request-, Session Object and the FormBean.

```
import java.io.IOException;
import javax.servlet.ServletException;

import com.cc.framework.adapter.struts.ActionContext;
import com.cc.framework.adapter.struts.FWAction;
import com.cc.framework.adapter.struts.FormActionContext;

public class UserEditAction extends FWAction {

    /**
     * @see com.cc.framework.adapter.struts.FWAction#doExecute(ActionContext)
     */
    public void doExecute(ActionContext ctx)
         throws IOException, ServletException {
         // Code see above
    }

    // ------------------------------------------------
    //              Event Handler
    // ------------------------------------------------

    /**
     * This Method is called when the Back-Button is pressed.
     * @param  ctx  FormActionContext
     */
    public void back_onClick(FormActionContext ctx) {
         ctx.forwardByName(Forwards.BACK);
    }

    /**
     * This Method is called when the Save-Button is pressed.
     * @param  ctx  FormActionContext
     */
    public void save_onClick(FormActionContext ctx) {
         // See next Chapter
    }

}
```

1.8 Validation and error presentation

In our example, the validation of our data is done using the validate()-method in the FormBean. The method is called within the UserEditAction, as soon as the Save button on our form has been clicked. The form can generate a corresponding visual note in front of the field in which an error has occurred. To do so, the error message is set in the ActionErrors-Collection, specifying the corresponding Property.

The following validation results in a message "Input required for Field: First Name" and displays a warning signal for the corresponding field if no input has been made there.

```
import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionMapping;

public class UserEditForm extends UserDisplayForm {

   /**
    * @see org.apache.struts.action.ActionForm#validate()
    */
   public ActionErrors validate(ActionMapping mapping,
         HttpServletRequest request) {

         ActionErrors errors = new ActionErrors();

         if ("".equals(firstName) ) {
               errors.add("firstName",
                     new ActionError("Input Required for Field: ", "First Name"));
         }

         if ("".equals(lastName) ) {
               errors.add("lastName",
                     new ActionError("Input Required for Field: ", "Last Name"));
         }

         return errors;
   }

}
```

The validation is triggered in the save-onClick() method. Any errors occuring thereby are set in the **FormActionContext**. The result of this is that on returning to the input page, the corresponding error messages are displayed.

If the validation was successful, the changes can be accepted in the database. Errors can also occur during this process. Such errors are not displayed in the input screen in our example, but in the screen from which we have come to the editing mode. For this, the error message is also set in the context, and then, the corresponding action is called. The CommonControls Trial Version contains the detailed source code for it.

In case of success, a corresponding message is passed to the user. For this, the text is set using the method addGlobalMessage() into the FormActionContext. Then, the input screen is quit again.

```
import java.io.IOException;
import javax.servlet.ServletException;

import com.cc.framework.adapter.struts.ActionContext;
import com.cc.framework.adapter.struts.FWAction;
import com.cc.framework.adapter.struts.FormActionContext;

public class UserEditAction extends FWAction {

    // other code see above ...

    public void save_onClick(FormActionContext ctx) {

          UserEditForm form = (UserEditForm) ctx.form();

          // Validate the Formdata
          ActionErrors errors = form.validate(ctx.mapping(), ctx.request());
          ctx.addErrors(errors);

          // If there are any Errors return and display a Message
          if (ctx.hasErrors()) {
               ctx.forwardToInput();
               return;
          }

          try {
               // In our Example we get the User-Object from the Session
               User user = (User) ctx.session().getAttribute("userobj");
               populateBusinessObject(ctx, user);
               user.update();
          }
          catch (Throwable t) {
               ctx.addGlobalError("Error: ", t);
               ctx.forwardByName(Forwards.BACK);
               return;
          }

          // Generate a Success Message
          ctx.addGlobalMessage("Data updated: ", form.getUserName());

          ctx.forwardByName(Forwards.SUCCESS);
    }

}
```

1.9 Tour End

The example has shown how interfaces can be generated more quickly by the use of Form tags. Adherence to a uniform design within the application is also guaranteed However, other designs can also be implemented by customizing the Painter. Various Designs can be used in parallel.

Features Form-Tag's:

- Supports the following form elements (Text, Plaintext, Textarea, Select, Button, Buttonsection, File, Password, Radio, Spin, Description, Checkbox, Section)
- Possibility for grouping form elements.
- Form elements can be easily declared with a label, description text, the required input etc. in the JSP-Page.
- Mapping of Form events to Event-Handler in the action class.
- Visualization in case of faulty inputs.
- Supporting 'Hover' effects in case of buttons.
- Design of the form can be defined in the JSP page or also on the server side.
- Design through Painterfactory can be matched to own StyleGuide (Corporate Identity).
- Same Look and Feel in Microsoft InternetExplorer > 5.x and Netscape Navigator > 7.x

1.10 Exkurs: Providing buttons with Hover effect

For depicting the Hover effect, one button is needed in the active state and one in the selected state. The active button is saved as a gif file with the prefix **btn** and the suffix **1.gif** (e.g. btnBack1.gif). For the Hover effect, one button is required with the ending **3.gif** (e.g. btnBack3.gif).

Name conventions and states for form buttons:

State	Example	Name convention
Active		btnXXX1.gif
Inactive		btnXXX2.gif
HOver		btnXXX3.gif
Pressed		btnXXX4.gif

The images are automatically exchanged as soon as the mouse pointer is moved over a form button. A JavaScript Eventhandler, which is registered for the MouseOver and MouseOut-Event in the file fw/def/jscript/controls.js, is responsible for this.

This script is automatically included in every HTML page by the default painter.

2 Glossary

C

CC

Common-Controls