

# Publish dynamic XML content with Cocoon 2, Part 1: Introduction to Cocoon 2

An open-source XML/XSLT publishing framework

Skill Level: Introductory

[Leigh Dodds \(leigh@xmlhack.com\)](mailto:leigh@xmlhack.com)

Developer and editor  
Ingenta, Ltd.

12 Mar 2002

Whereas EDI for years has provided a usable but expensive way for companies to exchange information in an automated manner, ebXML now provides a means for companies to integrate their processes much more easily. Based on XML, it provides a methodology for business to determine what information they should exchange and how, as well as a set of specifications to allow automation of the process. This tutorial gives an overview of ebXML, explaining how all of the pieces fit together.

## Section 1. Tutorial introduction

### What is Cocoon?

Cocoon is a Java server framework that allows the dynamic publishing of XML content using XSLT (XML Stylesheet Language-Transformation) transformations. By relying on XML to describe content, and XSLT as a means of transforming that content into multiple formats, Cocoon provides a platform for building applications with strong separation between content, logic, and presentation.

Cocoon uses the concept of a pipeline to describe the process of publishing content to the Web. A wide variety of reusable components are included, which can be configured to produce complex behavior with a minimum of Java development. For

example, using XML and XSLT alone, Cocoon can be used to:

- Serve static files as well as dynamically generated responses
- Map user requests transparently onto physical resources with an arbitrary amount of processing
- Perform both simple and multistage XSLT transformations
- Pass parameters dynamically to XSLT transformations
- Generate a wide variety of output formats including XML, HTML, PNG, JPEG, SVG, and PDF

This all adds up to a great deal of power that can be put to work using existing skills in XML and XSLT. Cocoon lets you produce dynamic Web sites with a minimum of fuss.

## Cocoon 1 and Cocoon 2

Cocoon is an open-source project being developed as part of the Apache XML effort. Cocoon 2 is a complete rewrite of the original Cocoon application and is the recommended version. New users should start directly with Cocoon 2, while existing Cocoon 1 users are encouraged to upgrade.

The aim of the Cocoon 2 project was to take the lessons learned during Cocoon 1 development and use them to engineer a more efficient and scalable platform. In particular, Cocoon 1 relied on the Document Object Model (DOM) API to pass XML data between components. The DOM is an inefficient means of passing data because a typical DOM tree can consume several times more memory than the original XML document. This severely limited Cocoon's scalability. Cocoon 2 is built around the SAX API, which is a more lightweight means of manipulating XML data.

Another key difference between the two versions of Cocoon centers on application management. In Cocoon 1, individual XML documents declared how they should be processed by including Cocoon processing instructions. This tied documents to specific processing, greatly restricting flexibility to reuse content in different ways. Cocoon 2 factors out management of processing into a configuration file known as a sitemap. This separates out the processing logic from the content itself, which in turn separates concerns among content, logic, and presentation.

Cocoon 2, because it is a more scalable and flexible platform than the original Cocoon application, is the focus of this tutorial.

## Who should take this tutorial?

To get the most from this tutorial, and Cocoon 2, readers should already be familiar with XML and XSLT. While Cocoon is a Java application, you don't need any in-depth Java experience to use it.

This tutorial covers the following:

- How to install and configure Cocoon
- An introduction to the principles of the Cocoon 2 architecture and its key components
- An introduction to the sitemap -- the means of managing Cocoon Web applications
- Example pipeline configurations that demonstrate how to construct a dynamic Web site using Cocoon and XSLT

## Tools

Cocoon is a Java Web application and must be run within a Java-Servlet-2.2-compliant servlet engine. The next section, [Installing and configuring Cocoon](#), includes details on how to download and install Jakarta Tomcat 4.0.1, the latest reference implementation of the servlet API.

Both Tomcat and Cocoon require installation of a Java 2 development kit. The Java 2 SDK, 1.3.1 can be downloaded from <http://java.sun.com/j2se/>.

To fetch the Cocoon source code so the application can be built locally, CVS also needs to be installed. See the CVS Web site, <http://www.cvshome.org/>, for further information.

