Grinder Webtest Documentation

Release 0.1

Automation Excellence

March 15, 2012

CONTENTS

You are reading the documentation for Grinder Webtest, a custom module for the Grinder load test framework designed to execute Visual Studio `.webtest` files.

CHAPTER

ONE

MOTIVATION

This module was developed in order to work around some perceived limitations of Grinder's built-in proxy-based HTTP recording. We found that in some cases, Fiddler was able to more correctly and accurately record the HTTP requests made by a web-enabled application. Since Fiddler is not designed for load testing, it became necessary to find an alternative method for running the scenarios it recorded.

Fiddler can export sessions in Visual Studio `.webtest` format, but Grinder has no native support for this. Hence, this module was developed, to allow transparent execution of these files as part of a load test. Since the `.webtest` format is plain-text XML, it's possible to extend it to allow parameterization, response capturing and verification, with a minimum of additional code.

CHAPTER

TWO

FEATURES

- Run an arbitrary number of tests, with logical grouping in test sets
- Global and local variable parameters
- Built-in and customizable macro functions
- Capturing and verifying response output using regular expressions
- Sequential, thread-based, random, or weighted test sequencing
- Automatic numbering of individual tests for logging and reporting purposes
- Correlating test runner that matches parameters in HTTP responses
- Four configurable levels of logging verbosity

CHAPTER

THREE

LICENSE

This software is open source, under the terms of the simplified BSD license.

CHAPTER FOUR

INSTALLATION

You can obtain Grinder Webtest by downloading a official release from the downloads page, then extracting it to a location on your disk. In later versions of Jython, you can install like this:

```
$ jython setup.py install
```

Alternatively, you can just build and run your tests directly in the source directory (it'll probably work better that way).

If you want a copy of the latest development version, clone it from Github:

```
$ git clone git://github.com/a-e/grinder-webtest.git
```

The only dependency aside from Grinder is Jython. Due to some limitations in earlier versions of Grinder, this module was designed to work with Jython 2.2.1. Grinder Webtest was developed using Grinder 3.2, and versions of Grinder as recent as 3.4 are known to be compatible with Jython 2.2.1. Grinder Webtest has not been tested with newer versions of Jython; if you try a newer version, please let us know whether or not it works.

CHAPTER

FIVE

SETUP

In the root directory of the Grinder Webtest branch, you will find an example `grinder_webtest.py` script, along with a `grinder.properties` that uses it. Refer to the Grinder documentation for more on how to use `grinder.properties`, and how to run test scripts.

You can use `grinder_webtest.py` as the template for your test script; all you need to do is include one or more `TestSets` containing `.webtest` files:

```
test_sets = [
    TestSet('test1.webtest'),
    TestSet('test2.webtest'),
]
```

Then create a `TestRunner` class:

```
TestRunner = get_test_runner(test_sets)
```

For the simplest tests, this is all you need to know. For more detail, refer to the *webtest.runner* documentation.

CHAPTER

SIX

RUNNING

To simplify the task of starting the Grinder console and/or agent, a `start.py` script is provided. This script reads configuration settings from `conf.py`.

First, copy `conf_example.py` to `conf.py`, and define the appropriate pathnames for your environment. On Linux, it might look something like this:

```
paths = {
    'java':          '/usr/bin/java',
    'jython':        '/usr/local/share/jython',
    'grinder':       '/usr/share/grinder',
    'properties':    './grinder.properties',
}
```

Save the changes to `conf.py`. You can run the tests in a single agent process like so:

```
$ jython start.py agent
```

Or, if you would like to use the console, ensure your `grinder.properties` has `grinder.useConsole=true`, then run the console:

```
$ jython start.py console
```

Then start agents in separate terminals. Refer to the Grinder docs for more information about the console, agents, and properties.

Please report bugs and feature requests to the issues page.

CHAPTER

SEVEN

DEVELOPMENT

If you'd like to work on developing grinder-webtest, fork the project from Github and clone your fork. You may also want to set up a virtual environment for installing any Python dependencies (pythonbrew is nice), activate it, then do:

```
$ pip install -r requirements.txt
```

Some unit tests are included in the `tests` directory, which you can run using pytest like this:

```
$ py.test
```

This will also generate an HTML-formatted coverage report in `htmlcov`. The tests should work using regular Python, so you don't need to muck about with Jython for this.

