Write a Custom Generator (2.1 legacy document)

Table of contents

Table of contents

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. Introduction

This Tutorial describes the steps necessary to write a basic Cocoon generator. Starting with a quick "Hello World" example and progressing to slightly more involved examples should give a good start to those whose applications call for extending Cocoon with a custom generator.

The intention is to provide:

- the basics of creating SAX events in a C2 generator
- a little understanding of the Avalon container contract as it relates to C2 generators
- a little understanding of the factors that would influence the decision about which xxxGenerator to extend

1.1. Purpose

The flexibility to extend the basic "Out of the box" functionality of Cocoon will be an important feature for Cocoon's viability as a broadly used application framework. Though the documentation on "Extending Cocoon" (at least at this writing) seems to have a hard time imagining applications for custom generators outside of the bizarre, I imagine several scenarios which could call for it:

- A datasource as yet undeveloped in Cocoon (e.g. event logs)
- Database driven applications for which XSP is either too awkward or holds too many performance questions. The need for high scalability will drive some (such as myself) to seek optimization in custom generators that just do not seem reasonable to expect out of the auto-generated code that XSPs produce. The current Performance Tips documentation seems to lead in this direction.
- Customized control over the caching behaviour if not provided for by other means.

1.2. Important

There are other options that should be considered before settling on a new generator. One notable consideration is the option of writing a Source that would fit your needs. See this discussion from the mailing list for an introduction to the idea. Of course, XSP should be considered - I have not seen any performance comparisons that quantify the benefit that can be had from a custom generator. Finally, be sure you understand the purpose and capabilities of all current standard Generators, as well as those in the scratchpad (for instance, there is a TextParserGenerator in the scratchpad at the moment which may be configurable enough to process the event log need mentioned above). Cocoon is a rapidly developing technology that may have anticipated your need. Because the documentation lags behind development, you may find more by examining the source directory and searching the mail archives for applicable projects.

1.3. Intended Audience

This Tutorial is aimed at users who have developed an understanding of the basics of Cocoon and have a need to begin extending it for their own purposes, or desire a deeper understanding of what goes on under the hood.

1.4. Prerequisites

Generator developers should have:

- Read Cocoon Concepts , as well as Extending Cocoon , and the broad overview of Avalon , the framework upon which Cocoon is built.
- An installed version of Cocoon if you want to follow the examples yourself (obviously).
- A good understanding of Java.
- Java SDK (1.2 or later) "installed".

2. Diving In

Let us start with a simple "Hello World" example:

2.1. Simple Example

Our goal will be to build the following document (or, more to the point, the SAX events that would correspond to this document).

<example>Hello World!</example>

An example of code that will send the correct SAX events down the pipeline:

import org.apache.cocoon.generation.AbstractGenerator; import org.xml.sax.helpers.AttributesImpl; import org.xml.sax.SAXException; public class HelloWorldGenerator extends AbstractGenerator { AttributesImpl emptyAttr = new AttributesImpl(); /** * Override the generate() method from AbstractGenerator. * It simply generates SAX events using SAX methods. * I haven't done the comparison myself, but this * has to be faster than parsing them from a string. */ public void generate() throws SAXException { // the org.xml.sax.ContentHandler is inherited // through org.apache.cocoon.xml.AbstractXMLProducer contentHandler.startDocument(); contentHandler.startElement("", "example", "example", emptyAttr); contentHandler.characters("Hello World!".toCharArray(),0, "Hello World!".length()); contentHandler.endElement("","example", "example"); contentHandler.endDocument(); } }

So, the basic points are that we extend AbstractGenerator, override its generate() method, call the relevant SAX methods on the contentHandler (inherited from AbstractGenerator) to start, fill and end the document. For information on the SAX api, see www.saxproject.org

A performance tip might be to keep an empty instance of AttributesImpl around to reuse for each element with no attributes. Also, the characters(char[] chars, int start, int end) begs to be overloaded with a version like characters(String justPutTheWholeThingIn) that handles the conversion to a character array and assumes you want from beginning to end, as is done in org.apache.cocoon.generation.AbstractServerPage. If you are not using namespaces, it is easy to imagine overloaded convenience implementations of the other SAX methods as well. You will probably want to set up a convenient BaseGenerator with helpers like this and extend it for your real Generators.

