Testing mod_perl 2.0 - Perl.com

By Geoffrey Young on May 22, 2003 12:00 AM

Last time, we looked at writing a simple Apache output filter - `Apache::Clean` - using the mod_perl 2.0 API. How did I know that the filter I presented really worked? I wrote a test suite for it, one that exercised the code against a live Apache server using the `Apache-Test` testing framework.

Writing a series of tests that executes against a live Apache server has become much simpler since the advent of `Apache-Test`. Although `Apache-Test`, as part of the Apache HTTP Test Project, is generic enough to be used with virtually any version of Apache (with or without mod_perl enabled), it comes bundled with mod_perl 2.0, making it the tool of choice for writing tests for your mod_perl 2.0 modules.

Testing, Testing, 1, 2, 3

There are many advantages to writing tests. For instance, maintaining the test suite as I coded `Apache::Clean` allowed me to test each functional unit as I implemented it, which made development easier. The individual tests also allowed me to be fairly certain that the module would behave as expected once distributed. As an added bonus, tests offer additional end-user documentation in the form of test scripts, supporting libraries and configuration files, available to anyone who wants to snoop around the distribution a bit. All in all, having a test suite increases the value of your code exponentially, while at the same time making your life easier.

Of course, these benefits come from having *any* testing environment, and are not limited to just `Apache-Test`. The particular advantage that `Apache-Test` brings to the table is the ease at which it puts a whole, pristine, and isolated Apache server at your disposal, allowing you to test and exercise your code in a live environment with a minimum of effort. No more `Apache::FakeRequest`, no more `httpd.conf` configurations strewn across development environments or corrupted with proof-of-concept handlers that keep you busy following non-bugs for half a day. No more mess, no more tears.

If you have ever used tools like `Test.pm` or `Test::More` as the basis for testing your modules, then you already know most of what using `Apache-Test` is going to look like. In fact, `Apache-Test` uses `Test.pm` under the hood, so the layout and syntax are similar. If you have never written a test before, (and shame on you)then An Introduction to Testing provides a nice overview of testing with Perl. For the most part, though, `Apache-Test` is really simple enough that you should be able to follow along here without any trouble or previous knowledge.

Leveraging the `Apache-Test` framework requires only a few steps - generating the test harness, configuring Apache to your specific needs, and writing the tests - each of which is relatively straightforward.

Generating the Test Harness

The first step to using `Apache-Test` is to tweak the `Makefile.PL` for your module. If you don't yet have a `Makefile.PL`, or are not familiar with how to generate one, then don't worry - all that is required is a simple call to `h2xs`, which provides us with a standard platform both for distributing our module and deploying the `Apache-Test` infrastructure.

```
$ h2xs -AXPn Apache::Clean
Defaulting to backward compatibility with perl 5.9.0
If you intend this module to be compatible with earlier perl versions, then
please
specify a minimum perl version with the -b option.

Writing Apache/Clean/Clean.pm
Writing Apache/Clean/Makefile.PL
Writing Apache/Clean/README
Writing Apache/Clean/t/1.t
Writing Apache/Clean/Changes
Writing Apache/Clean/MANIFEST
```

`h2xs` generates the necessary structure for our module, namely the `Clean.pm` template and the `Makefile.PL`, as well as the `t/` subdirectory where our tests and supporting files will eventually live. You can take some extra steps and shuffle the distribution around a bit (such as removing `t/1.t` and putting everything into `Apache-Clean/` instead of `Apache/Clean/`) but it is not required. Once you have the module layout sorted out and have replaced the generated `Clean.pm` stub with the actual `Clean.pm` filter from before, it's time to start preparing the basic test harness.

