Request Processing (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

1.1. Goal

This document tries to explain Apache Cocoon technically. We do this by describing what happens if somebody types in the URL of a simple Cocoon page.

1.2. Intended public

The reader should have a knowledge of:

- the Java 2 platform
- the javax.servlet extensions
- XML
- HTTP

2. The configuration assumptions

The sequence of events described in this document, depends on some assumptions with regard to the configuration of Cocoon. That's what's described here.

2.1. sitemap.xmap

The task of the sitemap is to define the pipelines that Cocoon will apply to URI's called in one's browser.

This is the minimal sitemap that is necessary. The lines here are included in the standard sitemap.xmap that comes with the distribution of Cocoon.

The sitemap is defined in ${cocoon}/sitemap.xmap.

```
<?xml version="1.0"?> <map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">
<!--===========================Components================================-->
<map:components> <map:generators default="file"> <map:generator name="file"
label="content" src="org.apache.cocoon.generation.FileGenerator"/> </map:generators>
<map:transformers default="xslt"> <map:transformer name="xslt"
src="org.apache.cocoon.transformation.XalanTransformer">
<use-request-parameters>false</use-request-parameters> </map:transformer>
</map:transformers> <map:serializers default="html"> <map:serializer name="html"
mime-type="text/html" src="org.apache.cocoon.serialization.HTMLSerializer"/>
</map:serializers> <map:selectors default="browser"> <map:selector name="browser"
factory="org.apache.cocoon.selection.BrowserSelector"> <browser name="explorer"
useragent="MSIE"/> <browser name="netscape" useragent="Mozilla"/> </map:selector>
</map:selectors> <map:matchers default="uri"> <map:matcher name="uri"
factory="org.apache.cocoon.matching.WildcardURIMatcher"/> </map:matchers>
</map:components>
<!--============================Pipelines================================-->
<map:pipelines> <map:pipeline> <map:match pattern="hello.html"> <map:generate
```

src="docs/samples/hello-page.xml"/> <map:transform src="stylesheets/page/simple-page2html.xsl"/> <map:serialize type="html"/> </map:match> </map:pipeline> </map:pipelines> </map:sitemap>

2.2. cocoon.xconf

cocoon.xconf is the file that defines the Avalon Components.

For our study, we need the standard cocoon.xconf file of Cocoon.

It can be found in ${cocoon}/WEB-INF/cocoon.xconf.

<?xml version="1.0"?> <cocoon version="2.0"> <!-- ========== General Components ========== --> <component role="org.apache.cocoon.components.parser.Parser" class="org.apache.cocoon.components.parser.JaxpParser"/> <component role="org.apache.cocoon.components.store.Store" class="org.apache.cocoon.components.store.MemoryStore"/> <component role="org.apache.cocoon.components.classloader.ClassLoaderManager" class="org.apache.cocoon.components.classloader.ClassLoaderManagerImpl"/> <component role="org.apache.cocoon.components.language.markup.MarkupLanguageSelector" class="org.apache.cocoon.core.container.StandaloneServiceSelector"> <component-instance name="xsp" class="org.apache.cocoon.components.language.markup.xsp.XSPMarkupLanguage"> <parameter name="prefix" value="xsp"/> <parameter name="uri" value="http://apache.org/xsp"/> <target-language name="java"> <parameter name="core-logicsheet" value="resource://org/apache/cocoon/components/language/markup/xsp/java/xsp.xsl"/> <builtin-logicsheet> <parameter name="prefix" value="xsp-request"/> <parameter name="uri" value="http://apache.org/xsp/request/2.0"/> <parameter name="href" value="resource://org/apache/cocoon/components/language/markup/xsp/java/request.xsl"/> </builtin-logicsheet> <builtin-logicsheet> <parameter name="prefix" value="xsp-response"/> <parameter name="uri" value="http://apache.org/xsp/response/2.0"/> <parameter name="href" value="resource://org/apache/cocoon/components/language/markup/xsp/java/response.xsl"/> </builtin-logicsheet> </target-language> </component-instance> <component-instance name="sitemap" class="org.apache.cocoon.components.language.markup.sitemap.SitemapMarkupLanguage"> <parameter name="prefix" value="map"/> <parameter name="uri" value="http://apache.org/cocoon/sitemap/1.0"/> <target-language name="java"> <parameter name="core-logicsheet" value="resource://org/apache/cocoon/components/language/markup/sitemap/java/sitemap.xsl"/> </target-language> </component-instance> </component> <component role="org.apache.cocoon.components.language.generator.ProgramGenerator" class="org.apache.cocoon.components.language.generator.ProgramGeneratorImpl"> <parameter name="auto-reload" value="true"/> </component> <!-- these components is used as a PoolController for the sitemap component pools --> <component role="org.apache.avalon.util.pool.PoolController" class="org.apache.cocoon.util.ComponentPoolController"/> <sitemap file="sitemap.xmap"/> </cocoon>

3. The sequence of things

3.1. Role of Tomcat

The role of Tomcat is to initialize the CocoonServlet, and to receive the HttpRequest and pass it on to the CocoonServlet.

