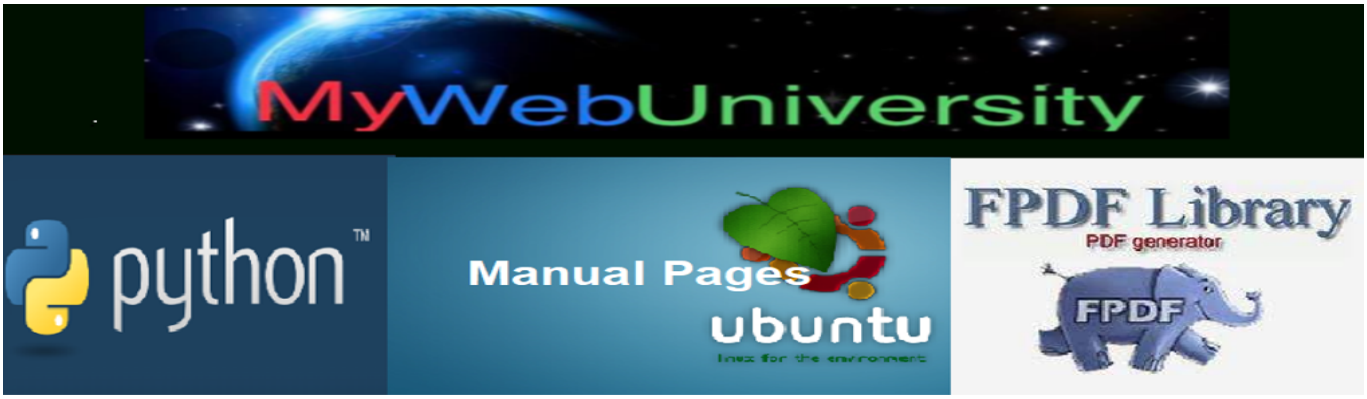

Rocky Enterprise Linux 9.2 Manual Pages on command 'Apache::Test.3pm'

$ man Apache::Test.3pm

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

NAME

Apache::Test - Test.pm wrapper with helpers for testing Apache

SYNOPSIS

use Apache::Test;

DESCRIPTION

Apache::Test is a wrapper around the standard "Test.pm" with helpers for testing an Apache server.

FUNCTIONS

plan

This function is a wrapper around "Test::plan":

plan tests => 3;

just like using Test.pm, plan 3 tests.

If the first argument is an object, such as an "Apache::RequestRec" object, "STDOUT" will be tied to it. The "Test.pm" global state will also be refreshed by calling "Apache::Test::test_pm_refresh". For example:

plan $r, tests => 7;

ties STDOUT to the request object $r.

If there is a last argument that doesn't belong to "Test::plan" (which expects a balanced hash), it's used to decide whether to continue with the test or to skip it all-together. This last argument can be:

? a "SCALAR"

the test is skipped if the scalar has a false value. For example:

plan tests => 5, 0;

But this won't hint the reason for skipping therefore it's better to use need():

plan tests => 5,
need 'LWP',
{ "not Win32" => sub { $^O eq 'MSWin32'} };

see "need()" for more info.

? an "ARRAY" reference

need_module() is called for each value in this array. The test is skipped if need_module() returns false (which happens when at least one C or Perl module from the list cannot be found).

Watch out for case insensitive file systems or duplicate modules with the same name. I.E. If you mean mod_env.c

need_module('mod_env.c') Not
need_module('env')

? a "CODE" reference

the tests will be skipped if the function returns a false value. For example:

plan tests => 5, need_lwp;

the test will be skipped if LWP is not available

All other arguments are passed through to Test::plan as is.

ok Same as Test::ok, see Test.pm documentation.

sok Allows one to skip a sub-test, controlled from the command line. The argument to sok() is a CODE reference or a BLOCK whose return value will be passed to ok(). By default behaves like ok(). If all sub-tests of the same test are written using sok(), and a test is executed as:

% ./t/TEST -v skip_subtest 1 3

only sub-tests 1 and 3 will be run, the rest will be skipped.

skip

Same as Test::skip, see Test.pm documentation.

test_pm_refresh

Normally called by Apache::Test::plan, this function will refresh the global state maintained by Test.pm, allowing "plan" and friends to be called more than once per-process. This function is not exported.

Functions that can be used as a last argument to the extended plan(). Note that for each

"need_*" function there is a "have_*" equivalent that performs the exact same function except that it is designed to be used outside of "plan()". "need_*" functions have the side effect of generating skip messages, if the test is skipped. "have_*" functions don't have this side effect. In other words, use "need_apache()" with "plan()" to decide whether a test will run, but "have_apache()" within test logic to adjust expectations based on older or newer server versions.

need_http11

    plan tests => 5, need_http11;

  Require HTTP/1.1 support.

need_ssl

    plan tests => 5, need_ssl;

  Require SSL support.

