wscltest user tutorial

Anton Lluis (Tonlluis) Fontgivell Mas
UOC - Universitat Oberta de Catalunya

tonlluis@uoc.edu
tonlluis@gmail.com

Copyright © 2009 Anton Lluis Fontgivell Mas

2009-02-04

1. Introduction

wscltest is a Comand Line tool to easy test webservices. It uses a working directory where a configuration file resides in along with xml request files. The configuration file defines the service's url and the operations to be called in sequential order.

wscltest is free software, licensed under GPL v. 2. You can find it at Source Forge (http://sourceforge.net/projects/wscltest)

This document is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported License. To view a copy of this license, you can click here (http://creativecommons.org/licenses/by-sa/3.0/) or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

2. Basic usage

To test a SOAP webservice's operation you send a request message and receive a response one. If you were to test several operations in a sequential order, how would you do it? A possibility would be to put every request message on a file, put all of these files in a folder and let the responses be saved in that folder too (so you can check if they're right). Well, that's how wscltest works: put the requests files in a directory, open a console, change the promt to that path, and execute *wscltest*; the responses will be saved in that folder.

But, in order to acomplish this, wscltest would need some basic information:

- the URL where the service resides
- the operations you were to call

- what's the file that contains the request for every operation
- what's the name of the files where the responses are going to be saved

That information is defined in the *configuration file* which has to be located in the same directory as the requests files.

Things would be clearer with an example. Let's begin with that *configuration file*. It's compulsory that its name be *Tests.xml*. Here is an example:

```
<?xml version="1.0"?>
<Test>
  <TestName>MathService</TestName>
  <ServiceUrl>http://localhost:8080/NumberService.asmx</ServiceUrl>
  <Operations>
    <Operation>
      <OperationName>AddNumbers</OperationName>
      <SoapAction>http://tempuri.org/NumberService/AddNumbers</SoapAction>
      <FileRequest>addRequest.xml</FileRequest>
      <FileResponse>addResponse.xml</FileResponse>
    </Operation>
    <Operation>
      <OperationName>SubtractNumbers</OperationName>
      <SoapAction>http://tempuri.org/NumberService/SubtractNumbers</SoapAction>
      <FileRequest>substractRequest.xml</FileRequest>
      <FileResponse>substractResponse.xml</FileResponse>
    </Operation>
  </Operations>
</Test>
```

If you put this file in a directory where there are also the two requests files (*addRequest.xml* and *substractRequest.xml*) then you can execute wscltest from that location and you'll receive the responses in the files *addResponse.xml* and *substractResponse.xml*. Of course the requests files must contain valid SOAP messages fot that service's operations, and the service itself must be operating. (By the way, the example service and its two operations are not arbitrary, they are taken from the example webservice in the mono web site, see "Writing a Webservice" article on mono-project (http://www.mono-project.com/Writing_a_WebService)).

It's handy to locate the command line in the folder and simply type wscltest:

```
prompt:~/myTests/numberServiceFolder> wscltest
```

But it's also possible to pass an argument especifying the path to the working directory (be it a relative path or an absolute one):

```
prompt:~> wscltest -wd myTests/numberServiceFolder
```

Here the option *-wd* especifies the location of the working directory, where the *Tests.xml* and the requests files must reside.

The elements "TestName" and "OperationName" are only informative, you can put there the name you want, without any impact on the service's call. There is another element that is informative: *WsdlUrl* (shown in an example in a latter section).

3. Automatic generation of files from a wsdl

A utility option of wscltest is -wsdl:

```
wscltest –wsdl http://localhost:8080/NumberService.asmx?wsdl
```

This way you generate templates of Tests.xml and request files for all the operations of the service described in the wsdl. They are only templates, so you need to edit them (but you are not forced to begin from scratch). When this option is used wscltest will call no operation, only the files will be created in the working directory.

This option accepts two types of wsdl locations: a url or a file. For the former it must begin with "http://" and the latter must be a path (relative or absolute, but not "url type" as "file:///"). The next examples show three different possibilities:

The wsdl file is located in the working directory and wscltest is executed from it:

```
wscltest –wsdl NumberService.wsdl
```

The wsdl is located in the working directory and wscltest is executed from another location:

```
wscltest –wsdl NumberService.wsdl –wd ../webservices/test
```

wscltest is executed from the working directory but the wsdl is not located there:

```
wscltest –wsdl /home/myhome/myWsdls/NumberService.wsdl
```

Important: Although a wsdl file may describe more than one service, this option will only consider the first it encounter. Besides, only SOAP 1.1 over http services and document/literal operations are contemplated.

4. Generating dinamically requests from previous responses

Imagine you have to substract 7 to the response of the AddNumbers operation and, for some obscure reasons, you cannot watch the contents of the addRequest.xml file (so you're not able to do the sum yourself and generate by hand the substractRequest.xml file). wscltest offers you two ways to generate a new request file in a "dinamic" way: by xslt or invoking an external program. Again, an example would clarify things; here is the new *Tests.xml*:

```
<?xml version="1.0"?>
<Test>
  <TestName>MathService</TestName>
  <WsdlUrl>http://localhost:8080/NumberService.asmx?wsdl</WsdlUrl>
  <ServiceUrl>http://localhost:8080/NumberService.asmx</ServiceUrl>
  <Operations>
    <Operation>
      <OperationName>AddNumbers</OperationName>
      <SoapAction>http://tempuri.org/NumberService/AddNumbers</SoapAction>
      <FileRequest>addRequest.xml</FileRequest>
      <FileResponse>addResponse.xml</FileResponse>
      <ProcessFiles>
        <Method>xslt</Method>
        <Name>addToSubstractRequest.xsl</Name>
        <InputFile>addResponse.xml</InputFile>
        <OutputFile>substractRequest.xml</OutputFile>
      </ProcessFiles>
    </Operation>
    <Operation>
      <OperationName>SubtractNumbers</OperationName>
      <SoapAction>http://tempuri.org/NumberService/SubtractNumbers</SoapAction>
      <FileRequest>substractRequest.xml</FileRequest>
      <FileResponse>substractResponse.xml</FileResponse>
    </Operation>
  </Operations>
</Test>
```