To begin, we need to modify the `Makefile.PL` significantly. The end result should look something like:

```
#!perl

use 5.008;

use Apache2 ();
use ModPerl::MM ();
use Apache::TestMM qw(test clean);
use Apache::TestRunPerl ();

# configure tests based on incoming arguments
Apache::TestMM::filter_args();
```

```
# provide the test harness
Apache::TestRunPerl->generate_script();

# now, write out the Makefile
ModPerl::MM::WriteMakefile(
  NAME      => 'Apache::Clean',
  VERSION   => '2.0',
  PREREQ_PM => { HTML::Clean      => 0.8,
                 mod_perl         => 1.9909, },
);
```

Let's take a moment to analyze our nonstandard `Makefile.PL`. We begin by importing a few new mod_perl 2.0 libraries. The first is `Apache2.pm`. In order to peacefully co-exist with mod_perl 1.0 installations, mod_perl 2.0 gives you the option of installing mod_perl relative to `Apache2/` in your `@INC`, as to avoid collisions with 1.0 modules of the same name. For instance, the mod_perl 2.0 `Apache::Filter` we used to write our output filter interface would be installed as `Apache2/Apache/Filter.pm`. Of course, ordinary calls that `require()` or `use()` `Apache::Filter` in mod_perl 2.0 code would fail to find the correct version (if one was found at all), since it was installed in a nonstandard place. `Apache2.pm` extends `@INC` to include any (existing) `Apache2/` directories so that `use()` and related statements work as intended. In our case, we need to `use() Apache2` in order to ensure that, no matter how the end-user configured his mod_perl 2.0 installation, we can find the rest of the libraries we need.

Secure in the knowledge that our `Makefile.PL` will be able to find all our other mod_perl 2.0 packages (wherever they live), we can proceed. `ModPerl::MM` provides the `WriteMakefile()` function, which is similar to the `ExtUtils::MakeMaker` function of the same name and takes the same options. The reason that you will want to use the `WriteMakefile()` from `ModPerl::MM` is that, through means highly magical, all of your mod_perl-specific needs are satisfied. For instance, your module will be installed relative to `Apache/` or `Apache2/`, depending on how mod_perl itself is installed. Other nice features are automatic inclusion of mod_perl's `typemap` and the header files required for XS-based modules, as well as magical cross-platform compatibility for Win32 compilation, which has been troublesome in the past.

Keep in mind that neither `Apache2.pm` nor `ModPerl::MM` are required in order to use `Apache-Test` - both are packages specific to mod_perl 2.0 and any handlers you may write for this version (as will be touched on later, `Apache-Test` can be used for mod_perl 1.0 based modules as well, or even Apache 1.3 or 2.0 modules independent of mod_perl, for that matter). The next package, `Apache::TestMM`, is where the real interface for `Apache-Test` begins.

`Apache::TestMM`, contains the functions we will need to configure the test harness. The first thing we do

is import the `test()` and `clean()` functions, which generate their respective `Makefile` targets so that we can run (and re-run) our tests. After that, we call the `filter_args()` function. This allows us to configure various parts of our tests on the command line using different options, which will be discussed later.

The final part of our configuration uses the `generate_script()` method from the `Apache::TestRunPerl` class, which writes out the script responsible for running our tests, `t/TEST`. It is `t/TEST` that will be invoked when a user issues `make test`, although the script can be called directly as well. While `t/TEST` can end up containing lots of information, if you crack it open, then you would see that the engine that really drives the test suite is rather simple.

```
use Apache::TestRunPerl ();
Apache::TestRunPerl->new->run(@ARGV);
```

Believe it or not, the single call to `run()` does all intricate work of starting, configuring, and stopping Apache, as well as running the individual tests we (still) have yet to define.

Despite the long explanations, the net result of our activity thus far has been a few modifications to a typical `Makefile.PL` so that it reflects the needs of both our mod_perl 2.0 module and our forthcoming use of the `Apache-Test` infrastructure. Next, we need to configure Apache for the tests specific to the functionality in our handler.

Configuring Apache

Ordinarily, there are many things you need to stuff into `httpd.conf` in order to get the server responding to requests, only some of which are related to the content the server will provide. The `Apache-Test` framework provides a minimal Apache configuration, such as default `DocumentRoot`, `ErrorLog`, `Listen`, and other settings required for normal operation of the server. In fact, with no intervention on your part, `Apache-Test` provides a configuration that enables you to successfully request `/index.html` from the server. Chances are, though, that you will need something above a basic configuration in order to test your module appropriately.

