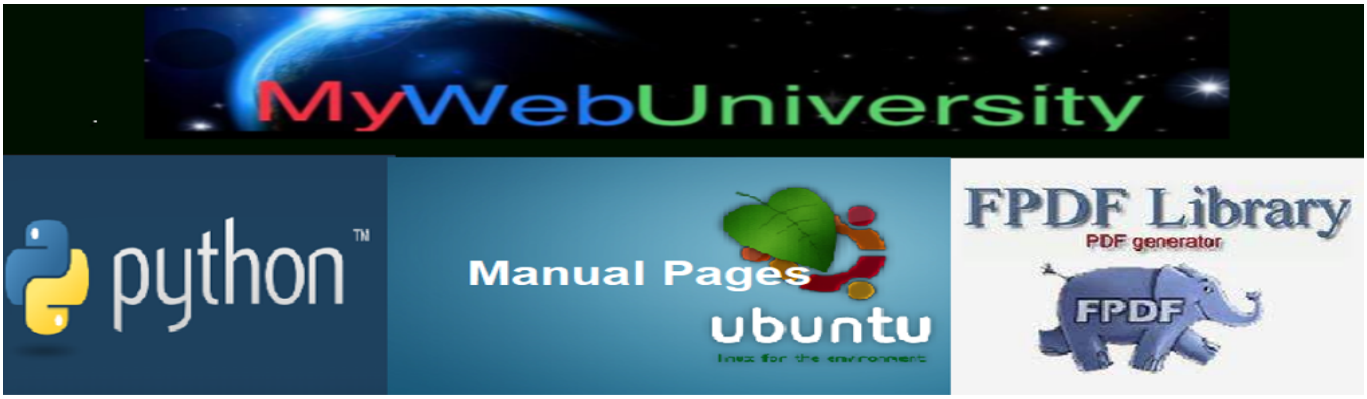

Rocky Enterprise Linux 9.2 Manual Pages on command 'Apache::TestUtil.3pm'

$ man Apache::TestUtil.3pm

Apache::TestUtil(3pm) User Contributed Perl Documentation Apache::TestUtil(3pm)

NAME

Apache::TestUtil - Utility functions for writing tests

SYNOPSIS

```
use Apache::Test;
use Apache::TestUtil;
ok t_cmp("foo", "foo", "sanity check");
t_write_file("filename", @content);
my $fh = t_open_file($filename);
t_mkdir("/foo/bar");
t_rmtree("/foo/bar");
t_is_equal($a, $b);
```

DESCRIPTION

"Apache::TestUtil" automatically exports a number of functions useful in writing tests.

All the files and directories created using the functions from this package will be automatically destroyed at the end of the program execution (via END block). You should not use these functions other than from within tests which should cleanup all the created directories and files at the end of the test.

FUNCTIONS

t_cmp()

```
t_cmp($received, $expected, $comment);
```

t_cmp() prints the values of $comment, $expected and $received. e.g.:

```
t_cmp(1, 1, "1 == 1?");
```

prints:

```
# testing : 1 == 1?
# expected: 1
# received: 1
```

then it returns the result of comparison of the $expected and the $received variables.

Usually, the return value of this function is fed directly to the ok() function, like this:

```
ok t_cmp(1, 1, "1 == 1?");
```

the third argument ($comment) is optional, mostly useful for telling what the comparison is trying to do.

It is valid to use "undef" as an expected value. Therefore:

```
my $foo;
t_cmp(undef, $foo, "undef == undef?");
```

will return a true value.

You can compare any two data-structures with t_cmp(). Just make sure that if you pass non-scalars, you have to pass their references. The datastructures can be deeply nested. For example you can compare:

```
t_cmp({1 => [2..3,{5..8}], 4 => [5..6]},
      {1 => [2..3,{5..8}], 4 => [5..6]},
      "hash of array of hashes");
```

You can also compare the second argument against the first as a regex. Use the "qr//" function in the second argument. For example:

```
t_cmp("abcd", qr/^abc/, "regex compare");
```

will do:

```
"abcd" =~ /^abc/;
```

This function is exported by default.

t_filepath_cmp()

This function is used to compare two filepaths via t_cmp(). For non-Win32, it simply uses t_cmp() for the comparison, but for Win32, Win32::GetLongPathName() is invoked to convert the first two arguments to their DOS long pathname. This is useful when there is a possibility the two paths being compared are not both represented by their long or short pathname.

This function is exported by default.

t_debug()

```
t_debug("testing feature foo");
t_debug("test", [1..3], 5, {a=>[1..5]});
```

t_debug() prints out any datastructure while prepending "#" at the beginning of each line, to make the debug printouts comply with "Test::Harness"'s requirements. This function should be always used for debug prints, since if in the future the debug printing will change (e.g. redirected into a file) your tests won't need to be changed.

the special global variable $Apache::TestUtil::DEBUG_OUTPUT can be used to redirect the output from t_debug() and related calls such as t_write_file(). for example, from a server-side test you would probably need to redirect it to STDERR:

```
sub handler {
  plan $r, tests => 1;
  local $Apache::TestUtil::DEBUG_OUTPUT = \*STDERR;
  t_write_file('/tmp/foo', 'bar');
  ...
}
```

left to its own devices, t_debug() will collide with the standard HTTP protocol during server-side tests, resulting in a situation both confusing difficult to debug. but STDOUT is left as the default, since you probably don't want debug output under normal circumstances unless running under verbose mode.