If you develop any cool new features or fix any bugs, please submit a **'pull request'_**!

CHAPTER

EIGHT

MODULES

8.1 `webtest.runner`

This module provides a high-level function called `get_test_runner`, which creates a `WebtestRunner` class for executing Visual Studio `.webtest` files found in one or more `TestSet`s.

8.1.1 Test Sets and Test Runners

In order to execute a `.webtest` file, you must wrap it in a `TestSet`, then create a `WebtestRunner` class via the `get_test_runner` function. Here is a simple example:

```
from webtest.runner import TestSet, get_test_runner

my_tests = [TestSet('my_test.webtest')]
TestRunner = get_test_runner(my_tests)
```

Each `TestSet` in the list will be executed by a single `WebtestRunner` instance. If you have more than one `.webtest` file to execute, you can pass all the filenames to a single `TestSet`:

```
my_tests = [TestSet('test1.webtest', 'test2.webtest', 'test3.webtest')]
TestRunner = get_test_runner(my_tests)
```

The above example would run three tests in sequential order, in the same `WebtestRunner` instance. This is recommended if there are dependencies between the tests (either they must always run in a given sequence, or there are captured variables shared between them–more on this shortly).

Another way would be to create a separate `TestSet` instance for each test:

```
my_tests = [
    TestSet('test1.webtest'),
    TestSet('test2.webtest'),
    TestSet('test3.webtest'),
]
TestRunner = get_test_runner(my_tests)
```

Here, each `.webtest` could be run in a separate `WebtestRunner` instance, and not necessarily in sequential order. You might take this approach if all three tests are independent, and have no need of running in sequence or sharing variables.

The `TestSet` might also be used for logical grouping of related tests. For example, if you have some tests for invoicing, and others for billing, you might create your TestRunner like this:

```
my_tests = [
    TestSet('invoice_1.webtest', 'invoice_2.webtest'),
    TestSet('billing_1.webtest', 'billing_2.webtest'),
]
TestRunner = get_test_runner(my_tests)
```

Here again, the two invoice tests will be run in order, in the same `WebtestRunner` instance, and the two billing tests will also be run in order, in the same `WebtestRunner` instance.

This covers the essentials of using this module; the next sections deal with how to initialize and capture variable values, and how to control the execution sequence.

8.1.2 Parameterization and Capturing

There are two critically important things that are often needed in web application testing: parameterization of values, and capturing of responses. These are handled by the inclusion of variables. To use variables, you must first determine which parts of your `.webtest` file need to be parameterized. Typically, you will parameterize the `Value` attribute of some element, for instance:

```
<FormPostParameter Name="UID" Value="wapcaplet" />
<FormPostParameter Name="PWD" Value="W0nderPet$" />
```

To turn these into parameters, insert an `ALL_CAPS` name surrounded by curly braces in place of the `Value` attribute's value:

```
<FormPostParameter Name="UID" Value="{USERNAME}" />
<FormPostParameter Name="PWD" Value="{PASSWORD}" />
```

Then, define values for these variables when you call `get_test_runner`, by passing a dictionary using the `variables` keyword:

```
my_vars = {
    'USERNAME': 'wapcaplet',
    'PASSWORD': 'W0nderPet$',
}
TestRunner = get_test_runner(my_tests, variables=my_vars)
```

Variables do not have a particular type; all variables are treated as strings, and must be defined as such in the `variables` dictionary.

This is just one way to set variable values, and would normally be used to initialize "global" variables that are used throughout all of your `.webtest` files. You may also initialize "local" variables in a single `.webtest` file by simply assigning the `ALL_CAPS` variable name a literal value when it is first referenced:

```
<FormPostParameter Name="INVOICE_ID" Value="{INVOICE_ID = 12345}" />
```

Here, `INVOICE_ID` is set to the value `12345`; any later reference to `INVOICE_ID` in the same `.webtest` file (or in other `.webtest` files that come later in the same `TestSet`) will evaluate to `12345`. See the `WebtestRunner.eval_expressions` method below for details.