To add additional settings to the defaults, we create the file `t/conf/extra.conf.in`, adding any required directories along the way. If `Apache-Test` sees `extra.conf.in`, then it would pull the file into its default configuration using an `Include` directive (after some manipulations we will discuss shortly). This provides a nice way of adding only the configuration data you require for your tests, and saves you from the need to worry about the mundane aspects of running the server.

One of the first aspects of `Apache::Clean` we should test is whether it can clean up a simple, static

HTML file. So, we begin our `extra.conf.in` with the following:

```
PerlSwitches -w

Alias /level @DocumentRoot@
<Location /level>
  PerlOutputFilterHandler Apache::Clean
  PerlSetVar CleanLevel 2
</Location>
```

This activates our output filter for requests to `/level`. Note the introduction of a new directive, `PerlSwitches`, which allows you to pass command line switches to the embedded perl interpreter. Here, we use it to enable warnings, similar to the way that `PerlWarn` worked in mod_perl 1.0. `PerlSwitches` can actually take any perl command line switch, which makes it a fairly useful and flexible tool. For example, we could use the `-I` switch to extend `@INC` in place of adding `use lib` statements to a `startup.pl`, or use `-T` to enable taint mode in place of the former `PerlTaintMode` directive, which is not part of mod_perl 2.0.

Next, we come to the familiar `Alias` directive, albeit with a twist. As previously mentioned, `Apache-Test` configures several defaults, including `DocumentRoot` and `ServerRoot`. One of the nice features of `Apache-Test` is that it keeps track of its defaults for you and provides some helpful variable expansions. In my particular case, the `@DocumentRoot@` variable in the `Alias` directive is replaced with the value of the default `DocumentRoot` that `Apache-Test` calculated for my build. The real configuration ends up looking like

```
Alias /level /src/perl.com/Apache-Clean-2.0/t/htdocs
```

when the tests are run. This is handy, especially when you take into consideration that your tests may run on different platforms.

The rest of the configuration closely resembles our example from last time - using the `PerlOutputFilterHandler` to specify `Apache::Clean` as our output filter, and `PerlSetVar` to specify the specific `HTML::Clean` level. The only thing missing before we have prepared our module enough to run our first test is some testable content in `DocumentRoot`.

As you can see from the `@DocumentRoot@` expansion in the previous example, `DocumentRoot` resolves to *`ServerRoot`*`/t/htdocs/`, so that is one place where we can put any documents we are interested in

retrieving for our tests. So, we create `t/htdocs/index.html` and place some useful content in it.

```
<i    ><strong>&quot;This is a test&quot;</strong></i   >
```

Our `index.html` contains a number of different elements that `HTML::Clean` can tidy, making it useful for testing various configurations of `Apache::Clean`.

Now we have all the Apache configuration that is required: some custom configuration directives in `t/conf/extra.conf.in` and some useful content in `t/htdocs/index.html`. All that is left to do is write the tests.

Writing the Tests

The Apache configuration we have created thus far provides a way to test `Apache::Clean` through `/level/index.html`. The result of this request should be that the default Apache content handler serves up `index.html`, applying our `PerlOutputFilterHandler` to the file before it is sent over the wire. Given the configured `PerlSetVar CleanLevel 2` we would expect the end results of the request to be

```
<i><b>&quot;This is a test&quot;</b></i>
```

where tags are shortened and whitespace removed but the `&quot;` entity is left untouched. Well, maybe this is not what *you* would have expected, but cracking open the code for `HTML::Clean` reveals that `level(2)` includes the `whitespace` and `shortertags` options, but not the `entities` option. This brings us to the larger issue of test design and the possibility that flawed expectations can mask true bugs - when a test fails, is the bug in the test or in the code? - but that is a discussion for another time.

