

Introduction

CoUnit is a *unit testing framework* for Cocoon applications.

It can be used to specify and execute unit tests for pipelines and stylesheets.

CoUnit was developed by Nico Verwer and Jan Willem Boer, and is open source software (actually, it is in the public domain).

CoUnit is not meant for testing the web-frontend of your application; [HttpUnit](#)¹ does a fine job in that realm, and so does [AntEater](#)². CoUnit helps you to test code that lives in the pipelines, transforming XML (SAX events). At the moment, it is particularly good at testing XSLT stylesheets. Flowscript is not yet covered.

WARNING

CoUnit can be used with Cocoon 2.1.6 and higher. If you need to use an earlier version of Cocoon, you will need to 'retrofit' the MountTableMatcher from the Cocoon repository after 11 October 2004.

Unit testing and test-driven development

Unit (and regression) testing has been one of the *best practices* in software development for a long time (I first came across it in an article in *Dr. Dobbs Journal*, February 1997). More recently it has become popular as part of the [eXtreme Programming](#)³ method, where it has evolved into *test-driven development*. There are many good books and web sites about unit testing, which you should have a look at if you have never heard about these methods:

- [JUnit](#)⁴ - dedicated to software developers using JUnit or one of the other XUnit testing frameworks.
- [TestDriven](#)⁵ - promoting techniques, tools, and general good will in the test-driven community.
- [Test Driven Development](#)⁶ - a page at objectmentor.com

Unit testing is about testing your code at a fine level of granularity; a single method or function. Regression testing extends this method by making a set of unit tests repeatable and automated, thus giving you an instant picture of the quality of your code.

Test-driven development takes this idea a step further, and demands that you do not write tests to check your code *after* it has been written, but write them *before* you write code, as part of the specification of what that code is supposed to do. The development cycle now becomes:

1. Write a test that specifies a tiny bit of functionality.
2. Ensure the test fails. (You haven't built the functionality yet!)
3. Write only the code necessary to make the test pass.
4. Refactor the code, ensuring that it has the simplest design possible for the functionality implemented until now.

On a project, this should be done following these rules:

1. After every change, run all tests.
2. Refactor the code if a test fails.

1. <http://www.httpunit.org/>
2. <http://aft.sourceforge.net/>
3. <http://www.extremeprogramming.org/>

3. Only check-in the code if all tests succeed.
4. When a bug is found, first write a test for it, then refactor the code.
5. Tests are organized in 'suites' – groups of tests for a particular bit of functionality.

There are many benefits to this approach, please see the references cited above.

Testing frameworks

Unit testing, and test-driven development, must be easy to do and take as little time as possible. If not, software developers will avoid it, because "it is a waste of time", and their managers won't allow it, because "it is a waste of time". And even if you have experienced the time savings (especially in non-trivial projects), it is nice to be able to do something useful with as little effort as possible, so you can concentrate on those things that are more difficult and time-consuming than they should be.

Unit-testing *frameworks* have been developed, which facilitate writing unit tests, run test suites, and generate reports. One of the most well-known is [JUnit](#)⁷, from which many other XUnit frameworks sprang. Each of these frameworks targets a specific language or development environment.

For the purpose of testing Cocoon applications, the following are interesting:

- [JUnit](#)⁸: Java
- [xUnit](#)⁹: Many other programming languages.
- [HttpUnit](#)¹⁰: Web applications.
- [Anteater](#)¹¹: Web services.
- [XMLUnit](#)¹²: XML; can test equality of XML documents, XSLT results, XPath expressions. Tests are written in Java or C#.
- [XSLTUnit](#)¹³: XSLT style sheet.

None of these does exactly what you need to test Cocoon pipelines, which is why I decided to build a specific unit-testing framework for this purpose. After a few iterations, this became CoUnit, which was first presented at the Cocoon [GetTogether 2004](#)¹⁴.

