htt User's Guide

by Marcel Schoen, Switzerland

January 23, 2011

Contents

Chapter 1

htt Functionality Overview

- The htt provides a large variety of HTTP-related functionality, useful for implementing all kinds of HTTP-based tests:
- Advanced HTTP protocol handling, including fine-grained timeout handling, request and response validation
- Simulating clients and servers, including startup and shutdown of server daemons. This allows to create mock-ups of back-end systems in more complex test situations, for example when the tested application needs to interact with a 3rd-party back-end system which is not available in the testing environment.
- Execution of external commandline tools, using their output as request or response data, or for validation purposes.
- Copying stream data (e.g. from a response) and re-using it in variables.

Chapter 2

Quickstart

This short chapter shows some simple, basic htt examples without explaining everything. Some parts should be obvious for anyone who has a solid knowledge of HTTP, everything else will be explained in more detail in the other chapters.

NOTE: Depending on the version and platform you are using `httest` on, there may be a bug which requires that every script ends with an empty line, below the last statement or variable definition!

2.1 Step 1: GET www.wikipedia.org

This first example sends a simple GET request to ”www.wikipedia.org” and checks that the page contains the word ”Wikipedia”, and the HTTP response code is 200.

Enter the following example code into a file named ”wikipedia.htt”:

```
CLIENT
_REQ www.wikipedia.org 80
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
END
```

Then execute the newly created script using the `httest` binary:

```
$ httest wikipedia.htt
```

And, given that the system you are running **httest** on has internet access, the following result should be displayed:

```
run wikipedia.htt
CLT0-0 start ...
_REQ www.wikipedia.org 80
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
>GET / HTTP/1.1
>Host: www.wikipedia.org
>
<HTTP/1.0 200 OK
<Date: Tue, 18 Jan 2011 12:38:51 GMT
...
...many more headers, and the HTML page...
...
<</body>
<</html> OK
```

Some explanations concerning the output:

- The first part (everything before the lines with the ">" prefix) shows the commands executed by the httest binary.
- All lines beginning with ">" show the data sent to the HTTP server.
- All lines beginning with "<" show the data received in the HTTP response.
- The "OK" at the end of the output signals a successful test completion.

2.2 Step 2: Externalize host and port values

Create a new file named ”values.htb” in the same directory like the ”wikipedia.htt” file. Insert these contents:

```
# Set host and port as variables
SET PORT=80
SET HOST=www.wikipedia.org
```

Then change the contents of the ”wikipedia.htt” file like this:

```
INCLUDE values.htb

CLIENT
_REQ $HOST $PORT
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
END
```

Now you can use that include file to use the same host and port value in multiple htt scripts, while being able to maintain those values in only one file.

Chapter 3

htt Commandline Interface

3.1 Files

The htt tool is available for multiple platforms. On Unix platforms, the htt provides two binaries:

- `httest`: The actual HTTP testing tool.
- `htntlm`: A tool required to perform an NTLM authentication.

On Windows, a number of other files (.exe and .dll) are required to run the htt.

3.2 Syntax

The htt binary uses the following commandline syntax:

```
    $ httest [OPTIONS] scripts
```

Where *scripts* is just one or multiple ".htt"-files, for example:

```
    $ httest -Ts testone.htt testtwo.htt
```

File names must be specified either relative to the current working directory, or with an absolute path.

3.3 Options

A list of all options can always be retrieved by using the ”–help” argument:

```
$ httest -Ts testone.htt testtwo.htt

httest is a script based tool for testing and benchmarking web
applications, web servers, proxy servers and web browsers. httest
can emulate clients and servers in the same test script, very
useful for testing proxys.

Usage: httest [OPTIONS] scripts

Options:
  -V --version           Print version number and exit
  -h --help              Display usage information (this message)
  -n --suppress          do no print start and OK|FAILED
  -s --silent            silent mode
  -e --error             log level error
  -w --warn              log level warn
  -d --debug             log level debug
  -L --list-commands     List all available script commands
  -C --help-command      Print help for specific command
  -T --timestamp         Time stamp on every run
  -S --shell             Shell mode

Examples:
httest script.htt

httest -Ts script.htt

Report bugs to http://sourceforge.net/projects/htt
```

Chapter 4

htt Script Structure

A htt test usually consists of a ".htt" file (the main script), and optionally some include files (usually with the suffix ".htb").