Given our configuration and expected results, we can craft a test that requests `/level/index.html`, isolates the content from the server response, then tests the content against our expectations. The file `t/01level.t` shown here does exactly that.

```
use strict;
use warnings FATAL => 'all';

use Apache::Test qw(plan ok have_lwp);
use Apache::TestRequest qw(GET);
```

```
use Apache::TestUtil qw(t_cmp);

plan tests => 1, have_lwp;

my $response = GET '/level/index.html';
chomp(my $content = $response->content);

ok ($content eq q!<i><b>&quot;This is a test&quot;</b></i>!);
```

`t/01level.t` illustrates a few of the things that will be common to most of the tests you will write. First, we do some bookkeeping and plan the number of tests that will be attempted using the `plan()` function from `Apache::Test` - in our case just one. The final, optional argument to `plan()` uses the `have_lwp()` function to check for the availability of the modules from the `libwww-perl` distribution. If `have_lwp()` returns true, then we know we can take advantage of the `LWP` shortcuts `Apache::TestRequest` provides. If `have_lwp()` returns false, then no tests are planned and the entire test is skipped at runtime.

After planning our test, we use the shortcut function `GET()` from `Apache::TestRequest` to issue a request to `/level/index.html`. `GET()` returns an `HTTP::Response` object, so if you are familiar with the `LWP` suite of modules you should feel right at home with what follows. Using the object in `$response` we isolate the server response using the `content()` method and compare it against our expected string. The comparison uses a call to `ok()`, which will report success if the two strings are equivalent.

Keep in mind that even though this example explicitly imported the `plan()`, `ok()`, `have_lwp()`, and `GET()` functions into our test script, that was just to illustrate the origins of the different parts of the test - each of these functions, along with just about all the others you may want, are exported by default. So, the typical test script will usually just call

```
use Apache::Test;
use Apache::TestRequest;
```

and go from there.

That is all there is to writing the test. In its simplest form, using `Apache-Test` involves pretty much the same steps as when writing tests using other Perl testing tools: `plan()` the number of tests in the script, do some stuff, and call `ok()` for each test you `plan()`. `Apache-Test` and its utility classes merely offer shortcuts that make writing tests against a running Apache server idiomatic.

Running the Tests

With all the preparation behind us - generating and customizing the `Makefile.PL`, configuring Apache with `extra.conf.in`, writing `index.html` and `01level.t` - we have all the pieces in place and can (finally) run our test.

There are a few different ways we can run the tests in a distribution, but all require that we go through the standard build steps first.

```
$ perl Makefile.PL -apxs /usr/local/apache2/bin/apxs
Checking if your kit is complete ...
Looks good
Writing Makefile for Apache::Clean

$ make
cp Clean.pm blib/lib/Apache2/Apache/Clean.pm
Manifying blib/man3/Apache::Clean.3
```

`Makefile.PL` starts the process by generating the `t/TEST` script via the call to `Apache::TestRunPerl->generate_script()`. The additional argument we pass, `-apxs`, is trapped by `Apache::TestMM::filter_args()` and is used to specify the Apache installation we want to test our code against. Here, I use `-apxs` to specify the location of the `apxs` binary in my local Apache DSO installation - for static builds you will want to use `-httpd` to point to the `httpd` binary instead. By the time `Makefile.PL` exits, we have our test harness and know where our server lives.

Running `make` creates our build directory, `blib/`, and installs `Clean.pm` locally so we can use it in our tests. Note that `ModPerl::MM` installed `Clean.pm` relative to `Apache2`, magically following the path of my current mod_perl 2.0 installation.