CoUnit is based on XSLTUnit, which is a single stylesheet developed by Eric van der Vlist of Dyomeda. XSLTUnit performs a single unit test on a XSLT style sheet by transforming a XML document, and comparing the result to a 'reference' output. Unfortunately, it falls short of being a framework, because it cannot execute test suites, and it doesn't do much reporting. These blind spots are filled in by CoUnit.

NOTE

XSLTUnit needs the [exsl:node-set\(\)](#)¹⁵ extension function, which is present in both Xalan-J and Saxon.

Requirements

Before starting to create code for CoUnit, I should have defined unit tests for it, but I didn't because I had no unit-testing framework to run them. What I did do, was to make a list of requirements which should ensure that CoUnit is a nice tool to work with:

- CoUnit must be transparent to the applications it tests. You should be able to set up your Cocoon application any way you like. No assumptions shall be made about sitemap-mounting structures,

7. <http://www.junit.org/>

14. <http://wiki.apache.org/cocoon/GT2004Notes>

15. <http://www.exslt.org/exsl/functions/node-set/index.html>

directories, symbolic links, etcetera. The only exception to this rule is, that you must create a directory called '_test' in the directory where your sitemap is.

- It must be easy to add tests and test suites. Overhead code that is not directly related to these must be minimized.
- It must be easy to run and generate reports for all tests belonging to an application, a test suite or an individual test by requesting their specific URIs.
- Multiple applications may exist in a Cocoon application independently. They must be testable independently.
- Tests can be written for stylesheets (XSLT or STX), pipeline fragments or pipelines. Later, flowscript might be added.

Using CoUnit to test a Cocoon-based application

Preparing CoUnit

To use CoUnit in your project or application, add the counit application directory to the Cocoon mount table with the uri-prefix "counit" (you may use a different uri-prefix). Alternatively, copy the counit directory ("counit2.0") to your webapp and rename it to "counit".

In order to find Cocoon applications that must be tested, CoUnit has its own mount-table. In order to test an application, you add it to the counit mount table. The counit mount table is located in the directory directly above the counit directory. This makes it easy to update counit to a new version without affecting your counit mount table. The mount table file name is counit-mount-table.xml. To add an application to the mount table, add a mount element with the attributes uri-prefix and src. The src attribute is the absolute path to the directory containing the sitemap. Unlike normal Cocoon mount-tables, it must not point to a sitemap directly. The uri-prefix attribute can be anything you like, and is the identifier with which the tests can be executed.

```
<mount-table>
  <mount uri-prefix="MyAppToTest"
        src="file:///C:/Projects/MyProject/build/webapp/" />
</mount-table>
```

The entries in the mount table must always have a slash as the last character in the src attribute.

Now you should be able to run counit by pointing your webbrowser to <http://server:port/counit/report/MyUriPrefix/>. It will show an empty document because no tests have been defined yet.

Defining a simple test

In this case, we assume we want to test an xsl stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:param name="to" />
    <xsl:template match="to">
      <xsl:value-of select="$to" />
    </xsl:template>
    <xsl:template match="*|node()">
      <xsl:copy>
```

```
<xsl:apply-templates select="@*|node()" />
</xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

The stylesheet receives a parameter to and replaces every element `<to>` by the value of the parameter to. We assume this stylesheet resides in the application directory and has the name `greeting.xsl`.

The tests will be organized in a directory called `.counit`, which is located in the application directory. Create a new directory `".counit"` in the application directory (where the `sitemap.xmap` resides). On Windows, this can only be done by using the **mkdir** command from a command window.

A test consists of three parts:

- an xml input document
- an expected xml output document
- the definition of the test

In our example, the xml input document is a small xml fragment that is the input to the component (in this case the xsl stylesheet) that we want to test.

```
<?xml version="1.0" encoding="UTF-8"?>
<greeting>Hello <to/>, how are you?</greeting>
```

Save this file in the `.counit` directory with the name `01-greeting-input.xml`.

