NAME

Catalyst::Manual::Tutorial::08_Testing - Catalyst Tutorial - Chapter 8: Testing

OVERVIEW

This is **Chapter 8 of 10** for the Catalyst tutorial.

DESCRIPTION

You may have noticed that the Catalyst Helper scripts automatically create basic `.t` test scripts under the `t` directory. This chapter of the tutorial briefly looks at how these tests can be used not only to ensure that your application is working correctly at the present time, but also provide automated regression testing as you upgrade various pieces of your application over time.

Source code for the tutorial in included in the */home/catalyst/Final* directory of the Tutorial Virtual machine (one subdirectory per chapter). There are also instructions for downloading the code in Catalyst::Manual::Tutorial::01_Intro.

For an excellent introduction to learning the many benefits of testing your Perl applications and modules, you might want to read 'Perl Testing: A Developer's Notebook' by Ian Langworth and chromatic.

RUNNING THE "CANNED" CATALYST TESTS

There are a variety of ways to run Catalyst and Perl tests (for example, `perl Makefile.PL` and `make test`), but one of the easiest is with the `prove` command. For example, to run all of the tests in the `t`

directory, enter:

```
$ prove -wl t
```

There will be a lot of output because we have the `-Debug` flag enabled in `lib/MyApp.pm` (see the `CATALYST_DEBUG=0` tip below for a quick and easy way to reduce the clutter). Look for lines like this for errors:

```
#   Failed test 'Request should succeed'
#   at t/controller_Books.t line 8.
# Looks like you failed 1 test of 3.
```

The redirection used by the Authentication plugins will cause several failures in the default tests. You can fix this by making the following changes:

1) Change the line in `t/01app.t` that reads:

```
ok( request('/')->is_success, 'Request should succeed' );
```

to:

```
ok( request('/login')->is_success, 'Request should succeed' );
```

2) Change the line in `t/controller_Logout.t` that reads:

```
ok( request('/logout')->is_success, 'Request should succeed' );
```

to:

```
ok( request('/logout')->is_redirect, 'Request should succeed' );
```

3) Change the line in `t/controller_Books.t` that reads:

```
ok( request('/books')->is_success, 'Request should succeed' );
```

to:

```
ok( request('/books')->is_redirect, 'Request should succeed' );
```

4) Add the following statement to the top of `t/view_HTML.t`:

```
use MyApp;
```

As you can see in the `prove` command line above, the `-l` option (or `--lib` if you prefer) is used to set the location of the Catalyst `lib` directory. With this command, you will get all of the usual development server debug output, something most people prefer to disable while running tests cases. Although you can edit the `lib/MyApp.pm` to comment out the `-Debug` plugin, it's generally easier to simply set the `CATALYST_DEBUG=0` environment variable. For example:

```
    $ CATALYST_DEBUG=0 prove -wl t
```

During the `t/02pod` and `t/03podcoverage` tests, you might notice the `all skipped: set TEST_POD to enable this test` warning message. To execute the Pod-related tests, add `TEST_POD=1` to the `prove` command:

```
    $ CATALYST_DEBUG=0 TEST_POD=1 prove -wl t
```

If you omitted the Pod comments from any of the methods that were inserted, you might have to go back and fix them to get these tests to pass. :-)

Another useful option is the `verbose` (`-v`) option to `prove`. It prints the name of each test case as it is being run:

```
    $ CATALYST_DEBUG=0 prove -vwl t
```

RUNNING A SINGLE TEST

You can also run a single script by appending its name to the `prove` command. For example:

```
    $ CATALYST_DEBUG=0 prove -wl t/01app.t
```

Also note that you can also run tests directly from Perl without `prove`. For example:

```
    $ CATALYST_DEBUG=0 perl -w -Ilib t/01app.t
```

ADDING YOUR OWN TEST SCRIPT

Although the Catalyst helper scripts provide a basic level of checks "for free," testing can become significantly more helpful when you write your own tests to exercise the various parts of your application. The Test::WWW::Mechanize::Catalyst module is very popular for writing these sorts of test cases. This module extends Test::WWW::Mechanize (and therefore WWW::Mechanize) to allow you to automate the action of a user "clicking around" inside your application. It gives you all the benefits of testing on a live system without the messiness of having to use an actual web server, and a real person to do the clicking.