2.1.1. What to Extend?

How did we choose to extend AbstractGenerator? Generators are defined by the org.apache.cocoon.generation.Generator interface. The only direct implementation of this of interest to us is AbstractGenerator, which gives a basic level of functionality. Another option would have been ServiceableGenerator, which would give us the added functionality of implenting the Avalon interface Serviceable , which would signal the container that handles all the components including our generator to give us a handle back to the ServiceManager during the startup of the container. If we needed to lookup a pooled database connection, or some other standard or custom Cocoon component, this is

what we would do. Most of the out of the box Generators extend ServiceableGenerator. Other abstract Generators you may choose to extend include the poorly named (IMHO) ServletGenerator , and AbstractServerPage . While these both introduce functionality specific to their eventual purpose - the JSP and XSP generators, they do make a convenient starting place for many other Generators.

2.1.2. Running The Sample

In order to run this sample, you will need to compile the code, deploy it into the cocoon webapp, and modify the sitemap to declare our generator and allow access to it via a pipeline.

2.1.2.1. Compile

Save this source as HelloWorldGenerator.java and compile it using

javac -classpath %PATH_TO_JARS%\cocoon.jar;%PATH_TO_JARS%\xml-apis.jar HelloWorldGenerator.java

Unfortunately for me, the exact name of your cocoon and xml-apis jars may vary with exactly which distribution, or CVS version you are using, since the community has taken to appending dates or versions at the end of the jar name to avoid confusion. Be sure to find the correct name on your system and substitute it in the classpath. Also, you have several options on where to find jars. If you have a source version that you built yourself, you may want to point to lib\core\ for them. If you have only the binary version, you can find them in WEB-INF\lib\

2.1.2.2. Deploy

Simply copy the class file into the %TOMCAT_HOME%\webapps\cocoon\WEB-INF\classes directory

If memory serves me, there have been occasional classloading problems in the past that may affect classloading. If your compiled classes are not recognized in the classes directory, try jar-ing them up and place them in WEB-INF\lib\ instead. That is probably where your real generators would go anyway - with a whole package of all your custom classes in one jar.

2.1.2.3. Sitemap Modifications

You need to do two things: in the map:generators section, add an element for your class:

<map:generator name="helloWorld" src="HelloWorldGenerator"/>

Then add a pipeline to sitemap.xmap which uses it:

... <map:match pattern="heyThere.xml"> <map:generate type="helloWorld"/> <map:serialize type="xml"/> </map:match> ...

And finally, our creation should be available at http://localhost:8080/cocoon/heyThere.xml

Depending on your exact setup, you may need to restart Tomcat (or whatever your servlet container is) to get there.

Notice that the <?xml version="1.0" encoding="UTF-8"?> declaration was added for us by the xml serializer at the beginning. If you need to modify this, the generator is not the appropriate place. The default encoding of UTF-8 could be overridden with iso-8859-1 for example by specifying an <encoding>iso-8859-1</encoding> child parameter inside the declaration for the xml serializer in your sitemap.

2.2. A Less Trivial Example

Moving on to a less trivial example, we will take some information out of the Request, and construct a slightly more involved document. This time, our goal will be the following document:

<doc> <uri>...</uri> <params> <param value="...">...</param> ... </params> <date>..</date> </doc>