Next, counit needs the expected output document. We will pass the parameter `"world"` to the transformation, so the expected output would be as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<greeting>Hello world, how are you?</greeting>
```

Save this file in the `.counit` directory with the name `01-greeting-output.xml`.

To finish the test, add the test to the testsuite. The testsuite is an xml file which describes a related set of tests. When testing the application, counit will automatically find the `testsuite.xml` file and execute all the tests described in the testsuite file. Create a file called `testsuite.xml` in the `.counit` directory. The testsuite file has an element `testsuite` as root element. A testsuite can contain other testsuite elements or testcase elements. The testcase element has an `id` attribute, an `input` and an `expect` attribute in which the test files can be referenced. The testcase element contains the transformation details which define which transformation will be executed on the input file.

```
<?xml version="1.0" encoding="UTF-8"?>

<testsuite>
  <testcase id="01-greeting" input="01-greeting-input.xml"
    expect="01-greeting-output.xml" description="A sample test">
    <xslt src="greeting.xsl">
      <parameter name="to" value="world"/>
    </xslt>
  </testcase>
</testsuite>
```

In the example we define the transformation as an xslt transformation with the stylesheet `greeting.xsl`. As parameter we define to to be `"world"`. A good practice is to use the `description` attribute for an adequate description of the test. The contents of the `description` attribute will be shown if the test fails, which can be very convenient to locate the problem.

Now you can browse to the count test suite again, and observe that your test is succesful. You should see a page similar to the one in the example below. If one or more tests fail, detailed diff information is given. To re-test this testcase, click on the name of the testcase. In a real-world test-driven situation, you would have written the test first, watched it fail, and then written the stylesheet.

In a textual context whitespace differences are often not relevant. If the testcase attribute ignore-whitespace, is set to true (which is the default), the whitespace in the output and expected output documents will be normalized. If this gives unwanted side-effects, use ignore-whitespace="false".

The screenshot shows a web browser window displaying the results of a CoUnit test suite. The path is `D:/Projecten/XMLontwikkelstraat/CoUnit/HelloWorld/`. A red banner at the top indicates "Failed (1 of 6 tests)". Below this, the test suite name "HelloWorld" is shown, followed by another red banner "Failed (1 of 5 tests)". A table lists the test cases, with "03-greetingParameterBad" highlighted. The details for this test case show the expected output as an XML error message: `<error:message> Bad words not allowed. Terminating! </error:message>`. The actual received output is: `<greeting> Hello meconium, how are you? </greeting>`. At the bottom, a green banner indicates "Passed (1 tests)" for the sub-suite `D:/Projecten/XMLontwikkelstraat/CoUnit/HelloWorld/ByeWorld/`.

[Show all tests, not only failed ones.](#)

Including input and expected output in testcase.xml

An alternative to this approach is to include the test input and expected output in the testsuite.xml file. For small tests this is more efficient than the separate files approach. The input and expected output are included directly in the testcase in the input and expect elements.

```
<testcase id="03-reverse" ignore-whitespace="true">
  <input>
    <text-to-reverse>
      The text in this element
      <embedded-element/>
      will be reversed.
    </text-to-reverse>
  </input>
  <expect>
    <text-to-reverse>
      tnele siht ni txet ehT
      <embedded-element/>
      .detrever eb lliw
    </text-to-reverse>
  </expect>
  <xslt src="reverse.xsl"/>
</testcase>
```

The input and expect elements can only contain one root element.

Note

This mechanism does not work for XML that can be interpreted by a Cocoon component, like the cinclude transformer. This is because thee cinclude and sourcewriting transformers are used in the testing process.

If the XML in input or expect elements includes instructions for these components, they will be interpreted and executed by the test pipeline and most likely result in strange error messages. To test cinclude and sourcewriting XML instructions, use the external files for input and expected output.

Alternatively, if the expected output contains cocoon instructions, and the input doesn't, you could use a filter stylesheet to transform cinclude- and sourcewriting elements in the output to other elements. See the [output-filter](#)¹⁶ option about the use of output filtering stylesheets. You could use the following filtering stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:ci="http://apache.org/cocoon/include/1.0"
  xmlns:filtered="http://cocoodev.org/counit/filtered"
  xmlns:source="http://apache.org/cocoon/source/1.0"
  version="1.0">

  <xsl:template match="ci:*|source:*">
    <xsl:element name="filtered:{local-name()}"
      namespace="http://cocoodev.org/counit/filtered">
      <xsl:attribute name="filtered:ns">
        <xsl:value-of select="namespace-uri()"/>
      </xsl:attribute>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:element>
  </xsl:template>

  <!-- copy everything else -->
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>

      <xsl:apply-templates select="node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

This stylesheet eliminates the problematic namespaces, but preserves the information about them. Using this stylesheet, the expected output can be stripped of cinclude and sourcewriting tags. For example:

