Testing with Apache-Test

Geoffrey Young

geoff@modperlcookbook.org

Goals

- Introduction to `Apache-Test`
- Perl module support
- C module support
- Automagic configuration
- Test-driven development basics
- Other Goodness™

The Code

http://people.apache.org/~geoff/

```
$ tar zxvf ApacheCon2006-code.tar.gz
$ cd ApacheCon2006-code

$ export PERL5LIB=`pwd`/Apache-Test/lib
```

Apache-Test

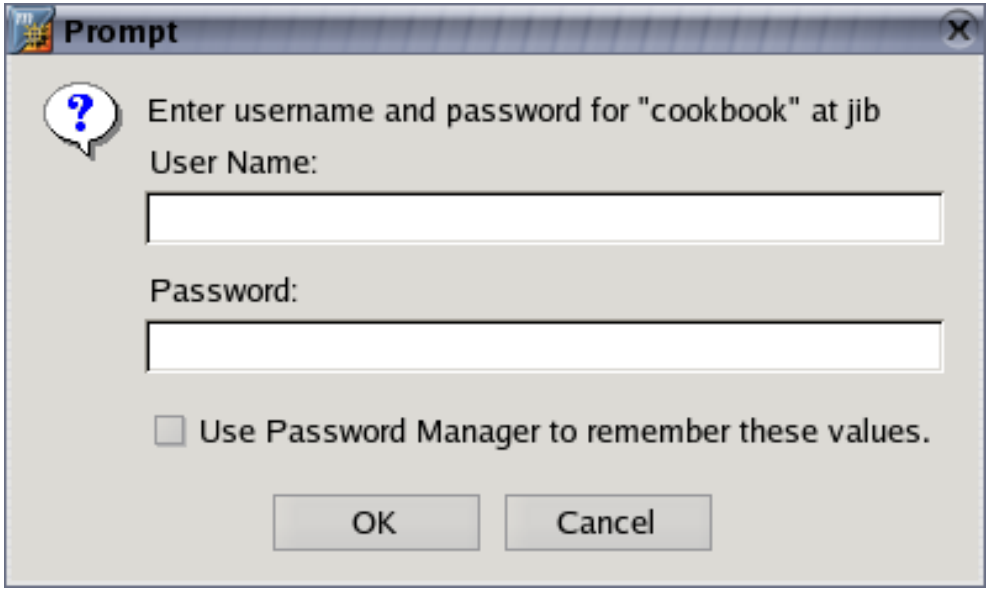
- Framework for testing Apache-based application components
- Provides tools to make testing Apache related things simple
  - configures, starts, and stops Apache
  - lets you focus on writing tests
  - provides HTTP-centric tools
- Gives you a self-contained, pristine Apache environment for testing

`Apache-Test` by Example

- Write a simple Perl handler
- Integrate `Apache-Test`
- Port the handler to C
- Show all kinds of cool stuff

```
package My::AuthenHandler;

use Apache2::Const -compile => qw(OK HTTP_UNAUTHORIZED);

use Apache2::RequestRec ();
use Apache2::Access ();

sub handler {

  my $r = shift;

  # Get the client-supplied credentials.
  my ($status, $password) = $r->get_basic_auth_pw;

  return $status unless $status == Apache2::Const::OK;

  # Perform some custom user/password validation.
  return Apache2::Const::OK if $r->user eq $password;

  # Whoops, bad credentials.
  $r->note_basic_auth_failure;
  return Apache2::Const::HTTP_UNAUTHORIZED;
}

1;
```

Voila!


Prompt
Enter username and password for "cookbook" at jib
User Name:
Password:
Use Password Manager to remember these values.
OK
Cancel

Testing, Testing... 1, 2, 3

1. Generate the test harness
2. Configure Apache
3. Write the tests

Step 1 - The Test Harness

- Generally starts from `Makefile.PL`
- There are other ways as well
  - illustrated later

Makefile.PL

```
use Apache::TestMM qw(test clean);
use Apache::TestRunPerl ();

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

# generate the test harness (t/TEST)
Apache::TestRunPerl->generate_script();
```

filter_args()

```
$ perl Makefile.PL -httpd /usr/local/apache/bin/httpd
```

t/TEST

- `generate_script()` creates the special file `t/TEST`
- `t/TEST` is the actual harness that coordinates testing activities
- called via `make test`
- can be called directly

```
$ t/TEST t/foo.t
```

Step 1 - The Test Harness

- Don't get bogged down with `Makefile.PL` details
- Lather, Rinse, Repeat

Testing, Testing... 1, 2, 3

1. Generate the test harness
2. Configure Apache

Step 2 - Configure Apache

- Apache needs a basic configuration to service requests
  - `ServerRoot` `t/`
  - `DocumentRoot` `t/htdocs`
  - `ErrorLog` `t/logs/error_log`
  - `Listen` `8529`
  - `LoadModule` `{all}`
- `Apache-Test` "intuits" these and creates its own `httpd.conf`
- Configures all that is required to `GET`

  `http://localhost:8529/index.html`

Adding to the Default Config

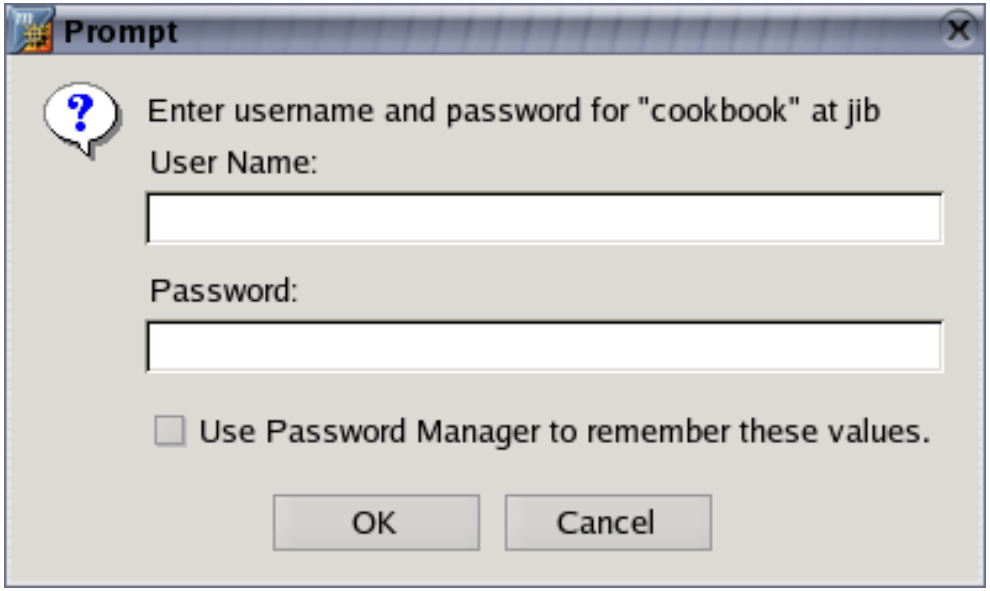
- When testing we generally need more than the default
- Instead of altering `httpd.conf` we augment it with a special file
- `t/conf/extra.conf.in`

extra.conf.in

- Same directives as `httpd.conf`
- Pulled into `httpd.conf` via `Include`
- Provides handy variable substitution