Variables can also be set to the result of a "macro"; this is useful if you need to refer to the current date (when the script runs), or for generating random alphanumeric values:

```
<FormPostParameter Name="INVOICE_DATE" Value="{TODAY = today(%y%m%d)}"/>
<FormPostParameter Name="INVOICE_ID" Value="{INVOICE_ID = random_digits(10)}"/>
```

See the `webtest.macro` module for details on using macros and defining custom macros.

Finally, and perhaps most importantly, if you need to set a variable's value from one of the HTTP responses in your `.webtest`, you can use a capture expression. For example, you may need to capture a session ID as it comes back from the login request, so that you can use it in subsequent requests. To do that, include a `<Capture>...</Capture>` element somewhere in the `<Request...>` tag. The `Capture` element can appear anywhere inside the `Request` element, though it makes the most chronological sense to put it at the end:

```
<Request Method="POST" Url="http://my.site/login" ...>
    ...
    <Capture>
        <![CDATA[{SESSION_ID = <sid>(.+)</sid>}]]>
    </Capture>
</Request>
```

This will look for `<sid>...</sid>` in the response body, and set the variable `SESSION_ID` equal to its contents. You capture an arbitrary number of variable values in this way, then refer to them later in the `.webtest` file (or in subsequent `.webtest` files in the same `TestSet`). See the `WebtestRunner.eval_capture` method below for additional details.

8.1.3 Sequencing

When calling `get_test_runner`, you can use the `sequence` keyword to control how the tests are executed. Using the same example as above:

```
my_tests = [
    TestSet('invoice_1.webtest', 'invoice_2.webtest'),
    TestSet('billing_1.webtest', 'billing_2.webtest'),
]
```

The default behavior is for each `WebtestRunner` instance (each Grinder worker thread, that is) to run all of the tests in sequential order. This is the same as passing `sequence='sequential'`:

```
TestRunner = get_test_runner(my_tests, sequence='sequential')
```

and would give this runtime behavior:

- Thread 0: invoice, billing
- Thread 1: invoice, billing
- ...

But perhaps you want the first thread to run the invoice tests, and the second thread to run the billing tests. To do this, pass `sequence='thread'` to `get_test_runner`:

```
TestRunner = get_test_runner(my_tests, sequence='thread')
```

Now, if you run two threads, then the first thread will run the first `TestSet`, and the second thread will run the second `TestSet`. If you have more than two threads, then the extra threads will cycle through the list of available `TestSet`s again:

- Thread 0: invoice
- Thread 1: billing
- Thread 2: invoice
- Thread 3: billing
- ...

Another option is to use random sequencing, so that each thread will choose a random `TestSet` to run:

```
TestRunner = get_test_runner(my_tests, sequence='random')
```

With this, you might end up with something like:

- Thread 0: billing
- Thread 1: billing
- Thread 2: invoice
- Thread 3: billing
- Thread 4: invoice
- ...

Finally, it's possible to assign a "weight" to each `TestSet`; this is similar to random sequencing, except that it allows you to control how often each each `TestSet` is run in relation to the others. For example, if you would like to run the billing tests three times as often as the invoice tests:

```
my_tests = [
    TestSet('invoice_1.webtest', 'invoice_2.webtest', weight=1),
    TestSet('billing_1.webtest', 'billing_2.webtest', weight=3),
]
TestRunner = get_test_runner(my_tests, sequence='weighted')
```

The `weight` of each `TestSet` can be any numeric value; you might use integers to obtain a relative frequency, like the example above, or you might use floating-point values to define a percentage. Here's the above example using percentages:

```
my_tests = [
    TestSet('invoice_1.webtest', 'invoice_2.webtest', weight=0.25),
    TestSet('billing_1.webtest', 'billing_2.webtest', weight=0.75),
]
TestRunner = get_test_runner(my_tests, sequence='weighted')
```

In other words, the invoice tests will be run 25% of the time, and the billing tests 75% of the time. In this case, you might end up with the following:

- Thread 0: billing
- Thread 1: billing
- Thread 2: invoice
- Thread 3: billing
- ...