The values of course will be filled in from the request, and will depend on choices we make later.

```
import org.apache.cocoon.generation.AbstractGenerator; import org.xml.sax.helpers.AttributesImpl; import org.xml.sax.SAXException; // for the setup() method import org.apache.cocoon.environment.SourceResolver; import java.util.Map; import org.apache.avalon.framework.parameters.Parameters; import org.apache.cocoon.ProcessingException; import java.io.IOException; // used to deal with the request parameters. import org.apache.cocoon.environment.ObjectModelHelper; import org.apache.cocoon.environment.Request; import java.util.Enumeration; import java.util.Date; public class RequestExampleGenerator extends AbstractGenerator { // Will be initialized in the setup() method and used in generate() Request request = null; Enumeration paramNames = null; String uri = null; // We will use attributes this time. AttributesImpl myAttr = new AttributesImpl(); AttributesImpl emptyAttr = new AttributesImpl(); public void setup(SourceResolver resolver, Map objectModel, String src, Parameters par) throws ProcessingException, SAXException, IOException { super.setup(resolver, objectModel, src, par); request = ObjectModelHelper.getRequest(objectModel); paramNames = request.getParameterNames(); uri = request.getRequestURI(); } /** * Implement the generate() method from AbstractGenerator. */ public void generate() throws SAXException { contentHandler.startDocument(); contentHandler.startElement("", "doc", "doc", emptyAttr); // <uri> and all following elements will be nested inside the doc element contentHandler.startElement("", "uri", "uri", emptyAttr); contentHandler.characters(uri.toCharArray(),0,uri.length()); contentHandler.endElement("", "uri", "uri"); contentHandler.startElement("", "params", "params", emptyAttr); while (paramNames.hasMoreElements()) { // Get the name of this request parameter. String param = (String)paramNames.nextElement(); String paramValue = request.getParameter(param); // Since we've chosen to reuse one AttributesImpl instance, // we need to call its clear() method before each use. We // use the request.getParameter() method to look up the value // associated with the current request parameter. myAttr.clear(); myAttr.addAttribute("","value","value","",paramValue); // Each <param> will be nested inside the containing <params> element. contentHandler.startElement("", "param", "param", myAttr); contentHandler.characters(param.toCharArray(),0,param.length()); contentHandler.endElement("","param", "param"); } contentHandler.endElement("","params", "params"); contentHandler.startElement("", "date", "date", emptyAttr); String dateString = (new Date()).toString(); contentHandler.characters(dateString.toCharArray(),0,dateString.length()); contentHandler.endElement("", "date", "date"); contentHandler.endElement("","doc", "doc"); contentHandler.endDocument(); } public void recycle() { super.recycle(); this.request = null; this.paramNames = null; this.parNames = null; this.uri = null; } }
```

2.2.1. Compile and Test

Save this code as RequestExampleGenerator.java and compile as before. You will need to add both avalon-framework.jar and avalon-excalibur.jar to your classpath this time. Besides finding the exact name of the jar as described above, you may now also have to ensure that you have the version of excalibur targeted to your jvm version - there is currently a version for JDK 1.4 and one for 1.2/1.3

For your sitemap, you will need to add a definition for this generator like <map:generator

name="requestExample" src="RequestExampleGenerator"/> and you will need a sitemap pipeline like:

<map:match pattern="howYouDoin.xml"> <map:generate type="requestExample"/> <map:serialize type="xml"/> </map:match>

At this point, you should be able to access the example at http://localhost:8080/cocoon/howYouDoin.xml?anyParam=OK&more=better

2.2.2. New Concepts

2.2.2.1. Lifecycle

First, notice that we now override the setup(...) and recycle() methods defined in AbstractGenerator. The ServiceManager that handles the lifecycle of all services in Cocoon, calls setup(..) before each new call to generate() to give the Generator information about the current request and its environment, and calls recycle() when it is done to enable it to clean up resources as appropriate. Our example uses only the objectModel which abstracts the Request, Response, and Context. We get a reference to the Request wrapper, and obtain an Enumeration of all the GET/POST parameters available.

The src and SourceResolver are provided to enable us to look up and use whatever source is specified in the pipeline setup. Had we specified <map:generate type="helloWorld" src="someSourceString"/> we would have used the SourceResolver to work with "someSourceString", whether it be a file, or url, etc.

We are also given a Parameters reference which we would use to obtain any parameter names and values which are children elements of our map:generate element in the pipeline.