Create the Configuration

- Our handler is an authentication handler



- Let's set up a protected URI using `t/conf/extra.conf.in`

extra.conf.in

```
Alias /authen @DocumentRoot@

<Location /authen>
  Require valid-user
  AuthType Basic
  AuthName "my test realm"

  PerlAuthenHandler My::AuthenHandler
</Location>
```

```
package My::AuthenHandler;

use Apache2::Const -compile => qw(OK HTTP_UNAUTHORIZED);

use Apache2::RequestRec ();
use Apache2::Access ();

sub handler {

  my $r = shift;

  # Get the client-supplied credentials.
  my ($status, $password) = $r->get_basic_auth_pw;

  return $status unless $status == Apache2::Const::OK;

  # Perform some custom user/password validation.
  return Apache2::Const::OK if $r->user eq $password;

  # Whoops, bad credentials.
  $r->note_basic_auth_failure;
  return Apache2::Const::HTTP_UNAUTHORIZED;
}

1;
```

extra.conf.in

```
Alias /authen @DocumentRoot@

<Location /authen>
  Require valid-user
  AuthType Basic
  AuthName "my test realm"

  PerlAuthenHandler My::AuthenHandler
</Location>
```

Whew!

- At this point we have...
  - autogenerated a minimal Apache configuration
  - configured a protected URL
  - created a mod_perl handler to do the authentication
- Now...

Testing, Testing... 1, 2, 3

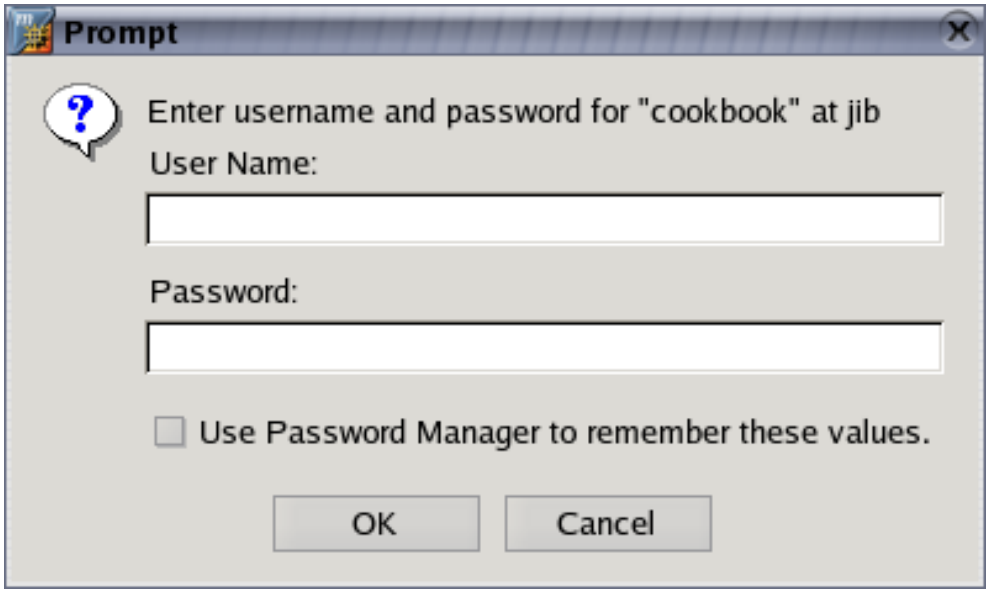
1. Generate the test harness
2. Configure Apache
3. Write the tests

What Should We Test?

- Tests need to be meaningful
- Tests ought to be thorough
- Tests should align the server response with our expected results

In Other Words...

- Our handler is an authentication handler



- Unknown users should fail
- Known users should pass

What Exactly is a Test?

- Tests are contained within a test file
- The test file acts as a client
- The client is scripted to
  - query the server
  - compare server response to expected results
  - indicate success or failure

The `t/` Directory

- Tests live in `t/`
  - `t/01basic.t`
- `t/` is also the `ServerRoot`
  - `t/htdocs`
  - `t/cgi-bin`
  - `t/conf`

Anatomy of a Test

- `Apache-Test` works the same way as `Test.pm`, `Test::More` and others
  - `plan()` the number of tests
  - call `ok()` for each test you plan
    - where `ok()` is any one of a number of comparison functions
  - All the rest is up to you

t/01basic.t

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

plan tests => 1, (need_lwp &&
                  need_auth &&
                  need_module('mod_perl.c'));

{
  my $uri = '/authen/index.html';

  my $response = GET $uri;
  ok $response->code == 401;
}
```

Apache::Test

- Provides basic `Test.pm` functions
  - `ok()`
  - `plan()`
- Also provides helpful `plan()` functions
  - `need_lwp()`
  - `need_module()`
  - `need_min_apache_version()`

plan()

- `plan()` the number of tests in the file

```
plan tests => 5;
```

- Preconditions can be specified

```
plan tests => 5, need_module('mod_foo');
```

- Failed preconditions will skip the entire test file

```
server localhost.localdomain:8529 started
t/01basic....skipped
        all skipped: cannot find module 'mod_foo.c'
All tests successful, 1 test skipped.
```

On Precondition Failures...

- A failed precondition is *not* the same as a failed test
- Failed precondition means "I cannot create a suitable environment"
- Failed test means "I fed a subroutine known data and it did *not* produce expected output"
- Failure needs to represent something very specific in order to be meaningful

Apache::TestRequest

- Provides a basic `LWP` interface
  - `GET()`
  - `POST()`
  - `HEAD()`
  - `GET_OK()`
  - `GET_BODY()`
  - more
- Note that these functions know which host and port to send the request to

```
GET '/authen/index.html';
```

  - request URI can be relative

HTTP::Response

- LWP base class
- Provides accessors to response attributes
  - code()
  - content()
  - content_type(), content_length(), etc
  - headers()
    - authorization()
- as well as some useful utility methods
  - as_string()
  - previous()

Testing, Testing... 1, 2, 3

1. Generate the test harness
2. Configure Apache
3. Write the tests
4. Run the tests

The Pain is Over

- Initial setup is a bit complex
  - you only do it once
- Running the tests is simple

```
$make test
$t/TEST
```

Hopefully, You Saw...