At this point, we can run our tests. Issuing `make test` will run all the tests in `t/`, as you might expect. However, we can run our tests individually as well, which is particularly useful when debugging. To run a specific test we call `t/TEST` directly and give it the name of the test we are interested in.

```
$ t/TEST t/01level.t
*** setting ulimit to allow core files
ulimit -c unlimited; t/TEST 't/01level.t'
/usr/local/apache2/bin/httpd  -d /src/perl.com/Apache-Clean-2.0/t
    -f /src/perl.com/Apache-Clean-2.0/t/conf/httpd.conf
```

```
        -DAPACHE2 -DPERL_USEITHREADS
using Apache/2.1.0-dev (prefork MPM)

waiting for server to start: ..
waiting for server to start: ok (waited 1 secs)
server localhost:8529 started
01level....ok
All tests successful.
Files=1, Tests=1,  4 wallclock secs ( 3.15 cusr +  0.13 csys =  3.28 CPU)
*** server localhost:8529 shutdown
```

As you can see, the server was started, our test was run, the server was shutdown, and a report was generated - all with what is really minimal work on our part. Major kudos to the `Apache-Test` developers for making the development of live tests as easy as they are.

Beyond the Basics

What we have talked about so far is just the basics, and the framework is full of a number of different options designed to make writing and debugging tests easier. One of these is the `Apache::TestUtil` package, which provides a number of utility functions you can use in your tests. Probably the most helpful of these is `t_cmp()`, a simple equality testing function that also provides additional information when you run tests in verbose mode. For instance, after adding `use Apache::TestUtil;` to our `01level.t` test, we can alter the call to `ok()` to look like

```
ok t_cmp(q!<i><b>&quot;This is a test&quot;</b></i>!, $content);
```

and the result would include expected and received notices (in addition to standard verbose output)

```
$ t/TEST t/01level.t -v
[lines snipped]
01level....1..1
# Running under perl version 5.009 for linux
# Current time local: Mon May  5 11:04:09 2003
# Current time GMT:   Mon May  5 15:04:09 2003
# Using Test.pm version 1.24
# expected: <i><b>&quot;This is a test&quot;</b></i>
# received: <i><b>&quot;This is a test&quot;</b></i>
```

```
ok 1
ok
All tests successful.
```

which is particularly helpful when debugging problems reported by end users of your code. See the `Apache::TestUtil` manpage for a long list of helper functions, as well as the `README` in the `Apache-Test` distribution for additional command line options over and above `-v`.

Of course, `01level.t` only tests one aspect of our `Clean.pm` output filter, and there is much more functionality in the filter that we might want verify. So, let's take a quick look at some of the other tests that accompany the `Apache::Clean` distribution.

One of the features of `Apache::Clean` is that it automatically declines processing non-HTML documents. The logic for this was defined in just a few lines at the start of our filter.

```
# we only process HTML documents
unless ($r->content_type =~ m!text/html!i) {
  $log->info('skipping request to ', $r->uri, ' (not an HTML document)');

  return Apache::DECLINED;
}
```

A good test for this code would be verifying that content from a plain-text document does indeed pass through our filter unaltered, even if it has HTML tags that `HTML::Clean` would ordinary manipulate. Our test suite includes a file `t/htdocs/index.txt` whose content is identical to the `index.html` file we created earlier. Remembering that we already have an Apache configuration for `/level` that inserts our filter into the request cycle, we can use a request for `/level/index.txt` to test the decline logic.

```
use Apache::Test;
use Apache::TestRequest;

plan tests => 1, have_lwp;

my $response = GET '/level/index.txt';
chomp(my $content = $response->content);

ok ($content eq q!<i><strong>&quot;This is a test&quot;</strong></i>!);
```

It may be obvious, but if you think about what we are really testing here it is not that the content is unaltered - that is just what we use to measure the success of our test. The real test is against the criterion that determines whether the filter acts on the content. If we wanted to be really thorough, then we could add

```
AddDefaultCharset On
```

to our `extra.conf.in` to test the `Content-Type` logic against headers that look like `text/html; charset=iso-8859-1` instead of just `text/html`. I actually have had more than one person comment that using a regular expression for testing the `Content-Type` is excessive - adding the `AddDefaultCharset On` directive shows that the regex logic can handle more runtime environments than a simple `$r->content_type eq 'text/html'` check. Oh, the bugs you will find, fix, and defend when you start writing tests.

