Portal: Using forms (2.1 legacy document)

Table of contents

1 Comments.....5

Table of contents

1 Overview 3
2 Including Applications 3
3 Building forms 4
3.1 A Sample 4
3.2 Several instances of an application 5
3.3 Extending the Sample 5

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. Overview

This document gives an overview over how to use forms within the portal engine.

The sample portal that comes with the Cocoon distribution contains a working sample for form handling.

2. Including Applications

The portal allows to include a complete web application (with forms, links etc.) that is build with Cocoon. Therefore it's possible to develop the forms like you usually do with Cocoon without thinking about the portal. When you are finished just include this application into the portal as a coplet. The following shows you how to configure such an application as a coplet.

For including complete applications, the portal offers a specific coplet adapter, the caching URI adapter. This is configured as the coplet base type: *CachingURICoplet*. So each coplet you configure has to use this base type.

For each application you have to configure a coplet data object in the profile:

... <coplet-data id="app-test-two" name="standard"> <title>Application Test</title> <coplet-base-data>CachingURICoplet</coplet-base-data> <attribute> <name>buffer</name> <value xsi:type="java:java.lang.Boolean" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> true </value> </attribute> <attribute> <name>handleParameters</name> <value xsi:type="java:java.lang.Boolean" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> true </value> </attribute> <attribute> <name>uri</name> <value xsi:type="java:java.lang.String" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> cocoon:/coplets/html/application </value> </attribute> <attribute> <name>temporary:application-uri</name> <value xsi:type="java:java.lang.String" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> cocoon://samples/flow/jxcalc/ </value> </attribute> </coplet-data> ...

As usual, the coplet data gets a unique id, a title and in this case the reference to the *CachingURICoplet*. In addition the *buffer* attribute is used to buffer the output of the coplet which avoids broken responses in the case of a malformed stream comming from the included application.

The *handleParameters* attribute has to be set to *true* as well, as the application has to handle it's own request parameters (for links or forms).

The *uri* attribute points to a Cocoon pipeline that will include your application. This is not the pipeline of your application itself. The starting URI for your application has to be configured using the attribute *temporary:application-uri*.

With this configuration you can configure instances of the coplet for each user. In addition a user can have several instances of the same application. If you look at the provided samples, you see two instances of a flow example and two instances of a forms sample at the same time for each user.

However, if you allow several instances per user of the same application, this application has to be developed with this aspect in mind. More about this topic later on.

So, basically this is all you have to do. Develop your application standalone without the example and

include it as outlined above. You can for example invoke the sample above (cocoon://samples/flow/jxcalc/) directly without the portal.

3. Building forms

In this chapter we demonstrate using a sample how to build forms that can be used within the portal. We will use Cocoon flow to define the logic for the form, but for your own form you can of course use a different approach as well. If you want to have complex forms, you can also use Cocoon forms (Woody), but you have to be careful with correctly using JavaScript on the client, which means you have to add the JavaScript to the response in the main portal pipeline and not by the form itself. Or you avoid using JavaScript on the client :)

As outlined in the previous chapter, you can define your form handling application without taking care about the portal, so let's start developing it.

3.1. A Sample

We will use flow for the logic. Each time the application is invoked, the same URI is used; this URI calls a flow function. Inside this function we check whether we have to display the form or a different (result) page. So our sitemap looks like this:

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

We have one single function in the flow, that checks if a session attribute already contains a value or not. If no value is stored in the session, this means that either the form has to be displayed or the user just submitted some values. So we have to distinguish these two cases as well:

```
... function form() { // is the value stored in the session? if (
cocoon.session.getAttribute("form") == null ) { // No: is this a submit? var name =
cocoon.request.getParameter("name"); if ( name == null ) { // No: display the form
cocoon.sendPage("page/form", {}); } else { // It's a submit, so process the value
cocoon.session.setAttribute("form", name); cocoon.sendPage("page/received", {"name" :
name}); } } else { // just display the value var name = cocoon.session.getAttribute("form");
cocoon.sendPage("page/content", {"name" : name}); } }...
```

This schema allows to use the same pipeline for all purposes. However, if you want a different design, you could e.g. use a different pipeline for processing the form data etc.

In each case, the view is called which is a Cocoon pipeline. For example the pipeline for the form reads an XML document that looks like this:

```
... <page> <title>Form</title> <content> <form method="post" action="form"> <para>Please
enter your <strong>name</strong>: <input type="text" name="name"/></para> <input
type="submit" name="submit" value="Enter"/> </form> </content> </page> }...
```

As already pointed out, you develop your application like you would do without the portal. Note in the sample above that the target of the form is the *form* pipeline, containing the call to the flow.

So, how does this work? Each link (or target of a form action) is rewritten by the portal engine. The link is transformed into an event. When now the user activates such a link (or form) the event is send to the portal and the portal updates the URI to call for the portlet. The *portal-html-eventlink* transformer does the rewriting of all links and forms in combination with the *portal-coplet* transformer that is one of the last transformers in the main portal pipeline.

Each application portlet that isn't changed during this single request/response cycle, isn't "activated" which means the corresponding application pipeline is not called.

The portal caches the response of an application and serves the coplet out of the cache until an action for this coplet is triggered.

3.2. Several instances of an application

If you allow the user to have several instances of the same application, the application has to be aware of this fact. Imagine that each instance should have its own data set, but as the user is the same, the session is shared by the instances. So, a unique identifier for each instance is required.

The portal framework already supplies this identifier and passes it as a request parameter to the pipeline of your application. The name of the parameter is *copletid* and the value is the unique id. In our sample, we pass this identifier to the flow script using an input module:

```
... <map:match pattern="form"> <map:call function="form"> <map:parameter
name="copletId" value="{request-param:copletid}"/> </map:call> </map:match> ...
```

In the flow script, we can now use this identifier to calculate a unique session key to store the information:

```
// get the coplet id var cid = cocoon.parameters["copletId"]; var key = cid + "/myform";
```

From now on you can use this key to get/store session attributes etc. In addition you can use all features available to flow, like looking up components and using them.

3.3. Extending the Sample

Although the sample is very simple (or more precisly the form is very simple) it demonstrates a possible way to build coplets that handle forms. You can use this as a guideline for your own forms.

If you need, e.g. validation of the form, you can simply add the validation in the flow script and return a corresponding view in the case of a validation error. If you need a more complex processing, e.g. sending an email, you can simply add this in the flow script as well. Or you can code the logic into a Java class and call this class from the flow script.

In addition, you can use Cocoon forms with all the nice features (widgets, validation, binding etc.) as well. You only have to provide a working set of stylesheets that work in the portal.

1. Comments

add your comments