---

## Section 2. Installing and configuring Cocoon

### Installing Tomcat

This section begins with instructions on how to install the Jakarta Tomcat servlet engine. The rest of the section provides detailed steps on how to download, install, and configure Cocoon 2, including instructions for both a quick installation using one of the prebuilt distributions as well as details of how to build Cocoon directly from source.

The final release of Jakarta Tomcat 4.0.1 is available from the release directory at <http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/>.

After downloading a binary distribution, installation is simply a matter of unzipping it into a suitable directory. Create an environment variable `CATALINA_HOME` that points to this directory. For example:

```
CATALINA_HOME=/usr/local/jakarta-tomcat-4.0.1.
```

Tomcat can be started and shut down by scripts in the `$CATALINA_HOME/bin` directory. By default, Tomcat is configured to run on port 8080. If this clashes with another existing application, then simply edit `$CATALINA_HOME/conf/server.xml`. Search for "8080" and replace it with an appropriate value.

Once Tomcat has been started with the `$CATALINA_HOME/bin/startup.sh` script, test the installation by pointing a browser to `http://localhost:8080`. A welcome page displays if the installation has been successful.

## Cocoon quick install

Cocoon 2 is also available from <http://cocoon.apache.org/mirror.cgi>. Once you've downloaded the distribution, follow the instructions in the `INSTALL.txt` file that comes with the distribution. Look for the `cocoon.war` file; this is the Cocoon Web application archive, which is suitable for deploying into a servlet engine.

Deploying Cocoon into Tomcat is simple. First, if Tomcat is running, shut down the server using the `$CATALINA_HOME/bin/shutdown.sh` script.

Now copy `cocoon.war` into the `$CATALINA_HOME/webapps` directory, and restart Tomcat.

Notice that Tomcat automatically unpacks the Web archive into `$CATALINA_HOME/webapps/cocoon`. Now, test the installation by pointing a Web browser at `http://localhost:8080/cocoon/`. There will be a pause while the server compiles its configuration files.

The following two panels discuss how to fetch the Cocoon source code and build it locally. It's possible to skip ahead to the last panel in this section, [Configuring Cocoon](#), at this point.

## Fetching the Cocoon source

As when conducting any long-term development with Cocoon, it's worthwhile to maintain a local copy of the source code and build the application locally. The

source code is publicly available through the Apache CVS (concurrent versioning system) server. Assuming CVS is already installed, the following steps allow you to download a copy of the Cocoon 2 source.

First, log in to the Apache CVS server:

```
cvcs -d :pserver:anoncvcs@cvcs.apache.org:/home/cvspublic login
```

When prompted for a password, type in `anoncvcs`. Next, enter the following:

```
cvcs -d :pserver:anoncvcs@cvcs.apache.org:/home/cvspublic -z3  
checkout -r cocoon_20_branch xml-cocoon2
```

This initiates the download of all required source files and libraries for building Cocoon. A subdirectory called `xml-cocoon2` will be created below the current working directory, and the distribution will be downloaded to that location.

## Compiling Cocoon

With the source code obtained, Cocoon must now be compiled to create the `cocoon.war` file for deploying into Tomcat. Cocoon is built using Ant, an open source Java build tool that's better suited to Java applications than make.

First, ensure that the `$JAVA_HOME` environment variable points to the directory where Java is installed. Then change to the `xml-cocoon2` directory created by the CVS download. The build process is started with the following command:

```
./build.sh -Dinclude.webapp.libs=yes webapp
```

This creates a `build` directory, into which the compiled code, libraries, and the `cocoon.war` file are placed. Once the build is complete, simply copy `cocoon.war` into `$CATALINA_HOME/webapps` and restart Tomcat.

Cocoon should now be running on `http://localhost:8080/cocoon/`.

## Configuring Cocoon

For basic operation there is very little else that needs to be configured in Cocoon. Apart from the *sitemap*, which is covered later in this article, there are only two configuration files for Cocoon. Both can be found in `$CATALINA_HOME/webapps/cocoon/WEB-INF`:

- `log.xconf`: Configures Cocoon logging. Cocoon uses Apache Log4J for logging internally (see [Resources](#)). By default, Cocoon writes log files to `$CATALINA_HOME\webapps\cocoon\WEB-INF\logs`.
- `cocoon.xconf`: Configures Cocoon caching, data sources, and a number

of other advanced options.

