# HowToCFormsInLenya

*Note: This is as much an Howto as it is a basis for discussion if this is the right approach to the problems. Please post you comments to this on the mailing list.*

## Editing in and using Cocoon Forms with Lenya

There are very few real life websites that do not contain any forms. But as of 1.2, Lenya has no mechanism neither to edit a form nor to handle the flow that forms imply most of the time. Even if you want to create a very simple contact form that will allow a user to leave a message to you into your website, the data that the users submits by filling in and submitting the form needs to go somewhere. And the user needs to get some feedback that his entry was received. This is where flow comes in and why forms and flow are often interrelated.

### Cocoon Forms Backgrounder

*Note: The Cocoon Forms Framework has been known as Woody before. If got renamed in Cocoon 2.1.5 to Cocoon Forms. The Forms block in Cocoon is still marked as unstable, but it is likely that it will not change too much anymore. Having said that you should make sure you use a version of Lenya that is compiled based on Cocoon 2.1.5 (not: Cocoon 2.1.4 or even earlier) as the names have changed to reflect the name change from Woody to Coocon Forms.*

The author does not intend to duplicate the Flow and Forms Howtos here. In case you are not yet familliar with the concepts of Cocoon Forms, please read the appropriate Cocoon documentation:

- http://cocoon.apache.org/2.1/userdocs/flow/index.html
- http://cocoon.apache.org/2.1/userdocs/forms/index.html

### Why all that hassle and not just use plain HTML forms ?

Of course you could use plain HTML forms in Lenya. But this doesn't meet todays requirements for a well done website anymore. If you are using plain HTML forms, you are entirely on your own, especially regarding:

- Validation and data type handling: HTML only knows a text input widget. It does not provide any means to make sure that in a date field for example there is a valid date entered. Of course you can parse whatever input you get back from your HTML form and implement your own error handling, but this is exactly what has been done by the CForms developers already.
- Advanved Widgets: CForms has some advanced features for really nice looking and nicely working forms available that you can use. They are all based on standards, not at all browser specific. See the Forms block examples in Cocoon. Sooner or later you would want to have this to make you site state of the art.

### Preparing Lenya for Cocoon Forms

If you compiled Lenya based on Cocooon 2.1.5, all Java classes that are necessary to use Control Flow and Cocoon Forms should already be in your webapp. It is just a matter of adding these components to the sitemap. There are four sitemap components that need to be added, two generators and one transformer.

Edit the {LENYA_WEBAPP}/sitemap.xmap file and add these lines in the apporpriate sections:

```
  <map:components>

    <map:generators default="file">

      [...]

      <map:generator logger="forms" name="forms" src="org.apache.cocoon.forms.generation.FormsGenerator"/>
      <map:generator label="content" logger="sitemap.generator.jx" name="jx" pool-grow="2" pool-max="16" pool-
min="2"
      src="org.apache.cocoon.generation.JXTemplateGenerator"/>

    </map:generators>

    [...]

    <map:transformers default="xslt">

      [...]
      <map:transformer logger="sitemap.transformer.jx" name="jx" pool-grow="2" pool-max="16" pool-min="2"
        src="org.apache.cocoon.transformation.JXTemplateTransformer"/>
      <map:transformer logger="forms" name="forms" src="org.apache.cocoon.forms.transformation.
FormsTemplateTransformer"/>
    </map:transformers>
```

*Note: It might be possible that it's sufficient to add these in the publication sitemap so that these changes would be local to a specific publication which would make it easier to move the publication around between different Lenya implementations. Someone might care to check this and alter this section accordingly.*

# Integrating Cocoon Forms and Flow into your Lenya Publication

Up to this point, the Integration is very straigt forward and should have been quite simple to manage. But now comes the (little bit) harder stuff. We should step back and think for a moment.

## Comparing Websites and Web Applications

A simple website is a collection of pages that are linked together. This is where the word "web" originally comes from. For most websites, this means that there is a finite number of pages. Each link in a page links to a distinct other page. This has nothing to do with static and dymamic websites at all. You can still have dynamic content inside each of your pages, such as the local weather for example. But you will still have one page that displays the weather information, that has a name like weather.html (or weather.php, weather.jsp, etc.) and will be called from a distinct link on some other page.

