reference-suite.wt

File: `/home/volker/webtest/reference-suite.wt`
Date: 12. Oct 2011, 14:37

Contents

Webtest test suites are simple text files in UTF-8 (without BOM). Line endings may be \n or \r\n and you are stupid *and* ugly if you use the later. The line length should not exeed 4000 characters.
Lines "starting" with a # are comments and are ignored. The # need not be the very first character on a line: # is considered 'start of comment' iff it is the first non-whitespace charater on a line.
Individual test cases are introduced like this (test case name surrounded by lines of - signs):

```
  ----------------------------------
  The Name of the Test Case
  ----------------------------------
```

See below: Global is such a testcase. Unfortunately Global is as very special testcase. So please skip to the "General Structure of Test Cases" test case.
Notes on test names: Test names should be uniq in one suite and they should not contain commas (",") in the name. Rational: This allows to select individual tests from a suite during a webtest run with the -tests option.

1 Global

```
----------------------------------------------
Global
----------------------------------------------
```

Note: You should not try to understand the "Global" section before familiarizing yourself with general testcases as Global is *not* a testcase by itself.
If the very first test in a test suite is named "Global" than this test will not be run but serve as a template for all subsequent tests: Settings, Variables, Header fields and Response checks are inherited from global to each test. The test may overwrite them. Body and Tag checks cannot be overwritten (as the do not contain some uniq id to identify them): Body and Tag conditions/checks from global are just added to each test.
Global is also used to keep cookies, e.g. login cookies which should be present in subsequent tests. So Global acts like a cookie jar.

```
GET http://unused.for/global

RESPONSE

    Normly we expect to get 200. Overwrite in test if you need to test for e.g. 404
    Status-Code == 200

SETTINGS

    Any request taking longer than 45 seconds is considered an error.
    Max-Time  :=  45000

CONST

    The global BaseUrl of our test server.
    BaseUrl  :=  http://localhost:8080
```

1.1 General Structure of Test Cases

```
----------------------------------------------
General Structure of Test Cases
----------------------------------------------
```

The most basic and simple test case: See if a server answers.

After the test case header (——-\n<Name>\n——-) several sections describe various aspects of the test. A Section is intruduced by an all caps section name starting at the first character of a line. Individual setting in a setting are indented by (at least) one tab '\t'. You may not indent with spaces.
The first section is special: It names the request method and URL and must be the first section, cannot be omitted an has no sub-elements. All other section are optional and may occur in any order. The current test below contains just the RESPONSE section.
Method may be "GET" or "POST". The URL must be a valid, full qualified URL. https works like expected.

```
GET http://host.to.ping/path.html
```

The response section: In this section the various header fields of the response can be checked.

```
RESPONSE
```

Check that the server answered with 200 status code.

```
81      Status-Code  ==  200
```

1.2 A simple example

```
-----------------------------------------------
87  A simple example
-----------------------------------------------
```

Let's make a POST-request, send some parameters and check the result.

```
POST http://www.domain.org/path/feedback
```

```
PARAM
```

send three parameters (automatically encoded) in request

```
94      name    :=  John Doe
95      city    :=  London
96      comment :=  Cool stuff :-)
```

```
RESPONSE
```

Check that the server answered with 200 status code.

```
99      Status-Code  ==  200
```

```
BODY
```

Check that the body contains some text

```
102     Txt  ~=  "Thank you for your feedback John Doe"
```

2 Constructing a request

A request is constructed from the following parts in a test:

- The URL is taken form `GET` or `POST` section
- Parameters which are sent from the `PARAM` section
- Manually set header fields from the `HEADER` section
- Cookies from the `SEND-COOKIE` section

Cookies are so special and complicated that the have their own section. See below under "Sending Cookies".