It may be good practice to abstract the source of your parameters so that they do not have to come from the Request object. For instance, the following code would allow us to abstract the origin of two parameters, param1 and param2:In RequestExampleGenerator.java, ... String param1 = null; String param2 = null; ... public void setup(SourceResolver resolver, Map objectModel, String src, Parameters par) throws ProcessingException, SAXException, IOException { ... param1 = par.getParameter("param1"); param2 = par.getParameter("param2"); } and in sitemap.xmap, ... <map:match pattern="abstractedParameters.xml"/> <map:act type="request"> <map:parameter name="parameters" value="true"/> <map:generate type="requestExample"> <parameter name="param1" value="{visibleName1}"/> <parameter name="param2" value="{visibleName2}"/> </map:generate> </map:act> </map:match> ...

As you can see, we have also hidden the internal name from the outside world who will use ?visibleName1=foo&visibleName2=bar

2.2.2.2. Nested Elements

In this example, nested elements are created simply by nesting complete startElement()/endElement pairs within each other. If we had a logic failure in our code and sent non-wellformed xml events down the pipeline, nothing in our process would complain (try it!). Of course, any transformers later in the pipeline would behave in an unpredictable manner.

2.2.2.3. Attributes

Finally, we've introduced the use of attributes. We chose to employ one attributesImpl, clearing it before each element. Multiple attributes for an element would simply be added by repeated calls to

addAttribute.

2.2.3. A Lesson

Before moving on, it is worth noting that after all this work, there is already a generator provided with Cocoon which does much of what we have accomplished here - org.apache.cocoon.generation.RequestGenerator which in the default configuration is probably available at http://localhost:8080/cocoon/request

2.3. Moving On

From here, we will move on to cover handling ugly pseudo-xml (like real world html) with CDATA blocks, employing some of the Avalon lifecycle method callbacks (Serviceable/Disposable), Database access, and Caching.

2.3.1. The Employee SQL Example Reworked

In the samples included with Cocoon, there is an example of a SQL query using XSP and ESQL. We will recreate part of that example below using the same HSQL database, which should be automatically configured and populated with data in the default build. If you find that you do not have that database set up, see the ESQL XSP sample for instructions on setting the datasource up. Do note that this specific task is handled in the ESQL XSP example in just a few lines of code. If your task is really this simple, there may be no need to create your own generator.

```
import org.apache.cocoon.generation.ServiceableGenerator; import
org.apache.avalon.framework.service.ServiceManager; import
org.apache.avalon.framework.service.ServiceException; import
org.apache.avalon.framework.service.ServiceSelector; import
org.apache.avalon.excalibur.datasource.DataSourceComponent; import
org.apache.cocoon.environment.SourceResolver; import
org.apache.avalon.framework.parameters.Parameters; import
org.apache.cocoon.environment.ObjectModelHelper; import
org.apache.cocoon.environment.Request; import org.apache.cocoon.caching.Cacheable;
import org.apache.cocoon.caching.CacheValidity; import
org.apache.cocoon.ProcessingException; import org.xml.sax.ContentHandler; import
org.xml.sax.SAXException; import org.xml.sax.helpers.AttributesImpl; import java.sql.*;
import java.util.Map; import java.util.Date; import
org.apache.avalon.framework.activity.Disposable; public class EmployeeGeneratorExample
extends ServiceableGenerator implements Cacheable, Disposable { public void dispose() {
super.dispose(); manager.release(datasource); datasource = null; } public void recycle() {
myAttr.clear(); super.recycle(); } public void setup(SourceResolver resolver, Map
objectModel, String src, Parameters par) { // Not neeed for this example, but you would get
request // and/or sitemap parameters here. } public void service(ServiceManager manager)
throws ServiceException{ super.service(manager); ServiceSelector selector =
(ServiceSelector) manager.lookup(DataSourceComponent.ROLE + "Selector");
this.datasource = (DataSourceComponent) selector.select("personnel"); } public void
generate() throws SAXException, ProcessingException { try { Connection conn =
this.datasource.getConnection(); Statement stmt = conn.createStatement(); ResultSet res =
stmt.executeQuery(EMPLOYEE_QUERY); //open the SAX event stream
contentHandler.startDocument(); myAttr.addAttribute("","date","date","", (new
Date()).toString()); //open root element contentHandler.startElement("","content",
```

