Caching (2.1 legacy document)

Table of contents

1 Comments.....7

Table of contents

1 Goal....3
2 Overview....3
3 How to Configure Caching....3
4 The Default Caching Algorithm....3
4.1 Simple Examples....3
4.2 Complex Example....4
4.3 Making Components Cacheable....4
5 Configuration....5
5.1 Configuration of Pipelines....5
5.1.1 Expiration of Content....5
5.1.2 Response Buffering....6
5.2 Configuration of Caches....6
5.3 Configuration of Stores....6
6 Additional Information for Developers....6
6.1 Java APIs....7
6.2 The XMLSerializer/XMLDeserializer....7
6.2.1 org.apache.cocoon.components.sax.XMLByteStreamCompiler....7
6.2.2 org.apache.cocoon.components.sax.XMLByteStreamInterpreter....7
6.2.3 Configuration....7

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. Goal

This document explains the basic caching algorithm of Apache Cocoon.

2. Overview

The caching algorithm of Cocoon has a very flexible and powerful design. The algorithms and components used are not hard coded into the core of Cocoon. They can be configured in the sitemap.

This document describes the components available for caching, how they can be configured and how to implement your own cacheable components.

3. How to Configure Caching

The caching can be turned on and off on a per pipeline setting in the sitemap. This means, for each *map:pipeline* section in a sitemap, it's possible to turn on/off caching and configure the caching algorithm.

The following example shows how to turn on caching for a pipeline:

```
<map:pipeline type="caching"> ... </map:pipeline>
```

If you know that it doesn't make sense to turn on caching for some of your pipelines, put them together in their own section and use:

```
<map:pipeline type="noncaching"> ... </map:pipeline>
```

As you might guess from how the caching is turned on (via a type attribute), you can have different caching (or better pipeline) implementation to choose from. This is similar to choose from a set of generators the generator to use in your pipeline etc. You will find in your main sitemap a section declaring all pipeline implementations. It's in the *map:components* section:

```
<map:pipes default="caching"> <map:pipe name="caching" src="..."/> <map:pipe
name="noncaching" src="..."/> </map:pipes>
```

Depending on your Cocoon installation you might have different implementations in that section. As with all components, you can define a default for all pipelines and override this whereever it makes sense.

4. The Default Caching Algorithm

The default algorithm uses a very easy but effective approach to cache a request: The pipeline process is cached up to the most possible point.

Therefore each component in the pipeline is queried by Cocoon if it supports caching. Several components, like the file generator or the xslt transformer support caching. However, dynamic components like the sql transformer or the cinclude transformer do not. Let's have a look at some examples:

4.1. Simple Examples

If you have the following pipeline:

Generator[type=file|src=a.xml] -> Transformer[type="xslt"|src=a.xsl] -> Serializer

The file generator is cacheable and generates a key which uses the src (or the filename) to build the key. The cache uses the last modification date of the xml file to test if the cached content is valid.

The xslt transformer is cacheable and generates a key which uses the filename to build the unique key. The cache validity object uses the last modification date of the xslt file.

The default serializer (html) supports the caching as well.

All three keys are used to build a unique key for this pipeline. The first time it is invoked its response is cached. The second time this pipeline is called, the cached content is get from the cache. If it is still valid, the cached content is directly send to the client.

4.2. Complex Example

Only part of the following pipeline is cached:

Generator[type=file|src=a.xml] -> Transformer[type="xslt"|src=a.xsl] -> Transformer[type=sql] -> Transformer[type="xslt"|src=b.xsl] -> Serializer

The file generator is cacheable and generates a key which uses the src (or the filename) to build the key. The cache uses the last modification date of the xml file to test if the cached content is valid.

The xslt transformer is cacheable and generates a key which uses the filename to build the unique key. The cache validity object uses the last modification date of the xslt file.

The sql transformer is not cacheable, so the caching algorithm stops at this point although the last transformer is cacheable again.

The cached response is the output of the first xslt transformer, so when the next request comes in and the cached content is valid, the cached content is directly feed into the sql transformer. The generator and the first xslt transformer are not executed.

4.3. Making Components Cacheable

This chapter is only for developers of own sitemap components. It details what you have to do when you want that your own sitemap components supports the caching.

Each sitemap component (generator or transformer) which might be cacheable must implement the CacheableProcessingComponent interface. When the pipeline is processed each sitemap component starting with the generator is asked if it implements this interface. This test stops either when the first component does not implement the CacheableProcessingComponent interface or when the first cacheable component is currently not cacheable for any reasons (more about this in a moment).

The CacheableProcessingComponent interface declares a method getKey() which must produce a unique key for this sitemap component inside the component space. For example the FileGenerator returns the source argument (the xml document read). All parameters/values which are used for the processing of the request by the generator must be used for this key. If, e.g. the request parameters are used by the component, it must build a key with respect to the current request parameters. The key can be any serializable java object.

If for any reason the sitemap component detects that the current request is not cacheable it can simply return null as the key. This has the same effect as not declaring the CacheableProcessingComponent interface.

Now after the key is build for this particular request, it is looked up in the cache if it exists. If not, the new request is generated and cached for further requests.

If a cached response is found for the key, the caching algorithm checks if this response is still valid. For this check each cacheable component returns a validity object when the method getValidity is invoked. (If a cacheable component returns null it is temporarily not cacheable, like returning null for the key.)

A SourceValidity object contains all information the component needs to verify if the cached content is still valid. For example the file generator stores the last modification date of the xml document parsed in the validity object.