2.1 Setting Request Header fields

```
-----------------------------------------------
Setting Request Header fields
-----------------------------------------------
```

You may specify any request header field in the HEADER section by just naming them and their value. Note: There is a special syntax for sending cookies: Refere to section Cookies.

```
GET http://host.to.ping/path.html
```
The header section: add special request header fields here.

```
HEADER
```

add the Accept-Language fields with given value.

```
Accept-Language  := de,fr,en
```

Quotes could be used around the value to include leading or trailing spaces, but request header fields normaly do not contain spaces at all.

2.2 Basic Authorization

```
-----------------------------------------------
Basic Authorization
-----------------------------------------------
```

To use basic authorization you can provide the Authorization header yourself, or you can use the special fake Basic-Authorization header.

```
GET http://host.to.ping/path.html
```

```
HEADER
```

This is a pure convenience feature. username and password are in cleartext and will be properly base64 encoded. No ***`Basic-Authorization`****" header will be sent, instead a correct* ***`Authorization`*** *header will be sent: Authorization: Basic=<base64 encoded user credentials>*

```
Basic-Authorization  :=  username:password
```

2.3 Sending Parameters and Files

```
-----------------------------------------------
Sending Parameters and Files
-----------------------------------------------
```

Arbitary parameters can be specified in the `PARAM` section. For `GET` requests they are appended to the given URL. `POST` requests are sent as `multipart/form-data` iff a file is uploaded and `application/x-www-form-urlencoded` if not. See below for forcing multipart posts.

```
POST http://my.blog/comment.html
```

```
PARAM
```

Parameter name and value are given like this.

```
date  :=  2010-03-04
```

As a parameter may have seveal values you need to quote values with spaces as a space is the delimiter for the different values.

```
name  :=  "Grigori Perelman"
text  :=  "A Proof of the Poincare Conjecture"
```

Sending multiple values of a parameter (e.g. checkbox)

```
177     status  :=  genius nerd
```

If one of the multiple values contains a space: Surround this *one with quotes*

```
181     categorie  :=  proof "hilbert problem" theorem
```

To send a file use the following syntax:

```
184     file  :=  @file:relative/path/to/document.pdf
```

Currently there is a bug: You may not send filenames with special characters (e.g. spaces)
Note: no space allowwed between '@file:' and the filename

2.4 Forcing multipart/form-data

```
------------------------------------------------
192 Forcing multipart/form-data
------------------------------------------------
```

To force the post to use "multipart/form-data" even if **no** file is uploaded: Use POST:mp as method.

```
POST:mp http://my.blog/comment.html

PARAM
199     name     :=  anonymous
200     comment  :=  Cool!
```

3 Testing the Response

There are several ways to test the response recieved:

- Header fields in the response are checked in the `RESPONSE` section
- Textual or binary search in the body in the `BODY` section
- Checking for tags in a html/xml body in the `TAG` section
- Checking cookies recieved in a Set-Cookie header in the `SET-COOKIE` section
- Validating (X)HTML and links (via Setting)

As cookies are complicated the have their own section (see below in CHecking recieved Cookies). Validation is triggered by a special setting (see below).

3.1 Checking Response Header Fields

```
------------------------------------------------
224 Checking Response Header Fields
------------------------------------------------
```

Recieved header fields can be accessed by their name, e.g. `Content-Type`. There are two special fields which can be checked allways, even if the server didn't include them in the response header: `Status-Code` and `Final-Url`. Status-Code is the numerical status code and Final-Url is the URL reached after doing all the redirects requested by the server.
There are several ways to test the recieved value.

```
GET http://host.to.ping/path.html

RESPONSE
```

Operator == is for real equality

```
Status-Code   ==  200
```

Operator ~= tests strings for "contains"

```
Content-Type  ~=  text/html
```

Operator /= tests for a regular expression matching the field value and is considered for expert use.

```
Strange-Field  /= (cat?|^dog?).+$
```

Operator _= tests for the field value starting with the given prefix

```
Other-Header  _= StartPrefix
```

