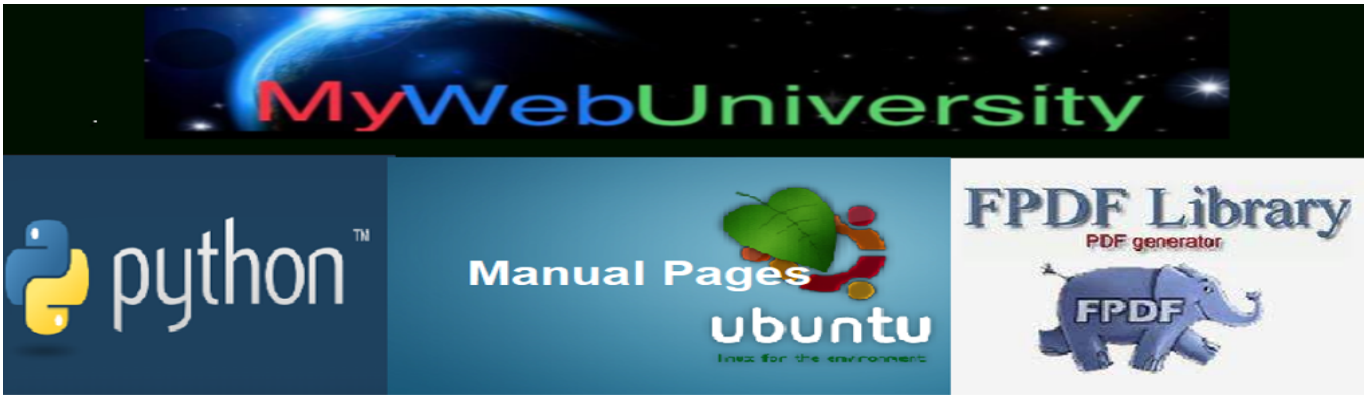

Rocky Enterprise Linux 9.2 Manual Pages on command 'Test::Tester.3perl'

$ man Test::Tester.3perl

Test::Tester(3perl) Perl Programmers Reference Guide Test::Tester(3perl)

NAME

Test::Tester - Ease testing test modules built with Test::Builder

SYNOPSIS

```
use Test::Tester tests => 6;
use Test::MyStyle;
check_test(
  sub {
    is_mystyle_eq("this", "that", "not eq");
  },
  {
    ok => 0, # expect this to fail
    name => "not eq",
    diag => "Expected: 'this'\nGot: 'that'",
  }
);
```

or

```
use Test::Tester tests => 6;
use Test::MyStyle;
check_test(
  sub {
    is_mystyle_qr("this", "that", "not matching");
  },
```

```
    {
      ok => 0, # expect this to fail
      name => "not matching",
      diag => qr/Expected: 'this'\s+Got: 'that'/,
    }
  );
or
  use Test::Tester;
  use Test::More tests => 3;
  use Test::MyStyle;
  my ($premature, @results) = run_tests(
    sub {
      is_database_alive("dbname");
    }
  );
  # now use Test::More::like to check the diagnostic output
  like($results[0]->{diag}, "/^Database ping took \\d+ seconds$"/, "diag");
```

DESCRIPTION

If you have written a test module based on Test::Builder then Test::Tester allows you to test it with the minimum of effort.

HOW TO USE (THE EASY WAY)

From version 0.08 Test::Tester no longer requires you to included anything special in your test modules. All you need to do is

```
  use Test::Tester;
```

in your test script before any other Test::Builder based modules and away you go.

Other modules based on Test::Builder can be used to help with the testing. In fact you can even use functions from your module to test other functions from the same module (while this is possible it is probably not a good idea, if your module has bugs, then using it to test itself may give the wrong answers).

The easiest way to test is to do something like

```
  check_test(
    sub { is_mystyle_eq("this", "that", "not eq") },
    {
```

```
      ok => 0, # we expect the test to fail
      name => "not eq",
      diag => "Expected: 'this'\nGot: 'that'",
    }
  );
```

this will execute the is_mystyle_eq test, capturing its results and checking that they are what was expected.