A web application is different. It does not consist of a finite number of pages but pages are generated as needed by appliation logic on the server. Each link in a page does not directly link to some other page but to a piece of code in the application logic which then will decide what page will get generated and send to the user's browser. In other words: You cannot tell what's behind a link.

## WYSIWYG for Web Applications ?

The basic concept of Lenya is WYSIWYG - What You See Is What You Get. For editing a website this means that the editor can navigate the site in authoring mode the same way the end user would do in live mode. It's just that in authoring mode there are the CMS menus that will allow the editor at any given point in the site to issue commands such as:

- I want to edit the page I am seeing
- I want to add a new page
- I want to delete a page

This concept is based on the assumption that the links between pages are static. As we found out already, this is not true for a web application. Think of a the typical shopping card for example. You will present a page to the user with the current content of the cart. On that form the user will be able to click on a number of links, such as "Add an item", "Change quantity", "Delete an item", "Checkout", "Login", "Create a profile".

If the user clicks on "Checkout" for example, the flow may fork. Depending on application logic on the server, the user might get a number of different pages after clicking the "Checkout" button:

- A page saying "We have received your order, thank you!"
- A login page asking to login if you already have an account.
- If the user is logged in, depending on his profile, maybe a selection of addresses to deliver to if he has multiple addresses in his profile. But this screen will never be displayed if the user only has one shipping address in his profile.

When you are building a webapp based on Cocoon Flow, so-called continuations will be used as links. They are random generated hashes which will allow the logic engine on the server to determine the point in the program logic on the server where the process left off when the page was displayed. In other words, you cannot tell what `http://www.yoursite.com/84715e431a200b113d6a257b31530d164c4f281b.continue` stands for. Only the application logic can and it will depend on the state of the current session instance what it means.

So unless we would implement the often tried method of "programming via mouse clicks" into Lenya (what's currently not on the roadmap and probably will not get there at all) this means that the paradigm of navigate and edit won't work for creating a web application. It would not even make too much sense if it was implemented, because you'd have to provoke any potential error condition to get to the error page and edit it. You probably don't want to work that way.

## Integrating the Control Flow into the Lenya site navigation

Using Lenya at all will probably only make sense if your are building a site which is mostly a website and will only have some parts that are more a web application. If for example you have a lot of just pages of information on your site, but you want to have a section on the site where users can order printed brochures, you need a way to edit the 90% of the site the same way you would want to if the other 10% weren't there.

So here is an approach of how to do this:

### Files needed for Cocoon Forms and Flow and their relationship with the Lenya framework

*Note: This is getting technical now. If you haven't yet taken the time to understand how Flow and CForms work together, you should do so now. Otherwise it will be difficult to follow this part.*

You need an entry point into the webapp part of the site that can be called from the website part of the site. This can be a link to a virtual document with the extension .flow. (You could give it any other name, just avoid using an extension that is used for any existing files for clarity. On the other hand, if you like security be obscurity, you can use .html here.)

The .flow link needs to match a pipeline which will make a call into the Javascript application logic. From that point on, the Javascript is in control. What it will typicall do is:

1. Instantiate a form instance from a form defintion. (Note: A form instance is NOT the layout of the form, but the model behind it.)
2. Instruct Cocoon / Lenya to render the form by actually rendering a form template with the actual form widgets from the form instance.

At this point, the user will probably fill in and submit that form to the continuation that the Flow engine has put into the template. Control is passed back to the application logic.

1.#3 The application logic will apply validation and decide if the form needs to be re-displayed because of validation failures.

1. Once the form passed validation, the data entered is available to the application logic.
2. The application logic can do whatever processing is required and will most likely display a result page that contains either some of the information entered on the form or information that got derived from the information entered. To achieve this, it will instruct Cocoon / Lenya to render a JX template with the business data injected into the pipeline.

There are four types of files needed in this process:

- The Flow script. This is a JavaScript file.
- The Form definition. This file will define the widgets available with their data types and validation rules. It does not contain any layout information.
- The Form template. This is the layout template for the form. The Cocoon Forms transformer will look for <ft:...> elements and replace them with actual HTML widgets.
- The JX template. This is a layout template in which either the Cocoon JXGenerator or the Cocoon JXTransformer will replace the ${variablename} elements with actual data.