This function is exported by default.

t_write_test_lib()

```
t_write_test_lib($filename, @lines)
```

t_write_test_lib() creates a new file at $filename or overwrites the existing file with the content passed in @lines. The file is created in a temporary directory which is added to @INC at test configuration time. It is intended to be used for creating temporary packages for testing which can be modified at run time, see the Apache::Reload unit tests for an example.

t_write_file()

```
t_write_file($filename, @lines);
```

t_write_file() creates a new file at $filename or overwrites the existing file with the content passed in @lines. If only the $filename is passed, an empty file will be

created.

If parent directories of $filename don't exist they will be automagically created.

The generated file will be automatically deleted at the end of the program's execution.

This function is exported by default.

t_append_file()

```
t_append_file($filename, @lines);
```

t_append_file() is similar to t_write_file(), but it doesn't clobber existing files and appends @lines to the end of the file. If the file doesn't exist it will create it.

If parent directories of $filename don't exist they will be automagically created.

The generated file will be registered to be automatically deleted at the end of the program's execution, only if the file was created by t_append_file().

This function is exported by default.

t_write_shell_script()

```
Apache::TestUtil::t_write_shell_script($filename, @lines);
```

Similar to t_write_file() but creates a portable shell/batch script. The created filename is constructed from $filename and an appropriate extension automatically selected according to the platform the code is running under.

It returns the extension of the created file.

t_write_perl_script()

```
Apache::TestUtil::t_write_perl_script($filename, @lines);
```

Similar to t_write_file() but creates a executable Perl script with correctly set shebang line.

t_open_file()

```
my $fh = t_open_file($filename);
```

t_open_file() opens a file $filename for writing and returns the file handle to the opened file.

If parent directories of $filename don't exist they will be automagically created.

The generated file will be automatically deleted at the end of the program's execution.

This function is exported by default.

t_mkdir()

```
  t_mkdir($dirname);
```

t_mkdir() creates a directory $dirname. The operation will fail if the parent directory doesn't exist.

If parent directories of $dirname don't exist they will be automagically created.

The generated directory will be automatically deleted at the end of the program's execution.

This function is exported by default.

t_rmtree()

```
  t_rmtree(@dirs);
```

t_rmtree() deletes the whole directories trees passed in @dirs.

This function is exported by default.

t_chown()

```
  Apache::TestUtil::t_chown($file);
```

Change ownership of $file to the test's User/Group. This function is noop on platforms where chown(2) is unsupported (e.g. Win32).

t_is_equal()

```
  t_is_equal($a, $b);
```

t_is_equal() compares any two datastructures and returns 1 if they are exactly the same, otherwise 0. The datastructures can be nested hashes, arrays, scalars, undefs or a combination of any of these. See t_cmp() for an example.

If $b is a regex reference, the regex comparison "$a =~ $b" is performed. For example:

```
  t_is_equal($server_version, qr{^Apache});
```

If comparing non-scalars make sure to pass the references to the datastructures.

This function is exported by default.

t_server_log_error_is_expected()

If the handler's execution results in an error or a warning logged to the error_log file which is expected, it's a good idea to have a disclaimer printed before the error itself, so one can tell real problems with tests from expected errors. For example when testing how the package behaves under error conditions the error_log file might be loaded with errors, most of which are expected.

For example if a handler is about to generate a run-time error, this function can be used as:

```
  use Apache::TestUtil;
```

```
...
sub handler {
    my $r = shift;
    ...
    t_server_log_error_is_expected();
    die "failed because ...";
}
```

After running this handler the error_log file will include:

```
*** The following error entry is expected and harmless ***
[Tue Apr 01 14:00:21 2003] [error] failed because ...
```

When more than one entry is expected, an optional numerical argument, indicating how many entries to expect, can be passed. For example:

```
t_server_log_error_is_expected(2);
```

will generate:

```
*** The following 2 error entries are expected and harmless ***
```

If the error is generated at compile time, the logging must be done in the BEGIN block at the very beginning of the file:

```
BEGIN {
    use Apache::TestUtil;
    t_server_log_error_is_expected();
}
use DOES_NOT_exist;
```

After attempting to run this handler the error_log file will include:

```
*** The following error entry is expected and harmless ***
[Tue Apr 01 14:04:49 2003] [error] Can't locate "DOES_NOT_exist.pm"
in @INC (@INC contains: ...
```

Also see "t_server_log_warn_is_expected()" which is similar but used for warnings.

This function is exported by default.

t_server_log_warn_is_expected()

"t_server_log_warn_is_expected()" generates a disclaimer for expected warnings.

See the explanation for "t_server_log_error_is_expected()" for more details.