Operator =_ tests for the field value ending with the given suffix

```
Something     =_ EndSuffix
```

For field which are numeric you may use <, <=, ==, >= or > with the usual meaning. If the field falue is not a numeric value the outcome is undefined.

```
Content-Length  > 500
```

For fields whose value is a RFC1123 date you may use also use <, <=, > and >=. (Only RFC1123 dates work, but others should not be used anyway...)

```
Expires  >=  Fri, 20 May 2011 12:59:19 GMT
```

To negate a condition: Prefix the whole *condition with a '!' character. Note: There is* no *!= operator. Use !field == val instead*

```
!App-Field ~= Illegal
```

To disalow the mere existence of a header field:

```
!Illegal-Header
```

Generally there is no need to quote field names or values to test against: field names do not contain whitespace or special characters and values are just the rest after the operator with leading and trailung whitespace trimmed. If you do need these leading or trailing whitespaces: Enclose the value with " marks:

```
Field-Name == "  spaces at begin and end are important for the test  "
```

3.2 Testing the Response Body

```
-----------------------------------------------
Testing the Response Body
-----------------------------------------------
```

There are three ways to test the content of the recieved body: Simple tests are specified in the BODY section, HTML tags can be checked in the TAG section: See checking HTML Tags below

```
GET http://host.to.ping/path.html

BODY
```

Txt is the whole text of the body

```
Txt  ==  Whole text of body
```

Attention: Only UTF-8 encoded bodys work well.

Bin is the whole text of the body as hexadecimal string

```
Bin  == 0daf23bcad873f94
```

Note: Syntax may change in the future

The same operators (without the numerical ones) like in the response section can be used with with Txt and Bin: ~=, _=, =_ and /= Test if boby contains "Hello World!"

```
302 Txt  ~=  Hello World
```

Use something like this to check if a binary file starts with the appropriate magic key (png magic key below)

```
306 Bin  _=  89504e470d0a1a0a
```

You can limit the range where the test applies by specifying start and end positions in the body: For Txt the positions are line numbers (starting from 0), for Bin the positions are byte numbers (again strting at 0). Negativ positions count backward, ommited positions mean start/end. (Like in python).

Line 4,5,6 or 7 contains text "Status: Fine"

```
315 Txt[3:7]  ~=  Status: Fine
```

Last line in body is "Success"

```
318 Txt[-1:]  ==  "Success"
```

Somewhere between byte 300 and 800 there is coffe

```
321 Bin[300:800] ~= caffebabe
```

3.3 Checking HTML Tags

```
------------------------------------------------
326 Checking HTML Tags
------------------------------------------------
```

HTML/XML tag/element checkings are placed in section TAG Syntax for the tags are like checktag. See documentation in tag.go. The syntax is a bit like

```
  tagSpec   :=  ['!'] [ numOp number ] { simpleTag | tagStructure }
  numOp     :=  { '<', '<=', '==', '>=', '>' }
  number    :=  "any number >= 0, e.g. 4 or 17"
  simpleTag :=  tagName [class...] [attribute...] [ { '==' | '=D='} content]
  tagName   :=  "the lower case tag name e.g. h2, div, iframe, ..."
  class     :=  [ '!' ] 'class='content
  attribute :=  [ '!' ] attrName'='content
  attrName  :=  "the lowercase attribute name, e.g. href, title, ..."
  content   :=  { '/' regexp '/' | pattern }
  regexp    :=  "a valid regular expression"
  pattern   :=  "a text pattern with * and ? as the usual wildcards"
  tagStruct :=  '[' '\n' moreIndnt { simpleTag | tagStructure } '\n' ']'
  moreIndnt :=  "more indentation by tabs/spc than previous/parent tagSpec."
```

```
GET http://host.to.ping/path.html
```

```
TAG
```

Check if any h2 tag with a class of 'home' and text content of 'Quality' is present. All other attributes are ignored.

```
349 h2 class=home == Quality
```