You may need to examine the test results in a more flexible way, for example, the diagnostic output may be quite long or complex or it may involve something that you cannot predict in advance like a timestamp. In this case you can get direct access to the test results:

```
  my ($premature, @results) = run_tests(
    sub {
      is_database_alive("dbname");
    }
  );
  like($result[0]->{diag}, "/^Database ping took \\d+ seconds$"/, "diag");
```

or

```
  check_test(
    sub { is_mystyle_qr("this", "that", "not matching") },
    {
      ok => 0, # we expect the test to fail
      name => "not matching",
      diag => qr/Expected: 'this'\s+Got: 'that'/,
    }
  );
```

We cannot predict how long the database ping will take so we use Test::More's like() test to check that the diagnostic string is of the right form.

HOW TO USE (THE HARD WAY)

This is here for backwards compatibility only

Make your module use the Test::Tester::Capture object instead of the Test::Builder one.

How to do this depends on your module but assuming that your module holds the Test::Builder object in $Test and that all your test routines access it through $Test then

providing a function something like this

```
sub set_builder
{
  $Test = shift;
}
```

should allow your test scripts to do

```
Test::YourModule::set_builder(Test::Tester->capture);
```

and after that any tests inside your module will captured.

TEST RESULTS

The result of each test is captured in a hash. These hashes are the same as the hashes returned by Test::Builder->details but with a couple of extra fields.

These fields are documented in Test::Builder in the details() function

ok

Did the test pass?

actual_ok

Did the test really pass? That is, did the pass come from Test::Builder->ok() or did it pass because it was a TODO test?

name

The name supplied for the test.

type

What kind of test? Possibilities include, skip, todo etc. See Test::Builder for more details.

reason

The reason for the skip, todo etc. See Test::Builder for more details.

These fields are exclusive to Test::Tester.

diag

Any diagnostics that were output for the test. This only includes diagnostics output after the test result is declared.

Note that Test::Builder ensures that any diagnostics end in a \n and it in earlier versions of Test::Tester it was essential that you have the final \n in your expected diagnostics. From version 0.10 onward, Test::Tester will add the \n if you forgot it. It will not add a \n if you are expecting no diagnostics. See below for help tracking down hard to find space and tab related problems.

depth

This allows you to check that your test module is setting the correct value for $Test::Builder::Level and thus giving the correct file and line number when a test fails. It is calculated by looking at caller() and $Test::Builder::Level. It should count how many subroutines there are before jumping into the function you are testing. So for example in

```
run_tests( sub { my_test_function("a", "b") } );
```

the depth should be 1 and in

```
sub deeper { my_test_function("a", "b") }
run_tests(sub { deeper() });
```

depth should be 2, that is 1 for the sub {} and one for deeper(). This might seem a little complex but if your tests look like the simple examples in this doc then you don't need to worry as the depth will always be 1 and that's what Test::Tester expects by default.

Note: if you do not specify a value for depth in check_test() then it automatically compares it against 1, if you really want to skip the depth test then pass in undef.

Note: depth will not be correctly calculated for tests that run from a signal handler or an END block or anywhere else that hides the call stack.

Some of Test::Tester's functions return arrays of these hashes, just like Test::Builder->details. That is, the hash for the first test will be array element 1 (not 0). Element 0 will not be a hash it will be a string which contains any diagnostic output that came before the first test. This should usually be empty, if it's not, it means something output diagnostics before any test results showed up.

SPACES AND TABS

Appearances can be deceptive, especially when it comes to emptiness. If you are scratching your head trying to work out why Test::Tester is saying that your diagnostics are wrong when they look perfectly right then the answer is probably whitespace. From version 0.10 on, Test::Tester surrounds the expected and got diag values with single quotes to make it easier to spot trailing whitespace. So in this example

```
# Got diag (5 bytes):
# 'abcd '
# Expected diag (4 bytes):
# 'abcd'
```

it is quite clear that there is a space at the end of the first string. Another way to solve this problem is to use colour and inverse video on an ANSI terminal, see below COLOUR below if you want this.