```
<testcase id="..." description="...">
  <input>

    <recipe-step name="teststep">
      <pipeline file="pipe.xml"/>
      <transform type="xslt" file="create.xml"/>
    </recipe-step>
  </input>
  <output-filter src="filter-cocoon-instructions.xml"/>

  <expect>
    <recipe-step name="teststep">
      <pipeline file="pipe.xml">
        <filtered:include
          filtered:ns="http://apache.org/cocoon/include/1.0"
          src="teststep/pipe.xml"/>
      </pipeline>
      <transform type="xslt" file="create.xml"/>
    </recipe-step>
  </expect>
```

16. #outputFilter

```
<xslt src="some_stylesheet.xsl"/>
</testcase>
```

Test a transformer

To test a transformer, the transformer must have been compiled into the Cocoon distribution in which CoUnit is running. As an example, we will write a test for the Cocoon i18n transformer.

A transformer can be tested by including a transformer element in the testcase. The src attribute must be set to the full java classname of the transformer. In this case we want to test the i18n transformer, so the src attribute would be org.apache.cocoon.transformation.I18nTransformer.

If the transformer needs configuration, this included in the configuration element. Parameters that would normally be declared in the map:transform element should be included within the transformer element. The input and expected output settings can be configured in the same way as the xslt testcases.

```
<testcase id="i18n" ignore-whitespace="true"
  description="Example test of i18n transformer">
  <input>

    <test>
      <i18n:date src-pattern="short" src-locale="en_US" locale="de_DE">
        12/24/01
      </i18n:date>
    </test>
  </input>
  <expect>

    <test>
      24.12.2001
    </test>
  </expect>
  <transformer src="org.apache.cocoon.transformation.I18nTransformer">
    <configuration>
      <catalogues default="cat">

        <catalogue id="cat" name="messages">
          <location>translations</location>
        </catalogue>
      </catalogues>
      <untranslated-text>untranslated</untranslated-text>

      <preload>en_US</preload>
      <preload catalogue="cat">de_DE</preload>
    </configuration>
    <parameter name="locale" value="en_US"/>
    <parameter name="untranslated-text" value="not translated"/>

  </transformer>
</testcase>
```

The testsuite.xml file

testsuite

The testsuite.xml file has one root element, the testsuite element. This element may contain other testsuite elements or testcase elements.

With the optional id attribute the testsuite can be given an id to identify the testsuite in the output. If no id is provided, the full path will serve as the id.

The element can also be used to include other testsuite.xml files. This makes it easy to create your tests in a modular way. The src attribute should contain the path to the other application to test, relative to this testsuite (= a directory with a .coint directory which contains the testsuite.xml file to mount).