```
-apxs
-httpd
-v
-help
-clean
-conf
```

Apache-Test fsck

- Every once in a while `Apache-Test` gets borked
- If you get stuck try cleaning and reconfiguring

```
$ t/TEST -clean
$ t/TEST -conf
```

- If that doesn't work, nuke everything

```
$ make realclean
$ rm -rf ~/.apache-test
```

Are you `ok`?

- `ok()` works, but is not descriptive
- luckily, we have options
  - `Apache::TestUtil`
  - `Test::More`

Apache::TestUtil

- Chocked full of helpful utilities
- `t_cmp()`

```
t_cmp($foo, $bar, 'foo is bar');
t_cmp($foo, qr/bar/, 'foo matches bar');
```

- `t_write_file($file, @lines);`
  - write out a file
  - clean it up after script execution completes
- `t_write_perl_script($file, @lines);`
  - same as `t_write_file()`
  - with compilation-specific shebang line
  - useful for writing out CGI scripts

`Test::More` functions

- Basic comparisons
  - `ok()`
  - `is()`
  - `like()`
- Intuitive comparisons
  - `isnt()`
  - `unlike()`
- Complex structures
  - `is_deeply()`
  - `eq_array()`

Using `Test::More`

- `Test::More` is the Perl-world standard
- You should use `Test::More` whenever possible
- Eventually `-withtestmore` will not be required

Getting Back to the Point...

- So far, we haven't actually tested anything useful
  - no username or password
- Let's add some real tests

```
my $uri = '/authen/index.html';

{
  my $response = GET $uri;

  is ($response->code,
      401,
      "no valid password entry");
}

{
  my $response = GET $uri, username => 'geoff', password => 'foo';

  is ($response->code,
      401,
      "password mismatch");
}

{
  my $response = GET $uri, username => 'geoff', password => 'geoff';

  is ($response->code,
      200,
      "geoff:geoff allowed to proceed");
}
```

From C to Shining C

- Let's create a C module that functions exactly the same as our Perl module

```
package My::AuthenHandler;

use Apache2::Const -compile => qw(OK HTTP_UNAUTHORIZED);

use Apache2::RequestRec ();
use Apache2::Access ();

sub handler {

  my $r = shift;

  # Get the client-supplied credentials.
  my ($status, $password) = $r->get_basic_auth_pw;

  return $status unless $status == Apache2::Const::OK;

  # Perform some custom user/password validation.
  return Apache2::Const::OK if $r->user eq $password;

  # Whoops, bad credentials.
  $r->note_basic_auth_failure;
  return Apache2::Const::HTTP_UNAUTHORIZED;
}

1;
```

```
#include "httpd.h"
#include "http_config.h"
#include "http_request.h"
#include "http_protocol.h"

module AP_MODULE_DECLARE_DATA my_authen_module;

static int authen_handler(request_rec *r) {
 ...
}

static void register_hooks(apr_pool_t *p)
{
  ap_hook_check_user_id(authen_handler, NULL, NULL, APR_HOOK_FIRST);
}

module AP_MODULE_DECLARE_DATA my_authen_module =
{
  STANDARD20_MODULE_STUFF,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  register_hooks
};
```

```
static int authen_handler(request_rec *r) {

  const char *sent_pw;

  /* Get the client-supplied credentials */
  int response = ap_get_basic_auth_pw(r, &sent_pw);

  if (response != OK) {
      return response;
  }

  /* Perform some custom user/password validation */
  if (strcmp(r->user, sent_pw) == 0) {
    return OK;
  }

  /* Whoops, bad credentials */
  ap_note_basic_auth_failure(r);
  return HTTP_UNAUTHORIZED;
}
```

Perl Makefile.PL

```
use Apache::TestMM qw(test clean);
use Apache::TestRunPerl ();

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

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

It Failed!

- Failures are a part of testing life
- Embrace failures
- In this case the failure isn't really a failure...
- Our environment isn't correct
- An incorrect environment can never yield "true" failures

Checking the Environment

- It's our job to check for the presence of the module we're testing

```
plan tests => 3, (need_lwp  &&
                  need_auth &&
                  need_module('mod_my_authen'));
```

The Real Problem

- Over in Perl-land, `ExtUtils::MakeMaker` took care of "compiling" our Perl module
  - put it in the proper place (`blib`)
  - added `blib` to `@INC`
- C modules rely on `apxs`, so we need to either compile them ourselves or tell `ExtUtils::MakeMaker` to do it for us
- Messing with `ExtUtils::MakeMaker` is hard
- `Apache-Test` has a better way

The `c-modules` Directory

- `Apache-Test` allows for special treatment of modules in `c-modules/`
- Modules placed in `c-modules/` will be
  - compiled via `apxs`
  - added to `httpd.conf` via `LoadModule`
- Similar to `lib/` and `blib/` in Perl

The Mechanics

- Modules should be placed in

  `c-modules/`*`name`*`/mod_`*`name`*`.c`

- where *`name`* matches C declaration minus `module`
- In our case

```
module AP_MODULE_DECLARE_DATA my_authen_module;
```

becomes

```
c-modules/my_authen/mod_my_authen.c
```

More Mechanics

- When the server environment is configured, the module will be added to `httpd.conf`

```
LoadModule my_authen_module /src/example/c-authen-auto-
  compile/c-modules/my_authen/.libs/mod_my_authen.so
```

But Wait, There's More

- If we can automatically compile and configure the loading of a module, why not fully configure it as well
- Enter automagic `httpd.conf` configuration