As with random sequencing, each thread will choose a `TestSet` at random, with the likelihood of choosing a particular `TestSet` being determined by the weight. This allows you to more closely mimic a real-world distribution of activity among your various test scenarios.

8.1.4 Before and After

If you have test steps that must be run at the beginning of testing (such as logging into the application), and/or steps that must be run at the end (such as logging out), you can encapsulate those in `TestSet`s and pass them using the `before_set` and `after_set` keywords. For example:

```
login = TestSet('login.webtest')
logout = TestSet('logout.webtest')
TestRunner = get_test_runner(my_tests, before_set=login, after_set=logout)
```

The `before_set` will be run when a `WebtestRunner` instance is created (normally when the Grinder worker thread starts up), and the `after_set` will be run when the instance is destroyed (the thread finishes execution, or is interrupted).

8.1.5 Classes and functions

Below is the API documentation of the classes and functions defined in this module.

`webtest.runner.`**`get_test_runner`**(*test_sets*, *variables={}*, *before_set=None*, *after_set=None*, *sequence='sequential'*, *think_time=500*, *scenario_think_time=500*, *verbosity='quiet'*, *macro_class=None*)

Return a `TestRunner` base class that runs `.webtest` files in the given list of `TestSet`s. This is the primary wrapper for executing your tests.

`variables` Default variables for all `TestRunner` instances. Each `TestRunner` instance will get their own copy of these, but passing them here lets you define defaults for commonly-used variables like server name, username, or password.

`test_sets` A list of TestSets, where each `TestSet` contains one or more webtests that must be run sequentially. Each `TestSet` will run in a single TestRunner instance, ensuring that they can share variable values.

`before_set` A `TestSet` that should be run when the `TestRunner` is initialized. Use this if all webtests need to perform the same initial steps, such as logging in.

`after_set` A `TestSet` that should be run when the `TestRunner` is destroyed. Use this if all webtests need to perform the same final steps, such as logging out.

`sequence` How to run the given test sets. Allowed values:

'sequential' Each thread runs all TestSets in order.

'random' Each thread runs a random TestSet for each `__call__`.

'weighted' Each thread runs a random TestSet, with those having a larger `weight` being run more often.

'thread' Thread 0 runs the first TestSet, Thread 1 runs the next, and so on. If there are fewer threads than TestSets, some TestSets will not be run. If there are more threads than TestSets, the extra threads start over at 0 again.

`think_time` Time in milliseconds to sleep between each request.

`scenario_think_time` Time in milliseconds to sleep between each scenario.

`verbosity` How chatty to be when logging. May be:

'debug' Everything, including response body

'info' Basic info, including request parameters and evaluated expressions

'quiet' Minimal info, including .webtest filename and test names

'error' Only log errors, nothing else

`macro_class` The class (not the instance) where macro functions are defined. If `None`, the `webtest.macro.Macro` class is used; pass a derived class if you want to define your own macros. See `webtest.macro` for how to define your own macros.

class `webtest.runner.`**`TestSet`**(**webtest_filenames*, ***kwargs*)

A collection of `.webtest` files that are executed sequentially, with an implied dependency between them.

webtest_filenames One or more `.webtest` XML filenames of tests to run together in this set

Optional keyword arguments:

weight A numeric indicator of how often to run this `TestSet`. Use this with the `sequence='weighted'` argument to `get_test_runner`.

class `webtest.runner.`**`WebtestRunner`**(***variables*)

A base class for `TestRunner` instances that will run `TestSet`s.

NOTE: This class is not meant to be instantiated or overridden directly–use the `get_test_runner` function instead.

classmethod **`set_class_attributes`**(*test_sets*, *before_set=None*, *after_set=None*, *sequence='sequential'*, *think_time=500*, *scenario_think_time=500*, *verbosity='quiet'*, *macro_class=None*)

Set attributes that affect all `WebtestRunner` instances.

See `get_test_runner` for what the parameters mean.

`eval_expressions`(*value*)

Parse the given string for variables or macros, and do any necessary variable assignment. Return the string with all expressions expanded.