For development purposes there is one change that is useful to make in `cocoon.xconf`. Locate the following entry:

```
<sitemap file="sitemap.xmap" reload-method="asynchron"
check-reload="yes" logger="sitemap"/>
```

Now, change the value of the `reload-method` attribute to `synchron`. This alters how Cocoon responds to changes in the sitemap, the central place from which a Cocoon application is configured. Changing the behavior to `synchron` means that Cocoon will act on changes immediately, before servicing any requests. The default asynchronous behavior means that it will read changes in the background, so they won't be immediately obvious. This can be frustrating during development when when immediate results are preferable, but isn't optimal for a live application that must continue to serve users as fast as possible.

---

## Section 3. The Cocoon 2 architecture

### Getting started

This section introduces the principles of the Cocoon 2 architecture. A number of key concepts are covered:

First, processing of an XML document can be broken down into several discrete steps. The combination of these steps describes a processing *pipeline*. A pipeline consists of input, some processing, and then output. Cocoon 2 uses SAX events as the glue between each processing step.

Second, each stage in a pipeline can be modeled with a particular kind of component. For example, a generator to produce input, and a serializer to produce output. Cocoon defines a number of different components and provides several implementations for each. These component types and their most useful implementations are reviewed in the [Pipeline components](#) panel later in this section.

Third, responding to user requests involves identifying the correct pipeline to service the request (along with its inputs), and then instructing the pipeline to carry out its processing to produce the response to the user.

Later sections in this article introduce the concept of the *sitemap*, which plugs those components together into pipelines that perform useful work.

## The pipeline model

There is a single concept that forms the key to the Cocoon 2 architecture: the *pipeline*. A pipeline consists of some input data, followed by a number of processing steps that act upon it. Each processing step accepts the output of the previous step as its input, until the end of the pipeline is reached and the final output is produced.

Pipelines are a simple concept. It's easy to decompose any complex processing task into a smaller series of steps that can be organized into a pipeline. This concept will be instantly familiar to UNIX users who are used to combining small, general applications (for example, `find`, `grep`, and `sort`) together to achieve specific tasks.

This illustrates another potential benefit of pipelines. Because each processing step has well-defined behavior coupled with fixed inputs and outputs, it becomes possible to create general, *reusable* pipeline components. This reuse allows for the construction of applications with little programming overhead.

## Pipeline components

Cocoon includes a number of general pipeline components that can be connected together in useful ways. These components can be grouped together into several distinct types, depending on the roles they play within a pipeline:

- Pipeline inputs -- generators and readers
- Processing steps -- transformers and actions
- Pipeline outputs -- serializers
- Conditional processing -- matchers and selectors

A Cocoon pipeline generally consists of at least a generator and a serializer, but may consist of any number of processing steps. Data is passed through a Cocoon pipeline as a SAX event.

The following four panels take a closer look at each of these components, and review some of the useful implementations that Cocoon provides.

## Inputs: generators and readers

*Generators* are responsible for reading a data source (for example, a file) and passing that data into the pipeline as a series of SAX events. The simplest generator is therefore a SAX parser. Generally, though, any data source that can be mapped

to a series of SAX events can become the basis for a generator.

There are a number of generators available in Cocoon. The most useful are:

- **FileGenerator:** Reads XML files from the file system or the Web
- **HTMLGenerator:** Reads HTML files from the file system or the Web
- **DirectoryGenerator:** Reads the file system to provide directory listings

*Readers* are a special case in the Cocoon pipeline model as they are not XML-aware components. Readers simply access an external resource and copy it directly to the response. They are commonly used to serve up static files such as images or CSS stylesheets. Readers can be viewed as self-contained pipelines; they generate the input data and serialize it to the response.