More and More Tests

What other aspects of the filter can we put to the test? If you recall from our discussion of output filters last time, one of the responsibilities of filters that alter content is to remove the generated `Content-Length` header from the server response. The relevant code for this in our filter was as follows.

```
# output filters that alter content are responsible for removing
# the Content-Length header, but we only need to do this once.
$r->headers_out->unset('Content-Length');
```

Here is the test for this bit of logic, which checks that the `Content-Length` header is indeed present for plain documents, but removed by our filter for HTML documents. Again, we will be using the existing `/level` URI to request both `index.txt` and `index.html`.

```
use Apache::Test;
use Apache::TestRequest;

plan tests => 2, have_lwp;

my $response = GET '/level/index.txt';
```

```
ok ($response->content_length == 58);

$response = GET '/level/index.html';
ok (! $response->content_length);
```

Note the use of the `content_length()` method on our `HTTP::Response` object to retrieve the `Content-Length` of the server response. Remember that you have all the methods from that class to choose from in your tests.

The final test we will take a look at is the example we used previous time to illustrate our filter does indeed co-exist with both mod_include and mod_cgi. As it turns out, the example was taken right from the test suite (always a good place from which to draw examples). Here is the `extra.conf.in` snippet.

```
Alias /cgi-bin @ServerRoot@/cgi-bin
<Location /cgi-bin>
  SetHandler cgi-script

  SetOutputFilter INCLUDES
  PerlOutputFilterHandler Apache::Clean

  PerlSetVar CleanOption shortertags
  PerlAddVar CleanOption whitespace
  Options +ExecCGI +Includes
</Location>
```

The nature of our test requires that both mod_include and a suitable CGI platform (either mod_cgi or mod_cgid) be available to Apache - without both of these, our tests are doomed to failure, so we need a way to test whether these modules are available to the server before planning the individual tests. Also required are some CGI scripts, the location of which is specified by expanding `@ServerRoot@`. To include these scripts, we could just create a `t/cgi-bin/` directory and place the relevant files in it. However, any CGI scripts we create would probably include a platform-specific shebang line like `#!/usr/bin/perl`. A better solution is to generate the scripts on-the-fly, specifying a shebang line that matches the version of Perl we are using to build and test the module.

Despite the extra work required, the test script used for this test is only a bit more complex than others we have seen so far.

```
```

```
use Apache::Test;
use Apache::TestRequest;
use Apache::TestUtil qw(t_write_perl_script);

use File::Spec::Functions qw(catfile);

plan tests => 4, (have_lwp &&
                  have_cgi &&
                  have_module('include'));

my @lines = <DATA>;
t_write_perl_script(catfile(qw(cgi-bin plain.cgi)), @lines[0,2]);
t_write_perl_script(catfile(qw(cgi-bin include.cgi)), @lines[1,2]);

my $response = GET '/cgi-bin/plain.cgi';
chomp(my $content = $response->content);

ok ($content eq q!<strong>/cgi-bin/plain.cgi</strong>!);
ok ($response->content_type =~ m!text/plain!);

$response = GET '/cgi-bin/include.cgi';
chomp($content = $response->content);

ok ($content eq q!<b>/cgi-bin/include.cgi</b>!);
ok ($response->content_type =~ m!text/html!);

__END__
print "Content-Type: text/plain\n\n";
print "Content-Type: text/html\n\n";
print '<strong><!--#echo var="DOCUMENT_URI" --></strong>';
```

The first thing to note is that we have joined the familiar call to `have_lwp()` with additional calls to `have_cgi()` and `have_module()`. The `Apache::Test` package comes with a number of handy shortcuts for querying the server for information. `have_cgi()` returns true if either mod_cgi or mod_cgid are installed. `have_module()` is more generic and can be used to test for either Apache C modules or Perl modules - for instance, `have_module('Template')` could be used to check whether the `Template Toolkit` is installed.