Unfortunately this is sometimes not enough, neither colour nor quotes will help you with problems involving tabs, other non-printing characters and certain kinds of problems inherent in Unicode. To deal with this, you can switch Test::Tester into a mode whereby all "tricky" characters are shown as \{xx}. Tricky characters are those with ASCII code less than 33 or higher than 126. This makes the output more difficult to read but much easier to find subtle differences between strings. To turn on this mode either call "show_space()" in your test script or set the "TESTTESTERSPACE" environment variable to be a true value. The example above would then look like

```
# Got diag (5 bytes):
# abcd\x{20}
# Expected diag (4 bytes):
# abcd
```

COLOUR

If you prefer to use colour as a means of finding tricky whitespace characters then you can set the "TESTTESTCOLOUR" environment variable to a comma separated pair of colours, the first for the foreground, the second for the background. For example "white,red" will print white text on a red background. This requires the Term::ANSIColor module. You can specify any colour that would be acceptable to the Term::ANSIColor::color function.

If you spell colour differently, that's no problem. The "TESTTESTERCOLOR" variable also works (if both are set then the British spelling wins out).

EXPORTED FUNCTIONS

($premature, @results) = run_tests(\&test_sub)

\&test_sub is a reference to a subroutine.

run_tests runs the subroutine in $test_sub and captures the results of any tests inside it. You can run more than 1 test inside this subroutine if you like.

$premature is a string containing any diagnostic output from before the first test.

@results is an array of test result hashes.

cmp_result(\%result, \%expect, $name)

\%result is a ref to a test result hash.

\%expect is a ref to a hash of expected values for the test result.

cmp_result compares the result with the expected values. If any differences are found it outputs diagnostics. You may leave out any field from the expected result and cmp_result will not do the comparison of that field.

cmp_results(\@results, \@expects, $name)

\@results is a ref to an array of test results.

\@expects is a ref to an array of hash refs.

cmp_results checks that the results match the expected results and if any differences are found it outputs diagnostics. It first checks that the number of elements in \@results and \@expects is the same. Then it goes through each result checking it against the expected result as in cmp_result() above.

($premature, @results) = check_tests(\&test_sub, \@expects, $name)

\&test_sub is a reference to a subroutine.

\@expect is a ref to an array of hash refs which are expected test results.

check_tests combines run_tests and cmp_tests into a single call. It also checks if the tests died at any stage.

It returns the same values as run_tests, so you can further examine the test results if you need to.

($premature, @results) = check_test(\&test_sub, \%expect, $name)

\&test_sub is a reference to a subroutine.

\%expect is a ref to an hash of expected values for the test result.

check_test is a wrapper around check_tests. It combines run_tests and cmp_tests into a single call, checking if the test died. It assumes that only a single test is run inside \&test_sub and include a test to make sure this is true.

It returns the same values as run_tests, so you can further examine the test results if you need to.

show_space()

Turn on the escaping of characters as described in the SPACES AND TABS section.

HOW IT WORKS

Normally, a test module (let's call it Test:MyStyle) calls Test::Builder->new to get the Test::Builder object. Test::MyStyle calls methods on this object to record information about test results. When Test::Tester is loaded, it replaces Test::Builder's new() method with one which returns a Test::Tester::Delegate object. Most of the time this object behaves as the real Test::Builder object. Any methods that are called are delegated to the

real Test::Builder object so everything works perfectly. However once we go into test mode, the method calls are no longer passed to the real Test::Builder object, instead they go to the Test::Tester::Capture object. This object seems exactly like the real Test::Builder object, except, instead of outputting test results and diagnostics, it just records all the information for later analysis.

CAVEATS

Support for calling Test::Builder->note is minimal. It's implemented as an empty stub, so modules that use it will not crash but the calls are not recorded for testing purposes like the others. Patches welcome.

SEE ALSO

Test::Builder the source of testing goodness. Test::Builder::Tester for an alternative approach to the problem tackled by Test::Tester - captures the strings output by Test::Builder. This means you cannot get separate access to the individual pieces of information and you must predict exactly what your test will output.

AUTHOR

Plan handling lifted from Test::More. written by Michael G Schwern <schwern@pobox.com>.

Test::Tester::Capture is a cut down and hacked up version of Test::Builder. Test::Builder was written by chromatic <chromatic@wgz.org> and Michael G Schwern <schwern@pobox.com>.

LICENSE