```
<testsuite>
  <testsuite id="HelloWorld">
    <testcase ...
    ... /testcase>
  </testsuite>
  <!-- included testsuite from /HelloWorld/ByeWorld/ -->
  <testsuite src="ByeWorld/" />
</testsuite>
```

testcase

The testcase element is a child element of testsuite. A testcase defines a single unit test. The id attribute is used to identify this unit test. The description attribute can be used to provide a more verbose identifier to this test and can be very useful to find out what is wrong when a test fails.

The input xml for the test is defined in the input attribute or in the input element. The attribute points to the location (relative to the testsuite.xml file) where the file can be found which contains the input; the element contains the input itself (the input xml is inline). In the latter case it is not possible to use xml which serves as instructions for Cocoon components like the cinclude and the sourcewriting transformers (and possible more in the future), because these instructions will be interpreted and executed by the CoUnit pipelines. To use these instructions as test input, use the input attribute instead.

The expected output xml is defined in the same way: either by the expect attribute or by the expect element. The same possibilities and restrictions apply.

Default any any whitespace differences are ignored when comparing the expected and received output. Set the ignore-whitespace attribute to false to achieve this. The default is true.

Which transformation will be applied to the input document is defined by one of the transformation elements. Currently available are xslt, stx and transformer.

```
<testcase id="01-reverse" ignore-whitespace="true"
  description="Each text should be reversed seperately">
  <input>
    <root-element>
      The text in this element
      <embedded-element/>
      will be reversed.
    </root-element>
  </input>
  <expect>
    <root-element>
      tnemele siht ni txet ehT
      <embedded-element/>
      .detrever eb lliw
    </root-element>
  </expect>
  [...]
</testcase>

<testcase id="02-testCInclude" input="cinclude/input.xml"
  expect="cinclude/expect.xml"
  description="cincludes should not be included in external test input">
  [...]
```

```
</testcase>
```

xslt

The xslt element is a child element of testcase and defines that an XSL transformation will be applied to the input document of the testcase.

The src attribute points to the XSL stylesheet to be applied. The path is relative to the root of the application (NOT to the testsuite.xml file).

Parameters used as input for the XSL transformation can be provided in this element as parameter elements, which should have a name and value attribute.

```
<testcase [...]>
  <xslt src="greeting.xsl">
    <parameter name="to" value="test"/>
  </xslt>
</testcase>
```

stx

The stx element is a child element of testcase and defines that an STX transformation will be applied to the input document of the testcase.

The src attribute points to the STX stylesheet to be applied. The path is relative to the root of the application (NOT to the testsuite.xml file).

Parameters used as input for the STX transformation can be provided in this element as parameter elements, which should have a name and value attribute.

```
<testcase [...]>
  <stx src="transform.stx">
    <parameter name="param" value="hello"/>
  </stx>
</testcase>
```

transformer

The transformer element is a child element of testcase and defines that a custom transformation will be applied to the input document of the testcase.

The src attribute should contain the fully qualified class name of the custom transformer. This transformer should have been compiled into the Cocoon distribution which CoUnit is running on, otherwise you will receive an error message containing the classname as output.

Any configuration, which would appear in the map:components section of the sitemap when declaring the transformer component, should be included in the configuration child element.

Any runtime parameters, which would appear as map:parameter in the map:transform element in the sitemap, should be included in parameter child elements.

```
<testcase [...]>
  <transformer
    src="org.apache.cocoon.transformation.ReplaceByRegExpTransformer">
    <configuration>
      <case-insensitive>true</case-insensitive>
    </configuration>
  </transformer>
</testcase>
```

```

</configuration>
<parameter name="regexp" value="[a-z]+[0-9]+"/>

<parameter name="replacement" value="(censored)"/>
</transformer>
</testcase>

```

output-filter

An output document may contain XML that is not interesting to the testcase. It can also contain dynamic content that changes with every call (for example a timestamp). This would make it impossible to make up a decent expected output to compare the results with. For these cases the output can be filtered with the `output-filter` element, which is a child element of the `testcase` element. The `src` attribute should point to a stylesheet which filters the output. The location in `src` is relative to the `testsuite.xml` file.

```

<testcase id="05-filter-output" description="Filter output">

  <input>
    ...
  </input>
  <output-filter src="filter-timestamp.xml"/>
  <expect>
    ...
  </expect>

  ...

```

The stylesheet which does the filtering can be something like:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:template match="@timestamp"
    <!-- remove -->
  </xsl:template>

  <xsl:template match="*|node()">
    <xsl:copy>
      <xsl:apply-templates select="*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>

```

For simple filters like exclusions and inclusions, the built-in filter generator can be used. In this case, the `src` attribute should be absent. First a default action should be provided in the `action` attribute. Actions can be `remove-all` and `preserve-all`. The first will result in recursively removing all elements down the tree, the second in recursively copying all elements down the tree. When an `output-filter` element is provided in the `testcase`, without an `action` attribute, the default action value is `remove-all`.