Magic

- `t/conf/extra.conf.in` has held our configuration thus far
- We can actually embed the config in our C module if we use `c-modules`

mod_example_ipc

```
 * To play with this sample module first compile it into a
 * DSO file and install it into Apache's modules directory
 * by running:
 *
 *    $ /path/to/apache2/bin/apxs -c -i mod_example_ipc.c
 *
 * Then activate it in Apache's httpd.conf file as follows:
 *
 *    LoadModule example_ipc_module modules/mod_example_ipc.so
 *
 *    <Location /example_ipc>
 *       SetHandler example_ipc
 *    </Location>

#if CONFIG_FOR_HTTPD_TEST

<Location /example_ipc>
   SetHandler example_ipc
</Location>

#endif
```

The Mechanics

- `mod_example_ipc`:

```
module AP_MODULE_DECLARE_DATA example_ipc_module;
```

becomes

```
c-modules/example_ipc/mod_example_ipc.c
```

Living in Harmony

- Using `Makefile.PL` has some obvious disadvantages
  - not everyone likes Perl
  - most people hate `ExtUtils::MakeMaker`
- Everyone can be happy
- Use both `Makefile.PL` and `makefile`
  - `makefile` for the stuff you like
  - `Makefile.PL` for test configuration

makefile

```
export APACHE_TEST_APXS ?= /apache/2.0.52/worker/perl-5.8.5/bin/apxs

all : Makefile
        $(MAKE) -f Makefile cmodules

Makefile :
        perl Makefile.PL

install :
        $(APACHE_TEST_APXS) -iac c-modules/example_ipc/mod_example_ipc.c

%: force
        @$(MAKE) -f Makefile $@
force: Makefile;
```

A Different `makefile`

```
export APACHE_TEST_APXS?=/apache/2.0.52/worker/perl-5.8.5/bin/apxs

t/TEST :
        perl -MApache::TestRun -e 'Apache::TestRun->generate_script()'

test : t/TEST
        t/TEST

install :
        $(APACHE_TEST_APXS) -iac c-modules/example_ipc/mod_example_ipc.c
```

example.t

```
use Apache::Test qw(-withtestmore);
use Apache::TestRequest;

plan tests => 20, need_module('mod_example_ipc');

foreach my $counter (1 .. 20) {

  my $response = GET_BODY '/example_ipc';

  like ($response,
        qr!Counter:</td><td>$counter!,
        "counter incremented to $counter");
}
```

Take Advantage of LWP

- Many of the things we do in Apache modules is complex
- Complex but still HTTP oriented
- LWP is a good tool for testing HTTP-specific things
- Like Digest authentication

An Aside on Digest Authentication

- Digest authentication uses a message digest to transfer the username and password across the wire
- Makes the Digest scheme (arguably) more secure than Basic
- Widespread adoption is made difficult because not all clients are RFC compliant
  - guess who?
- The most popular web server *is* RFC compliant

Reader's Digest

- RFC compliant clients and servers use the complete URI when computing the message digest
- Internet Explorer leaves off the query part of the URI when both transmitting the URI and computing the digest

Reader's Digest

- Given a request to `/index.html`

```
Authorization: Digest username="user1", realm="realm1",
qop="auth", algorithm="MD5", uri="/index.html",
nonce="Q9equ9C+AwA=195acc80cf91ce99828b8437707cafce78b11621",
nc=00000001, cnonce="3e4b161902b931710ae04262c31d9307",
response="49fac556a5b13f35a4c5f05c97723b32"
```

- Given a request to `/index.html?foo=bar`

```
Authorization: Digest username="user1", realm="realm1",
qop="auth", algorithm="MD5", uri="/index.html?foo=bar",
nonce="Q9equ9C+AwA=195acc80cf91ce99828b8437707cafce78b11621",
nc=00000001, cnonce="3e4b161902b931710ae04262c31d9307",
response="acbd18db4cc2f85cedef654fccc4a4d8"
```

MSIE's Digest

- Given a request to `/index.html`

```
Authorization: Digest username="user1", realm="realm1",
qop="auth", algorithm="MD5", uri="/index.html",
nonce="Q9equ9C+AwA=195acc80cf91ce99828b8437707cafce78b11621",
nc=00000001, cnonce="3e4b161902b931710ae04262c31d9307",
response="49fac556a5b13f35a4c5f05c97723b32"
```

- Given a request to `/index.html?foo=bar`

```
Authorization: Digest username="user1", realm="realm1",
qop="auth", algorithm="MD5", uri="/index.html",
nonce="Q9equ9C+AwA=195acc80cf91ce99828b8437707cafce78b11621",
nc=00000001, cnonce="3e4b161902b931710ae04262c31d9307",
response="49fac556a5b13f35a4c5f05c97723b32"
```

Working with MS Internet Explorer

The Digest authentication implementation in current Internet Explorer implementations has known issues, namely that `GET` requests with a query string are not RFC compliant. There are a few ways to work around this issue.

The first way is to use `POST` requests instead of `GET` requests to pass data to your program. This method is the simplest approach if your application can work with this limitation.

Since version 2.0.51 Apache also provides a workaround in the `AuthDigestEnableQueryStringHack` environment variable. If `AuthDigestEnableQueryStringHack` is set for the request, Apache will take steps to work around the MSIE bug and remove the request URI from the digest comparison. Using this method would look similar to the following.

Using Digest Authentication with MSIE:

```
BrowserMatch "MSIE" AuthDigestEnableQueryStringHack=On
```

See the `BrowserMatch` directive for more details on conditionally setting environment variables

AuthDigestAlgorithm Directive

Description:	Selects the algorithm used to calculate the challenge and response hases in digest authentication
Syntax:	`AuthDigestAlgorithm MD5|MD5-sess`
Default:	`AuthDigestAlgorithm MD5`

AuthDigestEnableQueryStringHack

- Developers could always work around the problem using POST
- As of Apache 2.0.51 administrators can work around the problem from httpd.conf

```
BrowserMatch MSIE AuthDigestEnableQueryStringHack=On
```

- Removes the query portion of the URI from comparison

Does It Work?

- How do you know it works?
- Here's what we need to verify
  - MSIE users can authenticate
  - RFC compliant users still can authenticate
  - if MSIE gets fixed, users can authenticate
  - Mismatches still fail
- Test-driven development begins!