The lines you have to attend are:

```
  <ProcessFiles>
    <Method>xslt</Method>
    <Name>addToSubstractRequest.xsl</Name>
    <InputFile>addResponse.xml</InputFile>
```

```
    <OutputFile>substractRequest.xml</OutputFile>
  </ProcessFiles>
```

This will instruct wscltest that, after invoking the *AddNumbers* operation, it parse *addResponse.xml* through *addToSubstractRequest.xsl* and produce *substractRequest.xml*.

Note that initially in the working directory you'd have only the configuration file (Tests.xml) and addRequest.xml. You could not have the substractRequest.xml because you don't know what information it would have to contain. It's wscltest, parsing the previous operation response with the xsl file, that produces the request needed to call the second operation.

Xslt is a convenient way to do that kind of transformations. But in some cases it could be easier to create the new request file the "old way": invoking an external program or script. The previous bunch of lines would become:

```
  <ProcessFiles>
    <Method>exec</Method>
    <Name>getSubstractFromAdd</Name>
    <InputFile>addResponse.xml</InputFile>
    <OutputFile>substractRequest.xml</OutputFile>
  </ProcessFiles>
```

Here *getSubstractFromAdd* is an executable file that takes *InputFile* and *OutputFile* elements as arguments, and must produce a file named as the second argument. Be aware that you cannot execute a mono program directly, neither a java one. In these cases you have to write a script that calls your program.

5. especifying a proxy

Of course you are not going to call all the time webservices that are hosted on your localhost, and maybe you'll have to access them through a proxy. In that case you have to specify it in the Tests.xml file:

```
<?xml version="1.0"?>
<Test>
  <TestName>MathService</TestName>
  <WsdlUrl>http://214.23.156.10/NumberService.asmx?wsdl</WsdlUrl>
  <ServiceUrl>http://214.23.156.10/NumberService.asmx</ServiceUrl>
  <Proxy>
    <Address>10.12.154.24</Address>
    <Port>80</Port>
  </Proxy>
  <Operations>
    <Operation>
      <OperationName>AddNumbers</OperationName>
      <SoapAction>http://tempuri.org/NumberService/AddNumbers</SoapAction>
      <FileRequest>addRequest.xml</FileRequest>
```

```
      <FileResponse>addResponse.xml</FileResponse>
      <ProcessFiles>
        <Method>xslt</Method>
        <Name>addToSubstractRequest.xsl</Name>
        <InputFile>addResponse.xml</InputFile>
        <OutputFile>substractRequest.xml</OutputFile>
      </ProcessFiles>
    </Operation>
    <Operation>
      <OperationName>SubtractNumbers</OperationName>
      <SoapAction>http://tempuri.org/NumberService/SubtractNumbers</SoapAction>
      <FileRequest>substractRequest.xml</FileRequest>
      <FileResponse>substractResponse.xml</FileResponse>
    </Operation>
  </Operations>
</Test>
```

6. load testing

You can do load testing specifying it on the root element:

```
<Test Threads="5" RunsPerThread="10">
...
</Test>
```

In the example above you are saying to wscltest that it launch 5 concurrent threads and that every thread execute sequentially 10 tests. When wscltest has done it you will find in the working directory 5 subdirectories with the name "thread_0", "thread_1" ... "thread_4", and inside every one of them a subdirectory for every run ("run_0" and so on). Inside every "run_x" subsubdirectory you will find the requests that have been sent and the responses that have been received for that run in that thread.

7. results summary report

At the end of the test there will appear at the command line a list with the execution time for every operation. If it is a load test the time is an average one.

8. Results report

When the test is done you'll find in the working directory two files: *Results.xml* and *Results.xsl*. The former is a xml report of the results where you'll find the highest and the lowest value of execution time,

bytes sent and bytes received for every operation, along with its average value and its standard deviation. This xml is associated to the *Results.xsl*, so you can open it with a browser to see the results in a more convenient view.

The *Results.xsl* will only be created if it doesn't exist in the working directory, so you can customize it in order to view the results as you like.

9. Logging

When you execute wscltest (whether it be to launch a test or to create template files) a file called "log" will appear at the working directory. As it's name suggest this is a log file. The default level of messages is INFO, so you can view the ERROR, WARNING and INFO messages. If you want to change the level you have to call wscltest with a new parameter: *-loglevel*, followed by the desired level. In the next example you are telling wscltest to log only messages of ERROR and WARNING levels:

```
wscltest -loglevel warning
```

A. Multiplatform

wscltest can run on whatever system that has a runtime implementation of Common Language Infrastructure. Mono Project (http://www.mono-project.com) provides an implementation for Linux and for Windows. Besides, in that latter platform you can run wscltest on .NET framework.