The main script consists of *include statements*, global or local *commands*, *variables*, *bodies* and comments:

- *Command*: An instruction to perform an action, like sending a request, setting a header in that request, evaluating the response etc. There are global and local commands (explained in the chapter "Commands").
- *Include*: A global command used to include re-usable scripting code from other files.
- *Variables*: Holder for values created during the execution of the script.
- *Body*: A block of code containing a number of commands.
- *Comment*: A line of non-executable text

Commands, variables and body types are explained in more detail in the corresponding chapters.

4.0.1 Comments

Comment lines must begin with the "#" character.

4.1 Structure Example

A simple htt script code example for showing the structure:

```
# Include some macros
INCLUDE macros.htb

# The actual test client part
CLIENT
   ...
END

# Optional: a re-used code block
BLOCK DOIT
   ...
END

# Optional: A test monitoring daemon
DAEMON
   ...
END
```

In the example above, `INCLUDE` is a global statement used to include another script file with some re-usable code blocks. It is followed by the `CLIENT` block which starts the actual HTTP test client, a block of re-usable functionality named ”DOIT” and a `DAEMON` body which contains some test monitoring functionality.

Chapter 5

htt Variables

Within a htt script, variables can store values read from files or created during the execution of the script.

5.1 Variable Names

A variable has a name and a value. Variable names are NOT case-sensitive, so a variable named ”VarA” can be used by refering to the name ”VARA” as well.

```
CLIENT
# Set variable with value "First"
_SET VarA=First

# This will overwrite the value of the
# variable "VarA", since it is internally
# the same variable as "VARA".
_SET VARA=Second
END
```

5.2 Variable Scope

A variable is defined either as a global variable, or as a local within a body (like `CLIENT`, `SERVER` etc.). Note that unlike as in common programming languages, local variables created in a `BLOCK` are also available in the invoking bodies.

So, if a `CLIENT` body invokes a `BLOCK` where a local variable ”A” is created, that variable will also become available in the invoking `CLIENT` body.

5.3 Setting A Variable

The syntax for setting a global variable is:

```
SET name=value
```

The syntax for setting a local variable in a body is:

```
_SET name=value
```

Examples:

```
SET MYVAR=SomeValue

CLIENT
_SET MYLOCAL=LocalValue
# Invoke block DOIT, makes variable "DoLocal"
# available in this CLIENT body as well.
_CALL DOIT
END

BLOCK DOIT
_SET DoLocal=anotherValue
END
```

5.4 Using Variables

In order to resolve a variable and use the value it is holding, its name must be prefixed with a ”$” character, just like in a shell script. Example:

```
# Set global Variables for host and port
SET MyHost=www.acme.com
SET MyPort=8080

CLIENT
# Invoke named block DOIT which
# sets the variable for the URI path
_CALL DOIT

_REQ $MyHost $MyPort
__GET $UriPath HTTP/1.1
__Host: www.acme.com
__
_WAIT
END

BLOCK DOIT
# Set URI path variable which will also
# become available in the invoking body.
_SET UriPath=/webapp
END
```

Done.

Chapter 6

Body Types

The following body types exist:

- `CLIENT` - The main HTTP test client logic.
- `SERVER` - Allows to start a HTTP server which will react and respond as defined in the given block. This allows to simulate HTTP back-end servers (like some application server) that may not be available in the testing environment.
- `BLOCK` - A block is a named section of the script with a defined begin and ending, not unlike functions or methods in common scripting or programming languages.
  - `BLOCK FINALLY` - A special block. The FINALLY block is always executed at the end of the test, and can be used for clean-up work.
  - `BLOCK ON_ERROR` - Another special block. A global error handler that will be executed when an error occurs in a test script. While some errors can be handled within a regular body (e.g. during the evaluation of a HTTP response), other errors (especially technical ones) sometimes can only be handled in this block.
- `DAEMON` - Other than a `SERVER`, a `DAEMON` does not implement any kind of listener. It is just a separate thread that can be used to monitor or control the execution of the test, e.g. implement timeout functionality to interrupt a hanging test etc.

6.1 Bodies And Threads

6.1.1 Parallel Execution

Note that all body blocks are executed in separate threads! So, if a htt script contains more than one `CLIENT` body, each one will be started in a separate thread, allowing to simulate two distinct, separate HTTP clients in one test case.

6.1.2 Execution Order

While multiple bodies are executed in parallel threads in general, `SERVER` and `DAEMON` bodies are always started before `CLIENT` bodies.