Fail if a h5 tag with content 'WRONG' is present

```
! h5 == WRONG
```

Whitespaces in text content is normalized: tabs and newlines are replaced by spaces, multiple spaces are collapsed to one and leading/trailing spaces are trimmed. I.e. the text content of

```
<p>  Hello   John Doe!
  Greetings!
</p>
```

is considered to be |Hello John Doe! Greetings!| and would be matched by

```
p == Hello John Doe! Greetings!
```

*but **not** by*

```
p == Hello    John Doe! Greetings!
```

Count occurences of this div tag with CSS class teaser: Must be exactly 3

```
=3 div class=teaser
```

span tags may not be present 2 times (0, 1, 3, .. or 17) is okay

```
!=2 span == xyz
```

Fail if there are more than 4 a-tags linking to /somewhere.html.

```
<5  a href=/somewhere.html
```

The rest of the numerical operators are <=, >= and > and work like expected. Negations are discuraged (but allowed): !<=, !>=, !< and !>

Tag structures (nested tags) are introduced by '[' and ended by ']' Each on a own line.

```
[
    div class=A
        div class=B
            ul
]
```

Negation '!' and counting operators may be placed before the [. Test if this div with span element occurs at least 5 times.

```
>4 [
        div class=X
            span == Test
   ]
```

Note: The following structure would match any teaser (the div) which contains as direct childs a h3- and a p-tag, regardless if there are other tags present: So

```
<div class="teaser>
  <h1>Super</h1>
  <h3>Freibier</h3>
  <span>Heute!</span>
  <p>Die ganze Woche Freibier fr alle</p>
</div>
```

Would match the following structure.

```
[
    div class=teaser
        h3 == Freibier
        p  == Die ganze Woche*
```

```
]
```

Checking for "deep" content: Consider the following html

```
<h2>  Hello<span>nice</span>World</h2>
```

The text content of the h2 is considered to be "Hello World" (note the trimming of spaces and the addition of a space between the two text nodes). To match the whole text content including nested tags use the "=D=" deep operator.

```
h2 =D= Hello nice World
```

3.4 Validating HTML and Links

```
------------------------------------------------
Validating HTML and Links
------------------------------------------------
```

If the response is a HTML or XHTML page you may validate the html and/or the links in the html. (X)HTML validation is done by connecting to the W3C Validator. If link checking is enabled, then all references in the html (link, a, and img tags) are requested and checked for a response code of 200 (maybe after redirecting).

```
GET http://www.domain.org/some/page.html
```

```
SETTING
```

Possible values are ***links****,* ***html*** *and* ***links+html****.*

```
Validate := links+html
```

4 Cookies

Cookies can be sent along with the request and recieved cookies (Set-Cookie) can be checked. Both have their own section.

4.1 Sending Cookies

```
------------------------------------------------
Sending Cookies
------------------------------------------------
```

A cookie is identified by the browser based on the trippel (name, domain, path). These three parts are given here separated by colons. Name is allways first, and you may omit the domain and/or the path: The path defaults to "/" and the domain to the domain of the request URL if this testcase.

```
GET http://www.some.url/home/user/xyz
```

```
SEND-COOKIE
```

Send cookie mySpecialCookie with value someValue along with the request. The domain set to www.some.url and path is set to /. This is important only if the original request results in a rederict to a different host or different path (where this cookie wouldn't be sent).

```
mySpecialCookie  :=  someValue
```

Full version of how to specifiy a cookie:

```
ownCookie:www.some.url:/home:secure  :=  theValue
```

This will send ownCookie (with value theValue) only to host www.some.url (and all subdomains like abc.www.some.url) and only to request with a path starting with /home and only to secure https request. Note: This cookie would not *be sent on the initial request given in the GET as this request isn't secure (it's http). But it* would *be sent e.g. to https://abc.some.url/home/login if the initial request redirects to this URL.*

You may omit ":secure" if sending to http is okay and you may omit domain and/or path as described above.