"content",myAttr); String currentDept = ""; boolean isFirstRow = true; boolean moreRowsExist = res.next() ? true : false; while (moreRowsExist) { String thisDept = attrFromDB(res, "name"); if (!thisDept.equals(currentDept)) { newDept(res,thisDept,isFirstRow); currentDept = thisDept; } addEmployee(res,attrFromDB(res,"id"), attrFromDB(res,"empName")); isFirstRow = false; if (!res.next()) { endDept(); moreRowsExist = false; } } //close root element contentHandler.endElement("","content","content"); //close the SAX event stream contentHandler.endDocument(); res.close(); stmt.close(); conn.close(); } catch (SQLException e) { throw new ProcessingException(e); } } public long generateKey() { // Default non-caching behaviour. We will implement this later. return 0; } public CacheValidity generateValidity() { // Default non-caching behaviour. We will implement this later. return null; } private DataSourceComponent datasource; private AttributesImpl myAttr = new AttributesImpl(); private String EMPLOYEE_QUERY = "SELECT department.name, employee.id, employee.name as empName " + "FROM department, employee " + "WHERE department.id = employee.department_id ORDER BY department.name"; private void endDept() throws SAXException { contentHandler.endElement("","dept","dept"); } private void newDept(ResultSet res, String dept, boolean isFirstRow) throws SAXException { if (!isFirstRow) { endDept(); } myAttr.clear(); myAttr.addAttribute("","name","name","",dept); contentHandler.startElement("","dept","dept",myAttr); } private void addEmployee(ResultSet res, String id, String name) throws SAXException { myAttr.clear(); myAttr.addAttribute("","id","id","",id); contentHandler.startElement("","employee","employee",myAttr); contentHandler.characters(name.toCharArray(),0,name.length()); contentHandler.endElement("","employee","employee"); } private String attrFromDB(ResultSet res, String column) throws SQLException { String value = res.getString(column); return (res.wasNull())?"":value; } }

2.3.2. Compile and Test

To compile this, you will now need the following on your classpath: avalon-excalibur.jar, avalon-framework.jar, cocoon.jar, xml-apis.jar (using whatever names they have in your distribution). When you compile this, you may receive some deprecation warnings. Do not worry about them - we will discuss that later.

To test it, copy it over to your WEB-INF\classes\ directory as before and add something like the following to your sitemap.xmap ...

... <map:generator name="employee" src="EmployeeGeneratorExample"/> ... <map:match pattern="employee.xml"> <map:generate type="employee"/> <map:serialize type="xml"/> </map:match> ...

2.3.3. New Concepts

2.3.3.1. Serviceable and Disposable

We've implemented the Avalon lifecycle interfaces Serviceable and Disposable. When Cocoon starts up (which happens when the servlet container starts up) the ServiceManager will call service(ServiceManager m) for our component as it works its way through all the components declared in the sitemap. The handle to ServiceManager is used to look up any other Avalon components that we need. Lookups happen in an abstracted way using a ROLE which enables us to change out implementations of each component without affecting previously written code. Our generator's ROLE by the way was defined in the Generator interface.

Similarly, when this instance of our generator is disposed of by the container, it will call the dispose() method to allow us to clean up any resources we held on to between invocations. Note that components can be pooled by the container. If we thought that our employee generator was going to see a lot of traffic, we might change its definition at the top of sitemap.xmap to include attributes like pool-max="16" so that multiple overlapping requests could be serviced without a log jam.

2.3.3.2. Datasource

We look up our HSQL database here by its name given in cocoon.xconf. If we had multiple datasources (say a backup development database and a live one), we could determine which one to use based on a simple configuration parameter in sitemap.xmap. We could get at configuration parameters using the Avalon interface Configurable.

Notice that we wait until generate() to request our connection from the pool - as we should. The problem is that we lose the benefit of using prepared statements since they would be destroyed when we returned the instance to the pool. At present, the implementation of org.apache.avalon.excalibur.datasource.DataSourceComponent does not support the pooling of statements.

2.3.3.3. Caching

open: Need more content here, or links to other docs.FIXME: This is still coming.

Introduce new code to implement Caching, discuss basic logic, and deprecation/move to Avalon. I could use some help here from Carsten, or someone who can quickly give an overview of the changes and plan.

1. Comments

add your comments