Tired

- Hack together some fix
- Hit it with a browser to make sure it works
- Move on
- Waste lots of time recreating bugs that *will* eventually show up

Wired

- Add a test to your `Apache-Test`-based framework
- Come up with basic conditions
- Write the code
- Run the test
- Add some edge cases
- Run the test
- Spend a little time fixing bugs that (probably) will show up

Bringing It All Together

- Let's write a test for the MSIE fix
- While we're at it we'll illustrate a few things
  - iterative test-driven development cycle
  - cool features of `Apache-Test` and `LWP`

t/conf/extra.conf.in

```
<IfModule mod_auth_digest.c>

  Alias /digest @DocumentRoot@

  <Location /digest>
    Require valid-user
    AuthType Digest
    AuthName realm1
    AuthDigestFile @ServerRoot@/realm1
  </Location>

</IfModule>
```

digest.t

```
use Apache::Test qw(-withtestmore);
use Apache::TestRequest;
use Apache::TestUtil qw(t_write_file);
use File::Spec;

plan tests => 4, need need_lwp,
                     need_module('mod_auth_digest');

# write out the authentication file - t/realm1
my $file = File::Spec->catfile(Apache::Test::vars('serverroot'),
                               'realm1');
t_write_file($file, <DATA>);

...

__DATA__
# user1/password1
user1:realm1:4b5df5ee44449d6b5fbf026a7756e6ee
```

need()

- `need()` serves as a `need_*()` aggregator
- compare

```
plan tests need_lwp && need_auth;
```

- to

```
plan tests need need_lwp, need_auth;
```

- `need()` shows the user everthing that's required at once

Apache::Test::vars()

- Allows access to configuration expansion variables
  - serverroot
  - httpd or apxs
- ServerRoot is required when writing files
  - Apache-Test changes directories from time to time
- Use File::Spec functions to concat
  - if you care about portability, that is

t_write_file()

- Exported by `Apache::TestUtil`

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

- Accepts a file and a list of lines

```
t_write_file($file, @lines);
```

- Write out the file
  - including any required directories
- Cleans up the file when script exits
  - including created directories

digest.t

```
use Apache::Test qw(-withtestmore);
use Apache::TestRequest;
use Apache::TestUtil qw(t_write_file);
use File::Spec;

plan tests => 4, need need_lwp,
                     need_module('mod_auth_digest');

# write out the authentication file - t/realm1
my $file = File::Spec->catfile(Apache::Test::vars('serverroot'),
                               'realm1');
t_write_file($file, <DATA>);

...

__DATA__
# user1/password1
user1:realm1:4b5df5ee44449d6b5fbf026a7756e6ee
```

• • •

```
my $url = '/digest/index.html';

{
  my $response = GET $url;

  is ($response->code,
      401,
      'no user to authenticate');
}

{
  # authenticated
  my $response = GET $url,
                  username => 'user1', password => 'password1';

  is ($response->code,
      200,
      'user1:password1 found');
}
```

MSIE Tests

- Ok, so we've proven that we can interact with Digest authentication
- Let's test our fix

t/conf/extra.conf.in

```
<IfModule mod_auth_digest.c>

  Alias /digest @DocumentRoot@

  <Location /digest>
    Require valid-user
    AuthType Digest
    AuthName realm1
    AuthDigestFile @ServerRoot@/realm1
  </Location>

  SetEnvIf X-Browser MSIE AuthDigestEnableQueryStringHack=On

</IfModule>
```

```
{
  # authenticated
  my $response = GET $url,
                  username => 'user1', password => 'password1';

  is ($response->code,
      200,
      'user1:password1 found');

  # set up for later
  $no_query_auth = $response->request->headers->authorization;
}

{
  # fake current MSIE behavior
  my $response = GET "$url?$query",
                  'X-Browser'   => 'MSIE',
                  Authorization => $no_query_auth;

  is ($response->code,
      200,
      'no query string in header + MSIE');
}
```

Failure!

- Of course it failed!
  - the correct code does not exist yet
- Writing the test first had two important effects
  - defined the interface
  - defined the behavior
- We often produce better code with just a little up-front thought

mod_auth_digest.c

```
else if (r_uri.query) {
  /* MSIE compatibility hack.  MSIE has some RFC issues - doesn't
  * include the query string in the uri Authorization component
  * or when computing the response component.  the second part
  * works out ok, since we can hash the header and get the same
  * result.  however, the uri from the request line won't match
  * the uri Authorization component since the header lacks the
  * query string, leaving us incompatable with a (broken) MSIE.
  *
  * workaround is to fake a query string match if in the proper
  * environment - BrowserMatch MSIE, for example.  the cool thing
  * is that if MSIE ever fixes itself the simple match ought to
  * work and this code won't be reached anyway, even if the
  * environment is set.
  */

  if (apr_table_get(r->subprocess_env,
                    "AuthDigestEnableQueryStringHack")) {
      d_uri.query = r_uri.query;
  }
}
```

Only the Beginning

- You're not finished yet!
- Our Criteria
  - MSIE users can authenticate
  - RFC compliant users still can authenticate
  - if MSIE gets fixed, users can authenticate
  - Mismatches still fail
- We have more tests to write

```
{
  # pretend MSIE fixed itself
  my $response = GET "$url?$query",
                  username    => 'user1', password => 'password1',
                  'X-Browser' => 'MSIE';

  is ($response->code,
      200,
      'a compliant response coming from MSIE');
}

{
  # this still bombs
  my $response = GET "$url?$query",
                   Authorization => $bad_query,
                   'X-Browser'   => 'MSIE';

  is ($response->code,
      400,
      'mismatched query string + MSIE');
}
```

Hopefully, you saw...

- a full digest test
- testing against an old version
- leaving a server running

Accomplishments

- Code that works as required
- Code that nobody else can break
  - as long as they run the tests
- Code that can be freely refactored or cleaned
  - formatting or whitespace changes
- Permanent place for what would otherwise be a manual intervention or one-off script

Let's Review...