Allowed expressions:

`{MY_VAR}` Expand to the current value of `MY_VAR`

`{MY_VAR = literal}` Assign `MY_VAR` a literal value, and expand to that value

`{macro_name(args)}` Expand to the result of `macro_name(args)`

`{MY_VAR = macro_name(args)}` Assign `MY_VAR` the result of calling `macro_name(args)`, and also expand to the resulting value. See the `webtest.macro` module for more information.

Any given value that does not match any of these forms is simply returned as-is. If you need to use literal `{` or `}` characters in a string, precede them with a backslash, like `\{` or `\}`.

The given value may contain multiple `{...}` expressions, and may have additional text before or after any expression. For example, if you have previously assigned two variables `FOO` and `BAR`:

```
>>> eval_expressions('{FOO = Hello}')
'Hello'
>>> eval_expressions('{BAR = world}')
'world'
```

you can combine them in an expression like this:

```
>>> eval_expressions('{FOO} {BAR}!')
'Hello world!'
```

The only caveat is that a `{...}` expression may not contain another `{...}` expression inside it.

`eval_capture`(*request*, *response*)

Evaluate any `Capture` expressions in the given request, and set variables to matching text in the given response. Return the number of capture expressions that were successfully evaluated.

In order for this to work, you should include a `Capture` element inside the `Request` element whose response you want to capture. Each expression inside `Capture` should follow this format:

```
{VAR_NAME = regexp}
```

Where `regexp` is a regular expression with parentheses around the part you want to capture (leave out the parentheses to capture the entire match). For example, if your response contains:

```
... <a href="http://python.org"> ...
```

And you want to store the URL into a variable called `HREF`, do:

```
{HREF = <a href="([^"]+)">}
```

If any capture expression is not found in the response, a `CaptureFailed` is raised. This makes them useful for verification too–if you want to ensure that a response contains expected text, just include a capture expression that looks for it. In this case, you can leave out the parentheses, since you don't need to capture any particular part, and if you don't need to keep the value for later, you can just assign it to a dummy variable.

For example, to verify that the response includes a form with a "submit" button:

```
{VERIFY = <form.*>.*<input type="submit".*>.*</form>}
```

You can include multiple `{VAR_NAME = regexp}` expressions in the same `Capture` element, to capture several variables from the same response, or to do several verifications.

Since regular expressions often contain characters that are precious to XML, such as `< > &` and so on, you can enclose your capture expressions in a `CDATA` block to prevent them from being interpreted as XML:

```
<Request ...>
    ...
    <Capture>
        <![CDATA[
            {VAR_A = <a href="([^"]+)">}
            {VAR_B = <b>([^<]+)</b>)}
        ]]>
    </Capture>
</Request>
```

You can include `{VAR_NAME}` references in the right-hand side of the expression; for example, if you have previously assigned a value to `ORDER_NUMBER`, and you want to capture the contents of a `div` having that order number as its `id`, you could do:

```
{ORDER_DIV = <div id="{ORDER_NUMBER}">(.*)</div>}
```

If `ORDER_NUMBER` was previously set to `12345`, then the above will be expanded to:

```
{ORDER_DIV = <div id="12345">(.*)</div>}
```

before matching in the response body.

8.2 `webtest.macro`

This module defines macro functions that can be invoked from a webtest. Macros provide a way of programmatically generating things like timestamps, random strings or numbers, or other strings that cannot be hard-coded.

Macro functions are defined as methods within the `Macro` class. Any methods defined in this module are built-in, so you can invoke them from any evaluated expression in a `.webtest` file.

All macro functions are invoked using a `lower_case` name followed by parentheses. If the macro accepts arguments, they are passed as a literal string, separated by commas. For example:

```
<FormPostParameter Name="TIMESTAMP" Value="{timestamp()}"/>
<FormPostParameter Name="INVOICE_DATE" Value="{today(%y%m%d)}"/>
<FormPostParameter Name="DUE_DATE" Value="{today_plus(7, %y%m%d)}"/>
```

You can also assign the return value of a macro to a variable, for later use:

```
<FormPostParameter Name="TIMESTAMP" Value="{NOW = timestamp()}"/>
<FormPostParameter Name="INVOICE_DATE" Value="{TODAY = today(%y%m%d)}"/>
<FormPostParameter Name="DUE_DATE" Value="{NEXT_WEEK= today_plus(7, %y%m%d)}"/>
```