  Not exported by default.

need_lwp

    plan tests => 5, need_lwp;

  Require LWP support.

need_cgi

    plan tests => 5, need_cgi;

  Requires mod_cgi or mod_cgid to be installed.

need_cache_disk

    plan tests => 5, need_cache_disk

  Requires mod_cache_disk or mod_disk_cache to be installed.

need_php

    plan tests => 5, need_php;

  Requires a PHP module to be installed (version 4 or 5).

need_php4

    plan tests => 5, need_php4;

  Requires a PHP version 4 module to be installed.

need_imagemap

    plan tests => 5, need_imagemap;

  Requires a mod_imagemap or mod_imap be installed

need_apache

    plan tests => 5, need_apache 2;

Requires Apache 2nd generation httpd-2.x.xx

plan tests => 5, need_apache 1;

Requires Apache 1st generation (apache-1.3.xx)

See also "need_min_apache_version()".

need_min_apache_version

Used to require a minimum version of Apache.

For example:

plan tests => 5, need_min_apache_version("2.0.40");

requires Apache 2.0.40 or higher.

need_apache_version

Used to require a specific version of Apache.

For example:

plan tests => 5, need_apache_version("2.0.40");

requires Apache 2.0.40.

need_min_apache_fix

Used to require a particular micro version from corresponding minor release

For example:

plan tests => 5, need_min_apache_fix("2.0.40", "2.2.30", "2.4.18");

requires Apache 2.0.40 or higher.

need_apache_mpm

Used to require a specific Apache Multi-Processing Module.

For example:

plan tests => 5, need_apache_mpm('prefork');

requires the prefork MPM.

need_perl

plan tests => 5, need_perl 'iolayers';

plan tests => 5, need_perl 'ithreads';

Requires a perl extension to be present, or perl compiled with certain capabilities.

The first example tests whether "PerlIO" is available, the second whether:

$Config{useithread} eq 'define';

need_min_perl_version

Used to require a minimum version of Perl.

For example:

```
  plan tests => 5, need_min_perl_version("5.008001");
```

requires Perl 5.8.1 or higher.

need_fork

Requires the perl built-in function "fork" to be implemented.

need_module

```
  plan tests => 5, need_module 'CGI';
  plan tests => 5, need_module qw(CGI Find::File);
  plan tests => 5, need_module ['CGI', 'Find::File', 'cgid'];
```

Requires Apache C and Perl modules. The function accept a list of arguments or a reference to a list.

In case of C modules, depending on how the module name was passed it may pass through the following completions:

1 need_module 'proxy_http.c'

If there is the .c extension, the module name will be looked up as is, i.e. 'proxy_http.c'.

2 need_module 'mod_cgi'

The .c extension will be appended before the lookup, turning it into 'mod_cgi.c'.

3 need_module 'cgi'

The .c extension and mod_ prefix will be added before the lookup, turning it into 'mod_cgi.c'.

need_min_module_version

Used to require a minimum version of a module

For example:

```
  plan tests => 5, need_min_module_version(CGI => 2.81);
```

requires "CGI.pm" version 2.81 or higher.

Currently works only for perl modules.

need

```
  plan tests => 5,
      need 'LWP',
          { "perl >= 5.8.0 and w/ithreads is required" =>
            ($Config{useperlio} && $] >= 5.008) },
          { "not Win32"                => sub { $^O eq 'MSWin32' },
            "foo is disabled"          => \&is_foo_enabled,
```

```
    },
    'cgid';
```

need() is more generic function which can impose multiple requirements at once. All requirements must be satisfied.

need()'s argument is a list of things to test. The list can include scalars, which are passed to need_module(), and hash references. If hash references are used, the keys, are strings, containing a reason for a failure to satisfy this particular entry, the values are the condition, which are satisfaction if they return true. If the value is 0 or 1, it used to decide whether the requirements very satisfied, so you can mix special "need_*()" functions that return 0 or 1. For example:

```
  plan tests => 1, need 'Compress::Zlib', 'deflate',
      need_min_apache_version("2.0.49");
```

If the scalar value is a string, different from 0 or 1, it's passed to need_module().

If the value is a code reference, it gets executed at the time of check and its return value is used to check the condition. If the condition check fails, the provided (in a key) reason is used to tell user why the test was skipped.

In the presented example, we require the presence of the "LWP" Perl module, "mod_cgid", that we run under perl >= 5.7.3 on Win32.