Testing, Testing... 1, 2, 3

1. Perl `Makefile.PL` foo
2. Apache `httpd.conf` foo
3. Write the tests

Step 1 - Makefile.PL

```
use Apache::TestMM qw(test clean);
use Apache::TestRunPerl ();

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

# generate the test harness (t/TEST)
Apache::TestRunPerl->generate_script();
```

Step 2 – Apache Foo

- Apache needs a basic configuration to service requests
  - `ServerRoot` `t/`
  - `DocumentRoot` `t/htdocs`
  - `ErrorLog` `t/logs/error_log`
  - `Listen` `8529`
  - `LoadModule`
- `Apache-Test` "intuits" these and creates its own `httpd.conf`
- We create `t/conf/extra.conf.in`

t/conf/extra.conf.in

```
Alias /basic @DocumentRoot@

<Location /basic>
  Require valid-user
  AuthType Basic
  AuthName myrealm

  PerlAuthenHandler Apache::BasicAuth
</Location>
```

```
package My::AuthenHandler;

use Apache2::Const -compile => qw(OK HTTP_UNAUTHORIZED);

use Apache2::RequestRec ();
use Apache2::Access ();

sub handler {

  my $r = shift;

  # Get the client-supplied credentials.
  my ($status, $password) = $r->get_basic_auth_pw;

  return $status unless $status == Apache2::Const::OK;

  # Perform some custom user/password validation.
  return Apache2::Const::OK if $r->user eq $password;

  # Whoops, bad credentials.
  $r->note_basic_auth_failure;
  return Apache2::Const::HTTP_UNAUTHORIZED;
}

1;
```

Step 3 – The Tests

- Standard Perl foo

```
plan tests => 3;

ok ($condition);

etc...
```

t/01basic.t

```
{
  my $response = GET '/basic/index.html';

  is ($response->code,
      401,
      "no valid password entry");
}

{
  my $response = GET '/basic/index.html',
                     username => 'geoff',
                     password => 'geoff';

  is ($response->code,
      200,
      "geoff:geoff allowed to proceed");
}
```

Client-Side Tests

- The test file acts as a web browser
- The client is scripted to
  - query the server
  - compare server response to expected results
  - indicate success or failure
- In short, the granularity rests in the client
  - that's where all the calls to `ok()` are

Server-Side Tests

- `Apache-Test` provides a mechanism for achieving granularity in the Apache runtime
- Highly magical (not really)
- Let's see it in action...

Apache::SSLLookup

- mod_ssl exposes a few optional functions in C
  - `is_https` `# true if https://`
  - `ssl_var_lookup` `# SSL_CLIENT_VERIFY`
- `Apache::SSLLookup` provides Perl glue
  - `Apache::SSLLookup->new()`
  - `is_https()`
  - `ssl_lookup()`

What to Test?

- Class
  - compiles
- Constructor
  - defined
  - returns an object of the proper class
  - returns an object with proper attributes
- Methods
  - defined
  - do something useful

Where To Test

- Clearly this API needs a real Apache + mod_perl runtime

```
$ perl -MApache::SSLLookup -e0
Can't load 'Apache/SSLLookup/SSLLookup.so'
 for module Apache::SSLLookup:  undefined
 symbol: MP_debug_level
```

- Apache-Test++
- But even here we have choices

Options

- Client-side test
  - do a bunch of stuff on the server and simply return
    - `OK` if it went OK
    - `500` if it didn't
  - testing in aggregate
- Server-side test
  - much more granular
  - each test can individually pass or fail
- It's all about where you call `ok()`

```
package TestSSL::02new;

use Apache::Test qw(-withtestmore);

use Apache::Const -compile => qw(OK);

sub handler {

  my $r = shift;

  plan $r, tests => 2;

  {
    use_ok('Apache::SSLLookup');
  }

  {
    can_ok('Apache::SSLLookup', 'new');
  }

  return Apache2::Const::OK
}
1;
```

A Brief Digression...

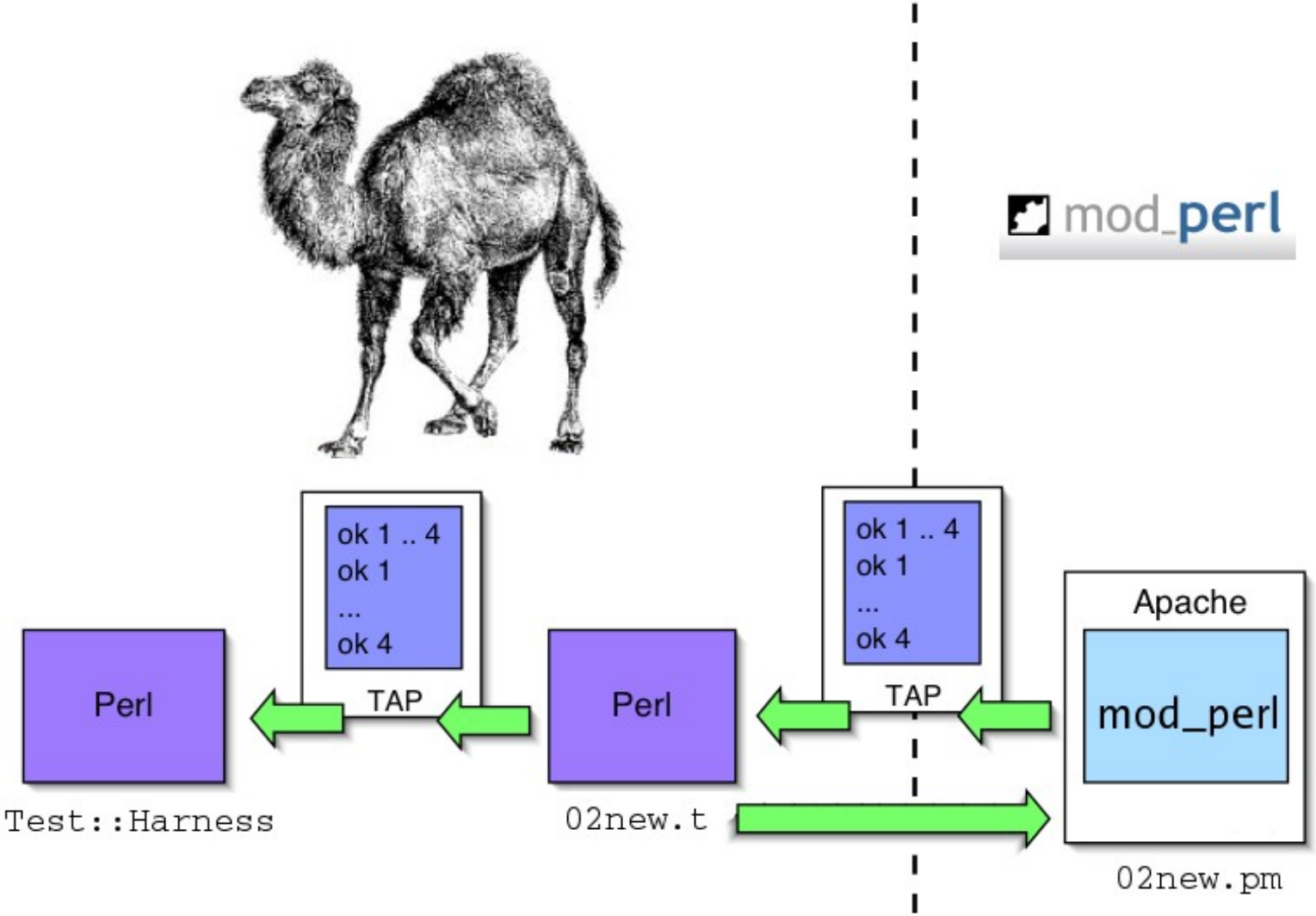
- TAP – the Test Anything Protocol
  - aka

```
1..2
ok 1
# this is a comment
not ok 2
```

- Documented in `Test::Harness::TAP`
- Why the name?