Next, the recursive processing mode can be switched by defining `preserve` and `remove` elements in the `output-filter` element. Both elements have a `select` attribute, which should contain an expression that matches the elements from which the mode will be switched to `preserve-all` or `remove-all`.

If elements are removed, this can result in an invalid document, because the output should have one and only one root element. To circumvent this, use the `use-root-element` attribute. The value of this attribute will be used as root element for the document.

Special care is needed when using namespace prefixes in the `select` attributes of the `preserve` and `remove` elements. Under the hood a filter stylesheet is generated, which doesn't include the proper namespace

declarations by default. To use a namespace prefix in the filter, the prefix should be declared by adding an attribute to the output-filter element. The local name of this attribute should be `include-namespace`, the prefix to the attribute should be the namespace prefix to include. Alternatively use the `name()` construction: `<preserve select="*[name()='prefix:element']"/>`. An example of a filter that removes elements and attributes from the output, and needs a container root-element:

```
<testcase id="06-filter-output" description="Filter output"
  xmlns:sense="sensical" xmlns:nonsense="nonsensical">

  <input>
    <nonsense:element>
      <nonsense:boring>
        <sense:texts>
          <text boring-attribute="true">This is some text to replace.</text>

          <nonsense:boring/>
        </sense:texts>
        <sense:texts>
          <text>This is text not meant to be replaced.</text>
        </sense:texts>
      </nonsense:boring>
    </nonsense:element>
  </input>
  <!--
    all elements are copied to the result tree, only the text
    is modified. We are only interested in the modifications,
    so remove the copied elements and attributes.
  -->
  <output-filter use-root-element="blah"
    sense:include-namespace="true"
    nonsense:include-namespace="true">
    <preserve select="sense:texts"/>

    <remove select="sense:texts/nonsense:boring"/>
    <remove select="@boring-attribute"/>
  </output-filter>
  <expect>
    <blah>
      <sense:texts>

        <text>This was an awesome text to replace.</text>
      </sense:texts>
      <sense:texts>
        <text>This is text not meant to be replaced.</text>
      </sense:texts>

    </blah>
  </expect>
```

Note the use of the `sense:include-namespace="true"` attribute in the above example to make sure the namespace can be used in the filter. An example of a filter that uses the `preserve-all` mode to begin with:

```
<testcase id="07-filter-output" description="Filter output">
  <input>

    <nonsense:element xmlns:nonsense="nonsensical">
      <nonsense:boring>
        <sense:texts xmlns:sense="sensical">
          <text>This is some text to replace.</text>
        </sense:texts>

        <sense:texts xmlns:sense="sensical">
          <text>This is text not meant to be replaced.</text>
        </sense:texts>
      </nonsense:boring>
    </nonsense:element>
  </input>
  <output-filter use-root-element="blah"
    preserve-all="true">
  </output-filter>
  <expect>
    <blah>
      <nonsense:boring>
        <sense:texts>
          <text>This was an awesome text to replace.</text>
        </sense:texts>
        <sense:texts>
          <text>This is text not meant to be replaced.</text>
        </sense:texts>
      </nonsense:boring>
    </blah>
  </expect>
```

```

        </nonsense:boring>
    </nonsense:element>

</input>
<output-filter action="preserve-all">
    <remove select="text"/>
</output-filter>
<expect>
    <nonsense:element xmlns:nonsense="nonsensical">

        <nonsense:boring>
            <sense:texts xmlns:sense="sensical">
                </sense:texts>
            <sense:texts xmlns:sense="sensical">
                </sense:texts>
            </nonsense:boring>

        </nonsense:element>
    </expect>
    [...]