Among those four types of files, only two of them are related to layout:

- The Form template
- The JX template

Therefore these two file types are candidates for being edited with Lenya and rendered through the Lenya framework. The other two types of files need to be edited manually directly on the server.

Note: *In practice it needs to be checked how that can be done in a useful way. So for now let's edit them manually.*

As long as there are no really good solutions to this (that will for example solve the problem that Kupu does not properly handle non-XHTML-elements in a page) a solution that worked for the author is:

- Create each template (no matter if Form template or JX template) as a document in Lenya.
- Set visibility to "false". This will make the pages invisible in the Lenya navigation menu.
- Edit the files manually on the server to add the JX and / or Form template (<ft:...>) elements
- Add matchers to the publication sitemap which will match only on these documents and run them through a JX and / or Forms transformer besides all transformers that apply the Lenya framework to the documents.

As a result, the forms and JX template pages will be displayed seamlessly with the Lenya navigation and styles. which is what we wanted!

**What happens if the form is redisplayed during auto validation? I guess the Lenya menu etc. will be included in the page, which is redisplayed? /roku**

**With the added diff not. You need tweak actualy more files. The problem is that *.continue is not linking back to the sitetree. That way the menu is not displayed. I will add my notes how to do it soon**

## Editing the Publication's sitemap

*Note: This is based on the "Registration" sample from the Coocoon documentation (see Link above).*

```
--- sitemap.xmap.orig   2005-01-08 11:37:20.000000000 +1100
+++ sitemap.xmap        2005-01-08 17:28:00.000000000 +1100
@@ -19,18 +19,27 @@

 <map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">

   <map:components/>

+  <map:flow language="javascript">
+    <map:script src="flow/registration.js"/>
+  </map:flow>
+
   <map:resources>
     <map:resource name="style-cms-page">
       <map:transform type="i18n">
         <map:parameter name="locale" value="{request:locale}"/>
       </map:transform>
       <map:transform src="../../xslt/util/page2xhtml.xsl">
         <map:parameter name="contextprefix" value="{request:contextPath}"/>
       </map:transform>
+      <map:transform type="forms"/>
+      <!-- <map:transform type="i18n">
+          <map:parameter name="locale" value="en-US"/>
+        </map:transform> -->
+      <map:transform src="resources/forms-samples-styling.xsl"/>
       <map:transform src="../../xslt/util/strip_namespaces.xsl"/>
       <map:select type="parameter">
         <map:parameter name="statusCode" value="{statusCode}"/>
         <map:when test="">
           <map:serialize/>
@@ -43,10 +52,36 @@
   </map:resources>

   <map:pipelines>

     <map:pipeline>
+      <map:match pattern="**/registration">
+        <map:call function="registration"/>
+      </map:match>
+
+      <map:match pattern="**/*.continue">
+        <map:call continuation="{2}"/>
+      </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="forms"/>
+<!--        <map:transform type="i18n">
+          <map:parameter name="locale" value="en-US"/>
+        </map:transform> -->
+        <map:transform src="resources/forms-samples-styling.xsl"/>
+        <map:serialize/>
+      </map:match>
+
+      <map:match pattern="registration-success-pipeline.jx">
+        <map:generate type="jx" src="forms/registration_success_jx.xml"/>
+        <map:serialize/>
+      </map:match>
+
       <map:match pattern="**">
         <map:mount uri-prefix="" src="publication-sitemap.xmap"/>
       </map:match>

       <map:handle-errors>
```

## Files needed for above example

To get these files for the registration sample, look in your cocoon source,

- cocoon-2.1.6/src/blocks/forms/samples/flow/registration.js
- cocoon-2.1.6/src/blocks/forms/samples/forms/registration_success_jx.xml
- cocoon-2.1.6/src/blocks/forms/samples/forms/registration_template.xml
- cocoon-2.1.6/src/blocks/forms/samples/forms/registration.xml

You'll also need to copy the xsl files across from the

- cocoon-2.1.6/src/blocks/forms/samples/resources

These all needs to be copied to your publication directory under the same directory names as they were in cocoon (flow, forms, resources)