## Processing: transformers and actions

*Transformers* are the main processing steps in a Cocoon pipeline. They accept SAX events as input, perform some useful processing, and then pass the results on down the pipeline as SAX events. One useful way to view a transformer is as a component that modifies a stream of SAX events as it passes through it. In this regard, they are similar to SAX filters.

The most widely used transformer is the XSLT Transformer. It feeds its input into an XSLT processor that performs an XSLT transformation. The results of the transform are then fed back into the pipeline as SAX events.

*Actions* are a means to plug additional dynamic behavior into a pipeline and are often custom-built for particular applications. However, some generic actions are bundled with Cocoon, for example, to carry out database interactions, form validation, sending mail, and so on. The successful completion of an action can also influence whether subsequent processing steps are carried out.

## Outputs: serializers

*Serializers* are the endpoints in Cocoon pipelines. They are responsible for taking a stream of SAX events produced either directly from a generator (in the shortest possible pipeline) or a previous processing step such as a transformer, and rendering them into a suitable format for the response. The specific format is dependent on the exact serializer being used.

The simplest serializer is the XML serializer, which simply turns the SAX events back into an XML document. Other serializers can produce HTML, plain text, PDF documents, and even images. These serializers all expect the SAX event stream to

conform to a particular XML vocabulary:

- HTML Serializer: Turns XHTML into valid HTML
- SVG Serializer: Turns SVG into JPEG or PNG images
- PDF Serializer: Turns XSL-FO into a PDF document

This ability to take XML content, process it, and serve it in multiple formats is the real power of the Cocoon framework.

## Conditionals: matchers and selectors

Any nontrivial pipeline is likely to involve some conditional sections. For example, the exact processing steps may depend on factors such as the request parameters, the user's browser, and so forth.

*Matchers* are the simplest of the two conditional components and are equivalent to simple `if` statements. If some condition is true, then a particular pipeline, or a section of a pipeline is evaluated.

The second type of conditional component is the *selector*, which is similar to an `if-then-else` statement. Selectors are used when one of several options are available, and are typically used to create conditional sections *within* a pipeline, whereas matchers are used to test whether a particular pipeline should be entered.

There are several implementations of each of these components. They all follow a common pattern of testing some aspect of the request (such as the host name, user-agent, parameter, or URL) or the user's session. Matching can be achieved using wildcard or regular expressions, while selectors typically enumerate all possible values.

## Putting pipelines to work

Now that we've seen the components that are commonly used to build Cocoon pipelines, it's important to put pipelines into context to see how they are put to work. A description of a logical cycle for receiving requests and serving responses can be summarized as follows:

1. Accept a request from a user.
2. Determine the correct pipeline that should be used to interpret this request and produce a response (using a matcher).

3. Construct the pipeline from available, preconfigured components.
4. Instruct the pipeline to service the request.
5. Return the response generated by the pipeline to the user, possibly caching the results for later use.

This is the basic request-response cycle that Cocoon uses to publish XML data to the Web. To manage this cycle, Cocoon provides an XML configuration file called a *sitemap*, which is explored in more detail in the following two sections.

This section has reviewed only a handful of the most common components used to build Cocoon pipelines. These reusable components are a direct benefit of the pipeline model, and their use allows for the creation of sophisticated processing with little or no programming beyond the creation of XSLT style sheets.

---

## Section 4. Sitemap basics

### Sitemap responsibilities

The sitemap is the central place in which a Cocoon Web site is managed, and it fulfills two functions:

- It's the place in which components are declared before being used in pipelines.
- It's the place in which pipelines are defined using the declared components.

This section deals with the first of these responsibilities, and introduces the basic structure of a sitemap. Pipeline configuration, complete with examples, will be covered in the next section, [The sitemap: defining pipelines](#) .

### Structure of a sitemap

The sitemap is an XML configuration file and therefore has a well-defined structure. The default Cocoon sitemap, `sitemap.xmap`, can be found in the Cocoon Web application directory, `$CATALINA_HOME/webapps/cocoon/sitemap.xmap`.