6.1.3 Lifespan

If a `CLIENT` body (or all of them, if there's more than one) ends, the test ends, and the htt process exits. The completion of `SERVER` and `DAEMON` bodies, on the other hand, will not automatically terminate the test.

6.1.4 Synchronization

There is no means of synchronization between those threads as of now.

6.2 CLIENT

A CLIENT body is the part of a htt script, where the actual HTTP requests are generated to perform some kind of test. Usually, there is just one single CLIENT body in a htt script, but there can be multiple ones; in that case, they are executed in parallel threads.

A htt test ends, when all CLIENT bodies complete their work.

A simple CLIENT example:

```
CLIENT
_REQ www.wikipedia.org 80
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
END
```

6.3 SERVER

A `SERVER` body can start a HTTP listener and react to incoming HTTP requests with predefined responses. This allows to simulate application backend servers that may not be available in a test environment.

There can be multiple `SERVER` bodies; in that case, they are executed in parallel threads.

A htt test does not end when the `SERVER` bodies complete their work, only when the `CLIENT` bodies have finished.

A simple `SERVER` example:

```
# Simple HTTP listener which just responds with
# a HTTP 200 return code, and an empty body.
SERVER
_RES
_WAIT
__HTTP/1.1 200 OK
__Content-Length: AUTO
__Content-Type: text/html
--

END
```

6.4 DAEMON

A DAEMON body does not start a listener, and also not necessarily send any HTTP request. It's purpose is to perform monitoring tasks during a test. For instance, if a CLIENT body sends a HTTP request that may hang for a long time due to slow response time of the server, the DAEMON may enforce an exit of the htt test process after a certain time, essentially implementing a timeout function.

This is only one possibility. A DAEMON can also monitor log files written during the test, execute external binaries etc.

There can be multiple DAEMON bodies; in that case, they are executed in parallel threads.

A htt test does not end when the DAEMON bodies complete their work, only when the CLIENT bodies have finished.

A simple DAEMON example:

```
# Timeout daemon thread, which exits the
# test process with a failure return code,
# if it takes longer than 30 seconds.
DAEMON
_SLEEP 30000
_DEBUG Test duration too long
_EXIT FAILED
END
```

6.5 BLOCK

A BLOCK body is not executed by itself like CLIENT or SERVER bodies; it just serves as a holder for a fragment of re-usable script code, similar to a function or method in common programming languages.

Every BLOCK must have a name, in order to distinguish them from each other, and to be able to be invoked from another body.

Just like high-level programming language functions or methods, a BLOCK can be parameterized, in a way very similar to shell scripts. The name of the BLOCK, mentioned in the previous paragraph, is just the first parameter (and it's mandatory). Within the BLOCK body, all parameter values are available as variables with simple numeric names, like in a shell script:

- $0 - The name of the BLOCK
- $1 - The first parameter after the name
- $2 - The second parameter after the name
- $3 - The third parameter, and so on

A simple BLOCK example, without parameters:

```
BLOCK SENDREQ
_REQ www.acme.com 80
__GET / HTTP/1.1
__Host: www.acme.com
__
_WAIT
END
```

The following example shows a BLOCK with some parameters:

```
# Block takes 3 parameters:
# 1: hostname, like www.acme.com
# 2: port
# 3: requested URI path
BLOCK SENDREQ
_REQ $1 $2
__GET $3 HTTP/1.1
__Host: $1
__
_WAIT
END
```

Chapter 7

htt Commands

Commands are the actual instructions that make up the test script, such as sending a request to a HTTP server, evaluating the response etc. There are two different types of commands:

- *Global* commands always begin with a capital letter (e.g. ”`INCLUDE`”) and can be placed in the ”root” of the script, outside of any body (block).
- *Local* commands always begin with an underscore (e.g. ”`_GET`”) and can only be placed within a *body*.

Some commands take arguments to perform their operation. Usually, the syntax is

```
    COMMAND [argument] [argument] [argument] ...
```

There is a special exception, the double-underscore local command (see section ”Local Commands” for details) which does not take a blank character between the command and its argument.

7.1 Global Commands

Global commands are used to perform initialization work, clean-up steps, start and end bodies etc.

Global commands can only be used within the "root" part of the script, i.e. outside of a body. It is not possible to use a global command within a `CLIENT`, `SERVER`, `DAEMON` or `BLOCK` body.