Marketing++

- Once TAP was properly branded things started happening
- There are now TAP implementations in
  - PHP (`test-more.php`)
  - C (`libtap`)
  - JavaScript (`TestSimple.js`)
- Once you can generate TAP all you need to do is feed it to `Test::Harness`


mod_perl
ok 1 .. 4
ok 1
...
ok 4
TAP
Perl
Test::Harness
ok 1 .. 4
ok 1
...
ok 4
TAP
Perl
02new.t
Apache
mod_perl
02new.pm

Magic

- Magical things happen if you follow a specific filesystem pattern
- In our case

  `t/response/TestSSL/02new.pm`

automagically generates

  `t/ssl/02new.t`

and an entry in

  `t/conf/httpd.conf`

t/ssl/02new.t

```
# WARNING: this file is generated, do not edit
# 01: Apache/TestConfig.pm:898
# 02: /Apache/TestConfig.pm:916
# 03: Apache/TestConfigPerl.pm:138
# 04: Apache/TestConfigPerl.pm:553
# 05: Apache/TestConfig.pm:584
# 06: Apache/TestConfig.pm:599
# 07: Apache/TestConfig.pm:1536
# 08: Apache/TestRun.pm:501
# 09: Apache/TestRunPerl.pm:80
# 10: Apache/TestRun.pm:720
# 11: Apache/TestRun.pm:720
# 12: t/TEST:28

use Apache::TestRequest 'GET_BODY_ASSERT';
print GET_BODY_ASSERT "/TestSSL__02new";
```

t/conf/httpd.conf

```
<Location /TestSSL__02new>
    SetHandler modperl
    PerlResponseHandler TestSSL::02new
</Location>
```

```
sub handler {

  plan shift, tests => 7;

  use_ok('Apache::SSLLookup');

  can_ok('Apache::SSLLookup', 'new');

  eval { my $r = Apache::SSLLookup->new() };

  like ($@, qr/Usage:/,  'new() requires arguments');

  eval { my $r = Apache::SSLLookup->new({}) };

  like ($@, qr/invoked by a 'unknown' object with no 'r' key/,
        'new() requires an object');

  eval { my $r = Apache::SSLLookup->new(bless {}, 'foo') };

  like ($@, qr/invoked by a 'foo' object with no 'r' key/,
        'new() requires an Apache2::RequestRec object');

  my $r = Apache::SSLLookup->new($r);

  isa_ok($r, 'Apache::SSLLookup');
  isa_ok($r, 'Apache2::RequestRec');

  return Apache2::Const::OK;
}
```

SSL

- We're testing an SSL interface
- Why not actually test it under SSL?

t/response/TestLive/01api.pm

```
sub handler {

  my $r = shift;

  plan $r, tests => 2;

  {
    $r = Apache::SSLLookup->new($r);

    SKIP : {
      skip 'apache 2.0.51 required', 1
        unless have_min_apache_version('2.0.51');

      ok($r->is_https,
         'is_https() returned true');
    }

    ok ($r->ssl_lookup('https'),
        'HTTPS variable returned true');
  }

  return Apache2::Const::OK;
}
```

t/live/01api.t

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

my $hostport = Apache::Test::config
               ->{vhosts}
               ->{TestLive}
               ->{hostport};

my $url = "https://$hostport/TestLive__01api/";

print GET_BODY_ASSERT $url;
```

t/conf/ssl/ssl.conf.in

```
PerlModule Apache::SSLLookup

<IfModule @ssl_module@>
  <VirtualHost TestLive>
    SSLEngine on
    SSLCertificateFile @SSLCA@/asf/certs/server.crt
    SSLCertificateKeyFile @SSLCA@/asf/keys/server.pem

    <Location /TestLive__01api>
      SetHandler modperl
      PerlResponseHandler TestLive::01api
    </Location>
  </VirtualHost>
</IfModule>
```

Hopefully, you saw...

- CA generation
- Hitting the running server with a browser

More Magic

- Take this code

```
package My::NeedsToBePreLoaded;

BEGIN {
  Apache->push_handlers(
     PerlChildInitHandler => sub { ... }
  );
}
```

- We need `PerlModule` or the `PerlChildInitHandler` won't run for our tests
  - or `startup.pl`, of course

The Solution

```
package TestMy::NeedsToBePreloaded::01compile;

use Apache::Test qw(-withtestmore);

sub handler {

  plan shift, tests => 1;

  use_ok (My::NeedsToBePreloaded);

  return 0;
}
1;

__DATA__
# make sure that the PerlChildInitHandler
# invokes the BEGIN block
PerlModule My::NeedsToBePreloaded
SetEnv PRELOAD test
```

t/conf/httpd.conf

```
# make sure that the PerlChildInitHandler
# invokes the BEGIN block
PerlModule My::NeedsToBePreloaded