Wildcard domains work (as in the browsers)

```
484 ownCookie:.domain.org  :=  theValue
```

Please note:

- *You cannot declare a cookie secure without a domain and a path.*
- *You cannot set expiration (just dont send it!).*
- *You cannot declare HttpOnly cookies (it's implicit, we) handle only http.*

4.2 Checking recieved cookies

```
------------------------------------------------
494 Checking recieved cookies
------------------------------------------------
```

The complicated stuff: Test wether server requests to set or delete a cookie.

```
GET http://www.domain.org/some/path/might/redirect
```

```
SET-COOKIE
```

You may use the common operators ==, ~=, _=, =_, /= to check the value:

```
503 name:domain.org:/path   ==  value
```

As described above omiting domain and/or path is possible with the usefull defaults.

```
507 name             ==  value
508 name:domain.org  ==  value
509 name:/path       ==  value
```

Cookies sent/set by the server can contain additional flags. You may check for them as follows:

```
513 name:domain.org:/path:Secure   == true
514 name:domain.org:/path:HttpOnly ==  false
515 name:domain.org:/path:MaxAge   ==  0
516 name:domain.org:/path:Expires  ~=  Nov 2013
517 name:domain.org:/path:Expires   >  Mon, 02 Jan 2006 15:04:05 MST
```

You may neither omit domain nor path in this syntax.

You may test if the server requested to delete the cookie with the following syntax:

```
522 name:domain.org:/path:Delete == true
```

It checks if the servers request would delete the cookie reliable in common browsers. Such cookies would be deleted in the Global too if Keep-Cookies is true

```
SETTING
```

Recieved cookies can be stored in the SEND-COOKIE section of the Global test. (Usefull for login/session cookies).

```
Keep-Cookies  :=  1
```

5 Variables and repeating tests

Tests may contain "variables" which get substituted before execution. Tests may be repeated and there are two types of variables which take different values on each repetition.
There are two different ways to "repeat" a test: Setting "Repeat" or "Tries" to a number greater 0.

- "Repeating" a test n times means executing the the test n times and reporting a success only if all n individual tests succeed. This is usefull for iterating over sequence or random variables. See above for examples.
- "Trying" means trying at most n times. Pass imediately if one run succeeds and fail if all n rounds fail. This is usefull to wait for some background job to complete and check this regularely.

5.1 Repeate a test multiple times

```
-----------------------------------------------
Repeate a test multiple times
-----------------------------------------------
```

Repeating a test `n` time will make `n` request and check all test conditions `n` times: It will report `n` times as much pass/fail as if run just once.

```
GET http://www.domain.org/show_random_fortune_cookie

RESPONSE
    Response-Status == 200

SETTING
```

Repeat this test 10 times: Make 10 requests and check 10 times the response code.

```
    Repeat  :=  10
```

5.2 Wait for Background Job / Trying

```
-----------------------------------------------
Wait for Background Job / Trying
-----------------------------------------------
```

Trying a test works a bit like repeating a test. The difference: If all conditions pass on one repetition the test and all it's conditions are marked as pass and no more repetitions are performed. Note: Combining repetition and trying is possible, but the result is currently undefined (aka buggy): The test status is solely determined by the last repetition.

```
GET http://some.host/job/123/detail.html

BODY
    Txt ~= Job 123 finished.

SETTING
```

Wait two seconds after each test

```
    Sleep  :=  2000
```

Try at most 60 times: Fail if not finished after approx 2 minutes.

```
Tries  :=  60
```

5.3 Variable Subtitutions

```
-----------------------------------------------
Variable Subtitutions
-----------------------------------------------
```

There are three types of variable substitutions, all can be used like shell variables: const variables (just sounds strange), sequence variables and random variables. A variable can be asigned a value and used as part of the URL, part of the header, response, body, tag or parameter values. The usage is allways the same for all three types: Occurences of ${<varname>} are replaced by the value of the varibale <varname>. But there are three ways to set a value for a variable in the three sections:

- CONST variables just take a single value, there use is obvious.
- SEQ (sequence) variables and
- RAND (random) values,

The only reasonable use for sequence and random variables is in a repeated test: SEQ and RAND values take values of a given list of possible values. SEQ cycles through the list in the given order, whereas RAND picks one value by random for each round of the test.
Notes:

- Variable names consist of characters only (no numbers, no underscore _).
- The following variable names are reserved for future use: "GLOBALID", "RANDOM", all variables starting with "ENV" and "NOW" (see below)
- Pay attention if variables are substituted in tag content as this might generate a regexp: If x takes value "xyz/" and the tag spec is e.g. "p == /abc*${x}" it will result in "/abc*xyz/ which is considered a regexp.

Usage of a variable: ${BaseUrl} is replace by const value set bolow. resulting in URL beeing http://my.blog/entries/show

```
GET ${BaseUrl}/show
```

```
PARAM
    month  :=  ${Month}
    year :=  ${Year}
    user :=  user-${Name}
```

```
TAG
```

Variable substitution is done in the content part of tags only: All other elements do not allow variables.

```
    h2 == Hello ${Name}!
```

```
CONST
```

Set value of BaseUrl to http://my.blog/entries.

```
    BaseUrl  :=  http://my.blog/entries
```

```
SEQ
```

Month will be 1 on first run of test, 2 on second, and so on. Will restart beeing 1 on 13th run of test.

```
    Month  :=  1 2 3 4 5 6 7 8 9 10 11 12
```

```
RAND
```

Year is one of the given four selected randomly on each test run.

```
Year  :=  2004 2005 2006 2007
```

Values with spaces like 'Emil Tom' must be enclosed in quotes as usual. Inside double quotes you may use standard go string escapings to create special characters e.g. a FEMAL SIGN.

```
Name  :=  Anna "Emil Tom" "Gender: \u2640" "Berta \"The Fat\" Bomb"
```

Name would be one of: Anna Emil Tom Gender: Berta "The Fat" Bomb

```
SETTING
```

Repeat the test 7 times.

```
Repeat :=  7
```

5.4 Special Variables

```
-----------------------------------------------
Special Variables
-----------------------------------------------
```

The following variables are provided by the system and cannot be redefined:

- "NOW" (currently the only one).

```
GET http://some.url

RESPONSE
```

NOW is the current time formated as RFC1123 (that is "Mon, 02 Jan 2006 15:04:05 MST"

```
Date   ==   ${NOW}
```

Now can be increased/decreased by adding/subtracting timespans. Formating remains RFC1123

```
Expires >  ${NOW + 3days}
```

If you need a different time format: add your own fmt definition in Go's time format after a '|' charcter. Time will be in UTC!

```
Last-Modified  >=  ${NOW - 5 hours + 10 minutes | Mon Jan _2 15:04:05 2006}
```

Possible modifiers are "second", "minute", "hour", "day", "week", "month" and "year" (all lower case, plural accepted). Output is in UTC time format (to prevent bug in Go)

6 Special Tests

6.1 Pre- and Post-Tasks

```
-----------------------------------------------
Pre- and Post-Tasks
-----------------------------------------------
```

You may execute (and test their outcome) arbitary commands before and after running the test. These commands are executed really before and after which means only once befor repeating or trying tests. The commands are search in PATH but you may give full paths. Use double quotes " to group words into one argument. If the command cannot be run, it aborts abnormaly or returns anything but 0 it is considered an error. The test itself is not performed after a failed BEFORE condition.

```
GET http://some.url
```

```
BEFORE
```

Everything in the BEFORE section will be executed before doing the actual GET request. All commands are are executed in the current working directory. Return value checking works like described in the AFTER section below.

```
714     bash -c "mysql -batch -u admin -p admin < setupdb.sql"
```

```
AFTER
```