When a response is cached all validity objects are stored together with the cached response in the cache. Actually the CachedResponse is stored which encapsulates all this information.

When a new response is generated and the key is build, the caching algorithm also collects all uptodate cache validity objects. So if the cached response is found in the cache these validity objects are compared. If they are valid (or equal) the cached response is used and feed into the pipeline. If they are not valid any more the cached response is removed from the cache, the new response is generated and then stored together with the new validity objects in the cache.

5. Configuration

The caching of Cocoon can be completely configured by different Avalon components. This chapter describes how the various components work together.

5.1. Configuration of Pipelines

Each pipeline can be configured with a buffer size, and each caching pipeline with the name of the Cache to use.

5.1.1. Expiration of Content

Utilize the pipeline expires parameter to dramatically reduce redundand requests. Even the most dynamic application pages have a reasonable period of time during which they are static. Even if a page doesn't change for just one minute, still use the expires parameter. Here is an example:

```
<map:pipeline> <map:parameter name="expires" value="access plus 1 minutes"/> ... </map:pipeline>
```

The value of the parameter is in a format borrowed from the Apache HTTP module mod_expires. Examples of other possible values are:

```
access plus 1 hours access plus 1 month access plus 4 weeks access plus 30 days access plus 1 month 15 days 2 hours
```

Imagine 1'000 users hitting your web site at the same time. Say that they are split into 5 groups, each of which has the same ISP. Most ISPs use intermediate proxy servers to reduce traffic, hense improving their end user experience and also reducing their operating costs. In our case the 1'000 end user requests will result in just 5 requests to Cocoon.

After the first request from each group reaches the server, the expires header will be recognized by the proxy servers which will serve the following requests from their cache. Keep in mind however that most proxies cache HTTP GET requests, but will not cache HTTP POST requests.

To feel the difference, set an expires parameter on one of your pipelines and load the page with the

browser. Notice that after the first time, there are no access records in the server logs until the specified time expires.

This parameter has effect on all pipeline implementations, even on the non caching ones. Remember, the caching does not take place in Cocoon, it's either in a proxy inbetween Cocoon and the client or in the client itself.

5.1.2. Response Buffering

Each pipeline can buffer the response, before it is send to the client. The default buffer size is unlimited (-1), which means when all bytes of the response are available on the server, they are send with one command directly to the client.

Of course, this slows down the response as the whole response is first buffered inside Cocoon and then send to the client instead of directly sending the parts of the response when they are available. But on the other hand this is very important for error handling. If you don't buffer the response and an error occurs, you might get corrupt pages. Example: you have a pipeline that already send some content to the client and now an exception occurs. This exception "calls" the error handler that generates a new response that is appended to the already send content. If content is already send to the client there is no way of reverting this! So buffering in these cases makes sense.

If you have a stable application running in production where the error handler is never invoked, you can turn off the buffering, by setting the buffer to *0*.

You can set the buffer to any other value higher than 0 which means the content of the response is buffered in Cocoon until the buffer is full. If the buffer is full it's flushed and the next part of the response is buffered again. If you know the maximum size of your content than you can fine tune the buffer handling with this.

You can set the default buffer size for each pipeline implementation at the declaration of the pipeline. Example:

```
<map:pipe name="noncaching" src="..."> <parameter name="outputBufferSize"
value="2048"/> </map:pipe>
```

The above configuration sets the buffer size to *2048* for the non caching pipeline. Please note, that the parameter element does not have the sitemap namespace!

You can override the buffer size in each *map:pipeline* section:

```
<map:pipeline type="noncaching"> <map:parameter name="outputBufferSize"
value="4096"/> ... </map:pipeline>
```

The above parameters sets the buffer size to *4096* for this particular pipeline. Please note, that the parameter element does have the sitemap namespace!

5.2. Configuration of Caches

Each cache can be configured with the store to use.

5.3. Configuration of Stores

Have a look at the store configuration.

6. Additional Information for Developers

6.1. Java APIs

For more information on the java apis refer directly to the javadocs of Cocoon.

The most important packages are:

1. org.apache.cocoon.caching: This package declares all interfaces for caching.
2. org.apache.cocoon.components.pipeline: The interfaces and implementations of the pipelines.

6.2. The XMLSerializer/XMLDeserializer

The caching of the sax events is implemented by two Avalon components: The XMLSerializer and the XMLDeserializer. The XMLSerializer gets sax events and creates an object which is used by the XMLDeserializer to recreate these sax events.

6.2.1. org.apache.cocoon.components.sax.XMLByteStreamCompiler

The XMLByteStreamCompilercompiles sax events into a byte stream.

6.2.2. org.apache.cocoon.components.sax.XMLByteStreamInterpreter

The XMLByteStreamInterpreter is the counterpart of the XMLByteStreamCompiler. It interprets the byte stream and creates sax events.

6.2.3. Configuration

The XMLSerializer and XMLDeserialzer are two Avalon components which can be configured in the cocoon.xconf:

```
<xml-serializer class="org.apache.cocoon.components.sax.XMLByteStreamCompiler"/>
<xml-deserializer class="org.apache.cocoon.components.sax.XMLByteStreamInterpreter"/>
```

You must assure that the correct (or matching) deserializer is configured for the serializer.

Both components are poolable, so make sure you set appropriate pool sizes for these components. For more information on component pooling have a look at the Avalon documentation.

1. Comments

add your comments