Test-Driven Apache Module Development

Geoffrey Young
geoff@modperlcookbook.org
Goals

• Introduction to Apache-Test
• Perl module support
• C module support
• Automagic configuration
• Test-driven development basics
• Other Goodness™
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!
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
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();
t/TEST

- t/TEST is generated by the call to generate_script()
- 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
  – DocumentRoot
  – ErrorLog
  – Listen

• Content is also generally useful
Apache-Test Defaults

• Apache-Test provides server defaults
  – ServerRoot t/
  – DocumentRoot t/htdocs
  – ErrorLog t/logs/error_log
  – Listen 8529

• Also provides an initial index.html
  http://localhost:8529/index.html

• You will probably need more than the default settings
Adding to the Default Config

• Supplement default `httpd.conf` with custom configurations

• Define `t/conf/extra.conf.in`
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>
Testing, Testing... 1, 2, 3

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

• \texttt{plan()} the number of tests

• \texttt{call ok()} for each test you plan
  – where \texttt{ok()} is any one of a number of comparison functions

• All the rest is up to you
use Apache::Test;
use Apache::TestRequest;

plan tests => 1, (need_lwp &&
 need_auth &&
 need_module('mod_perl.c'));
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
  ```perl
  plan tests => 5;
  ```
- Preconditions can be specified
  ```perl
  plan tests => 5, need_lwp;
  ```
- 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
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::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
  – 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()
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;
}

http://www.modperlcookbook.org/
Testing, Testing... 1, 2, 3

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

$ make test
$ t/TEST t/01basic.t
$ t/TEST t/01basic.t -verbose
    -preamble
    'PerlLogHandler "sub { warn shift->as_string; 0 }"'
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 \texttt{ok}\? 

- \texttt{ok()} works, but is not descriptive
- luckily, we have options
  - \texttt{Apache::TestUtil}
  - \texttt{Test::More}
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::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
Test::More functions

• Basic comparisons
  – ok()
  – is()
  – like()

• Intuitive comparisons
  – isnt()
  – unlike()

• Complex structures
  – is_deeply()
  – eq_array()
use Apache::Test;
use Apache::TestRequest;
use Apache::TestUtil;

plan tests => 1, (need_lwp &&
    need_auth &&
    need_module('mod_perl.c'));
{
    my $uri = '/authen/index.html';

    my $response = GET $uri;

    ok t_cmp($response->code,
        401,
        "no valid password entry");
}
server localhost.localdomain:8529 started
t/authen03....1..1
ok 1 - no valid password entry
ok
All tests successful.

server localhost.localdomain:8529 started
t/authen03....1..1
not ok 1 - no valid password entry

#   Failed test (t/authen03.t at line 18)
#       got: '200'
#   expected: '401'
# Looks like you failed 1 test of 1.
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");
}
```c
#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;
}
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;
}
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();
The 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
- 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;
```
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
use Apache::Test qw(:withtestmore);
use Apache::TestRequest;

use Test::More;

plan tests => 20;

foreach my $counter (1 .. 20) {

    my $response = GET_BODY '/example_ipc';

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

http://www.modperlcookbook.org/
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
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"
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?
  – MSIE users can authenticate
  – RFC compliant users still can authenticate
  – if MSIE gets fixed, users can authenticate

• 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
Wiried

• 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
<IfModule mod_auth_digest.c>

    Alias /digest @DocumentRoot@

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

</IfModule>
use Apache::Test qw(withtestmore);
use Apache::TestRequest;
use Apache::TestUtil qw(t_write_file);
use File::Spec;

use Test::More;

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

# write out the authentication file
my $file = File::Spec->catfile(Apache::Test::vars('serverroot'),
    'realm1');
t_write_file($file, <DATA>);

...

__DATA__
# user1/password1
user1:realm1:4b5df5ee44449d6b5fbf026a7756e6ee
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
  
  ```perl
  use Apache::TestUtil qw(t_write_file);
  ```

• **Accepts a file and a list of lines**
  
  ```perl
  t_write_file($file, @lines);
  ```

• **Write out the file**
  
  – including any required directories

• **Cleans up the file when script exits**
  
  – including created directories
use Apache::Test qw(:withtestmore);
use Apache::TestRequest;
use Apache::TestUtil qw(t_write_file);
use File::Spec;

use Test::More;

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

# write out the authentication file
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
<IfModule mod_auth_digest.c>

    Alias /digest @DocumentRoot@

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

</IfModule>
<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=

</IfModule>
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
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 incompatible 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
• 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');
{
    # 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');
}
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
Server-Side Tests

- So far, we have been using *.	 tests to act as clients
- Apache-Test provides a mechanism for running server-side tests
- Highly magical
- Currently, only supported for Perl handlers or PHP scripts
  - no magic for C modules (or other embedded languages, like python or parrot) yet
Say What?

• mod_ssl exposes a few optional functions
  – is_https()
  – ssl_var_lookup()

• 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

- **Method**
  - defined
  - do something useful
Options

• Client-side test
  – run a bunch of tests and return OK
  – if one test fails, return 500
  – 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::01new;

use Apache::Test qw(-withtestmore);

use Apache2::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;
use Apache::TestRequest 'GET_BODY_ASSERT';
print GET_BODY_ASSERT "[/TestSSL__01new]";
Magic

• Just like with the c-modules/directory, magical things happen if you follow a specific pattern

• In our case
  t/response/TestSSL/01new.pm

  automagically generates
  t/ssl/01new.t

  and an entry in t/conf/httpd.conf
<Location /TestSSL__01new>
    SetHandler modperl
    PerlResponseHandler TestSSL::01new
</Location>
sub handler {
  my $r = shift;
  plan $r, tests => 4;

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

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

  {
    eval {
      $r = Apache::SSLLookup->new(bless {}, 'foo')
    }
    like ($@,
      qr/'new' invoked by a `foo' object with no `r' key/,
      'new() requires an Apache::RequestRec object');
  }

  {
    $r = Apache::SSLLookup->new($r);
    isa_ok($r, 'Apache::SSLLookup');
  }

  return Apache2::Const::OK;
}
sub handler {

    my $r = shift;

    plan $r, tests => 3;

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

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

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

        ok(defined $r->is_https,
            'is https returned a defined value');
    }

    return Apache2::Const::OK;
}
SSL

- We're testing an SSL interface
- Why not actually test it under SSL
sub handler {

    my $r = shift;

    plan $r, tests => 2;

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

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

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

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

    return Apache2::Const::OK;
}
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;
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>
Where is Apache-Test?

• mod_perl 2.0
• CPAN
• httpd-test project
  – http://httpd.apache.org/test/
  – test-dev@httpd.apache.org
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/

• All the tests in the perl-framework part of the httpd-test project
Slides

• These slides freely available at some long URL you will never remember...

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

• Linked to from my homepage

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