```

CoUnit components and options

Caution

Het is duidelijker om hier het plaatje uit de ontwerp-SVG over te nemen. Met wat voorbeelden.

The CoUnit entrypoints are the test/**- and report/**-patterned matches in the main CoUnit sitemap. They produce the testdata, the latter in a nice HTMLized format.

The main test pipeline which is the source for the entry points, is the **-patterned match. Here the testcase.xml is read and the testcases prepared. The test pipeline consists of several steps, which will be explained below. If something goes wrong, the test pipeline can be debugged on step level. The pipeline can be "paused" on every step. This is done by using the cocoon-view url parameter. For example <http://127.0.0.1:8888/counit/HelloWorld/?cocoon-view=write-source> will execute the tests up to the write-source step.

The testcases are executed within the transformations pipeline. Each test type (xslt and transformer) has its own match. Errors in this pipeline are caught and treated as test output. Custom CoUnit transformation test types should be added in the transformations pipeline.

mount-table

The first step in the test pipeline is the mount-table matcher. In this step the uri part after "counit" is converted to the actual location of the application that has to be tested. After this step, the {src} pipeline parameter contains the location.

find-testsuite

With the given location, the step find-testsuite will look for the testsuite.xml file, which should be located in the .counit directory at that location. It is also possible to tell this step to look into a subdirectory instead of the root directory. This is done by adding the path to the URL, like <http://127.0.0.1:8888/counit/HelloWorld/ByeWorld/>. The HelloWorld part is looked up in the counit mount-table and will be converted into a path (or result in an error message if the mount entry is not found). The ByeWorld part is treated as a subdirectory of this path.

The testcase.xml file can contain references to other testsuites. These will be included in this step.

It is can be desirable to execute only one test at a time, for example to test a failed testcase after fixing it. To execute one testcase in any of the found testsuite.xml files, add the testcase id to the path: <http://127.0.0.1:8888/counit/HelloWorld/ByeWorld/thefirsttestcase>. This will look for a testcase with id "thefirsttestcase" and execute that testcase only. The testcase doesn't necessarily have to be a testcase in the ByeWorld testsuite.xml, but can also be a testcase from a "mounted" testsuite.xml. A link to execute a testcase is automatically provided in the HTML output if a testcase fails.

prepare-xslt and prepare-stx

These steps prepare the testcases which have defined an xslt or stx as the transformation to test. The preparation is in fact the brewing of a cinclude instruction, which will tell the cinclude transformer to call the transform-with-ss pipeline. This will be a complete POST request which does not use the internal cocoon:/mechanism. This makes it possible to handle the errors and treat any error data as output. If the internal cocoon:/mechanism would have been used, cocoon will present an ugly error page and stop the test pipeline.

The XSL file to call and the input xml are included as POST parameters. The input xml can be passed by two ways: via the @input attribute on the testcase element, which contains a URI, which points to the location of an XML input document, or by the input element, which contains the XML directly. In the first case the called transform-with-ss pipeline will generate from the URI, in the latter case it will use the stream generator to read the XML from the request parameter.

Because the cinclude (and write-source) steps will be executed after this step, it is not possible to use cinclude or sourcewriting instructions in the input- or expected XML if these are included in the testsuite.xml document. These instructions would be executed by the following steps. Use the URI method for input if the input contains special instructions for cocoon components.

Any parameters that are defined in the xslt or stx element will be passed in the same request and will be passed through to the stylesheet transformation.

Instead of using a URI in the src attribute, a generated stylesheet can be used as well. In this case, don't include a src attribute and include two child elements in the xslt or stx element: an input element and a transformation element like xslt, stx or transformer. The input will be passed through the given transformation and should result in a generated stylesheet.

Generated stylesheets can be nested with no limits except the physical limits of your computer and your patience.