A sitemap is structured according to the basic outline given in the following XML fragment. Note that there is a specific sitemap namespace, `http://apache.org/cocoon/sitemap/1.0`, used to identify sitemap elements. Also note that the sitemap is divided into two high-level sections, `map:components` and `map:pipelines`, which reflect the sitemap's two responsibilities.

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

  <map:components>
    <!-- component declarations -->
    <map:generators/>
    <map:readers/>
    <map:transformers/>
    <map:actions/>
    <map:serializers/>
    <map:actions/>
    <map:matchers/>
    <map:selectors/>
  </map:components>

  <map:pipelines>
    <!-- pipeline definitions -->
  </map:pipelines>

</map:sitemap>
```

The declarations for each type of component are grouped together within specific elements. For example, all generators' declarations can be found within the `map:generators` element.

## Declaring sitemap components

Components are declared in the sitemap with the general idiom outlined in the following example. There are a few items worth noting:

- *Component-type* is the name of a specific type of component; for example, the `generators` element contains generator declarations.
- Each component must have a unique *name* attribute. Names are used to refer to the components elsewhere in the sitemap.
- Each component must identify its *implementation*. It is possible to have more than one instance of a component sharing the same implementation, but declared with different names.
- A *default* component can be identified. This is used when a component instance isn't specifically named.
- Components can be passed parameters by the sitemap. It's therefore possible to have several instances of the same component but with

different parameters. The parameter elements are specific to each component.

```
<map:component-types
  default="component-name">

  <map:component-type
    name="component-name"
    src="implementation">

    <!-- component-type
      specific parameters -->

  </map:component-type>
</map:component-types>
```

For specific examples of component declarations, read through the first section of the default sitemap, `$CATALINA_HOME/webapps/cocoon/sitemap.xmap`.

## Extensibility

The Cocoon framework loads components based on the declarations in the sitemap using Java's dynamic, class-loading capability. To be dynamically loaded and plugged into pipelines, each component must implement a specific Java interface depending on its type. For example, all generators must implement the `org.apache.cocoon.generation.Generator` interface.

Relying on interfaces to describe each component type means that Cocoon's capabilities can easily be extended simply by writing new implementations of these interfaces, and adding the appropriate declarations to the sitemap.

Consider this simple scenario: There is a large amount of legacy data in the CSV (comma separated values) format that must be available in both the original raw format and as HTML. One approach would be to write a simple application that reads the CSV data and outputs it as XML. This imposes some extra management overhead due to the need to batch-process new data as it arrives, and store both the original and XML formats.

Better integration with Cocoon can be achieved by writing a custom generator such as `com.mycompany.CSVGenerator`, which is capable of parsing a CSV file to directly generate SAX events. This class can then be plugged into Cocoon by declaring it within the sitemap, as follows:

```
<map:generator name="csv" src="com.mycompany.CSVGenerator"/>
```

This component can now be used as a means to feed CSV data into Cocoon pipelines, where it can be transformed, manipulated, and serialized into multiple

formats. The original CSV files can still be delivered using a `Reader` component. Because there is only one source data file, and all dynamic transformation is carried out by Cocoon, there is less management overhead. Obviously, this example could be applied to many different kinds of document formats, and extended to include retrieving documents from content management systems, and so on.

---

## Section 5. The sitemap: defining pipelines

### Sitemap configurations

This section details some examples of sitemap configurations that demonstrate how the components introduced in the previous sections can be combined to perform useful work. The focus will be on the Cocoon sitemap configuration, rather than on the details of the individual transformations and XML formats. Creating the latter can be a separate exercise for readers.

The examples assume that the following directory structure has been created under `$CATALINA_HOME/webapps/cocoon` (referred to as `$COCOON_HOME` from now on):

```
/static      Static HTML document
/content     XML content
/styles      CSS style sheets
/transforms  XSLT style sheets
```

**Tip:** When experimenting with Cocoon, it's useful to do so from a clean environment. Start by creating a new directory under `$CATALINA_HOME/webapps/`, e.g. `cocoon-dev`. Then copy over both `$COCOON_HOME/cocoon.xconf` and the `$COCOON_HOME/WEB-INF` directory, which contains all the required Cocoon classes. This will create a separate Cocoon Web application that can be found at `http://localhost:8080/cocoon-dev/`.

Finally, create a new `sitemap.xmap` in this directory so that experiments can take place without disturbing the original examples and documentation.

### Component declarations

The following configuration examples assume that their declarations have been made in the sitemap:

```
<map:generators default="file">
  <map:generator name="file"
    src="org.apache.cocoon.generation.FileGenerator"/>
</map:generators>

<map:transformers default="xslt">
  <map:transformer name="xslt"
    src="org.apache.cocoon.transformation.TraxTransformer"/>
</map:transformers>

<map:readers default="resource">
  <map:reader name="resource"
    src="org.apache.cocoon.reading.ResourceReader"/>
</map:readers>

<map:serializers default="html">
  <map:serializer name="xml" mime-type="text/xml"
    src="org.apache.cocoon.serialization.XMLSerializer"/>
  <map:serializer name="html" mime-type="text/html"
    src="org.apache.cocoon.serialization.HTMLSerializer"/>
  <map:serializer name="svg2png"
    src="org.apache.cocoon.serialization.SVGSerializer"
    mime-type="image/png"/>
  <map:serializer name="fo2pdf"
    src="org.apache.cocoon.serialization.FOPSerializer"
    mime-type="application/pdf"/>
</map:serializers>

<map:matchers default="wildcard">
  <map:matcher name="wildcard"
    src="org.apache.cocoon.matching.WildcardURIMatcher"/>
</map:matchers>
```

## Serving a static document

Using Cocoon to serve up a static document is a useful initial example. The following pipeline description demonstrates how this is achieved using a reader. Obviously, it's preferable to let an existing Web server handle static files, but this provides a simple example to illustrate working with pipelines.

```
<map:pipelines>
  <map:pipeline>
    <map:match pattern="index.html">
      <map:read src="static/index.html" mime-type="text/html"/>
    </map:match>
  </map:pipeline>
</map:pipelines>
```

First, note that the pipeline is defined by the `map:pipeline` child of the `map:pipelines` element, which should contain all pipeline definitions.

The matcher component is used to associate this pipeline with a request using a *match pattern*. In this instance, a request for the document "index.html" (for example, `http://localhost:8080/cocoon/index.html`) will trigger this

pipeline.

The processing that must happen when the pipeline is triggered is then defined. Here, a reader is instructed to send the file `$_COCOON_HOME/static/index.html` to the user with the mime-type `text/html`. The location of the file is entirely independent of the URL path used to request it.

Explicitly associating an individual file with a pipeline in this way would obviously involve a great deal of work. Luckily, the matcher component makes it possible to avoid this.

## Using wildcards

The initial example can be expanded upon to demonstrate the use of wildcards to match fragments of the requested URL. The pipeline has also been extended slightly to allow the delivery of CSS stylesheets as well as HTML documents. Doing this involves simply adding an extra matcher to the existing pipeline. This has the same effect as declaring a new pipeline, because Cocoon will check both patterns.

```
<map:pipeline>
  <map:match pattern="*.css">
    <map:read src="styles/{1}.css" mime-type="text/css"/>
  </map:match>
  <map:match pattern="**.*.html">
    <map:read src="static/{1}.html" mime-type="text/html"/>
  </map:match>
</map:pipeline>
```

The wildcard matcher allows two kinds of wildcards, both of which are illustrated here. First, a single asterisk matches any numbers of characters *except* a forward slash. A double asterisk matches any number of characters *including* a forward slash. The text matched with these patterns is available to the other sitemap components and can be referenced as `{1}`, `{2}`, `{3}`, and so on, depending on how many wildcards are used.

In the above example, a request for `http://localhost:8080/mysite.css` will match the CSS pattern, and the value "mysite" will be assigned to `{1}`. The reader will then send back `$_COCOON_HOME/styles/mysite.css` to the user with the correct mime-type. A request for `http://localhost:8080/styles/mysite.css` would *not* match the pattern because it specifies only a single asterisk.

The revised HTML match pattern uses a double asterisk as the wildcard. So, a request for `http://localhost:8080/help/help.html` will be successfully matched, and `{1}` will be assigned the value `help/help`. The reader will then send `$_COCOON_HOME/static/help/help.html` back to the user.

It's worth noting that Cocoon processes match patterns sequentially in the order they are defined in the sitemap. Cocoon processes the request according to the first pattern that successfully matches. Therefore, ordering within the sitemap is significant, and care should be taken to place the most specific matches first within a sitemap. For example, a match pattern for `index.html` should be declared before `*.html`, otherwise it will never be matched.

## Performing a transformation

At least three components are needed to carry out a transformation: a generator to read the XML document, a transformer to carry out the transformation, and a serializer to deliver the results. Here's how they are pieced together to effect a transformation.

First, declare the pipeline and the match pattern used to trigger it:

```
<map:pipe>
  <map:match pattern="content/*.html">
```

Next, add a generator to read the XML documents from the `content` directory:

```
<map:generate src="content/{1}.xml"/>
```

Next, add a transformer to transform the XML document using a specified style sheet:

```
<map:transform src="transforms/content2html.xsl"/>
```

Finally, use a serializer to turn the results of the transformation into an HTML document to return to the user:

```
<map:serialize type="html"/>
</map:match>
</map:pipe>
```

Requesting the URL `http://localhost:8080/content/document.html` triggers this pipeline and causes Cocoon to first parse `document.xml`, and then transform it using `$(COCOON_HOME)/transforms/content2html.xsl`, before delivering the results back to the browser.

More complex transformations can be achieved by adding more transformers to the pipeline. Again, wildcards have been used to avoid having to define the actual inputs to the pipeline, by using the results of the matching process.

## Generating other formats

Now that we've seen how to dynamically generate HTML from XML content, it's worth considering how to deliver other document formats. Cocoon supports this through the use of custom serializers.

When delivering XML rather than HTML as the result of a transformation, the XML serializer must be used. If a transformation to generate an RSS file for content syndication (see [Resources](#)) is available, then a pipeline can be written that includes the following fragment:

```
<map:transform src="transforms/content2rss.xsl"/>
<map:serialize type="xml"/>
```

Notice that a specific serializer has been selected with the `type` attribute on the serializer component. The value of the attribute matches the name of one of the serializers from the [Component declarations](#) panel earlier in this section.

SVG (Scalable Vector Graphics) is an XML format for describing line diagrams. Normally, a browser plugin is required to view SVG documents. However, Cocoon includes a serializer that is capable of creating a JPEG or a PNG image directly from an SVG document. This serializer could be invoked as follows:

```
<map:transform src="transforms/content2svg.xsl"/>
<map:serialize type="svg2png"/>
```

Cocoon also supports the creation of PDF files directly from XSL-FO documents. Again, simply ensure that the style sheet produces the correct document format for input into the specialized serializer:

```
<map:transform src="transforms/content2fo.xsl"/>
<map:serialize type="fo2pdf"/>
```

## Passing parameters

It's often useful to pass parameters to an XSLT transformation. Cocoon supports

parameter passing from within the sitemap. The first way to achieve this is with the `map:parameter` element. Here's an example:

```
<map:transform src="transforms/content2html.xsl">
  <map:parameter name="myFixedParam" value="fixed-value"/>
  <map:parameter name="myDynamicParam" value="{1}"/>
</map:transform
```

Both the name and value of the parameter are specified as attributes of the `map:parameter` element. It's possible to pass both fixed and dynamic parameters into style sheets with this method. The second parameter element will have its value set by the first wildcard in a match pattern. As long as the style sheet includes an `xsl:param` element, the parameter will be correctly passed to the transformation.

The alternative way to pass parameters to a style sheet allows the passing of all of the URL request parameters. For example, if a request for `http://localhost:8080/content.html?param1=value1&param2=value2` causes this pipeline to be triggered, then two parameters (`param1` and `param2`) will be passed to the style sheet.