<Location /TestMy__NeedsToBePreloaded__01compile>
    SetEnv PRELOAD test
    SetHandler perl-script
    PerlHandler TestMy::NeedsToBePreloaded::01compile
</Location>
```

__DATA__

- Stuff Apache foo into __DATA__
- Apache-Test converts to httpd.conf
  - with substitutions, like @DocumentRoot@
- Localized to generated <Location>
  - PerlModule, Alias, etc are special
- Use <Base></Base> to force top-level configuration

EOP

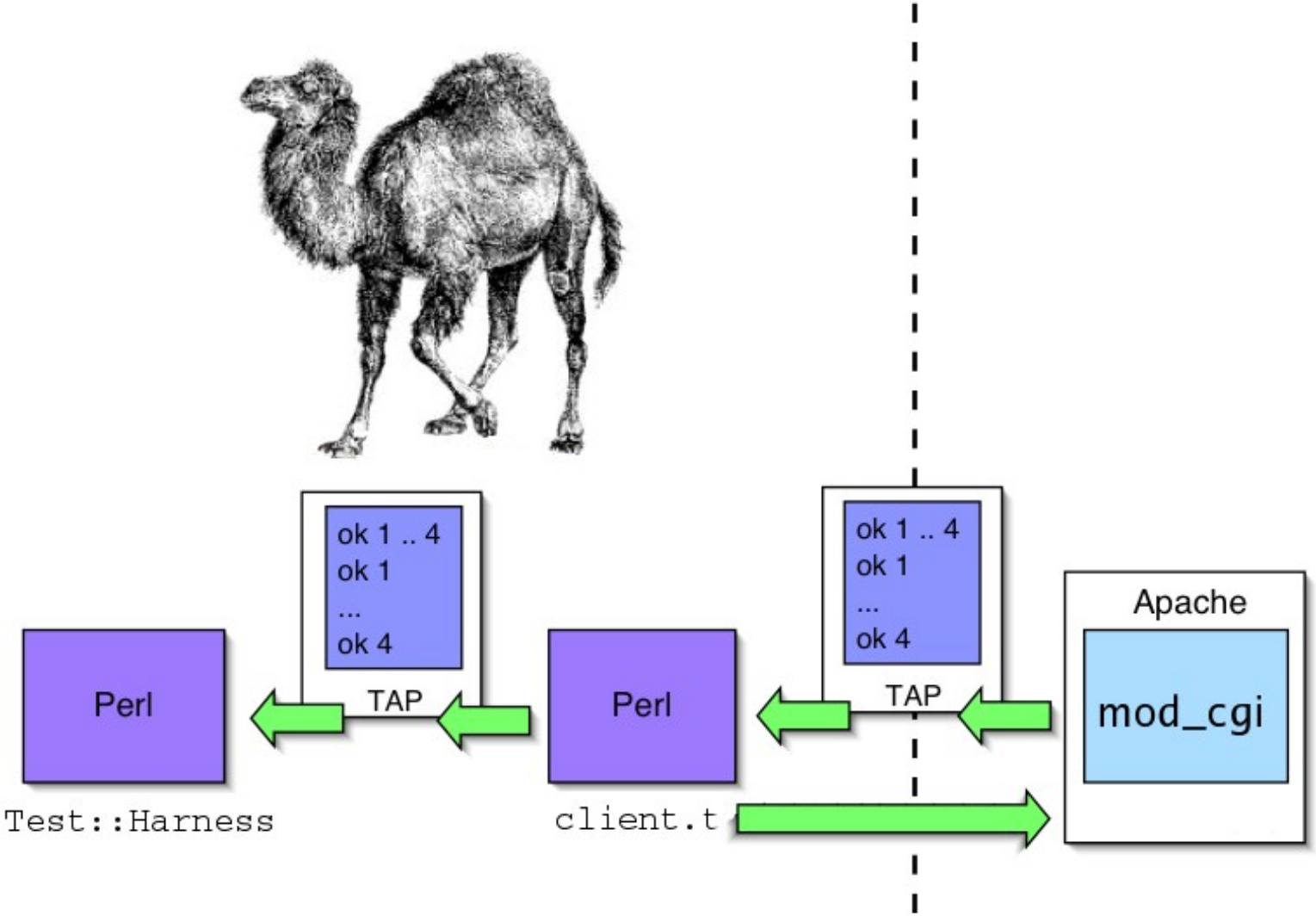
- `Apache-Test` is an Equal Opportunity Platform
- You can now use `Apache-Test` to test PHP too!
  - server-side
  - client-side
- `make test` runs PHP!

t/response/TestAPI/01php.php

```
<?php

  include 'test-more.php';

  plan(7);

  ok(true, 'ok() pass');
  ok(false, 'ok() fail');

  is('foo', 'foo', 'is() pass');

  isnt('foo', 'bar', 'isnt() pass');

  like('foo', '/oo/', 'like() pass');

  unlike('foo', '/oo/', 'unlike() fail');

  diag('diag()');
?>
```

The Server Side

- mod_perl and PHP are just examples
- As long as you can get TAP from the server to `Test::Harness` you're golden
- Automagic client generation is helpful but not required
  - write a CGI script that calls `plan()` and `ok()`
  - `GET` the script from `t/foo.t`
  - `print` CGI script response to `STDOUT`
- `Apache-Test` takes care of the hard parts


ok 1 .. 4
ok 1
...
ok 4
TAP
Perl
Test::Harness
ok 1 .. 4
ok 1
...
ok 4
TAP
Perl
client.t
Apache
mod_cgi

Huh?

- I know you guys are tired... but
- I just gave you the keys to the kingdom
- If Perl is your game, you can test anything

Complex

- The cool answer we'll dissect in detail
  - `WebService-CaptchasDotNet`
- Two versions of `t/80cgi.t`
- `t/80cgi-simple.t`
  - assumes `Apache-Test` is present
  - like you would for work
- `t/80cgi.t`
  - does not assume `Apache-Test`
  - like you would for CPAN

t/80cgi-simple.t

```
use File::Spec::Functions qw(catfile);
use Apache::Test qw(-withtestmore);
use Apache::TestRequest;
use Apache::TestUtil qw(t_write_perl_script);

plan tests => 4, need need_lwp, need_cgi;

Test::More->builder->current_test(4);


my $file = catfile(Apache::Test::vars('documentroot'),
                   'test.cgi');

t_write_perl_script($file, <DATA>);

print GET_BODY_ASSERT '/test.cgi';

__END__
# some cgi script that uses is(), ok(), etc...
```

t/80cgi-simple.t

```
# whee, no shebang line!

print "Content-Type: text/plain\n\n";

use Test::More no_plan => 1;
Test::More->builder->no_header(1);

is ('this',
    'cool',
    "isn't this cool?");
```

t/conf/extra.conf.in

```
# make sure .cgi files are executed

<Directory @DocumentRoot@>
  Options +ExecCGI
  AddHandler cgi-script .cgi
</Directory>
```

Where is `Apache-Test`?

- mod_perl 2.0
- CPAN

More Information

- perl.com
  - http://www.perl.com/pub/a/2003/05/22/testing.html
- Apache-Test tutorial
  - http://perl.apache.org/docs/general/testing/testing.html
- Apache-Test manpages
- *mod_perl Developer's Cookbook*
  - http://www.modperlcookbook.org/

Slides

- These slides freely available at some long URL you will never remember…

http://www.modperlcookbook.org/~geoff/slides/ApacheCon

- Linked to from my homepage

http://www.modperlcookbook.org/~geoff/