Error Handling (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. Error Handling

During the execution of a Cocoon pipeline exceptions may occur within the involved components like generators, transformers etc. There are two possibilities to deal with them: The one would be not to handle them explicitly in the sitemap, which causes them to be logged and a default Cocoon error page to be displayed in the browser. The second is to define an error handling by using the sitemap tag <map:handle-errors>. Therein you are able to define any pipeline, that is executed in case of an exception occurred and displays an appropriate page.

1.1. ExceptionSelector

The ExceptionSelector allows to realize conditional error handling within <map:handle-errors>-tags depending on the type of the occurred exception. Each exception is configured centrally at the selector in the sitemap by associating a symbolic name to its class.

Furthermore it is possible to define, what exceptions are to be "unrolled". This means, that if an exception has been rethrown embedded in another exception, this original exception can be considered for choosing the correct error handling.

Example:

```
<map:selector name="exception" src="org.apache.cocoon.selection.ExceptionSelector">
<exception name="processing" class="ProcessingException" unroll="true"/> <exception
name="sax" class="SAXException"/> <exception name="application"
class="ApplicationException"/> </map:selector> ... <map:pipeline> <map:match
pattern="resource"> ... </map:match> <map:handle-errors> <map:select type="exception">
<map:when test="processing">...</map:when> <map:when test="sax">...</map:when>
<map:when test="application">...</map:when> </map:select> </map:handle-errors>
</map:pipeline>
```

Let's consider the following nested exceptions to occur:

1. ProcessingException (ApplicationException): The ProcessingException is unrolled, so the error pipeline for "application" will be executed.
2. ProcessingException (ValidationException): Since ValidationException is not configured at all and therefore unknown, the ProcessingException is not unrolled even if unrolling is enabled. Therefore the pipeline for "processing" will be executed.
3. SAXException (ApplicationException): The unrolling is not enabled for SAXException, so the pipeline for "sax" will be executed.

Please notice that the selector configuration is processed from top to bottom and stops at the first matching exception. Therefore the most specific classes must be configured first. This behaviour is the same as with Java catch statements.

1.2. XPathExceptionSelector

The XPathExceptionSelector is an extension to the standard selector described above. It adds the possibility to configure additional conditions for each exception type by using JXPath expressions, that operate on the exception object. This configuration is also done centrally at the selector in the sitemap, where symbolic names are defined for all specific error situations.

Example:

```
<map:selector name="exception"
src="org.apache.cocoon.selection.XPathExceptionSelector"> <exception name="Denied"
class="AuthenticationFailure"> <xpath name="PasswordWrong" test="authCode=10"/>
<xpath name="PasswordExpired" test="errorCode=11"/> <xpath name="AccessForbidden"
test="errorCode&gt;11"/> </exception> </map:selector> ... <map:pipeline> <map:match
pattern="login"> ... </map:match> <map:handle-errors> <map:select type="exception">
<map:when test="PasswordWrong">...</map:when> <map:when
test="PasswordExpired">...</map:when> <map:when
test="AccessForbidden">...</map:when> <map:when test="Denied">...</map:when>
<map:otherwise>...</map:otherwise> </map:select> </map:handle-errors> </map:pipeline>
```

In this example the exception AuthenticationFailure is configured under name "Denied". Additionally three further conditions "PasswordWrong", "PasswordExpired" and "AccessForbidden" are defined by using JXPath expressions. Therefore instances of AuthenticationFailure are expected to have methods getAuthCode() and getErrorCode(). Now the error handler defined for resource "login" has five branches: If situation "PasswordWrong" occurs, which means that an AuthenticationFailure exception with auth code 10 has been thrown, the first error pipeline is executed. If the error code equals to 11 the second pipeline is executed, if it is greater that 11 the third one and all other AuthenticationFailure errors are handled by the fourth one. In any other error situation the fifth branch would be chosen.

Please notice that the selector stops when it finds the first JXPath expression in the configuration that matches:

Example:

```
<map:selector name="exception"
src="org.apache.cocoon.selection.XPathExceptionSelector"> <exception name="application"
class="ApplicationException"> <xpath name="error3" test="errorCode&gt;3"/> <xpath
name="error6" test="errorCode&gt;6"/> </exception> </map:selector> ... <map:pipeline>
<map:match pattern="processForm"> ... </map:match> <map:handle-errors> <map:select
type="exception"> <map:when test="error6">...</map:when> <!-- handler 1 --> <map:when
test="error3">...</map:when> <!-- handler 2 --> </map:select> </map:handle-errors>
</map:pipeline>
```

If an ApplicationException with error code 9 occurs, handler 2 is executed since error situation "error3" is configured before "error6" at the selector even if the expression for "error6" also evaluates to "true".