Note that some commands can are available in two forms, both as global and local command.

The following global commands perform an operation by themselves:

- `SET` - Sets a global variable.
- `INCLUDE` - Includes content from a file.
- `GO` -
- `EXEC` -
- `TIMEOUT` -
- `AUTO_CLOSE` -
- `FILE` -

While the global commands listed below are used to start or end a body (a block of script code):

- `CLIENT` - Starts a `CLIENT` body.
- `SERVER` - Starts a `SERVER` body.
- `DAEMON` - Starts a `DAEMON` body.
- `END` - Ends the current body.

All these commands are explained in more detail in TODO-REF.

7.2 Local Commands

Local commands always begin with an underscore (”_”) character. There are a large number of local commands, and they are explained in more detail in TODO-REF.

Double Underscore Command

Please note that the double-underscore is also a local command! But contrary to all other commands, with the double-underscore there's no blank character between the command and it's parameters.

The reason for this somewhat peculiar design is that this command is used to insert a line of data into the output stream of a HTTP request. And a line always ends with two invisible characters for the line-break and line-feed. Now, in some cases it is necessary to calculate the exact length of a line in order to know the correct length of the HTTP request body; in that case, the length of a line displayed by a text editor can be used directly as the length of the actual line of HTTP data for body length calculations.

7.3 List Of Global Commands

7.3.1 AUTO_CLOSE

Syntax:

```
AUTO_CLOSE on|off
```

Purpose:

Closes the current connection automatically.

7.3.2 BLOCK

Syntax:

```
BLOCK <name>
```

Purpose:

Begins a body with scripting code that can be invoked from anywhere else, using the `CALL` or `_CALL` command.

The `BLOCK` must be closed using the `END` command.

7.3.3 CLIENT

Syntax:

```
CLIENT [<number of concurrent clients>]
```

Purpose:

Begin of a body which starts one or multiple threads running a test client for sending HTTP requests to a server, and evaluating the response.

The number of threads started depends on the value of the argument for the number of concurrent clients. If none is given, it defaults to 1 and only a single thread will be started. Otherwise, the same `CLIENT` body will be executed in the specified number of parallel threads.

The CLIENT must be closed using the END command.

7.3.4 DAEMON

Syntax:

```
DAEMON
```

Purpose:

Begins a body with scripting code that is executed in a separate thread, and can be used to perform tasks like monitoring the tests, enforcing timeouts etc.

The DAEMON must be closed using the END command.

7.3.5 END

Syntax:

```
END
```

Purpose:

Ends the current body (either a CLIENT, SERVER, DAEMON or BLOCK body).

7.3.6 EXEC

Syntax:

```
EXEC <shell command>
```

Purpose:

Executes the given command, e.g. an external shell script. Note: The execution will not join any of the CLIENT or SERVER threads (TODO: WAS GENAU HEISST DAS?)

7.3.7 FILE

Syntax:

```
FILE <name>
```

Purpose:

Creates a temporary file with the given name. TODO: WHAT FOR? EMPTY? HOW TO USE?

7.3.8 GO

Syntax:

```
GO
```

Purpose:

Starts all `CLIENT`, `SERVER` and `DAEMON` bodies that have been defined up to this point.

TODO: WANN IST DAS NOETIG? BEI EINEM EINFACHEN CLIENT JA NICHT.

7.3.9 INCLUDE

Syntax:

```
INCLUDE <include file>
```

Purpose:

Includes the given file into the current script code. The file name parameter is processed relative to the working directory of the current process, if it is not an absolute path.

7.3.10 SERVER

Syntax:

```
SERVER [<SSL>:]<addr_port> [<number of concurrent servers>]
```

Purpose:

Begin of a body which starts one or multiple threads running a test server for responding to HTTP requests in predefined ways.

The number of threads started depends on the value of the argument for the number of concurrent servers. If none is given, it defaults to 1 and only a single thread will be started. Otherwise, the same `SERVER` body will be executed in the specified number of parallel threads.

The `SERVER` must be closed using the `END` command.

Please see chapter TODO-REF for details about how to correctly set up SSL connections.

7.3.11 TIMEOUT

Syntax:

```
TIMEOUT <timeout in ms>
```

Purpose:

Sets the global TCP socket connection timeout.

7.4 List Of Local Commands

7.4.1 ¨(Double Underscore)

Syntax:

```
__<data>
```

Purpose:

Bla bla