It's possible to put more than one requirement into a single hash reference, but be careful that the keys will be different.

It's also important to mention to avoid using:

```
  plan tests => 1, requirement1 && requirement2;
```

technique. While test-wise that technique is equivalent to:

```
  plan tests => 1, need requirement1, requirement2;
```

since the test will be skipped, unless all the rules are satisfied, it's not equivalent for the end users. The second technique, deploying "need()" and a list of requirements, always runs all the requirement checks and reports all the missing requirements. In the case of the first technique, if the first requirement fails, the second is not run, and the missing requirement is not reported. So let's say all the requirements are missing Apache modules, and a user wants to satisfy all of these and run the test suite again. If all the unsatisfied requirements are reported at once, she will need to rebuild Apache once. If only one requirement is reported at a time, she will have to rebuild Apache as many times as there are elements in the "&&"

statement.

Also see plan().

under_construction

plan tests => 5, under_construction;

skip all tests, noting that the tests are under construction

skip_reason

plan tests => 5, skip_reason('my custom reason');

skip all tests. the reason you specify will be given at runtime. if no reason is given a default reason will be used.

Additional Configuration Variables

basic_config

my $basic_cfg = Apache::Test::basic_config();

$basic_cfg->write_perlscript($file, $content);

"basic_config()" is similar to "config()", but doesn't contain any httpd-specific information and should be used for operations that don't require any httpd-specific knowledge.

config

my $cfg = Apache::Test::config();

my $server_rev = $cfg->{server}->{rev};

...

"config()" gives an access to the configuration object.

vars

my $serverroot = Apache::Test::vars->{serverroot};

my $serverroot = Apache::Test::vars('serverroot');

my($top_dir, $t_dir) = Apache::Test::vars(qw(top_dir t_dir));

"vars()" gives an access to the configuration variables, otherwise accessible as:

$vars = Apache::Test::config()->{vars};

If no arguments are passed, the reference to the variables hash is returned. If one or more arguments are passed the corresponding values are returned.

Test::More Integration

There are a few caveats if you want to use Apache::Test with Test::More instead of the default Test backend. The first is that Test::More requires you to use its own "plan()" function and not the one that ships with Apache::Test. Test::More also defines "ok()" and

"skip()" functions that are different, and simply "use"ing both modules in your test script will lead to redefined warnings for these subroutines.

To assist Test::More users we have created a special Apache::Test import tag, ":withtestmore", which will export all of the standard Apache::Test symbols into your namespace except the ones that collide with Test::More.

```
use Apache::Test qw(:withtestmore);
use Test::More;
plan tests => 1;           # Test::More::plan()
ok ('yes', 'testing ok');  # Test::More::ok()
```

Now, while this works fine for standard client-side tests (such as "t/basic.t"), the more advanced features of Apache::Test require using Test::More as the sole driver behind the scenes.

Should you choose to use Test::More as the backend for server-based tests (such as "t/response/TestMe/basic.pm") you will need to use the "-withtestmore" action tag:

```
use Apache::Test qw(-withtestmore);
sub handler {
    my $r = shift;
    plan $r, tests => 1;           # Test::More::plan() with
                                   # Apache::Test features
    ok ('yes', 'testing ok');      # Test::More::ok()
}
```

"-withtestmore" tells Apache::Test to use Test::More instead of Test.pm behind the scenes.

Note that you are not required to "use Test::More" yourself with the "-withtestmore" option and that the "use Test::More tests => 1" syntax may have unexpected results.

Note that Test::More version 0.49, available within the Test::Simple 0.49 distribution on CPAN, or greater is required to use this feature.

Because Apache:Test was initially developed using Test as the framework driver, complete Test::More integration is considered experimental at this time - it is supported as best as possible but is not guaranteed to be as stable as the default Test interface at this time.

Apache::TestToString Class

The Apache::TestToString class is used to capture Test.pm output into a string. Example:

```
Apache::TestToString->start;
```

```
plan tests => 4;

ok $data eq 'foo';

...

# $tests will contain the Test.pm output: 1..4\nok 1\n...

my $tests = Apache::TestToString->finish;
```

SEE ALSO

The Apache-Test tutorial: <http://perl.apache.org/docs/general/testing/testing.html>.

Apache::TestRequest subclasses LWP::UserAgent and exports a number of useful functions for sending request to the Apache test server. You can then test the results of those requests.

Use Apache::TestMM in your Makefile.PL to set up your distribution for testing.

AUTHOR

Doug MacEachern with contributions from Geoffrey Young, Philippe M. Chiasson, Stas Bekman and others.

Questions can be asked at the test-dev <at> httpd.apache.org list For more information see: http://httpd.apache.org/test/.