# FlowAndXsltHelloWorld

## Page title (cookbook approach, tutorial , ...)

- TARGET-AUDIENCE: ***beginner*** advanced expert
- COCOON-RELEASES: 2.0.3, 2.0.4
- DOCUMENT-STATUS: ***draft*** reviewed released

---

### What you will get from this page

A start on using XSLT to transform output generated by JX and Flow as used in FlowHelloWorld. I read the FlowHelloWorld tutorial and found it helpful, but slightly too simple. It uses JX to generate XHTML directly. I wanted to use JX to generate XML, which could then be transformed using XSLT to any format I desire. This is my attempt.

If any experts here know a better way to do this, please change this page. I tried several alternatives that involved less modification to the initial FlowHello World example, but couldn't get anything to work without the internal pipeline.

### Your basic skills

- You have read the FlowHelloWorld example, tried it out, and understand it.
- You have basic knowledge of XSLT.

### Technical prerequisites

- A cleanly installed version of Cocoon (see BeginnerInstallation)

### Setting up

This project has almost the same structure as the FlowHelloWorld example. Make a directory called, for example, `hello-xslt` in your Cocoon directory. In `hello-xslt`, make the directories `flow`, `documents` and `stylesheets`.

#### The Sitemap

The sitemap, saved as `sitemap.xmap` in your `hello-xslt` directory, is slightly modified from the normal Hello, World example and looks like this:

{{{<?xml version="1.0" encoding="UTF-8"?>

<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">
<map:components>
<map:generators default="file">

<map:generator label="content,data" logger="sitemap.generator.jxt"
name="jxt" src="org.apache.cocoon.generation.JXTemplateGenerator"/>

</map:generators>

<map:transformers default="xslt"/>

<map:serializers default="html"/>
<map:matchers default="wildcard"/>
<map:selectors default="browser"/>

<map:actions/>
<map:pipes default="caching"/>
</map:components>

<map:views/>
<map:resources/>
<map:action-sets/>

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

<map:pipelines>
<map:component-configurations>
<global-variables/>
</map:component-configurations>

<map:pipeline>
<!-- This is what triggers when somebody visits the page.-->
<map:match pattern="">
<map:call function="hello"/>
</map:match>

<!-- In the FlowHelloWorld example, there was a match on *.jxt here.
I took it out because my JXT files do not map directly to output. -->
</map:pipeline>

<!-- An internal pipeline can be called from within Cocoon, but does not
map to URLs entered in the browser -->
<map:pipeline internal-only="true">

<!-- The flowscript invokes this part of the pipeline. It's internal-only so
that the flowscript can serve it, but people can't happen upon it. -->
<map:match pattern="internal/hello">
<map:generate type="jxt" src="documents/hello.jxt"/>
<map:transform type="xslt" src="stylesheets/hello.xsl"/>
<map:serialize type="xhtml"/>
</map:match>

</map:pipeline>

</map:pipelines>
</map:sitemap>
}}}

### The Flowscript

The flowscript should be saved as `hello-xslt/flow/hello.js`.

It has been modified to call the internal pipeline, and to be a bit more extroverted, saying "Hello" to more than just the world.

```
function hello() {
  var name =  "World";
  var name2 = "Everything Else";
  cocoon.sendPage("internal/hello", {"name" : name, "name2" : name2} );
}
```

### The JXT

This file goes in `hello-xslt/documents/hello.jxt`

I should pause here to say that, with the exception of sitemap.xmap, which Cocoon expects, these names are all arbitrary. `hello-xslt` could be anything you want, and the other names could be anything as long as you change them in the sitemap.

On to the JXT file:

```
<?xml version="1.0"?>
<document>
  <hello>Hello ${name}!</hello>
  <hello>Hello ${name2}!</hello>
</document>
```

### The XSLT

Finally, the xslt stylesheet should be saved as `hello-xlst/stylesheets/hello.xsl`. It looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="//document">
<html>
<body>
<xsl:for-each select="hello">
<p><xsl:value-of select="."/></p>
</xsl:for-each>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

This simple stylesheet does nothing more than transform the <hello></hello> elements the JXT spits out into paragraphs.

## Running the Example

Assuming a standard installation, browse to a URL like this one:

`http://localhost:8080/cocoon/hello-xslt/`

Modify as needed to suit your Cocoon installation.

## Learning from This Example

If you compare this example with FlowHelloWorld, which I recommend, you can see that each file has changed in some way:

- The sitemap still calls the flowscript, but it handles the result differently. Instead of handling JXT requests generically (matching *.jxt) and serializing them as XHTML, it provides a resource for the flowscript to call based on the individual request. The generic method maps one JXT file to one output page; the internal-resource method would allow one JXT file to serve different requests, probably using different stylesheets (e.g. an xml-to-html sheet and an xml-to-pdf sheet.)
- The JXT file outputs XML instead of XHTML. This separates content from presentation. The JXT is taking program data and forming it as XML data; this can then be used in different ways.
- The flowscript is almost the same, but calls a *specific* resource in the pipeline rather than just calling a JXT file and letting the generic match take it from there.

This is the beginnings of a framework for a flexible application with separation of content and presentation. In the real world, your flowscript would call business logic rather than just assigning variables

When using flow to serve individual pages with no user input, rather than forms, the effort required may seem a little redundant. That is, in `sitemap.xmap` you have one match that calls the function and another match that renders its output. I believe this is the nature of the beast, and that for sites with a mixture of interactive and non-interactive dynamic content, the consistency of using flow/jxt for everything will probably outweigh the inconvenience of defining both entry and exit points.

---

## page metadata

- AUTHOR: JasonStitt
- REVIEWED-BY:
- REVIEWER-CONTACT: