

1 Running and Developing Tests with the Apache::Test Framework

1.1 Description

The title is self-explanatory :)

The `Apache::Test` framework was designed for creating test suits for products running on Apache httpd webserver (not necessarily `mod_perl`). Originally designed for the `mod_perl` Apache module, it was extended to be used for any Apache module.

This chapter is talking about the `Apache::Test` framework, and in particular explains how to:

1. **run existing tests**
2. **setup a testing environment for a new project**
3. **develop new tests**

For other `Apache::Test` resources, see the References section at the end of this document.

1.2 Basics of Perl Modules Testing

The tests themselves are written in Perl. The framework provides an extensive functionality which makes the tests writing a simple and therefore enjoyable process.

If you have ever written or looked at the tests most Perl modules come with, `Apache::Test` uses the same concept. The script `t/TEST` is running all the files ending with `.t` it finds in the `t/` directory. When executed a typical test prints the following:

```
1..3      # going to run 3 tests
ok 1      # the first  test has passed
ok 2      # the second test has passed
not ok 3  # the third  test has failed
```

Every `ok` or `not ok` is followed by the number which tells which sub-test has succeeded or failed.

`t/TEST` uses the `Test::Harness` module which intercepts the `STDOUT` stream, parses it and at the end of the tests print the results of the tests running: how many tests and sub-tests were run, how many succeeded, skipped or failed.

Some tests may be skipped by printing:

```
1..0 # all tests in this file are going to be skipped.
```

Usually a test may be skipped when some feature is optional and/or prerequisites are not installed on the system, but this is not critical for the usefulness of the test. Once you test that you cannot proceed with the tests and it's not a must pass test, you just skip it.

By default `print()` statements in the test script are filtered out by `Test::Harness`. if you want the test to print what it does (if you decide to debug some test) use `-verbose` option. So for example if your test does this:

```
print "# testing : feature foo\n";  
print "# expected: $expected\n";  
print "# received: $received\n";  
ok $expected eq $received;
```

in the normal mode, you won't see any of these prints. But if you run the test with *t/TEST -verbose*, you will see something like this:

```
# testing : feature foo  
# expected: 2  
# received: 2  
ok 2
```

When you develop the test you should always put the debug statements there, and once the test works for you do not comment out or delete these debug statements. This is because if some user reports a failure in some test, you can ask him to run the failing test in the verbose mode and send you back the report. It'll be much easier to understand what the problem is if you get these debug printings from the user.

In the section Writing Tests several helper functions which make the tests writing easier are discussed.

For more details about the `Test::Harness` module please refer to its manpage. Also see the `Test` manpage about Perl's test suite.

1.3 Prerequisites

In order to use `Apache::Test` it has to be installed first.

Install `Apache::Test` using the familiar procedure:

```
% cd Apache-Test  
% perl Makefile.PL  
% make && make test && make install
```

If you install `mod_perl 2.0`, you get `Apache::Test` installed as well.

1.4 Running Tests

It's much easier to copy-cat things, than creating from scratch. It's much easier to develop tests, when you have some existing system that you can test, see how it works and build your own testing environment in a similar fashion. Therefore let's first look at how the existing test environments work.

You can look at the `modperl-2.0's` or `httpd-test's` (*perl-framework*) testing environments which both use `Apache::Test` for their test suites.

1.4.1 Testing Options

Run:

```
% t/TEST -help
```

to get the list of options you can use during testing. Most options are covered further in this document.

1.4.2 Basic Testing

Running tests is just like for any CPAN Perl module; first we generate the *Makefile* file and build everything with *make*:

```
% perl Makefile.PL [options]
% make
```

Now we can do the testing. You can run the tests in two ways. The first one is usual:

```
% make test
```

but it adds quite an overhead, since it has to check that everything is up to date (the usual make source change control). Therefore you have to run it only once after *make* and for re-running the tests it's faster to run the tests directly via:

```
% t/TEST
```

When *make test* or *t/TEST* are run, all tests found in the *t* directory (files ending with *.t* are recognized as tests) will be run.

1.4.3 Individual Testing

To run a single test, simply specify it at the command line. For example to run the test file *t/protocol/echo.t*, execute:

```
% t/TEST protocol/echo
```

Notice that you don't have to add the *t/* prefix and *.t* extension for the test filenames if you specify them explicitly, but you can have these as well. Therefore the following are all valid commands:

```
% t/TEST    protocol/echo.t
% t/TEST t/protocol/echo
% t/TEST t/protocol/echo.t
```

The server will be stopped if it was already running and a new one will be started before running the *t/protocol/echo.t* test. At the end of the test the server will be shut down.

When you run specific tests you may want to run them in the verbose mode, and depending on how the test was written, you may get more debug information under this mode. This mode is turned on with *-verbose* option:

```
% t/TEST -verbose protocol/echo
```

You can run groups of tests at once. This command:

```
% ./t/TEST modules protocol/echo
```

will run all the tests in *t/modules/* directory, followed by *t/protocol/echo.t* test.

1.4.4 Repetitive Testing

By default when you run the test without *-run-tests* option, the server will be started before the testing and stopped at the end. If during a debugging process you need to re-run tests without a need to restart the server, you can start the server once:

```
% t/TEST -start-httpd
```

and then run the test(s) with *-run-tests* option many times:

```
% t/TEST -run-tests
```

without waiting for the server to restart.

When you are done with tests, stop the server with:

```
% t/TEST -stop-httpd
```

When the server is started you can modify *.t* files and rerun the tests without restarting the server. However if you modify response handlers, you must restart the server for changes to take an effect. However the changes are done in the perl code only, it's possible to orrange for Apache::Test to handle the code reload without restarting the server.

The *-start-httpd* option always stops the server first if any is running.

Normally when *t/TEST* is run without specifying the tests to run, the tests will be sorted alphabetically. If tests are explicitly passed as arguments to *t/TEST* they will be run in a specified order.

1.4.5 Parallel Testing

Sometimes you need to run more than one Apache::Test framework instances at the same time. In this case you have to use different ports for each instance. You can specify explicitly which port to use, using the *-port* configuration option. For example to run the server on port 34343:

```
% t/TEST -start-httpd -port=34343
```

or by setting an environment variable *APACHE_TEST_PORT* to the desired value before starting the server.

Specifying the port explicitly may not be the most convenient option if you happen to run many instances of the `Apache::Test` framework. The `-port=select` option comes to help. This option will automatically pick for the next available port. For example if you run:

```
% t/TEST -start-httpd -port=select
```

and there is already one server from a different test suite which uses the default port 8529, the new server will try to use a higher port.

There is one problem that remains to be resolved though. It's possible that two or more servers running `-port=select` will still decide to use the same port, because when the server is configured it only tests whether the port is available but doesn't call `bind()` immediately. Therefore there is a race condition here, which needs to be resolved. Currently the workaround is to start the instances of the `Apache::Test` framework with a slight delay between each other. Depending on the speed of your machine, 4-5 seconds can be a good choice. That's approximately the time it takes to configure and start the server on a quite slow machine.

1.4.6 Verbose Mode

In case something goes wrong you should run the tests in the verbose mode:

```
% t/TEST -verbose
```

In this case the test may print useful information, like what values it expects and what values it receives, given that the test is written to report these. In the silent mode (without `-verbose`) these printouts are filtered out by `Test::Harness`. When running in the *verbose* mode usually it's a good idea to run only problematic tests to minimize the size of the generated output.

When debugging problems it helps to keep the *error_log* file open in another console, and see the debug output in the real time via `tail(1)`:

```
% tail -f t/logs/error_log
```

Of course this file gets created only when the server starts, so you cannot run `tail(1)` on it before the server starts. Every time `t/TEST -clean` is run, *t/logs/error_log* gets deleted, therefore you have to run the `tail(1)` command again, when the server is started.

1.4.7 Colored Trace Mode

If your terminal supports colored text you may want to set the environment variable `APACHE_TEST_COLOR` to 1 to enable the colored tracing when running in the non-batch mode, which makes it easier to tell the reported errors and warnings, from the rest of the notifications.

1.4.8 Controlling the Apache::Test's Signal to Noise Ratio

In addition to controlling the verbosity of the test scripts, you can control the amount of information printed by the `Apache::Test` framework itself. Similar to Apache's log levels, `Apache::Test` uses these levels for controlling its signal to noise ratio:

```
emerg alert crit error warning notice info debug
```

where *emerg* is the for the most important messages and *debug* for the least important ones.

Currently the default level is *info*, therefore any messages which fall into the *info* category and above (*notice*, *warning*, etc). This tracing level is unrelated to the Apache's `LogLevel` mechanism, which Apache-Test sets to `debug` in `t/conf/httpd.conf` and you can override it `t/conf/extra.conf.in`.

Let's assume you have the following code snippet:

```
use Apache::TestTrace;
warning "careful, perl on the premises";
debug "that's just silly";
```

If you want to get only *warning* messages and above, use:

```
% t/TEST -trace=warning ...
```

now only the warning message

```
careful, perl on the premises
```

will be printed. If you want to see the *debug* messages you can change the default level using *-trace* option:

```
% t/TEST -trace=debug ...
```

now the last example will print both messages.

By default the messages are printed to `STDERR`, but can be redirected to a file. Refer to the `Apache::TestTrace` manpage for more information.

Finally you can use methods: `emerg()`, `alert()`, `crit()`, `error()`, `warning()`, `notice()`, `info()` and `debug()` in your client and server side code. This is useful for example if you have some debug tracing that you don't want to be printed during the normal `make test`. However if some users have a problem you can ask them to run the test suite with the trace level of 'debug' and voila they can send you the extra debug output. Moreover all these functions use `Data::Dumper` to dump arguments which are references to perl structures. So for example your code may look like:

```
use Apache::TestTrace;
...
my $data = { foo => bar };
debug "my data", $data;
```

and only when run with `-trace=debug` it'll output.

```
my data
$VAR1 = {
    'foo' => 'bar'
};
```

Normally it will not print anything.

1.4.9 Stress Testing

1.4.9.1 The Problem

When we try to test a stateless machine (i.e. all tests are independent), running all tests once ensures that all tested things properly work. However when a state machine is tested (i.e. where a run of one test may influence another test) it's not enough to run all the tests once to know that the tested features actually work. It's quite possible that if the same tests are run in a different order and/or repeated a few times, some tests may fail. This usually happens when some tests don't restore the system under test to its pristine state at the end of the run, which may influence other tests which rely on the fact that they start on pristine state, when in fact it's not true anymore. In fact it's possible that a single test may fail when run twice or three times in a sequence.

1.4.9.2 The Solution

To reduce the possibility of such dependency errors, it's important to run random testing repeated many times with many different pseudo-random engine initialization seeds. Of course if no failures get spotted that doesn't mean that there are no tests inter-dependencies, unless all possible combinations were run (exhaustive approach). Therefore it's possible that some problems may still be seen in production, but this testing greatly minimizes such a possibility.

The `Apache::Test` framework provides a few options useful for stress testing.

- **-times**

You can run the tests N times by using the `-times` option. For example to run all the tests 3 times specify:

```
% t/TEST -times=3
```

- **-order**

It's possible that certain tests aren't cleaning up after themselves and modify the state of the server, which may influence other tests. But since normally all the tests are run in the same order, the potential problem may not be discovered until the code is used in production, where the real world testing hits the problem. Therefore in order to try to detect as many problems as possible during the testing process, it's may be useful to run tests in different orders.

This is of course mostly useful in conjunction with *-times=N* option.

Assuming that we have tests a, b and c:

- **-order=rotate**

rotate the tests: a, b, c, a, b, c

- **-order=repeat**

repeat the tests: a, a, b, b, c, c

- **-order=random**

run in the random order, e.g.: a, c, c, b, a, b

In this mode the seed picked by `srand()` is printed to `STDOUT`, so it then can be used to rerun the tests in exactly the same order (remember to log the output).

- **-order=SEED**

used to initialize the pseudo-random algorithm, which allows to reproduce the same sequence of tests. For example if we run:

```
% t/TEST -order=random -times=5
```

and the seed 234559 is used, we can repeat the same order of tests, by running:

```
% t/TEST -order=234559 -times=5
```

Alternatively, the environment variable `APACHE_TEST_SEED` can be set to the value of a seed when *-order=random* is used. e.g. under `bash(1)`:

```
% APACHE_TEST_SEED=234559 t/TEST -order=random -times=5
```

or with any shell program if you have the `env(1)` utility:

```
$ env APACHE_TEST_SEED=234559 t/TEST -order=random -times=5
```

1.4.9.3 Resolving Sequence Problems

When this kind of testing is used and a failure is detected there are two problems:

1. First is to be able to reproduce the problem so if we think we fixed it, we could verify the fix. This one is easy, just remember the sequence of tests run till the failed test and rerun the same sequence once again after the problem has been fixed.
2. Second is to be able to understand the cause of the problem. If during the random test the failure has happened after running 400 tests, how can we possibly know which previously running tests has caused to the failure of the test 401. Chances are that most of the tests were clean and don't have inter-dependency problem. Therefore it'd be very helpful if we could reduce the long sequence to a

minimum. Preferably 1 or 2 tests. That's when we can try to understand the cause of the detected problem.

1.4.9.4 Apache::TestSmoke Solution

Apache::TestSmoke attempts to solve both problems. When it's run, at the end of each iteration it reports the minimal sequence of tests causing a failure. This doesn't always succeed, but works in many cases.

You should create a small script to drive Apache::TestSmoke, usually *t/SMOKE.PL*. If you don't have it already, create it:

```
#file:t/SMOKE.PL
#-----
#!perl

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

use FindBin;
use lib "$FindBin::Bin/../../Apache-Test/lib";
use lib "$FindBin::Bin/../../lib";

use Apache::TestSmoke ();

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

Usually *Makefile.PL* converts it into *t/SMOKE* while adjusting the perl path, but you can create *t/SMOKE* in first place as well.

t/SMOKE performs the following operations:

1. Runs the tests randomly until the first failure is detected. Or non-randomly if the option *-order* is set to *repeat* or *rotate*.
2. Then it tries to reduce that sequence of tests to a minimum, and this sequence still causes to the same failure.
3. It reports all the successful reductions as it goes to STDOUT and report file of the format: *smoke-report-<date>.txt*.

In addition the systems build parameters are logged into the report file, so the detected problems could be reproduced.

4. Goto 1 and run again using a new random seed, which potentially should detect different failures.

Currently for each reduction path, the following reduction algorithms are applied:

1. Binary search: first try the upper half then the lower.

2. Random window: randomize the left item, then the right item and return the items between these two points.

You can get the usage information by executing:

```
% t/SMOKE -help
```

By default you don't need to supply any arguments to run it, simply execute:

```
% t/SMOKE
```

If you want to work on certain tests you can specify them in the same way you do with *t/TEST*:

```
% t/SMOKE foo/bar foo/tar
```

If you already have a sequence of tests that you want to reduce (perhaps because a previous run of the smoke testing didn't reduce the sequence enough to be able to diagnose the problem), you can request to do just that:

```
% t/SMOKE -order=rotate -times=1 foo/bar foo/tar
```

-order=rotate is used just to override the default *-order=random*, since in this case we want to preserve the order. We also specify *-times=1* for the same reason (override the default which is 50).

You can override the number of `srand()` iterations to perform (read: how many times to randomize the sequence), the number of times to repeat the tests (the default is 10) and the path to the file to use for reports:

```
% t/SMOKE -times=5 -iterations=20 -report=../myreport.txt
```

Finally, any other options passed will be forwarded to *t/TEST* as is.

1.4.10 RunTime Configuration Overriding

After the server is configured during `make test` or with *t/TEST -config*, it's possible to explicitly override certain configuration parameters. The override-able parameters are listed when executing:

```
% t/TEST -help
```

Probably the most useful parameters are:

- **-preamble**

configuration directives to add at the beginning of *httpd.conf*. For example to turn the tracing on:

```
% t/TEST -preamble "PerlTrace all"
```

- **-postamble**

configuration directives to add at the end of *httpd.conf*. For example to load a certain Perl module:

```
% t/TEST -postamble "PerlModule MyDebugMode"
```

- **-user**

run as user *nobody*:

```
% t/TEST -user nobody
```

- **-port**

run on a different port:

```
% t/TEST -port 8799
```

- **-servername**

run on a different server:

```
% t/TEST -servername test.example.com
```

- **-httpd**

configure an httpd other than the default (that apxs figures out):

```
% t/TEST -httpd ~/httpd-2.0/httpd
```

- **-apxs**

switch to another apxs:

```
% t/TEST -apxs ~/httpd-2.0-prefork/bin/apxs
```

For a complete list of override-able configuration parameters see the output of `t/TEST -help`.

1.4.11 Request Generation and Response Options

We have mentioned already the most useful run-time options. Here are some other options that you may find useful during testing.

- **-ping**

Ping the server to see whether it runs

```
% t/TEST -ping
```

Ping the server and wait until the server starts, report waiting time.

```
% t/TEST -ping=block
```

This can be useful in conjunction with *-run-tests* option during debugging:

```
% t/TEST -ping=block -run-tests
```

normally, *-run-tests* will immediately quit if it detects that the server is not running, but with *-ping=block* in effect, it'll wait indefinitely for the server to start up.

- **-head**

Issue a HEAD request. For example to request */server-info*:

```
% t/TEST -head /server-info
```

- **-get**

Request the body of a certain URL via GET.

```
% t/TEST -get /server-info
```

If no URL is specified */* is used.

Also you can issue a GET request but to get only headers as a response (e.g. useful to just check *Content-length*)

```
% t/TEST -head -get /server-info
```

GET URL with authentication credentials:

```
% t/TEST -get /server-info -username dougm -password domination
```

(please keep the password secret!)

- **-post**

Generate a POST request.

Read content to POST from string:

```
% t/TEST -post /TestApache__post -content 'name=dougm&company=covalent'
```

Read content to POST from STDIN:

```
% t/TEST -post /TestApache__post -content - < foo.txt
```

Generate a content body of 1024 bytes in length:

```
% t/TEST -post /TestApache__post -content x1024
```

The same but print only the response headers, e.g. useful to just check *Content-length*:

```
% t/TEST -post -head /TestApache__post -content x1024
```

- **-header**

Add headers to (-get|-post|-head) request:

```
% t/TEST -get -header X-Test=10 -header X-Host=example.com /server-info
```

- **-ssl**

Run all tests through mod_ssl:

```
% t/TEST -ssl
```

- **-http11**

Run all tests with HTTP/1.1 (KeepAlive) requests:

```
% t/TEST -http11
```

- **-proxy**

Run all tests through mod_proxy:

```
% t/TEST -proxy
```

-

The debugging options *-debug* and *-breakpoint* are covered in the Debugging Tests section.

For a complete list of available switches see the output of `t/TEST -help`.

1.4.12 Batch Mode

When running in the batch mode and redirecting STDOUT, this state is automatically detected and the *no color* mode is turned on, under which the program generates a minimal output to make the log files useful. If this doesn't work and you still get all the mess printed during the interactive run, set the `APACHE_TEST_NO_COLOR=1` environment variable.

1.5 Setting Up Testing Environment

We will assume that you setup your testing environment even before you have started coding the project, which is a very smart thing to do. Of course it'll take you more time upfront, but it'll will save you a lot of time during the project developing and debugging stages. The extreme programming methodology says that tests should be written before starting the code development.

1.5.1 Know Your Target Environment

In the following demonstration and mostly through the whole document we assume that the test suite is written for a module running under `mod_perl 2.0`. You may need to adjust the code and the configuration files to the `mod_perl 1.0` syntax, if you work with that generation of `mod_perl`. If your test suite needs to work with both `mod_perl` generations refer to the porting to `mod_perl 2.0` chapter. Of course it's quite possible that what you test doesn't have `mod_perl` at all, in which case, again, you will need to make adjustments to work in the given environment.

1.5.2 Basic Testing Environment

So the first thing is to create a package and all the helper files, so later on we can distribute it on CPAN. We are going to develop an `Apache::Amazing` module as an example.

```
% h2xs -AXn Apache::Amazing
Writing Apache/Amazing/Amazing.pm
Writing Apache/Amazing/Makefile.PL
Writing Apache/Amazing/README
Writing Apache/Amazing/test.pl
Writing Apache/Amazing/Changes
Writing Apache/Amazing/MANIFEST
```

`h2xs` is a nifty utility that gets installed together with Perl and helps us to create some of the files we will need later.

However we are going to use a little bit different files layout, therefore we are going to move things around a bit.

We want our module to live in the *Apache-Amazing* directory, so we do:

```
% mv Apache/Amazing Apache-Amazing
% rmdir Apache
```

From now on the *Apache-Amazing* directory is our working directory.

```
% cd Apache-Amazing
```

We don't need the *test.pl*, as we are going to create a whole testing environment:

```
% rm test.pl
```

We want our package to reside under the *lib* directory, so later we will be able to do live testing, without rerunning make every time we change the code:

```
% mkdir lib
% mkdir lib/Apache
% mv Amazing.pm lib/Apache
```

Now we adjust the *lib/Apache/Amazing.pm* to look like this:

```
#file:lib/Apache/Amazing.pm
#-----
package Apache::Amazing;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();

$Apache::Amazing::VERSION = '0.01';

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    $r->content_type('text/plain');
    $r->print("Amazing!");
    return Apache::OK;
}
1;
__END__
... pod documentation goes here...
```

The only thing it does is setting the *text/plain* header and responding with *"Amazing!"*.

Next adjust or create the *Makefile.PL* file:

```
#file:Makefile.PL
#-----
require 5.6.1;

use ExtUtils::MakeMaker;

use lib qw(../bilib/lib lib );

use Apache::TestMM qw(test clean); #enable 'make test'

# prerequisites
my %require =
(
    "Apache::Test" => "", # any version will do
);
my @scripts = qw(t/TEST);

# accept the configs from command line
Apache::TestMM::filter_args();
Apache::TestMM::generate_script('t/TEST');

WriteMakefile(
    NAME          => 'Apache::Amazing',
    VERSION_FROM  => 'lib/Apache/Amazing.pm',
    PREREQ_PM     => \%require,
    clean         => {
```



```

        FILES => "@{ clean_files() }",
    },
    ($] >= 5.005 ?
        (ABSTRACT_FROM => 'lib/Apache/Amazing.pm',
          AUTHOR         => 'Stas Bekman <stas (at) stason.org>',
        ) : ()
    ),
);

sub clean_files {
    return [@scripts];
}

```

Apache::TestMM will do a lot of thing for us, such as building a complete Makefile with proper *'test'* and *'clean'* targets, automatically converting *.PL* and *conf/*.in* files and more.

As you see we specify a prerequisites hash with *Apache::Test* in it, so if the package gets distributed on CPAN, CPAN.pm shell will know to fetch and install this required package.

Next we create the test suite, which will reside in the *t* directory:

```
% mkdir t
```

First we create *t/TEST.PL* which will be automatically converted into *t/TEST* during *perl Makefile.PL* stage:

```

#file:t/TEST.PL
#-----
#!perl

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

use lib qw(lib);

use Apache::TestRunPerl ();

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

```

Assuming that *Apache::Test* is already installed on your system and Perl can find it. If not you should tell Perl where to find it. For example you could add:

```
use lib qw(Apache-Test/lib);
```

to *t/TEST.PL*, if *Apache::Test* is located in a parallel directory.

As you can see we didn't write the real path to the Perl executable, but *#!perl*. When *t/TEST* is created the correct path will be placed there automatically.

Next we need to prepare extra Apache configuration bits, which will reside in *t/conf*:

```
% mkdir t/conf
```

We create the *t/conf/extra.conf.in* file which will be automatically converted into *t/conf/extra.conf* before the server starts. If the file has any placeholders like *@documentroot@*, these will be replaced with the real values specific for the used server. In our case we put the following configuration bits into this file:

```
#file:t/conf/extra.conf.in
#-----
# this file will be Include-d by @ServerRoot@/httpd.conf

# where Apache::Amazing can be found
PerlSwitches -I@ServerRoot@/../lib
# preload the module
PerlModule Apache::Amazing
<Location /test/amazing>
    SetHandler modperl
    PerlResponseHandler Apache::Amazing
</Location>
```

As you can see we just add a simple *<Location>* container and tell Apache that the namespace */test/amazing* should be handled by *Apache::Amazing* module running as a *mod_perl* handler. Notice that:

```
SetHandler modperl
```

is *mod_perl* 2.0 configuration, if you are running under *mod_perl* 1.0 use:

```
SetHandler perl-script
```

which also works for *mod_perl* 2.0.

Now we can create a simple test:

```
#file:t/basic.t
#-----
use strict;
use warnings FATAL => 'all';

use Apache::Amazing;
use Apache::Test;
use Apache::TestUtil;
use Apache::TestRequest 'GET_BODY';

plan tests => 2;

ok 1; # simple load test

my $url = '/test/amazing';
my $data = GET_BODY $url;

ok t_cmp(
    "Amazing!",
    $data,
    "basic test",
);
```

Now create the *README* file.

```
% touch README
```

Don't forget to put in the relevant information about your module, or arrange for `ExtUtils::Maker::WriteMakefile()` to do this for you with:

```
#file:Makefile.PL
#-----
WriteMakefile(
    #...
    dist => {
        PREOP => 'pod2text lib/Apache/Amazing.pm > $(DISTVNAME)/README',
    },
    #...
);
```

in this case *README* will be created from the documentation POD sections in *lib/Apache/Amazing.pm*, but the file has to exist for *make dist* to succeed.

and finally we adjust or create the *MANIFEST* file, so we can prepare a complete distribution. Therefore we list all the files that should enter the distribution including the *MANIFEST* file itself:

```
#file:MANIFEST
#-----
lib/Apache/Amazing.pm
t/TEST.PL
t/basic.t
t/conf/extra.conf.in
Makefile.PL
Changes
README
MANIFEST
```

That's it. Now we can build the package. But we need to know the location of the *apxs* utility from the installed httpd server. We pass its path as an option to *Makefile.PL*:

```
% perl Makefile.PL -apxs ~/httpd/prefork/bin/apxs
% make
% make test

basic.....ok
All tests successful.
Files=1, Tests=2, 1 wallclock secs ( 0.52 cusr + 0.02 csys = 0.54 CPU)
```

To install the package run:

```
% make install
```

Now we are ready to distribute the package on CPAN:

```
% make dist
```

will create the package which can be immediately uploaded to CPAN. In this example the generated source package with all the required files will be called: *Apache-Amazing-0.01.tar.gz*.

The only thing that we haven't done and hope that you will do is to write the POD sections for the `Apache::Amazing` module, explaining how amazingly it works and how amazingly it can be deployed by other users.

1.5.3 Extending Configuration Setup

Sometimes you need to add extra *httpd.conf* configuration and perl startup specific to your project that uses `Apache::Test`. This can be accomplished by creating the desired files with an extension *.in* in the *t/conf/* directory and running:

```
panic% t/TEST -config
```

which for each file with the extension *.in* will create a new file, without this extension, convert any template placeholders into real values and link it from the main *httpd.conf*. The latter happens only if the file have the following extensions:

- **.conf.in**

will add to *t/conf/httpd.conf*:

```
Include foo.conf
```

- **.pl.in**

will add to *t/conf/httpd.conf*:

```
PerlRequire foo.pl
```

- **other**

other files with *.in* extension will be processed as well, but not linked from *httpd.conf*.

Files whose name matches the following pattern:

```
/\..last\.(conf|pl).in$/
```

will be included very last in *httpd.conf*.

As mentioned before the converted files are created, any special token in them are getting replaced with the appropriate values. For example the token `@ServerRoot@` will be replaced with the value defined by the `ServerRoot` directive, so you can write a file that does the following:

```
#file:my-extra.conf.in
#-----
PerlSwitches -I@ServerRoot@/../lib
```

and assuming that the *ServerRoot* is *~/modperl-2.0/t/*, when *my-extra.conf* will be created, it'll look like:

```
#file:my-extra.conf
#-----
PerlSwitches -I~/modperl-2.0/t/./lib
```

The valid tokens are defined in `%Apache::TestConfig::Usage` and also can be seen in the output of `t/TEST -help`'s *configuration options* section. The tokens are case insensitive.

1.5.4 Special Configuration Files

Some of the files in the *t/conf* directory have a special meaning, since the `Apache::Test` framework uses them for the minimal configuration setup. But they can be overridden:

- if the file *t/conf/httpd.conf.in* exists, it will be used instead of the default template (in *Apache/Test-Config.pm*).
- if the file *t/conf/extra.conf.in* exists, it will be used to generate *t/conf/extra.conf* with `@variable@` substitutions.
- if the file *t/conf/extra.conf* exists, it will be included by *httpd.conf*.
- if the file *t/conf/modperl_extra.pl* exists, it will be included by *httpd.conf* as a `mod_perl` file (`PerlRequire`).

1.5.5 Inheriting from System-wide httpd.conf

`Apache::Test` tries to find a global *httpd.conf* file and inherit its configuration when autogenerating *t/conf/httpd.conf*. For example it picks `LoadModule` directives.

It's possible to explicitly specify which file to inherit from using the `-httpd_conf` option. For example during the build:

```
% perl Makefile.PL -httpd_conf /path/to/httpd.conf
```

or during the configuration:

```
% t/TEST -conf -httpd_conf /path/to/httpd.conf
```

Certain projects need to have a control of what gets inherited. For example if your global *httpd.conf* includes a directive:

```
LoadModule apreq_module "/home/joe/apache2/modules/mod_apreq.so"
```

And you want to run the test suite for `Apache::Request 2.0`, inheriting the above directive will load the pre-installed *mod_apreq.so* and not the newly built one, which is wrong. In such cases it's possible to tell the test suite which modules shouldn't be inherited. In our example `Apache-Request` has the following code in *t/TEST.PL*:

```

use base 'Apache::TestRun';
$Apache::TestTrace::Level = 'debug';
main:-->new->run(@ARGV);

sub pre_configure {
    my $self = shift;
    # Don't load an installed mod_apreq
    Apache::TestConfig::autoconfig_skip_module_add('mod_apreq.c');
}

```

it subclasses `Apache::TestRun` and overrides the `pre_configure` method, which excludes the module `mod_apreq.c` from the list of inherited modules (notice that the extension is `.c`).

1.6 Apache::Test Framework's Architecture

In the previous section we have written a basic test, which doesn't do much. In the following sections we will explain how to write more elaborate tests.

When you write the test for Apache, unless you want to test some static resource, like fetching a file, usually you have to write a response handler and the corresponding test that will generate a request which will exercise this response handler and verify that the response is as expected. From now we may call these two parts as client and server parts of the test, or request and response parts of the test.

In some cases the response part of the test runs the test inside itself, so all it requires from the request part of the test, is to generate the request and print out a complete response without doing anything else. In such cases `Apache::Test` can auto-generate the client part of the test for you.

1.6.1 Developing Response-only Part of a Test

If you write only a response part of the test, `Apache::Test` will automatically generate the corresponding test part that will generate the response. In this case your test should print `'ok 1'`, `'not ok 2'` responses as usual tests do. The autogenerated request part will receive the response and print them out automatically completing the `Test::Harness` expectations.

The corresponding request part of the test is named just like the response part, using the following translation:

```
$response_test =~ s|t/[^\s]+/Test([^\s]+)/(.*)\.pm$|t/\L$1\E/$2.t|;
```

so for example `t/response/TestApache/write.pm` becomes: `t/apache/write.t`.

If we look at the autogenerated test `t/apache/write.t`, we can see that it starts with the warning that it has been autogenerated, so you won't attempt to change it. Then you can see the trace of the calls that generated this test, in case you want to figure out how the test was generated. And finally the test loads the `Apache::TestRequest` module, imports the GET shortcut and prints the response's body if it was successful. Otherwise it dies to flag the problem with the server side. The latter is done because there is nothing on the client side, that tells the testing framework that things went wrong. Without it the test will be skipped, and that's not what we want.

```
use Apache::TestRequest 'GET_BODY_ASSERT';
print GET_BODY_ASSERT "/TestApache__write";
```

As you can see the request URI is autogenerated from the response test name:

```
$response_test =~ s|.*(?:[^\w]+)/(.*)\.pm$|/$1__$2|;
```

So *t/response/TestApache/write.pm* becomes: */TestApache__write*.

Now a simple response test may look like this:

```
#file:t/response/TestApache/write.pm
#-----
package TestApache::write;

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

use constant BUFSIZ => 512; #small for testing
use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    $r->content_type('text/plain');

    $r->write("1..2\n");
    $r->write("ok 1")
    $r->write("not ok 2")

    Apache::OK;
}
1;
```

[F] `Apache::Const` is `mod_perl` 2.0's package, if you test under 1.0, use the `Apache::Constants` module instead [/F].

The configuration part for this test will be autogenerated by the `Apache::Test` framework and added to the autogenerated file *t/conf/httpd.conf* when `make test` or `t/TEST -configure` is run. In our case the following configuration section will be added:

```
<Location /TestApache__write>
    SetHandler modperl
    PerlResponseHandler TestApache::write
</Location>
```

You should remember to run:

```
% t/TEST -configure
```

so the configuration file will be re-generated when new tests are added.

Also notice that if you manually add configuration the `<Location>` path can't include `' : '` characters in the first segment, due to Apache security protection on WinFU platforms. So please make sure that you don't create entries like:

```
<Location /Foo::bar/>
```

You can include `' : '` characters in the further segments, so this is OK:

```
<Location /tests/Foo::bar/>
```

Of course if your code is not intended to run on WinFU you can ignore this detail.

1.6.2 Developing Response and Request Parts of a Test

But in most cases you want to write a two parts test where the client (request) parts generates various requests and tests the responses.

It's possible that the client part tests a static file or some other feature that doesn't require a dynamic response. In this case, only the request part of the test should be written.

If you need to write the complete test, with two parts, you proceed just like in the previous section, but now you write the client part of the test by yourself. It's quite easy, all you have to do is to generate requests and check the response. So a typical test will look like this:

```
#file:t/apache/cool.t
#-----
use strict;
use warnings FATAL => 'all';

use Apache::Test;
use Apache::TestUtil;
use Apache::TestRequest 'GET_BODY';

plan tests => 1; # plan one test.

Apache::TestRequest::module('default');

my $config = Apache::Test::config();
my $hostport = Apache::TestRequest::hostport($config) || '';
t_debug("connecting to $hostport");

my $received = GET_BODY "/TestApache__cool";
my $expected = "COOL";

ok t_cmp(
    $expected,
    $received,
    "testing TestApache::cool",
);
```


See the `Apache::TestUtil` manpage for more info on the `t_cmp()` function (e.g. it works with regexs as well).

And the corresponding response part:

```
#file:t/response/TestApache/cool.pm
#-----
package TestApache::cool;

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

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    $r->content_type('text/plain');

    $r->write("COOL");

    Apache::OK;
}
1;
```

Again, remember to run `t/TEST -clean` before running the new test so the configuration will be created for it.

As you can see the test generates a request to `/TestApache__cool`, and expects it to return `"COOL"`. If we run the test:

```
% ./t/TEST t/apache/cool
```

We see:

```
apache/cool....ok
All tests successful.
Files=1, Tests=1, 1 wallclock secs ( 0.52 cusr + 0.02 csys = 0.54 CPU)
```

But if we run it in the debug (verbose) mode, we can actually see what we are testing, what was expected and what was received:

```
apache/cool...1..1
# connecting to localhost:8529
# testing : testing TestApache::cool
# expected: COOL
# received: COOL
ok 1
ok
All tests successful.
Files=1, Tests=1, 1 wallclock secs ( 0.49 cusr + 0.03 csys = 0.52 CPU)
```

So in case in our simple test we have received something different from `COOL` or nothing at all, we can immediately see what's the problem.

The name of the request part of the test is very important. If `Apache::Test` cannot find the corresponding test for the response part it'll automatically generate one and in this case it's probably not what you want. Therefore when you choose the filename for the test, make sure to pick the same `Apache::Test` will pick. So if the response part is named: `t/response/TestApache/cool.pm` the request part should be named `t/apache/cool.t`. See the regular expression that does that in the previous section.

1.6.3 Developing Test Response Handlers in C

If you need to exercise some C API and you don't have a Perl glue for it, you can still use `Apache::Test` for the testing. It allows you to write response handlers in C and makes it easy to integrate these with other Perl tests and use Perl for request part which will exercise the C module.

The C modules look just like standard Apache C modules, with a couple of differences to:

- **a**
help them fit into the test suite
- **b**
allow them to compile nicely with Apache 1.x or 2.x.

The `httpd-test` ASF project is a good example to look at. The C modules are located under: `httpd-test/perl-framework/c-modules/`. Look at `c-modules/echo_post/echo_post.c` for a nice simple example. `mod_echo_post` simply echos data that is POSTed to it.

The differences between various tests may be summarized as follows:

- If the first line is:

```
#define HTTPD_TEST_REQUIRE_APACHE 1
```

or

```
#define HTTPD_TEST_REQUIRE_APACHE 2
```

then the test will be skipped unless the version matches. If a module is compatible with the version of Apache used then it will be automatically compiled by `t/TEST` with `-DAPACHE1` or `-DAPACHE2` so you can conditionally compile it to suit different httpd versions.

In addition to the single-digit form,

```
#define HTTPD_TEST_REQUIRE_APACHE 2.0.48
```

and

```
#define HTTPD_TEST_REQUIRE_APACHE 2.1
```

are also supported, allowing for conditional compilation based on criteria similar to *have_min_apache_version()*.

- If there is a section bounded by:

```
#if CONFIG_FOR_HTTPD_TEST
...
#endif
```

in the *.c* file then that section will be inserted verbatim into *t/conf/httpd.conf* by *t/TEST*.

There is a certain amount of magic which hopefully allows most modules to be compiled for Apache 1.3 or Apache 2.0 without any conditional stuff. Replace XXX with the module name, for example *echo_post* or *random_chunk*:

- You should:

```
#include "apache_httpd_test.h"
```

which should be preceded by an:

```
#define APACHE_HTTPD_TEST_HANDLER XXX_handler
```

apache_httpd_test.h pulls in a lot of required includes and defines some constants and types that are not defined for Apache 1.3.

- The handler function should be:

```
static int XXX_handler(request_rec *r);
```

- At the end of the file should be an:

```
APACHE_HTTPD_TEST_MODULE(XXX)
```

where XXX is the same as that in *APACHE_HTTPD_TEST_HANDLER*. This will generate the hooks and stuff.

1.6.4 Request and Response Methods

If you have LWP (libwww-perl) installed its *LWP::UserAgent* serves as an user agent in tests, otherwise *Apache::TestClient* tries to emulate partial LWP functionality. So most of the LWP documentation applies here, but the *Apache::Test* framework provides shortcuts that hide many details, making the test writing a simple and swift task. Before using these shortcuts *Apache::TestRequest* should be loaded, and its *import()* method will fetch the shortcuts into the caller namespace:

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

Request generation methods issue a request and return a response object (*HTTP::Response* if LWP is available). They are documented in the *HTTP::Request::Common* manpage. The following methods are available:

- **GET**

Issues the GET request. For example, issue a request and retrieve the response content:

```
$url = "$location?foo=1&bar=2";  
$res = GET $url;  
$str = $res->content;
```

To set request headers, supply them after the `$url`, e.g.:

```
$res = GET $url, 'Content-type' => 'text/html';
```

- **HEAD**

Issues the HEAD request. For example issue a request and check that the response's *Content-type* is *text/plain*:

```
$url = "$location?foo=1&bar=2";  
$res = HEAD $url;  
ok $res->content_type() eq 'text/plain';
```

- **POST**

Issues the POST request. For example:

```
$content = 'PARAM=%33';  
$res = POST $location, content => $content;
```

The second argument to POST can be a reference to an array or a hash with key/value pairs to emulate HTML `<form>` POSTing.

- **PUT**

Issues the PUT request.

- **OPTIONS**

META: ???

These are two special methods added by the `Apache::Test` framework:

- **UPLOAD**

This special method allows to upload a file or a string which will look as an uploaded file to the server. To upload a file use:

```
UPLOAD $location, filename => $filename;
```

You can add extra request headers as well:

```
UPLOAD $location, filename => $filename, 'X-Header-Test' => 'Test';
```

To upload a string as a file, use:

```
UPLOAD $location, content => 'some data';
```

● UPLOAD_BODY

Retrieves the content from the response resulted from doing UPLOAD. It's equal to:

```
my $body = UPLOAD(@_)->content;
```

For example, this code retrieves the content of the response resulted from file upload request:

```
my $str = UPLOAD_BODY $location, filename => $filename;
```

Once the response object is returned, various response object methods can be applied to it. Probably the most useful ones are:

```
$content = $res->content;
```

to retrieve the content of the response and:

```
$content_type = $res->header('Content-type');
```

to retrieve specific headers.

Refer to the `HTTP::Response` manpage for a complete reference of these and other methods.

A few response retrieval shortcuts can be used to retrieve the wanted parts of the response. To apply these simply add the shortcut name to one of the request shortcuts listed earlier. For example instead of retrieving the content part of the response via:

```
$res = GET $url;
$str = $res->content;
```

simply use:

```
$str = GET_BODY $url;
```

● RC

returns the *response code*, equivalent to:

```
$res->code;
```

For example to test whether some URL is bogus:

```
use Apache::Const 'NOT_FOUND';
ok GET_RC('/bogus_url') == NOT_FOUND;
```

You usually need to import and use `Apache::Const` constants for the response code comparisons, rather than using codes' corresponding numerical values directly. You can import groups of code as well. For example:

```
use Apache::Const ':common';
```

Refer to the `Apache::Const` manpage for a complete reference. Also you may need to use `APR` and `mod_perl` constants, which reside in `APR::Const` and `ModPerl::Const` modules respectively.

- **OK**

tests whether the response was successful, equivalent to:

```
$res->is_success;
```

For example:

```
ok GET_OK '/foo';
```

- **STR**

returns the response (both, headers and body) as a string and is equivalent to:

```
$res->as_string;
```

Mostly useful for debugging, for example:

```
use Apache::TestUtil;
t_debug POST_STR '/test.pl', content => 'foo';
```

- **HEAD**

returns the headers part of the response as a multi-line string.

For example, this code dumps all the response headers:

```
use Apache::TestUtil;
t_debug GET_HEAD '/index.html';
```

- **BODY**

returns the response body and is equivalent to:

```
$res->content;
```

For example, this code validates that the response's body is the one that was expected:

```
use Apache::TestUtil;
ok GET_BODY('/index.html') eq $expect;
```

- **BODY_ASSERT**

Same as the BODY shortcut, but will assert if the request has failed. So for example if the test's output is generated on the server side, the client side may only need to print out what the server has sent and we want it to report that the test has failed if the request has failed:

```
use Apache::TestUtil;
print GET_BODY_ASSERT "/foo"
```

1.6.5 Other Request Generation helpers

META: these methods need documentation

Request part:

```
Apache::TestRequest::scheme('http'); #force http for t/TEST -ssl
Apache::TestRequest::module($module);
my $config = Apache::Test::config();
my $hostport = Apache::TestRequest::hostport($config);
```

Getting the request object? `Apache::TestRequest::user_agent()`

1.6.6 Starting Multiple Servers

By default the `Apache::Test` framework sets up only a single server to test against.

In some cases you need to have more than one server. If this is the situation, you have to override the `maxclients` configuration directive, whose default is 1. Usually this is done in `t/TEST.PL` by subclassing the parent test run class and overriding the `new_test_config()` method. For example if the parent class is `Apache::TestRunPerl`, you can change your `t/TEST.PL` to be:

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

use lib "../lib"; # test against the source lib for easier dev
use lib map {"../blib/$_", "../../blib/$_"} qw(lib arch);

use Apache::TestRunPerl ();

package MyTest;

our @ISA = qw(Apache::TestRunPerl);

# subclass new_test_config to add some config vars which will be
# replaced in generated httpd.conf
sub new_test_config {
    my $self = shift;

    $self->{conf_opts}->{maxclients} = 2;

    return $self->SUPER::new_test_config;
}

MyTest->new->run(@ARGV);
```

1.6.7 Multiple User Agents

By default the `Apache::Test` framework uses a single user agent which talks to the server (this is the LWP user agent, if you have LWP installed). You almost never use this agent directly in the tests, but via various wrappers. However if you need a second user agent you can clone these. For example:

```
my $ua2 = Apache::TestRequest::user_agent()->clone;
```

1.6.8 Hitting the Same Interpreter (Server Thread/Process Instance)

When a single instance of the server thread/process is running, all the tests go through the same server. However if the `Apache::Test` framework was configured to to run a few instances, two subsequent sub-tests may not hit the same server instance. In certain tests (e.g. testing the closure effect or the `BEGIN` blocks) it's important to make sure that a sequence of sub-tests are run against the same server instance. The `Apache::Test` framework supports this internally.

Here is an example from `ModPerl::Registry` closure tests. Using the counter closure problem under `ModPerl::Registry`:

```
#file:cgi-bin/closure.pl
#-----
#!perl -w
print "Content-type: text/plain\r\n\r\n";

# this is a closure (when compiled inside handler()):
my $counter = 0;
counter();

sub counter {
    #warn "$$";
    print ++$counter;
}
```

If this script get invoked twice in a row and we make sure that it gets executed by the same server instance, the first time it'll return 1 and the second time 2. So here is the gist of the request part that makes sure that its two subsequent requests hit the same server instance:

```
#file:closure.t
#-----
...
my $url = "/same_interp/cgi-bin/closure.pl";
my $same_interp = Apache::TestRequest::same_interp_tie($url);

# should be no closure effect, always returns 1
my $first = req($same_interp, $url);
my $second = req($same_interp, $url);
ok t_cmp(
    1,
    $first && $second && ($second - $first),
    "the closure problem is there",
);
sub req {
```



```

my($same_interp, $url) = @_;
my $res = Apache::TestRequest::same_interp_do($same_interp,
                                              \&GET, $url);

return $res ? $res->content : undef;
}

```

In this test we generate two requests to *cgi-bin/closure.pl* and expect the returned value to increment for each new request, because of the closure problem generated by `ModPerl::Registry`. Since we don't know whether some other test has called this script already, we simply check whether the subtraction of the two subsequent requests' outputs gives a value of 1.

The test starts by requesting the server to tie a single instance to all requests made with a certain identifier. This is done using the `same_interp_tie()` function which returns a unique server instance's identifier. From now on any requests made through `same_interp_do()` and supplying this identifier as the first argument will be served by the same server instance. The second argument to `same_interp_do()` is the method to use for generating the request and the third is the URL to use. Extra arguments can be supplied if needed by the request generation method (e.g. headers).

This technique works for testing purposes where we know that we have just a few server instances. What happens internally is when `same_interp_tie()` is called the server instance that served it returns its unique UUID, so when we want to hit the same server instance in subsequent requests we generate the same request until we learn that we are being served by the server instance that we want. This magic is done by using a fixup handler which returns OK only if it sees that its unique id matches. As you understand this technique would be very inefficient in production with many server instances.

1.7 Writing Tests

All the communications between tests and `Test::Harness` which executes them is done via STDOUT. I.e. whatever tests want to report they do by printing something to STDOUT. If a test wants to print some debug comment it should do it starting on a separate line, and each debug line should start with `#`. The `t_debug()` function from the `Apache::TestUtil` package should be used for that purpose.

1.7.1 Defining How Many Sub-Tests Are to Be Run

Before sub-tests of a certain test can be run it has to declare how many sub-tests it is going to run. In some cases the test may decide to skip some of its sub-tests or not to run any at all. Therefore the first thing the test has to print is:

```
1..M\n
```

where `M` is a positive integer. So if the test plans to run 5 sub-tests it should do:

```
print "1..5\n";
```

In `Apache::Test` this is done as follows:

```
use Apache::Test;
plan tests => 5;
```

1.7.2 *Skipping a Whole Test*

Sometimes when the test cannot be run, because certain prerequisites are missing. To tell `Test::Harness` that the whole test is to be skipped do:

```
print "1..0 # skipped because of foo is missing\n";
```

The optional comment after `# skipped` will be used as a reason for test's skipping. Under `Apache::Test` the optional last argument to the `plan()` function can be used to define prerequisites and skip the test:

```
use Apache::Test;
plan tests => 5, $test_skipping_prerequisites;
```

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 `have()`:

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

- **an ARRAY reference**

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

```
plan tests => 5, [qw(mod_index mod_mime)];
```

- **a CODE reference**

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

```
plan tests => 5, \&have_lwp;
```

the test will be skipped if LWP is not available

There is a number of useful functions whose return value can be used as a last argument for `plan()`:

- **have_module()**

`have_module()` tests for presense of Perl modules or C modules *mod_**. It accepts a list of modules or a reference to the list. If at least one of the modules is not found it returns a false value, otherwise it returns a true value. For example:

```
plan tests => 5, have_module qw(Chatbot::Eliza CGI mod_proxy);
```

will skip the whole test unless both Perl modules `Chatbot::Eliza` and `CGI` and the C module *mod_proxy.c* are available.

- **have_min_module_version()**

Used to require a minimum version of a module

For example:

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

requires `CGI.pm` version 2.81 or higher.

Currently works only for perl modules.

- **have()**

`have()` called as a last argument of `plan()` can impose multiple requirements at once.

`have()`'s arguments can include scalars, which are passed to `have_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 valuees are the condition, which are satisfaction if they return true. If the value is a scalar it's used as is. 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.

For example:

```
plan tests => 5,
    have 'LWP',
        { "perl >= 5.8.0 is required" => ($] >= 5.008) },
        { "not Win32"                  => sub { $^O eq 'MSWin32' } },
        { "foo is disabled"            => \&is_foo_enabled,
        },
    'cgid';
```

In this example, we require the presense of the LWP Perl module, `mod_cgid`, that we run under perl `>= 5.7.3` on Win32, and that `is_foo_enabled` returns true. If any of the requirements from this list fail, the test will be skipped and each failed requiremnt will print a reason for its failure.

- **have_perl()**

`have_perl('foo')` checks whether the value of `$Config{foo}` or `$Config{usefoo}` is equal to *'define'*. For example:

```
plan tests => 2, have_perl 'ithreads';
```

if Perl wasn't compiled with `-Duseithreads` the condition will be false and the test will be skipped.

Also it checks for Perl extensions. For example:

```
plan tests => 5, have_perl 'iolayers';
```

tests whether `PerlIO` is available.

- **have_min_perl_version()**

Used to require a minimum version of Perl.

For example:

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

requires Perl 5.8.1 or higher.

- **have_threads()**

`have_threads` checks whether threads are supported by both Apache and Perl.

```
plan tests => 2, have_threads;
```

- **under_construction()**

this is just a shortcut to skip the test while printing:

```
"skipped: this test is under construction";
```

For example:

```
plan tests => 2, under_construction;
```

- **have_lwp()**

Tests whether the Perl module `LWP` is installed.

- **have_http11()**

Tries to tell `LWP` that sub-tests need to be run under HTTP 1.1 protocol. Fails if the installed version of `LWP` is not capable of doing that.

- **have_cgi()**

tests whether `mod_cgi` or `mod_cgid` is available.

- **have_apache()**

tests for a specific generation of httpd. For example:

```
plan tests => 2, have_apache 2;
```

will skip the test if not run under the 2nd Apache generation (httpd-2.x.xx).

```
plan tests => 2, have_apache 1;
```

will skip the test if not run under the 1st Apache generation (apache-1.3.xx).

- **have_min_apache_version**

Used to require a minimum version of Apache. For example:

```
plan tests => 5, have_min_apache_version("2.0.40");
```

requires Apache 2.0.40 or higher.

- **have_apache_version**

Used to require a specific version of Apache.

For example:

```
plan tests => 5, have_apache_version("2.0.40");
```

requires Apache 2.0.40.

1.7.3 *Skipping Numerous Tests*

Just like you can tell `Apache::Test` to run only specific tests, you can tell it to run all but a few tests.

If all files in a directory `t/foo` should be skipped, create:

```
#file:t/foo/all.t
#-----
print "1..0\n";
```

Alternatively you can specify which tests should be skipped from a single file `t/SKIP`. This file includes a list of tests to be skipped. You can include comments starting with `#` and you can use the `*` wildcard for multiply files matching.

For example if in `mod_perl 2.0` test suite we create the following file:

1.7.4 Reporting a Success or a Failure of Sub-tests

```
#file:t/SKIP
#-----
# skip all files in protocol
protocol

# skip basic cgi test
modules/cgi.t

# skip all filter/input_* files
filter/input*.t
```

In our example the first pattern specifies the directory name *protocol*, since we want to skip all tests in it. But since the skipping is done based on matching the skip patterns from *t/SKIP* against a list of potential tests to be run, some other tests may be skipped as well if they match the pattern. Therefore it's safer to use a pattern like this:

```
protocol/*.t
```

The second pattern skips a single test *modules/cgi.t*. Note that you shouldn't specify the leading *t/*. The *.t* extension is optional, so you can tell:

```
# skip basic cgi test
modules/cgi
```

The last pattern tells `Apache::Test` to skip all the tests starting with *filter/input*.

1.7.4 Reporting a Success or a Failure of Sub-tests

After printing the number of planned sub-tests, and assuming that the test is not skipped, the tests is running its sub-tests and each sub-test is expected to report its success or failure by printing *ok* or *not ok* respectively followed by its sequential number and a new line. For example:

```
print "ok 1\n";
print "not ok 2\n";
print "ok 3\n";
```

In `Apache::Test` this is done using the `ok()` function which prints *ok* if its argument is a true value, otherwise it prints *not ok*. In addition it keeps track of how many times it was called, and every time it prints an incremental number, therefore you can move sub-tests around without needing to remember to adjust sub-test's sequential number, since now you don't need them at all. For example this test snippet:

```
use Apache::Test;
use Apache::TestUtil;
plan tests => 3;
ok "success";
t_debug("expecting to fail next test");
ok "";
ok 0;
```

will print:

```
1..3
ok 1
# expecting to fail next test
not ok 2
not ok 3
```

Most of the sub-tests perform one of the following things:

- test whether some variable is defined:

```
ok defined $object;
```

- test whether some variable is a true value:

```
ok $value;
```

or a false value:

```
ok !$value;
```

- test whether a received from somewhere value is equal to an expected value:

```
$expected = "a good value";
$received = get_value();
ok defined $received && $received eq $expected;
```

1.7.5 Skipping Sub-tests

If the standard output line contains the substring *# Skip* (with variations in spacing and case) after *ok* or *ok NUMBER*, it is counted as a skipped test. `Test::Harness` reports the text after *# Skip\S*\s+* as a reason for skipping. So you can count a sub-test as a skipped as follows:

```
print "ok 3 # Skip for some reason\n";
```

or using the `Apache::Test`'s `skip()` function which works similarly to `ok()`:

```
skip $should_skip, $test_me;
```

so if `$should_skip` is true, the test will be reported as skipped. The second argument is the one that's sent to `ok()`, so if `$should_skip` is true, a normal `ok()` sub-test is run. The following example represent four possible outcomes of using the `skip()` function:

```
skip_subtest_1.t
-----
use Apache::Test;
plan tests => 4;

my $ok      = 1;
my $not_ok  = 0;

my $should_skip = "foo is missing";
skip $should_skip, $ok;
skip $should_skip, $not_ok;
```

1.7.5 Skipping Sub-tests

```
$should_skip = '';
skip $should_skip, $ok;
skip $should_skip, $not_ok;
```

now we run the test:

```
% ./t/TEST -run-tests -verbose skip_subtest_1
skip_subtest_1....1..4
ok 1 # skip foo is missing
ok 2 # skip foo is missing
ok 3
not ok 4
# Failed test 4 in skip_subtest_1.t at line 13
Failed 1/1 test scripts, 0.00% okay. 1/4 subtests failed, 75.00% okay.
```

As you can see since `$should_skip` had a true value, the first two sub-tests were explicitly skipped (using `$should_skip` as a reason), so the second argument to `skip` didn't matter. In the last two sub-tests `$should_skip` had a false value therefore the second argument was passed to the `ok()` function. Basically the following code:

```
$should_skip = '';
skip $should_skip, $ok;
skip $should_skip, $not_ok;
```

is equivalent to:

```
ok $ok;
ok $not_ok;
```

However if you want to use `t_cmp()` or some other function call in the arguments to `ok()` that won't quite work since the function will be always called no matter whether the first argument will evaluate to a true or a false value. For example, if you had a function:

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

and now you want to run this sub-test if `HTTP::Date` is available, changing it to:

```
my $should_skip = eval { require HTTP::Date } ? "" : "missing HTTP::Date";
skip $should_skip, t_cmp($expected, $received, $comment);
```

will still run `t_cmp()` even if `HTTP::Date` is not available. Therefore it's probably better to code it in this way:

```
if (eval {require HTTP::Date}) {
    ok t_cmp($expected, $received, $comment);
}
else {
    skip "Skip HTTP::Date not found";
}
```


1.7.6 Running only Selected Sub-tests

`Apache::Test` also allows to write tests in such a way that only selected sub-tests will be run. The test simply needs to switch from using `ok()` to `sok()`. Where the argument to `sok()` is a CODE reference or a BLOCK whose return value will be passed to `ok()`. If sub-tests are specified on the command line only those will be run/passed to `ok()`, the rest will be skipped. If no sub-tests are specified, `sok()` works just like `ok()`. For example, you can write this test:

```
#file:skip_subtest_2.t
#-----
use Apache::Test;
plan tests => 4;
sok {1};
sok {0};
sok sub {'true'};
sok sub {''};
```

and then ask to run only sub-tests 1 and 3 and to skip the rest.

```
% ./t/TEST -verbose skip_subtest_2 1 3
skip_subtest_2....1..4
ok 1
ok 2 # skip skipping this subtest
ok 3
ok 4 # skip skipping this subtest
ok, 2/4 skipped: skipping this subtest
All tests successful, 2 subtests skipped.
```

Only the sub-tests 1 and 3 get executed.

A range of sub-tests to run can be given using the Perl's range operand:

```
% ./t/TEST -verbose skip_subtest_2 2..4
skip_subtest_2....1..4
ok 1 # skip asking this subtest
not ok 2
# Failed test 2
ok 3
not ok 4
# Failed test 4
Failed 1/1 test scripts, 0.00% okay. 2/4 subtests failed, 50.00% okay.
```

In this run, only the first sub-test gets executed.

1.7.7 Todo Sub-tests

In a safe fashion to skipping specific sub-tests, it's possible to declare some sub-tests as *todo*. This distinction is useful when we know that some sub-test is failing but for some reason we want to flag it as a todo sub-test and not as a broken test. `Test::Harness` recognizes *todo* sub-tests if the standard output line contains the substring `# TODO` after `not ok` or `not ok NUMBER` and is counted as a todo sub-test. The text afterwards is the explanation of the thing that has to be done before this sub-test will succeed. For example:

1.7.8 Making it Easy to Debug

```
print "not ok 42 # TODO not implemented\n";
```

In `Apache::Test` this can be done with passing a reference to a list of sub-tests numbers that should be marked as *todo* sub-test:

```
plan tests => 7, todo => [3, 6];
```

In this example sub-tests 3 and 6 will be marked as *todo* sub-tests.

1.7.8 Making it Easy to Debug

Ideally we want all the tests to pass, reporting minimum noise or none at all. But when some sub-tests fail we want to know the reason for their failure. If you are a developer you can dive into the code and easily find out what's the problem, but when you have a user who has a problem with the test suite it'll make his and your life much easier if you make it easy for the user to report you the exact problem.

Usually this is done by printing the comment of what the sub-test does, what is the expected value and what's the received value. This is a good example of debug friendly sub-test:

```
#file:debug_comments.t
#-----
use Apache::Test;
use Apache::TestUtil;
plan tests => 1;

t_debug("testing feature foo");
$expected = "a good value";
$received = "a bad value";
t_debug("expected: $expected");
t_debug("received: $received");
ok defined $received && $received eq $expected;
```

If in this example `$received` gets assigned a *bad value* string, the test will print the following:

```
% t/TEST debug_comments
debug_comments....FAILED test 1
```

No debug help here, since in a non-verbose mode the debug comments aren't printed. If we run the same test using the verbose mode, enabled with `-verbose`:

```
% t/TEST -verbose debug_comments
debug_comments....1..1
# testing feature foo
# expected: a good value
# received: a bad value
not ok 1
```

we can see exactly what's the problem, by visual examination of the expected and received values.

It's true that adding a few print statements for each sub tests is cumbersome, and adds a lot of noise, when you could just tell:

```
ok "a good value" eq "a bad value";
```

but no fear, `Apache::TestUtil` comes to help. The function `t_cmp()` does all the work for you:

```
use Apache::Test;
use Apache::TestUtil;
ok t_cmp(
    "a good value",
    "a bad value",
    "testing feature foo");
```

`t_cmp()` will handle undef'ined values as well, so you can do:

```
my $expected;
ok t_cmp(undef, $expected, "should be undef");
```

Finally you can use `t_cmp()` for regex comparisons. This feature is mostly useful when there may be more than one valid expected value, which can be described with regex. For example this can be useful to inspect the value of `$@` when `eval()` is expected to fail:

```
eval {foo();}
if ($@) {
    ok t_cmp(qr/^expecting foo/, $@, "func eval");
}
```

which is the same as:

```
eval {foo();}
if ($@) {
    t_debug("func eval");
    ok $@ =~ /^expecting foo/ ? 1 : 0;
}
```

1.7.9 Tie-ing STDOUT to a Response Handler Object

It's possible to run the sub-tests in the response handler, and simply return them as a response to the client which in turn will print them out. Unfortunately in this case you cannot use `ok()` and other functions, since they print and don't return the results, therefore you have to do it manually. For example:

```
sub handler {
    my $r = shift;

    $r->print("1..2\n");
    $r->print("ok 1\n");
    $r->print("not ok 2\n");

    return Apache::OK;
}
```

now the client should print the response to STDOUT for `Test::Harness` processing.

If the response handler is configured as:

```
SetHandler perl-script
```

STDOUT is already tied to the request object `$r`. Therefore you can now rewrite the handler as:

```
use Apache::Test;
sub handler {
    my $r = shift;

    Apache::Test::test_pm_refresh();
    plan tests => 2;
    ok "true";
    ok "";

    return Apache::OK;
}
```

However to be on the safe side you also have to call `Apache::Test::test_pm_refresh()` allowing `plan()` and friends to be called more than once per-process.

Under different settings STDOUT is not tied to the request object. If the first argument to `plan()` 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:

```
use Apache::Test;
sub handler {
    my $r = shift;

    plan $r, tests => 2;
    ok "true";
    ok "";

    return Apache::OK;
}
```

Yet another alternative to handling the test framework printing inside response handler is to use `Apache::TestToString` class.

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

```
use Apache::Test;
sub handler {
    my $r = shift;

    Apache::TestToString->start;

    plan tests => 2;
    ok "true";
    ok "";

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

```

    $r->print($output);

    return Apache::OK;
}

```

In this example `Apache::TestToString` intercepts and buffers all the output from `Test.pm` and can be retrieved with its `finish()` method. Which then can be printed to the client in one shot. Internally it calls `Apache::Test::test_pm_refresh()` to make sure `plan()`, `ok()` and other functions() will work correctly more than one test is running under the same interpreter.

1.7.10 Helper Functions

`Apache::TestUtil` provides other helper functions, useful for writing tests, not mentioned in this tutorial:

```

t_cmp()
t_debug()
t_append_file()
t_write_file()
t_open_file()
t_mkdir()
t_rmtree()
t_is_equal()
t_write_perl_script()
t_write_shell_script()
t_chown()
t_server_log_error_is_expected()
t_server_log_warn_is_expected()
t_client_log_error_is_expected()>
t_client_log_warn_is_expected()>

```

See the `Apache::TestUtil` manpage for more information.

1.7.11 Auto Configuration

If the test is comprised only from the request part, you have to manually configure the targets you are going to use. This is usually done in `t/conf/extra.conf.in`.

If your tests are comprised from the request and response parts, `Apache::Test` automatically adds the configuration section for each response handler it finds. For example for the response handler:

```

package TestResponse::nice;
... some code
1;

```

it will put into `t/conf/httpd.conf`:

```

<Location /TestResponse__nice>
    SetHandler modperl
    PerlResponseHandler TestResponse::nice
</Location>

```

If you want to add some extra configuration directives, use the `__DATA__` section, as in this example:

```
package TestResponse::nice;
... some code
1;
__DATA__
PerlSetVar Foo Bar
```

These directives will be wrapped into the `<Location>` section and placed into `t/conf/httpd.conf`:

```
<Location /TestResponse__nice>
    SetHandler modperl
    PerlResponseHandler TestResponse::nice
    PerlSetVar Foo Bar
</Location>
```

This autoconfiguration feature was added to:

- simplify (less lines) test configuration.
- ensure unique namespace for `<Location ...>`'s.
- force `<Location ...>` names to be consistent.
- prevent clashes within main configuration.

1.7.11.1 Forcing Configuration Sections into the Top Level

If some directives are supposed to go to the base configuration, i.e. not to be automatically wrapped into `<Location>` block, you should use a special `<Base>..</Base>` block:

```
__DATA__
<Base>
    PerlSetVar Config ServerConfig
</Base>
PerlSetVar Config LocalConfig
```

Now the autogenerated section will look like this:

```
PerlSetVar Config ServerConfig
<Location /TestResponse__nice>
    SetHandler modperl
    PerlResponseHandler TestResponse::nice
    PerlSetVar Config LocalConfig
</Location>
```

As you can see the `<Base>..</Base>` block has gone. As you can imagine this block was added to support our virtue of laziness, since most tests don't need to add directives to the base configuration and we want to keep the configuration sections in tests to a minimum and let Perl do the rest of the job for us.

1.7.11.2 Bypassing Auto-Configuration

In more complicated cases, usually when virtual hosts containers are involved, the auto-configuration might stand in a way and you will simply want to bypass it. If that's the case, put the configuration inside the `<NoAutoConfig>..</NoAutoConfig>` container. For example:

```
<NoAutoConfig>
  <VirtualHost TestPreConnection::note>
    PerlPreConnectionHandler TestPreConnection::note

    <Location /TestPreConnection__note>
      SetHandler modperl
      PerlResponseHandler TestPreConnection::note::response
    </Location>
  </VirtualHost>
</NoAutoConfig>
```

Notice, that the internal sections will be still parsed, tokens `@var@` will be substituted and `VirtualHost` sections will be rewritten with an automatically assigned port number and `ServerName`.

1.7.11.3 Virtual Hosts

`Apache::Test` automatically assigns an unused port for the virtual host configuration. Just make sure that you use the package name in the place where you usually specify a *hostname:port* value. For example for the following package:

```
#file:MyApacheTest/Foo.pm
#-----
package MyApacheTest::Foo;
...
1;
__END__
<VirtualHost MyApacheTest::Foo>
  <Location /test_foo>
    ....
  </Location>
</VirtualHost>
```

After running:

```
% t/TEST -conf
```

Check the auto-generated `t/conf/httpd.conf` and you will find what port was assigned. Of course it can change when more tests which require a special virtual host are used.

Now in the request script, you can figure out what port that virtual host was assigned, using the package name. For example:

```
#file:test_foo.t
#-----
use Apache::TestRequest;

my $module = "MyApacheTest::Foo";
my $config = Apache::Test::config();
Apache::TestRequest::module($module);
my $hostport = Apache::TestRequest::hostport($config);

print GET_BODY_ASSERT "http://$hostport/test_foo";
```

1.7.11.4 Running Pre-Configuration Code

Sometimes you need to setup things for the test. This usually includes creating directories and files, and populating the latter with some data, which will be used at request time. Instead of performing that operation in the client script every time a test is run, it's usually better to do it once when the server is configured. If you wish to run such a code, all you have to do is to add a special subroutine `APACHE_TEST_CONFIGURE` in the response package (assuming that that response package exists). When server is configured (`t/TEST -conf`) it scans all the response packages for that subroutine and if found runs it.

`APACHE_TEST_CONFIGURE` accepts two arguments: the package name of the file this subroutine is defined in and the `Apache::TestConfig` configuration object.

Here is an example of a package that uses such a subroutine:

```
package TestDirective::perlmodule;

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

use Apache::Test ();

use Apache::RequestRec ();
use Apache::RequestIO ();
use File::Spec::Functions qw(catfile);

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->puts($ApacheTest::PerlModuleTest::MAGIC || '');

    Apache::OK;
}

sub APACHE_TEST_CONFIGURE {
    my ($class, $self) = @_;

    my $vars = $self->{vars};
    my $target_dir = catfile $vars->{documentroot}, 'testdirective';
```



```

    my $magic = __PACKAGE__;
    my $content = <<EOF;
package ApacheTest::PerlModuleTest;
\${ApacheTest::PerlModuleTest::MAGIC} = '$magic';
1;
EOF
    my $file = catfile $target_dir,
        'perlmodule-vh', 'ApacheTest', 'PerlModuleTest.pm';
    $self->writefile($file, $content, 1);
}
1;

```

In this example's function a directory is created. Then a file with some perl code as a content is created.

1.7.11.5 Controlling the Configuration Order

Sometimes it's important in which order the configuration section of each response package is inserted. Apache::Test controls the insertion order using a special token `APACHE_TEST_CONFIG_ORDER`. To decide on the configuration insertion order, Apache::Test scans all response packages and tries to match the following pattern:

```
/APACHE_TEST_CONFIG_ORDER\s+([+-]?\d+)/
```

So you can assign any integer number (positive or negative). If the match fails, it's assumed that the token's value is 0. Next a simple numerical search is performed and those configuration sections with lower token value are inserted first.

It's not specified how sections with the same token value are ordered. This usually depends on the order the files were read from the disk, which may vary from machine to machine and shouldn't be relied upon.

As already mentioned by default all configuration sections have a token whose value is 0, meaning that their ordering is unimportant. Now if you want to make sure that some section is inserted first, assign to it a negative number, e.g.:

```
# APACHE_TEST_CONFIG_ORDER -150
```

Now if a new test is added and it has to be the first, add to this new test a token with a negative value whose absolute value is higher than -150, e.g.:

```
# APACHE_TEST_CONFIG_ORDER -151
```

or

```
# APACHE_TEST_CONFIG_ORDER -500
```

Decide how big the gaps should be by thinking ahead. This is similar to the Basic language line numbering ;) In any case, you can always adjust other tests' token if you need to squeeze a number between two consequent integers.

If on the other hand you want to ensure that some test is configured last, use the highest positive number, e.g.:

```
# APACHE_TEST_CONFIG_ORDER 100
```

If some other test needs to be configured just before the one we just inserted, assign a token with a lower value, e.g.:

```
# APACHE_TEST_CONFIG_ORDER 99
```

1.7.12 Threaded versus Non-threaded Perl Test's Compatibility

Since the tests are supposed to run properly under non-threaded and threaded perl, you have to worry to enclose the threaded perl specific configuration bits in:

```
<IfDefine PERL_USEITHREADS>
    ... configuration bits
</IfDefine>
```

Apache::Test will start the server with -DPERL_USEITHREADS if the Perl is ithreaded.

For example PerlOptions +Parent is valid only for the threaded perl, therefore you have to write:

```
<IfDefine PERL_USEITHREADS>
    # a new interpreter pool
    PerlOptions +Parent
</IfDefine>
```

Just like the configuration, the test's code has to work for both versions as well. Therefore you should wrap the code specific to the threaded perl into:

```
if (have_perl 'ithreads'){
    # ithread specific code
}
```

which is essentially does a lookup in \$Config{useithreads}.

1.7.13 Retrieving the Server Configuration Data

The server configuration data can be retrieved and used in the tests via the configuration object:

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

1.7.13.1 Module Magic Number

The following code retrieves the major and minor MMN numbers.

```
my $cfg = Apache::Test::config();
my $info = $cfg->{httpd_info};

my $major = $info->{MODULE_MAGIC_NUMBER_MAJOR};
my $minor = $info->{MODULE_MAGIC_NUMBER_MINOR};

print "major=$major, minor=$minor\n";
```

For example for MMN 20011218:0, this code prints:

```
major=20011218, minor=0
```

1.8 Debugging Tests

Sometimes your tests won't run properly or even worse will segfault. There are cases where it's possible to debug broken tests with simple print statements but usually it's very time consuming and ineffective. Therefore it's a good idea to get yourself familiar with Perl and C debuggers, and this knowledge will save you a lot of time and grief in a long run.

1.8.1 Under C debugger

mod_perl-2.0 provides built in 'make test' debug facility. So in case you get a core dump during make test, or just for fun, run in one shell:

```
% t/TEST -debug
```

in another shell:

```
% t/TEST -run-tests
```

then the *-debug* shell will have a (gdb) prompt, type where for stacktrace:

```
(gdb) where
```

You can change the default debugger by supplying the name of the debugger as an argument to *-debug*. E.g. to run the server under ddd:

```
% ./t/TEST -debug=ddd
```

META: list supported debuggers

If you debug mod_perl internals you can set the breakpoints using the *-breakpoint* option, which can be repeated as many times as needed. When you set at least one breakpoint, the server will start running till it meets the *ap_run_pre_config* breakpoint. At this point we can set the breakpoint for the mod_perl code, something we cannot do earlier if mod_perl was built as DSO. For example:

```
% ./t/TEST -debug -breakpoint=modperl_cmd_switches \
    -breakpoint=modperl_cmd_options
```

will set the *modperl_cmd_switches* and *modperl_cmd_options* breakpoints and run the debugger.

If you want to tell the debugger to jump to the start of the *mod_perl* code you may run:

```
% ./t/TEST -debug -breakpoint=modperl_hook_init
```

In fact *-breakpoint* automatically turns on the debug mode, so you can run:

```
% ./t/TEST -breakpoint=modperl_hook_init
```

1.8.2 Under Perl debugger

When the Perl code misbehaves it's the best to run it under the Perl debugger. Normally started as:

```
% perl -debug program.pl
```

the flow control gets passed to the Perl debugger, which allows you to run the program in single steps and examine its states and variables after every executed statement. Of course you can set up breakpoints and watches to skip irrelevant code sections and watch after certain variables. The *perldebug* and the *perldebug-tut* manpages are covering the Perl debugger in fine details.

The `Apache::Test` framework extends the Perl debugger and plugs in LWP's debug features, so you can debug the requests. Let's take test *apache/read* from *mod_perl* 2.0 and present the features as we go:

META: to be completed

run *.t* test under the perl debugger

```
% t/TEST -debug perl t/modules/access.t
```

run *.t* test under the perl debugger (nonstop mode, output to *t/logs/perldb.out*)

```
% t/TEST -debug perl=nostop t/modules/access.t
```

turn on *-v* and LWP trace (1 is the default) mode in `Apache::TestRequest`

```
% t/TEST -debug lwp t/modules/access.t
```

turn on *-v* and LWP trace mode (level 2) in `Apache::TestRequest`

```
% t/TEST -debug lwp=2 t/modules/access.t
```

1.8.3 Tracing

To get Start the server under *strace*(1):

```
% t/TEST -debug strace
```

The output goes to *t/logs/strace.log*.

Now in a second terminal run:

```
% t/TEST -run-tests
```

Beware that *t/logs/strace.log* is going to be very big.

META: can we provide `strace(1)` opts if we want to see only certain syscalls?

1.9 Using Apache::Test to Speed up Project Development

When developing a project, as the code is written or modified it is desirable to test it at the same time. If you write tests as you code, or even before you code, Apache::Test can speed up the modify-test code development cycle. The idea is to start the server once and then run the tests without restarting it, and make the server reload the modified modules behind the scenes. This of course works only if you modify plain perl modules. If you develop XS/C components, you have no choice but to restart the server before you want to test the modified code.

First of all, your perl modules need to reside under the *lib* directory, the same way they reside in *blib/lib*. In the section Basic Testing Environment we've already arranged for that. If *Amazing.pm* resides in the top-level directory, it's not possible to perform `'require Apache::Amazing'`. Only after running `make`, the file will be moved to *blib/lib/Apache/Amazing.pm*, which is when we can load it. But you don't want to run `make` every time you change the file. It's both annoying and error-prone, since at times you'd do some change, try to verify it and it will appear to be wrong, and you will try to understand why, whereas in reality you just forgot to run `make` and the server was testing against the old unmodified version in *blib/lib*. Of you course if you always run `make test` it'll always do the right thing, but it's not the most effecient way to undertake when you want to test a specific test and you do it every few seconds.

The following scenario will make you a much happier Perl developer.

First, we need to instruct Apache::Test to modify `@INC`, which we could do in *t/conf/modperl_extra.pl* or *t/conf/extra.conf.in*, but the problem is that you may not want to keep that change in the released package. There is a better way, if the environment variable `APACHE_TEST_LIVE_DEV` is set to a true value, `Apache::Test` will automatically add the *lib/* directory if it exists. Executing:

```
% APACHE_TEST_LIVE_DEV=1 t/TEST -configure
```

will add code to add */path/to/Apache-Amazing/lib* to `@INC` in *t/conf/modperl_inc.pl*. This technique is convenient since you don't need to modify your code to include that directory.

Second, we need to configure `mod_perl` to use `Apache::Reload` to automatically reload the module when it's changed, by adding following configuration directives to *t/conf/extra.conf.in*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "Apache::Amazing"
```

(For more information about `Apache::Reload`, depending on the used `mod_perl` generation, refer to the `mod_perl` 1.0 documentation or the `Apache::Reload` manpage for `mod_perl` 2.0.)

now we execute:

```
% APACHE_TEST_LIVE_DEV=1 t/TEST -configure
```

which will generate `t/conf/extra.conf` and start the server:

```
% t/TEST -start
```

from now on, we can modify *Apache/Amazing.pm* and repeatedly run:

```
% t/TEST -run basic
```

without restarting the server.

1.10 Writing Tests Methodology

META: to be completed

1.10.1 When Tests Should Be Written

- **A New feature is Added**

Every time a new feature is added new tests should be added to cover the new feature.

- **A Bug is Reported**

Every time a bug gets reported, before you even attempt to fix the bug, write a test that exposes the bug. This will make much easier for you to test whether your fix actually fixes the bug.

Now fix the bug and make sure that test passes ok.

It's possible that a few tests can be written to expose the same bug. Write them all -- the more tests you have the less chances are that there is a bug in your code.

If the person reporting the bug is a programmer you may try to ask her to write the test for you. But usually if the report includes a simple code that reproduces the bug, it should probably be easy to convert this code into a test.

1.11 Other Webserver Regression Testing Frameworks

- **Puffin**

Puffin is a web application regression testing system. It allows you to test any web application from end to end based application as if it were a "black box" accepting inputs and returning outputs.

It's available from <http://puffin.sourceforge.net/>

1.12 References

- **more Apache-Test documentation**

Testing mod_perl 2.0 <http://www.perl.com/pub/a/2003/05/22/testing.html>

Apache::Test manpage

Apache-Test README

- **extreme programming methodology**

Extreme Programming: A Gentle Introduction: <http://www.extremeprogramming.org/>.

Extreme Programming: <http://www.xprogramming.com/>.

See also other sites linked from these URLs.

1.13 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

1.14 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

1	Running and Developing Tests with the Apache::Test Framework	1
1.1	Description	2
1.2	Basics of Perl Modules Testing	2
1.3	Prerequisites	3
1.4	Running Tests	3
1.4.1	Testing Options	4
1.4.2	Basic Testing	4
1.4.3	Individual Testing	4
1.4.4	Repetitive Testing	5
1.4.5	Parallel Testing	5
1.4.6	Verbose Mode	6
1.4.7	Colored Trace Mode	6
1.4.8	Controlling the Apache::Test's Signal to Noise Ratio	7
1.4.9	Stress Testing	8
1.4.9.1	The Problem	8
1.4.9.2	The Solution	8
1.4.9.3	Resolving Sequence Problems	9
1.4.9.4	Apache::TestSmoke Solution	10
1.4.10	RunTime Configuration Overriding	11
1.4.11	Request Generation and Response Options	12
1.4.12	Batch Mode	14
1.5	Setting Up Testing Environment	14
1.5.1	Know Your Target Environment	15
1.5.2	Basic Testing Environment	15
1.5.3	Extending Configuration Setup	20
1.5.4	Special Configuration Files	21
1.5.5	Inheriting from System-wide httpd.conf	21
1.6	Apache::Test Framework's Architecture	22
1.6.1	Developing Response-only Part of a Test	22
1.6.2	Developing Response and Request Parts of a Test	24
1.6.3	Developing Test Response Handlers in C	26
1.6.4	Request and Response Methods	27
1.6.5	Other Request Generation helpers	31
1.6.6	Starting Multiple Servers	31
1.6.7	Multiple User Agents	32
1.6.8	Hitting the Same Interpreter (Server Thread/Process Instance)	32
1.7	Writing Tests	33
1.7.1	Defining How Many Sub-Tests Are to Be Run	33
1.7.2	Skipping a Whole Test	34
1.7.3	Skipping Numerous Tests	37
1.7.4	Reporting a Success or a Failure of Sub-tests	38
1.7.5	Skipping Sub-tests	39
1.7.6	Running only Selected Sub-tests	41
1.7.7	Todo Sub-tests	41

Table of Contents:

1.7.8 Making it Easy to Debug	42
1.7.9 Tie-ing STDOUT to a Response Handler Object	43
1.7.10 Helper Functions	45
1.7.11 Auto Configuration	45
1.7.11.1 Forcing Configuration Sections into the Top Level	46
1.7.11.2 Bypassing Auto-Configuration	47
1.7.11.3 Virtual Hosts	47
1.7.11.4 Running Pre-Configuration Code	48
1.7.11.5 Controlling the Configuration Order	49
1.7.12 Threaded versus Non-threaded Perl Test's Compatibility	50
1.7.13 Retrieving the Server Configuration Data	50
1.7.13.1 Module Magic Number	50
1.8 Debugging Tests	51
1.8.1 Under C debugger	51
1.8.2 Under Perl debugger	52
1.8.3 Tracing	52
1.9 Using Apache::Test to Speed up Project Development	53
1.10 Writing Tests Methodology	54
1.10.1 When Tests Should Be Written	54
1.11 Other Webserver Regression Testing Frameworks	54
1.12 References	55
1.13 Maintainers	55
1.14 Authors	55