If you want to define your own custom macros, create a derived class containing your custom methods:

```python
from webtest.macro import Macro

class MyMacro (Macro):
    # Zero-argument macro
    def pi(self):
        return '3.14159'

    # One-argument macro
    def square(self, num):
        return int(num) ** 2

    # Two-argument macro
    def multiply(self, x, y):
        return int(x) * int(y)
```

Then, tell your test runner to use your macro class:

```
TestRunner = get_test_runner( ... , macro_class=MyMacro)
```

Any `.webtest` files that are executed by this TestRunner will be able to call your custom macro methods, optionally storing their results in variables:

```
<FormPostParameter Name="NUM1" Value="{PI = pi()}" />
<FormPostParameter Name="NUM2" Value="{square(5)}" />
<FormPostParameter Name="NUM3" Value="{PRODUCT = multiply(3, 6)}" />
```

All macro arguments are strings; if your arguments are intended to be numeric, you must convert them yourself. The return value will also be converted to a string; it's converted automatically when your macro is invoked, but you may want to convert it yourself to ensure you get exactly what you want.

8.2.1 Classes and functions

class `webtest.macro.`**`Macro`**

Functions that can be invoked from a webtest.

`invoke`(*macro_name*, *args*)

Invoke `macro_name`, passing `args`, and return the result. This is a helper method and should not be called as a macro.

`now`(*format='%Y-%m-%d %H:%M:%S'*)

Return the current date/time in the given format. For example, `%m%d%y` for `MMDDYY` format. See the datetime module documentation for allowed format strings. For example, `today(%y-%m-%d)` might return `2011-10-06`.

random_alphanumeric(*length*)
Generate a random alphanumeric string of the given length. For example, `random_alphanumeric(5)` might return `F31B9`.

random_digits(*length*)
Generate a random string of digits of the given length. For example, `random_digits(5)` might return `28571`.

random_letters(*length*)
Generate a random string of letters of the given length. For example, `random_letters(5)` might return `KPDLE`.

timestamp()
Return a timestamp (number of seconds since the epoch). For example, `timestamp()` might return `1317917454`.

today(*format='%Y-%m-%d %H:%M:%S'*)
Return the current date/time in the given format. For example, `%m%d%y` for `MMDDYY` format. See the datetime module documentation for allowed format strings. For example, `today(%y-%m-%d)` might return `2011-10-06`.

today_plus(*days*, *format='%Y-%m-%d'*)
Return today plus some number of days, in the given format. For example, `today_plus(9, %Y-%m-%d)` might return `2011-10-15`.

8.3 webtest.correlate

This module provides a subclass of `WebtestRunner` designed to aid you with finding parameter values in HTTP responses. It does this by attempting to match parameter names found in each request to the response body of any HTTP request that preceded it.

To use it, simply call `get_correlation_runner` in the same way that you call `get_test_runner`. When you run your main script, the log output will include additional information about which responses contain certain parameter names; this is useful in determining where you might be able to capture parameter values.

Note that the correlating test runner is much slower and more memory-intensive than the normal test runner (not to mention it can produce some gigantic log files). Use this only during development of your scripts, and never for an actual load test!

class `webtest.correlate.`**CorrelationRunner**(***variables*)
A WebtestRunner that correlates requests and responses.

correlate(*filename*, *request*)
Attempt to correlate parameters in the given request to any responses already received for the current webtest file.

run_test_set(*test_set*)
Overridden from WebtestRunner base class, to record the response for each request.

`webtest.correlate.`**get_correlation_runner**(*test_sets*, *before_set=None*, *after_set=None*, *sequence='sequential'*, *think_time=500*, *verbosity='debug'*, *variables={}*)
Return a `TestRunner` base class that runs `.webtest` files in the given list of `TestSet`s, and does correlation of request parameters with responses.

All arguments to this function have the same meaning as their counterparts in `get_test_runner`, with the possible exception of `verbosity`–the correlating runner is more verbose, printing certain output about found correlations regardless of the `verbosity` setting.