Everything in the AFTER section will be executed after doing the actual request and running the tests on the response. An AFTER condition (same for BEFORE conditions) is considered a failure if the return value is != 0.

```
721     /home/tester/bin/python check-state.py
722     bash -c "if [ -f threaddump.bin ]; then exit 1; else exit 0; fi"
```

6.2 Logfile Checking

```
-----------------------------------------------
727 Logfile Checking
-----------------------------------------------
```

Checking additions to one ore more logfiles can be checked in the section LOG:
Please note: If log rotation happens during the test, then the test is not accurate as it might miss a log record. Writing log files is typically done asyncronousely so we might miss entries. Please add some (long enough) sleep time to your tests if the internal grace period of 250 ms isn't long enough (e.g. for emails).
The format of a log file condition is:

```
  [!]  <path/to/logfile>  <op>  <pattern>
```

with <op> one of the following operators:

```
  ~=   contains
  /=   regexp match
  _=   line starts with
  =_   line ends with
  >    logfile did grow more than given number of bytes (unimplemented)
  <    logfile did not grow more (unimplemented)
```

The rest of the operators known from the RESPONSE section are undefined. Please note: No spaces in the path to the logfile. Before running any tests the length of the logfile is recorded. After running all the tests (that is after all repetitions and tries) the log file is openend and the new lines are matched against the conditions.

```
GET http://some.url

LOG
```

Make sure no ERROR is is reported in the log file

```
756     ! server/log/error.log ~= ERROR
```

Make sure the following warning is logged:

```
759     ../server/log/access.log /= .*WARNING.*Unauthorized access from.*
```

Make sure that a email is recieved

```
762     /var/mail/testuser _= Subject: Weekly Traffic Summary
```

Make sure this request does not fill the disk if repeated: Logfile must not grow less than 250 bytes. (Currently unimplemented)

```
767     server/log/error.log < 250
```

7 Miscelaneous

7.1 Settings

```
-----------------------------------------------
Settings
-----------------------------------------------
```

Various setting can be applied to each and every test in the SETTING section. Currently the following are implemented:

- Repeat
- Tries
- Sleep
- Max-Time
- Keep-Cookies
- Dump
- Abort
- Validate

```
GET http://host.to.ping/path.html

SETTING
```

Number of repetitions of the test. Set to 0 to "disable" this test.

```
Repeat  :=  12
```

Number of tries this test is executed at most. The test passes if one try succeeds (the rest of the possible tries are skipped) and fails if all tries fail.

```
Tries    := 5
```

Setting both Tries and Repeat to values > 1 is (currently) undefined.

Time in ms to sleep after test

```
Sleep   :=  250
```

Fail if answer is not recievd in less than 300 ms.

```
Max-Time   :=  300
```

Keep (store in Global) cookies set by the server answer. See below. Use 0 to turn storage off.

```
Keep-Cookies  :=  1
```

Dump (see below in Debugging)

```
Dump   :=  0
```

Abort test suite and fail immediately if this tests fails. Usefull to skip test which cannot be tested because some setup task failed.

```
Abort  :=  1
```

Check the recieved html. See below in Validating) Possible values are ***links****,* ***html*** *and* ***links+html****.*

```
Validate := links+html
```

7.2 Debuging

```
-----------------------------------------------
Debuging
```

Setting Dump to 1 will dump the whole request/response talk to a .dump-file. The filename is constructed from the test name. Setting Dump to 3 will save the response body to a file.

```
GET http://some.url
```

```
SETTING
```

Turn dumping on with 1 or 2. Will dump to file "Debuging.dump" 1 will create a new file wheras 2 will append to an existing one. 1 and 2 will dump the whole wiretalk of request and response while 3 will just dump the recieved response body (and create a new file each time)

```
840         Dump  :=  1
```