WoodySample

A simple Woody example

In this example we will show how to create a simple registration form using Woody and flowscript. We will follow to following steps:

1.Create a form definition file
1.Create a template file for the Woody template transformer
1.Write a bit of flowscript
1.Add some pipelines to the sitemap

Here is a screenshot of the form we're going to create:

http://outerthought.net/~bruno/images/woody_registrationform_initial.png

Note: *This example is included in the Woody block, you may want to play with it first before reading the explanation below*.

Create a form definition file

Below the form definition file is displayed. This lists all the widgets in the form, together with their configuration information.

```
<wd:form
  xmlns:wd="http://apache.org/cocoon/woody/definition/1.0">

  <wd:widgets>
    <wd:field id="name" required="true">
      <wd:label>Name:</wd:label>
      <wd:datatype base="string">
        <wd:validation>
          <wd:length min="2"/>
        </wd:validation>
      </wd:datatype>
    </wd:field>

    <wd:field id="email" required="true">
      <wd:label>Email address:</wd:label>
      <wd:datatype base="string">
        <wd:validation>
          <wd:email/>
        </wd:validation>
      </wd:datatype>
    </wd:field>

    <wd:field id="age">
      <wd:label>Your age:</wd:label>
      <wd:datatype base="long">
        <wd:validation>
          <wd:range min="0" max="150"/>
        </wd:validation>
      </wd:datatype>
    </wd:field>

    <wd:field id="password" required="true">
      <wd:label>Password:</wd:label>
      <wd:datatype base="string">
        <wd:validation>
          <wd:length min="5" max="20"/>
        </wd:validation>
      </wd:datatype>
    </wd:field>

    <wd:field id="confirmPassword" required="true">
      <wd:label>Re-enter password:</wd:label>
      <wd:datatype base="string">
        <wd:validation>
          <wd:assert test="password = confirmPassword">
            <wd:failmessage>The two passwords are not equal.</wd:failmessage>
          </wd:assert>
        </wd:validation>
      </wd:datatype>
    </wd:field>

    <wd:booleanfield id="spam">
      <wd:label>Send me spam</wd:label>
    </wd:booleanfield>
  </wd:widgets>

</wd:form>
```

All elements are in the Woody Definition namespace: **wd**.

Every definition file has a **<wd:form>** element as the root element.

The child widgets of the form are defined inside the **<wd:widgets>** element. As you can see, most of the widgets are **field** widgets. The **field** widget is the most important widget in Woody. It is very flexible because it can be associated with different datatypes and with a selection list. See the WoodyWidgetReference for more information on this and other widgets.

A nice feature is that the **wd:label** tags can contain mixed content. On the one hand, this can be used to provide rich formatting in the label. But it also enables you to put i18n-elements in there, to be interpreted by the I18NTransformer. This way, internationalisation is done using standard Cocoon techniques.

Create a template file for the WoodyTemplateTransformer

Here's the template for our registration form example:

```
<html xmlns:wt="http://apache.org/cocoon/woody/template/1.0"
  xmlns:wi="http://apache.org/cocoon/woody/instance/1.0">
  <head>
    <title>Registration form</title>
  </head>
  <body>
    <h1>Registration</h1>
    <wt:form-template action="#{$continuation/id}.continue" method="POST">
      <wt:widget-label id="name"/>
      <wt:widget id="name"/>
      <br/>
      <wt:widget-label id="email"/>
      <wt:widget id="email"/>
      <br/>
      <wt:widget-label id="age"/>
      <wt:widget id="age"/>
      <br/>
      <wt:widget-label id="password"/>
      <wt:widget id="password">
        <wi:styling type="password"/>
      </wt:widget>
      <br/>
      <wt:widget-label id="confirmPassword"/>
      <wt:widget id="confirmPassword">
        <wi:styling type="password"/>
      </wt:widget>
      <br/>
      <wt:widget id="spam"/>
      <wt:widget-label id="spam"/>
      <br/>

      <input type="submit"/>
    </wt:form-template>
  </body>
</html>
```

The Woody-specific elements here are in the "Woody Template" namespace: **wt**.