8.4 `webtest.parser`

Provides classes for parsing Visual Studio `.webtest` XML files.

The Visual Studio `.webtest` XML format does not seem to be documented anywhere, so all the elements and attributes used here are inferred from the `.webtest` files written by Fiddler.

This module defines three important classes:

`Webtest` Parses a `.webtest` file and gets a list of `Request` objects

`Request` Stores attributes and contents relevant to a single HTTP(S) request

`WebtestHandler` Used internally by the sax XML parser

The `Webtest` class is the one you're most likely to use directly. Simply provide the name of a `.webtest` XML file:

```
>>> my_test = Webtest('my_test.webtest')
```

If you provide a filename to the constructor, parsing is done automatically. Alternatively, you can delay parsing until later:

```
>>> my_test = Webtest()
...
>>> my_test.load('my_test.webtest')
```

After parsing, the `Webtest` object will contain a list of `Request` objects. You can print all requests in summarized form:

```
>>> print(my_test)
```

Or iterate over the requests and do something with them:

```
>>> for request in my_test.requests:
...     do_something(request)
```

The `.webtest` file is expected to have one or more `Request` elements, similar to the one below:

```
<Request Method="POST" Url="http://www.example.com/">
  <Headers>
    <Header Name="Content-Type" Value="text/plain" />
  </Headers>
  <FormPostHttpBody ContentType="text/plain">
    <FormPostParameter Name="username" Value="phil" />
    <FormPostParameter Name="session_id" Value="12345" />
  </FormPostHttpBody>
</Request>
```

Each `<Request>...</Request>` defines a single HTTP request to a particular URL, using a method of GET or POST. Headers are enclosed in a `<Header .../>` element, and any parameters are sent using `<FormPostParameter .../>`, both of which have `Name` and `Value` attributes.

Two additional elements are understood by this parser:

`<Description>...</Description>` A human-readable string that describes what the request does

`<Capture>...</Capture>` A block of expressions that may be used to capture or verify content in the body of the response for this request

This module is designed to be used with the `webtest.runner` module, which is specifically designed to work with the Grinder load test framework, but the parser defined here is not Grinder-specific, and can be used for more general-purpose parsing of `.webtest` files.

exception `webtest.parser.`**`MalformedXML`**

Raised when any malformed XML is encountered.

class `webtest.parser.`**`Request`**(*attrs*, *line_number=0*)

Store attributes pertaining to an HTTP Request, including:

url The full URL path of the request

headers A list of `(Name, Value)` for the request header

parameters A list of `(Name, Value)` for the request parameters

`add_header`(*attrs*)

Add a header `(Name, Value)` pair to the request.

attrs A `dict` including 'Name' and 'Value' items

If the 'Name' or 'Value' attributes are not defined, or if the 'Name' attribute is empty, nothing is added.

`add_parameter`(*attrs*)

Add a parameter `(Name, Value)` pair to the request.

attrs A `dict` including 'Name' and 'Value' items

If the 'Name' or 'Value' attributes are not defined, or if the 'Name' attribute is empty, nothing is added.

`captures`()

Return capture expressions as a list of strings.

Normally, `self.capture` will be a literal block of text as it appeared inside the `Capture` element; it may contain extra spaces and newlines. This method strips out the extra newlines and whitespace, and converts to a list for easy iteration over each capture expression.

class `webtest.parser.`**`Webtest`**(*filename='`*)

Webtest XML file parser.

`load`(*filename*)

Load and parse the given `.webtest` XML file, and store the list of requests found in it.

`parse`(*filename*)

Load and parse the given `.webtest` XML file, and store the list of requests found in it.

class `webtest.parser.`**`WebtestHandler`**

Content handler for xml.sax parser.

`characters`(*data*)

Called when character data is found inside an element.

`endElement`(*name*)

Called when a closing XML tag is found.

`setDocumentLocator`(*locator*)

Set the document locator, for tracking line numbers. This is set by the parser, and is used in `startElement`.

`startElement`(*name*, *attrs*)

Called when an opening XML tag is found. If any badly-formed XML is encountered, a `MalformedXML` exception is raised.

PYTHON MODULE INDEX