For generation of the CGI scripts, we use the `t_write_perl_script()` function from the `Apache::TestUtil` package. `t_write_perl_script()` takes two arguments, the first of which is the

name of the file to generate, relative to the `t/` directory in the distribution. If the file includes a path, any necessary directories are automatically created. In the interests of portability, we use `catfile()` from the `File::Spec::Functions` package to join the file with the directory. In general, you will want to keep `File::Spec` and its associated classes in mind when writing your tests - you never know when somebody is going to try and run them on Win32 or VMS. The second argument to `t_write_perl_script()` is a list of lines to append to the file after the (calculated) shebang line.

Although `t_write_perl_script()` cleans up any generated files and directories when the test completes, if we were to intercept `include.cgi` before removal it would look similar to something we would have written ourselves.

```
#!/src/bleedperl/bin/perl
# WARNING: this file is generated, do not edit
# 01: /src/bleedperl/lib/site_perl/5.9.0/i686-linux-thread-multi/
      Apache/TestUtil.pm:129
# 02: 06mod_cgi.t:18
print "Content-Type: text/html\n\n";
print '<strong><!--#echo var="DOCUMENT_URI" --></strong>';
```

As you probably have guessed by now, just as we ran tests against scripts in the (generated) `t/cgi-bin/` directory, we can add other directories to `t/` for other kinds of tests. For instance, we can create `t/perl-bin/` to hold standard `ModPerl::Registry` scripts (remember, you don't need to generate a shebang line for those). We can even create `t/My/` to hold a custom `My::ContentGenerator` handler, which can be used just like any other Perl module during Apache's runtime. All in all, you can simulate practically any production environment imaginable.

But Wait, There's More!

The tests presented here should be enough to get you started writing tests for your own modules, but they are only part of the story. If you are interested in seeing some of the other tests written to support this article, the `Apache::Clean` distribution is full of all kinds of different tests and test approaches, including some that integrate custom handlers as well as one that tests the POD syntax for the module. In fact, you will find 26 different tests in 12 test files there, free for the taking.

Stuck using mod_perl 1.0? One of the best things about `Apache-Test` is that it is flexible and intelligent enough to be used for mod_perl 1.0 handlers as well. In fact, the recent release of `Apache-Test` as a CPAN module outside of the mod_perl 2.0 distribution makes it even easier for all mod_perl developers to take advantage of the framework. For the most part, the instructions in this article should be enough to get you going writing tests for 1.0-based modules - the only changes specific to 1.0 modules rest in the

`Makefile.PL`. I took the time to whip up a version of `Apache::Clean` for mod_perl 1.0 that parallels the functionality in these articles, which you can find next to the 2.0 version. The 1.0 distribution runs against the exact same `*.t` files (where applicable) and includes a sample 1.0 `Makefile.PL`.

Personally, I don't know how I ever got along without `Apache-Test`, and I'm sure that once you start using it you will feel the same. Secretly, I'm hoping that `Apache-Test` becomes so popular that end-users start wrapping their bug reports up in little, self-contained, `Apache-Test`-based tarballs so anyone can reproduce the problem.

More Information

This article was derived from Recipe 7.7 in the *mod_perl Developer's Cookbook*, adjusted to accommodate both mod_perl 2.0 and changes in the overall `Apache-Test` interface that have happened since publication. Despite these differences, the recipe is useful for its additional descriptions and coverage of features not discussed here. You can read Recipe 7.7, as well as the rest of Chapter 7 from the book's website. Also, in addition to the `Apache-Test` manpage and `README` there is also the `Apache-Test` tutorial on the mod_perl Project website, all of which are valuable sources of information.

Thanks

The `Apache-Test` project is the result of the tireless efforts of many, many developers - far too many to name individually here. However, there has has been a recent surge of activity as `Apache-Test` made its way to CPAN, especially in making it more platform aware and solving a few back compatibility problems with the old `Apache::test` that ships with mod_perl 1.0. Special thanks are due to Stas Bekman, David Wheeler, and Randy Kobes for helping to polish `Apache-Test` on Win32 and Mac OS X without requiring major changes to the API.