

# Test::Builder::Tester

by brian d foy  
bdfoy@cpan.org



Testing is a core part of Perl culture, but what am I to do when I want to write my own `Test::*` module? Typically, I use `Test::Builder`, which I don't cover here. Once I've write my special test code, I want to test that, too. Perl testing works through the Test Anywhere Protocol, which outputs the test results. That's a bit inconvenient when I want to get the results in my program instead of on my terminal. The `Test::Builder::Tester` module is going to handle all of the hard work for me.

## ■ A bit of history-----

When I first got into Perl testing, there weren't many test modules yet. Michael Schwern created `Test::More` came out a bit later and showed everyone that we could use the same `ok()` functions that everyone was making on their own. Along with that, `Test::More` handled the test counts, so we didn't have to do that on our own anymore either.

Things got a bit sticky when I wanted to test more complex things. I could use `ok()`, `is()`, and `like()`, but I still had to write a lot of code around those to prepare the values and set up the tests, when I really wanted a single line that did the whole thing. For instance, assume that I want to test a URI to ensure that I have the right host. I could do that manually (and probably incorrectly) by extracting the host myself every time I want to do that:

```
use Test::More tests => 1;
my $uri = 'http://example.com/foo.html';
... code to extract host ...
ok( $host, 'example.com');
```

Why should I do all of that work? I want to do it in one shot with as much code reuse as possible. For that, I created `Test::URI` and added an `uri_host_ok()` subroutine:

```
use Test::More tests => 1;
use Test::URI;

my $uri = 'http://example.com/foo.html';
uri_host_ok( $uri, 'example.com' );
```

That's much better. Everything I need to do is contained in that one subroutine. My test file concentrates on what I'm trying to test, not how I test it. Now that I have my module with its specialized test subroutines, I need to test those too. I

need to test my subroutine to check its output for both success and failure.

## ■ Test output for passing tests -----

Test output shows up on both standard output and standard error. The test information which shows the `ok` and `not ok` messages goes to `stdout`. The comments and diagnostics, including the messages that tell me about test failures, goes to `stderr`. I need to intercept both of these to check my tests. It's not a hard task and there are plenty of modules that can intercept output, but I don't want to think about that.

`Test::Builder::Tester` handles the interception for me. Here's a bit of code to test my `uri_host_ok()` subroutine. I set it up by pulling in `Test::Builder::Tester` and tell it how many tests I'm going to run. Since I can only set the plan once, I tell `Test::Builder::Tester` the count of all tests. I also pull in `Test::More` so I can test that my module loads:

```
use Test::Builder::Tester tests => 2;
use Test::More;

use_ok( 'Test::URI' );

{
my $uri_string =
    "http://www.example.com/index.html";

test_out( "ok 1" );

uri_host_ok(
    $uri_string,
    'www.example.com' );

test_test("uri_host_ok with string");
}
```

Before I run `uri_host_ok()`, I call the `test_out()` subroutine to tell `Test::Builder::Tester` what I expect to see on `stdout`. Since I expect this test to pass, I don't need to look at `stderr`. If the `uri_host_ok()` test fails, it's `stdout` won't be just `ok 1`.

Once I tell `test_out()` what to expect, I run `uri_host_ok()`. `Test::Builder::Tester` intercepts the output and holds onto it. When I call `test_test()`, `Test::Builder::Tester` tells me if it saw what I expected.