A generated stylesheet can not contain imports which have a relative URL as href attribute. In these cases you will receive a cryptic error message about the cocoon: protocol that is not recognized.

prepare-transformer

The prepare-transformer step prepares testcases which are meant to test a custom transformer, i.e. which have a transformer element. The preparation consists of two things. Firstly an instruction for the sourcewriting transformer is made up to generate a small sitemap which declares the transformer and calls it with the input xml. Secondly a cinclude instruction is prepared to call the do-transformation pipeline from the main sitemap. This pipeline will actually mount the newly generated sitemap.

The new sitemap will be generated in the CoUnit directory and will most likely have a unique name. In a following step the generated sitemap will be deleted again, but if the pipeline has been halted by using the cocoon-view parameter or by a fatal error, you might end up with garbage in the CoUnit directory.

The mechanism to include the input xml is the same as the previous step, see the information in [prepare-xslt](#)¹⁷ about this.

If the transformer needs configuration, any parameters can be included in the configuration element which is a child element of transformer. These will be copied into the generated sitemap. Any run-time parameters to the transformation can be included in the parameters element. These will be included into the generated sitemap as map:parameter elements.

(new transformation test types)

Any custom CoUnit transformation test types like xslt and transformer should be prepared with a stylesheet on this place in the testpipeline. The stylesheet should include cinclude elements to call the transformation with a full URL.

prepare-filter

The output-filter instruction is interpreted by the prepare-filter step. This step writes a temporary stylesheet to disk if the built-in filter features are used. After that, it extends the cinclude instruction which triggers the transformation with a parameter which contains the URI to the stylesheet (generated or user-defined). The sitemap resource "post-testcase" will pick up this parameter and execute the stylesheet. Any user defined CoUnit transformation should include a call to post-testcase if the output should be filterable.

write-source

This step will execute the sourcewriting instructions that have been prepared by previous steps. After this step, the temporary sitemaps have been written to disk.

cinclude

This step will execute the cinclude instructions that have been prepared by previous steps. In fact this means that this steps does the actual transformation, because the cinclude tags refer to the transformation pipeline. After this step the output of each transformation is included in the testcases in the output element.

prepare-cleanup and cleanup

The prepare-cleanup step includes an instruction to the sourcewriting transformer to delete the generated sitemap. The cleanup step performs the deletion.

include-expected-output

The step include-expected-output is only necessary for testcases that don't contain an expect element, but use the @expect attribute that contains an URI. This step creates a new expect element and includes the XML from the given URI in this element.

remove-whitespace

It can be very annoying to make sure the expect element resembles the exact output of the transformation if the output contains unexpected whitespace or leaves out whitespace where you would expect it. To circumvent this, any whitespace is normalized by default. If this gives unwanted side-effects, the @ignore-whitespace attribute on the testcase can be used. This step normalizes whitespace from any text() node if the attribute is NOT set to false.

17. #input-xml

evaluate-results

Evaluate-results compares the contents of the expect element with the contents of the output element (see [cinclde](#)¹⁸). This is done by using the xsltunit 0.2 library. The result of the comparison is added to the testcase.

format-results

This step removes the garbage from the output of all the preceding steps, and formats the results of the comparison to resemble the JUnit XML output.

html-report (optional)

If the report/** pipeline is used, the html-report step will be executed as the last step. This step formats the JUnit results and shows them to a nice HTML representation.

Project resources

- **Mailing lists:** <http://lists.cocoondev.org/mailman/listinfo>
- **SVN:** <http://svn.cocoondev.org/repos/counit>
- **Cocoon Wiki:** <http://wiki.apache.org/cocoon/UnitTestingStylesheetsAndPipelines>
- **Get the code**²²

Frequently asked questions

Q: I always get a NullPointerException trying to run unit tests, even those that come with CoUnit. A: You need to update Cocoon to version 2.1.6 or higher, or at least update the MountTableMatcher to a version later than 11 October 2004. On that date, Sylvain Wallez fixed [bug #31637](#)²³.

Links

CoUnit 2.0 on Google Code
<http://code.google.com/p/counit/>

18. #cinclde

23. http://issues.apache.org/bugzilla/show_bug.cgi?id=31637