```
<map:transform src="transforms/content2html.xsl">
  <map:parameter name="use-request-parameters" value="true"/>
</map:transform
```

This method is useful when there is a varying number of parameters that may be passed in a request. However, this does incur a small performance cost, because Cocoon is less able to cache the results of transformations that use this method than those using fixed parameters. If any of the URL parameters alters, even if not directly used by the style sheet, the cached results will not be used.

---

## Section 6. Summary

### Where next?

This tutorial covered the fundamental principles of developing Cocoon 2 applications.

After explaining how to install Cocoon, I introduced the pipeline model that defines the basic architecture of the Cocoon framework. I then examined the concept of a

pipeline made up of individual processing steps. Processing at each stage in a pipeline is carried out by different kinds of components, which are classified by the roles they play in passing data through a pipeline. Generators are responsible for feeding data into a pipeline where it can be manipulated using transformers, and results are delivered in multiple formats using serializers.

Next, I covered the basics of the Cocoon sitemap. The sitemap is an XML configuration file that sits at the heart of the Cocoon framework and is responsible for declaring individual components, as well as defining how those components are used to construct pipelines. Declaring components within the sitemap provides Cocoon with a great deal of extensibility, allowing the plug-and-play addition of new implementations. I provided a number of examples that demonstrate how to:

- Use matchers and wildcards to flexibly associate content with pipeline processing
- Serve static files using readers
- Transform XML content with XSLT style sheets using transformer components
- Use different kinds of serializers, and specific XSLT transformations to deliver content in many different formats

This tutorial has only touched on the capabilities that Cocoon 2 offers. Later tutorials in this series will discuss in more detail the various Cocoon components, particularly selectors and actions. XML Server Pages, a Cocoon technology for adding custom processing to Cocoon pipelines using Java, will also be introduced.

# Resources

## Learn

- Take the other tutorials in this developerWorks [series on Cocoon](#).
- Follow the continuing development of [Apache Cocoon](#) at the official project home page.
- The [Cocoon users mailing list](#) is a great source of advice from a thriving community of users.
- For more information about Apache Log4J, which Cocoon uses for logging internally, visit the [Logging Services documentation](#) (Log4j Project).
- Read the [user documentation](#) for more information on Cocoon components.
- Read [XSL Formatting Objects \(XSL-FO\) basics](#) and [XSL-FO advanced techniques](#), two tutorials by Doug Tidwell (*developerWorks*, February 2003) for more information on how to use XSLT to convert XML documents into formatting objects and then the Apache XML Project's FOP (Formatting Object to PDF) tool to convert those objects into PDF files.
- Robert DuCharme's [Transforming XML column](#) contains regular practical advice on using XSLT.
- Read an [Introduction to XSL Formatting Objects](#) by Dave Pawson.
- Find more [resources on XSL and XSL Formatting Objects](#) at the W3C.
- Learn about [FOP](#), the formatting engine used to generate PDFs in Cocoon.
- Follow the progress of [SVG](#) at the W3C.
- While you're at it, you can learn the fundamentals of SVG in the tutorial ["Introduction to Scalable Vector Graphics"](#) (*developerWorks*, March 2004).
- Read ["Understanding SAX"](#) by Nick Chase (*developerWorks*, July 2003) for a great introduction to the SAX API.
- Read [the RSS 1.0 specification](#).
- Find [additional tutorials](#) about XSL, SVG, and more in the [XML zone](#) on developerWorks.

## Get products and technologies

- Find and register RSS feeds at [Syndic8.com](#), the RSS community site.
- [IBM WebSphere Studio Application Developer](#) is an easy-to-use, integrated development environment for building, testing, and deploying J2EE™ applications, including generating XML documents from DTDs and schemas.

## About the author

### Leigh Dodds



Leigh Dodds is team leader of the Research and Technology Group at Ingenta, Ltd. He has five years of experience in developing on the Java platform, and he has spent the last three years working with XML and related technologies. Leigh is also a contributing editor to [xmlhack.com](http://xmlhack.com), and has been writing the regular "XML-Deviant" column on [XML.com](http://XML.com) since February 2000. He holds a bachelor's degree in biological science, and a master's in computing. When he's not wrestling with pointy brackets, Leigh can be found making silly noises with his son, Ethan.