3.1.1. Initialize CocoonServlet

This is done by calling CocoonServlet.init(ServletConfig). This is the standard servlet way to initialize a servlet.

3.1.2. Pass HttpRequest

On reception of a HttpRequest, Tomcat calls CocoonServlet.service(HttpRequest, HttpResponse). This is also standard.

3.2. Initialization

3.2.1. Overview

The steps that happen on initialization, are:

3.2.1.1. Find the classpath

Cocoon needs to know the classpath for compilation of the files it generates itself. This is where the classpath is stored.

3.2.1.2. Find the init file

The init file (normally cocoon.xconf, as defined in ${cocoon}/WEB-INF/web.xml) contains the necessary information for Cocoon to decide which classes to use for which roles (refer to Avalon).

This is a feature that is added for increased configurability. If you were developing a one time solution, the information in this file would normally be hard coded, but the use of this file increases potential reusability.

3.2.1.3. Read the init file

The init file is an xml file (normally cocoon.xconf) which describes the classes to use for which roles.

"Roles" are a concept of Avalon.

The handling of cocoon.xconf goes as follows:

1. Get the parser: This is something necessary for bootstrapping: cocoon.xconf contains the parser to be used by Cocoon, but cocoon.xconf is an xml file that has to be parsed itself. That's why Cocoon gets a default parser out of the System properties (this refers to the environment variable $org.apache.cocoon.components.parser.Parser of the OS). If no parser is defined in the environment, Cocoon will use org.apache.cocoon.components.parser.JaxpParser (a hard-coded default).
2. Get the components: Cocoon uses roles (refer to Avalon) as its working classes. Each role is implemented by one or more real classes (components, again an Avalon concept). This is where

they are retrieved.
3. Get the sitemap: Here the location of the sitemap is retrieved. The actual compilation of the sitemap occurs in the HttpRequest handling.

3.2.2. UML sequence diagram

You can find it here.

3.3. HttpRequest handling

3.3.1. Overview

When the CocoonServlet gets a HttpRequest from the servlet engine, it sets up an Environment (a HttpEnvironment in this case) and passes that to Cocoon. The Environment exists of Request, Response, and some servlet info (such as requested URI and the servlet's path).

This Cocoon object lets the Environment decide which sitemap to use, and passes the sitemap filename along with the Environment to a Manager.

This one puts a Handler to work: it checks whether there already exists a Handler with a compiled version of the sitemap. If not, it creates one. This is what happens then:

1. The Handler creates a File object with the asked URL.
2. The Manager sets the Serviceable and the Configuration of the Handler. (These are Avalon things).
3. If necessary, the Manager asks the Handler to regenerate its sitemap class. (FIXME: As of today, 2000-11-08, I'm not sure if the "if necessary" check is working). Regeneration exists in:
    1. The Handler gets the "program-generator" Component from its Serviceable.
    2. The load() method of this ProgramGeneratorImpl is called.
    3. The ProgramGeneratorImpl gets the "markup-language" (in this case it will get a SitemapMarkupLanguage) and "programming-language" (being JavaLanguage) Components.
    4. The ProgramGeneratorImpl asks the SitemapMarkupLanguage to generate code.
    5. Then it asks the JavaLanguage to load the code. The JavaLanguage does this by creating a Javac object, setting its variables, and asking it to compile. Then it loads the class.
    6. Then its back to the ProgramGeneratorImpl who tells the JavaLanguage to instantiate the just loaded class.
4. At last, the sitemapManager asks the Handler to process the Environment, and the Handler just forwards this request to the generated sitemap class.

3.3.2. UML sequence diagram

You can find it here.

4. Description of classes

4.1. CocoonServlet

org.apache.cocoon.servlet.CocoonServlet

This is the contact point for the servlet engine. It sets up the environment and passes all the work to a Cocoon object.

4.2. Cocoon

org.apache.cocoon.Cocoon

While this sounds to be the most important part of the Cocoon application, it is not. It is merely a Serviceable, meaning that it does some administrative work and gets other classes to work.

4.3. ConfigurationBuilder

org.apache.avalon.ConfigurationBuilder

This one generates a Configuration out of a xml file.

4.4. Parser

org.apache.cocoon.components.parser.Parser

An interface that takes an xml file and throws SAX events to the outside.

4.5. Configuration

org.apache.avalon.Configuration

This is an Avalon interface. It assigns classes to roles. If an object needs a class for a specific role, it can ask a Configuration which class it has to use.

4.6. ProgramGenerator

org.apache.cocoon.components.language.programming.ProgrammingLanguage

Generates programs.

4.7. SitemapMarkupLanguage

org.apache.cocoon.components.language.markup.sitemap.SitemapMarkupLanguage

This one knows the markup of the sitemap, and helps writing the source file of the sitemap class.

4.8. JavaLanguage

org.apache.cocoon.components.language.programming.java.JavaLanguage

This takes care for outputing Java code as source of the sitemap class.

4.9. ResourcePipeline

org.apache.cocoon.sitemap.ResourcePipeline

Holds the various steps that have to be taken when executing a pipeline.

1. Comments

add your comments