The **<wt:widget-label>** tag will cause the label of a widget to be inserted at the location of the tag. The **<wt:widget>** tag will cause the XML representation of a widget to be inserted at the location of that tag. The inserted XML will be in the "Woody Instance" namespace: **wi**.

The XML representation of the widget will then be translated to HTML by an XSLT stylesheet (woody-samples-styling.xsl in our case – see sitemap snippets below). This XSLT only has to handle individual widgets, and not the page as a whole, and is thus not specific for one form but can be reused across forms.

For certain widgets it may be necessary to provide extra presentation hints, such as the width of a text box, the style of a selection list (drop down, radio buttons, ...) or class and style attribute values. This can be done by putting a wi:styling element inside the wt:widget element. This element is in the wi namespace because it will be copied literally. The attributes and/or content of the wi:styling element depend on what is supported by the particular stylesheet used.

As an alternative to the template approach, you could also use the WoodyGenerator, which will generate an XML representation of the whole form, and style that with a custom-written XSLT. Using the Woody template transformer is much easier though.

Write a bit of flowscript

Flowscript is Cocoon's solution to handling the flow of a web interaction. It is probably unlike anything you've seen before, because it is based on the concept of *continuations*. If you had not heard of it before, be sure to read about it, or otherwise you won't understand how the code below works.

Here's the flowscript for our example, `registration.js`:

```
cocoon.load("resource://org/apache/cocoon/woody/flow/javascript/woody2.js");

function registration() {
    var form = new Form("forms/registration.xml");

    form.showForm("registration-display-pipeline");

    var model = form.getModel();
    var bizdata = { "username" : model.name }
    cocoon.sendPage("registration-success-pipeline", bizdata);
}
```

The flowscript works as follows:

First we create a Form object, specifying the form definition file to be used. The Form object is actually a javascript wrapper around the "real" Java form instance object.

Then the showForm function is called on the form object. This will (re)display the form to the user until validation of the form succeeded. As parameter to the showForm function, we pass the sitemap pipeline to be used to display the form.

Finally we get some data from the form (the entered name), and call a sitemap pipeline to display this data. This pipeline is based on the JXTemplate generator.

Note that the javascript Form object, and the showForm function, have more features not shown here, such as:

- ability to specify a validation function for custom validation
- ability to specify bizdata to be passed on to the form display pipeline
- ability to load and save the data in the form from and to Java beans or XML documents using the Woody binding framework

Add some pipelines to the sitemap

First of all, do not forget to register the `registration.js` file in the map:flow section of the sitemap, as follows:

```
<map:flow language="javascript">
  <map:script src="flow/registration.js"/>
</map:flow>
```

And here are the pipelines we need:

```
<map:match pattern="registration">
  <map:call function="registration"/>
</map:match>

<map:match pattern="*.continue">
  <map:call continuation="{1}"/>
</map:match>

<map:match pattern="registration-display-pipeline">
  <map:generate src="forms/registration_template.xml"/>
  <map:transform type="woody"/>
  <map:transform type="i18n">
    <map:parameter name="locale" value="en-US"/>
  </map:transform>
  <map:transform src="resources/woody-samples-styling.xsl"/>
  <map:serialize/>
</map:match>

<map:match pattern="registration-success-pipeline">
  <map:generate type="jx" src="forms/registration_success.jx"/>
  <map:serialize/>
</map:match>
```

The first two are for managing the flowscript: when someone hits the registration URL, we call the registration function in our flowscript.

When a form is submitted, it will be matched by the second matcher, `*.continue`, which will continue the execution of the flowscript.

The third matcher is for displaying the form, and uses the Woody template transformer.

The fourth pipeline is for showing the "success" page using the JXTemplate generator, here is the contents of the registration_succcess.jx page:

```
<html>
  <head>
    <title>Registration successful</title>
  </head>
  <body>
    Registration was successful for ${username}!
  </body>
</html>
```

Historical note on actions

Long ago, when animals still spoke french and actions were still the way to do webapplications in Cocoon, the sample on this page was based on actions (the map:act construct from the sitemap) and XSP. You can find that sample WoodyActionSample.