This function is exported by default.

t_client_log_error_is_expected()

"t_client_log_error_is_expected()" generates a disclaimer for expected errors. But in contrast to "t_server_log_error_is_expected()" called by the client side of the script.

See the explanation for "t_server_log_error_is_expected()" for more details.

For example the following client script fails to find the handler:

```
use Apache::Test;
use Apache::TestUtil;
use Apache::TestRequest qw(GET);
plan tests => 1;
t_client_log_error_is_expected();
my $url = "/error_document/cannot_be_found";
my $res = GET($url);
ok t_cmp(404, $res->code, "test 404");
```

After running this test the error_log file will include an entry similar to the following snippet:

```
*** The following error entry is expected and harmless ***
[Tue Apr 01 14:02:55 2003] [error] [client 127.0.0.1]
File does not exist: /tmp/test/t/htdocs/error
```

When more than one entry is expected, an optional numerical argument, indicating how many entries to expect, can be passed. For example:

```
t_client_log_error_is_expected(2);
```

will generate:

```
*** The following 2 error entries are expected and harmless ***
```

This function is exported by default.

t_client_log_warn_is_expected()

"t_client_log_warn_is_expected()" generates a disclaimer for expected warnings on the client side.

See the explanation for "t_client_log_error_is_expected()" for more details.

This function is exported by default.

t_catfile('a', 'b', 'c')

This function is essentially "File::Spec->catfile", but on Win32 will use "Win32::GetLongpathName()" to convert the result to a long path name (if the result is an absolute file). The function is not exported by default.

t_catfile_apache('a', 'b', 'c')

This function is essentially "File::Spec::Unix->catfile", but on Win32 will use "Win32::GetLongpathName()" to convert the result to a long path name (if the result is an absolute file). It is useful when comparing something to that returned by Apache, which uses a Unix-style specification with forward slashes for directory separators.

The function is not exported by default.

t_start_error_log_watch(), t_finish_error_log_watch()

This pair of functions provides an easy interface for checking the presence or absence of any particular message or messages in the httpd error_log that were generated by the httpd daemon as part of a test suite. It is likely, that you should proceed this with a call to one of the t_*_is_expected() functions.

```
t_start_error_log_watch();
do_it;
ok grep {...} t_finish_error_log_watch();
```

Another usage case could be a handler that emits some debugging messages to the error_log. Now, if this handler is called in a series of other test cases it can be hard to find the relevant messages manually. In such cases the following sequence in the test file may help:

```
t_start_error_log_watch();
GET '/this/or/that';
t_debug t_finish_error_log_watch();
```

t_start_file_watch()

```
Apache::TestUtil::t_start_file_watch('access_log');
```

This function is similar to "t_start_error_log_watch()" but allows for other files than "error_log" to be watched. It opens the given file and positions the file pointer at its end. Subsequent calls to "t_read_file_watch()" or "t_finish_file_watch()" will read lines that have been appended after this call.

A file name can be passed as parameter. If omitted or undefined the "error_log" is opened. Relative file name are evaluated relative to the directory containing "error_log".

If the specified file does not exist (yet) no error is returned. It is assumed that it will appear soon. In this case "t_{read,finish}_file_watch()" will open the file silently and read from the beginning.

t_read_file_watch(), t_finish_file_watch()

```
local $/ = "\n";
$line1=Apache::TestUtil::t_read_file_watch('access_log');
$line2=Apache::TestUtil::t_read_file_watch('access_log');
@lines=Apache::TestUtil::t_finish_file_watch('access_log');
```

This pair of functions reads the file opened by "t_start_error_log_watch()". As does the core "readline" function, they return one line if called in scalar context, otherwise all lines until end of file.

Before calling "readline" these functions do not set $/ as does "t_finish_error_log_watch". So, if the file has for example a fixed record length use this:

```
{
  local $/=\$record_length;
  @lines=t_finish_file_watch($name);
}
```

t_file_watch_for()

```
@lines=Apache::TestUtil::t_file_watch_for('access_log',
                                          qr/condition/,
                                          $timeout);
```

This function reads the file from the current position and looks for the first line that matches "qr/condition/". If no such line could be found until end of file the function pauses and retries until either such a line is found or the timeout (in seconds) is reached.

In scalar or void context only the matching line is returned. In list context all read lines are returned with the matching one in last position.

The function uses "\n" and end-of-line marker and waits for complete lines.

The timeout although it can be specified with sub-second precision is not very accurate. It is simply multiplied by 10. The result is used as a maximum loop count. For the intented purpose this should be good enough.

Use this function to check for logfile entries when you cannot be sure that they are already written when the test program reaches the point, for example to check for messages that are written in a PerlCleanupHandler or a PerlLogHandler.

```
ok t_file_watch_for 'access_log', qr/expected log entry/, 2;
```

This call reads the "access_log" and waits for maximum 2 seconds for the expected entry to appear.

AUTHOR

Stas Bekman <stas@stason.org>, Torsten F?rtsch <torsten.foertsch@gmx.net>

SEE ALSO

perl(1)