1.3. Error Handler Hierarchy

The tag <map:handle-errors> may be attached to any <map:pipeline> or the <map:pipelines> tag of the root sitemap or a subsitemap. Therefore it is possible to define two kinds of error handlers: A default handler may be defined within <map:pipelines> for applying to all resources of a sitemap. Alternatively individual handlers may be configured for sets of resources within <map:pipeline>.

Example:

```
<map:pipelines> <map:pipeline name="pipe1"> <map:match pattern="res1"> ...
</map:match> <map:handle-errors> <!-- this is an individual handler for pipe1 -->
</map:handle-errors> </map:pipeline> <map:pipeline name="pipe2"> <map:match
pattern="res2"> ... </map:match> </map:pipeline> <map:pipeline name="pipe3">
<map:match pattern="res3"> ... </map:match> </map:pipeline> <map:handle-errors> <!-- this
is the default handler for the whole sitemap --> </map:handle-errors> </map:pipelines>
```

In conjunction with the ExceptionSelector resp. the XPathExceptionSelector it is possible to define a hierarchy of error handlers for an application. The behaviour then is the following: If an error occurs within a pipeline, Cocoon at first checks if an individual handler for this pipeline is defined. If so and it is able to handle the error due to its selection the processing terminates. Otherwise Cocoon looks for a default handler of the current sitemap. If one is found it is called. Now there is the same behaviour as above: If it can handle the exception the processing terminates otherwise the searching proceeds within the pipeline where the subsitemap is mounted. This goes on until the default handler of the root sitemap has been considered. If an error could not be handled at all, it is processed by the Cocoon engine in the end.

Please notice that a <map:otherwise> breaks the hierarchy since all errors will be handled on this level. Therefore all levels above will be called never.

Example:

Root sitemap: <map:pipelines> <map:pipeline> <map:mount uri-prefix="sub" src="sub/"/> <map:handle-errors> <map:select type="exception"> <map:when test="resourceNotFound">...</map:when> </map:select> </map:handle-errors> </map:pipeline> <map:handle-errors> <map:generate src="generalerror.htm"/> <map:serialize/> </map:handle-errors> </map:pipelines> Subsitemap: <map:pipelines> <map:pipeline> <map:match pattern="processForm"> ... </map:match> <map:handle-errors> <map:select type="exception"> <map:when test="validation">...</map:when> </map:select> </map:handle-errors> </map:pipeline> <map:handle-errors> <map:select type="exception"> <map:when test="application">...</map:when> </map:select> </map:handle-errors> </map:pipelines>

Let's consider four situations concerning the above example:

1. A ValidationException occurs, because for instance the user entered an invalid value: The defined pipeline's handler is called. Since it has a matching <map:when>-section it is able to handle such an exception and therefore the processing is finished.
2. An ApplicationException occurs, because for instance a database connection has failed: The pipeline's handler is not able to handle the exception, so next the subsitemap's default handler is called. It has a matching <map:when>-section and is therefore able to handle the exception.
3. A ResourceNotFoundException occurs, because for instance some file is missing. Both the pipeline's and the subsitemaps' handlers are not able to handle it. Now Cocoon proceeds after the mount point of the subsitemap and finds its pipeline's handler next. It is able to handle a ResourceNotFoundException and therefore produces the output in this case.
4. A NullPointerException occurs, because something went completely wrong in the application: All handlers are not configured for such an exception and so the root sitemaps default handler will apply to it showing a general error page.

When handling exceptions in error handlers one has to take care about recursion when working with redirects. Consider the following sitemap:

Example:

<map:pipelines> <map:pipeline> <map:match pattern="resource"> ... <map:transformer type="foo"/> ... </map:match> <map:match pattern="error"> ... <map:transformer type="foo"/> ... </map:match> <map:handle-errors> <map:select type="exception"> <map:when test="connection"> <map:act type="redirect" src="cocoon:/error"/> </map:when> </map:select> </map:handle-errors> </map:pipeline> </map:pipelines>

This configuration may lead to an infinite loop: Imagine to call "resource" where the FooTransformer throws a ConnectionException, because the connection to a backend system has broken. The defined

error handler will handle it and the used action internally redirects to resource "error". This resource itself uses the FooTransformer to get some data from the backend, which of cause also causes a ConnectionException. This is handled by the error handler, which redirects to resource "error" and so on. Such an infinite loop may also occur when using several "nested" redirects, i.e. the error handler redirects to a resource, which redirects to another resource, which might produce the original exception.

When defining error handlers for an application such situation must be avoided. An easy rule would be: An error handling routine must never redirect to a resource for which the routine itself is responsible and which might produce the same error as just handled.

1. Comments

add your comments