To create a sample test case, open the `t/live_app01.t` file in your editor and enter the following:

```
    #!/usr/bin/env perl

    use strict;
    use warnings;
    use Test::More;
```

```
# Need to specify the name of your app as arg on next line
# Can also do:
#   use Test::WWW::Mechanize::Catalyst "MyApp";

BEGIN { use_ok("Test::WWW::Mechanize::Catalyst" => "MyApp") }

# Create two 'user agents' to simulate two different users ('test01' & 'test02')
my $ua1 = Test::WWW::Mechanize::Catalyst->new;
my $ua2 = Test::WWW::Mechanize::Catalyst->new;

# Use a simplified for loop to do tests that are common to both users
# Use get_ok() to make sure we can hit the base URL
# Second arg = optional description of test (will be displayed for failed tests)
# Note that in test scripts you send everything to 'http://localhost'
$_->get_ok("http://localhost/", "Check redirect of base URL") for $ua1, $ua2;
# Use title_is() to check the contents of the <title>...</title> tags
$_->title_is("Login", "Check for login title") for $ua1, $ua2;
# Use content_contains() to match on text in the html body
$_->content_contains("You need to log in to use this application",
    "Check we are NOT logged in") for $ua1, $ua2;

# Log in as each user
# Specify username and password on the URL
$ua1->get_ok("http://localhost/login?username=test01&password=mypass", "Login 'test01'");
# Could make user2 like user1 above, but use the form to show another way
$ua2->submit_form(
    fields => {
        username => 'test02',
        password => 'mypass',
    });

# Go back to the login page and it should show that we are already logged in
$_->get_ok("http://localhost/login", "Return to '/login'") for $ua1, $ua2;
$_->title_is("Login", "Check for login page") for $ua1, $ua2;
$_->content_contains("Please Note: You are already logged in as ",
    "Check we ARE logged in" ) for $ua1, $ua2;

# 'Click' the 'Logout' link (see also 'text_regex' and 'url_regex' options)
$_->follow_link_ok({n => 4}, "Logout via first link on page") for $ua1, $ua2;
$_->title_is("Login", "Check for login title") for $ua1, $ua2;
$_->content_contains("You need to log in to use this application",
    "Check we are NOT logged in") for $ua1, $ua2;

# Log back in
$ua1->get_ok("http://localhost/login?username=test01&password=mypass",
    "Login 'test01'");
$ua2->get_ok("http://localhost/login?username=test02&password=mypass",
    "Login 'test02'");
# Should be at the Book List page... do some checks to confirm
$_->title_is("Book List", "Check for book list title") for $ua1, $ua2;

$ua1->get_ok("http://localhost/books/list", "'test01' book list");
$ua1->get_ok("http://localhost/login", "Login Page");
$ua1->get_ok("http://localhost/books/list", "'test01' book list");

$_->content_contains("Book List", "Check for book list title") for $ua1, $ua2;
# Make sure the appropriate logout buttons are displayed
$_->content_contains("/logout\">User Logout</a>",
    "Both users should have a 'User Logout'") for $ua1, $ua2;
$ua1->content_contains("/books/form_create\">Admin Create</a>",
    "'test01' should have a create link");
$ua2->content_lacks("/books/form_create\">Admin Create</a>",
    "'test02' should NOT have a create link");

$ua1->get_ok("http://localhost/books/list", "View book list as 'test01'");
```

```
    # User 'test01' should be able to create a book with the "formless create" URL
    $ua1->get_ok("http://localhost/books/url_create/TestTitle/2/4",
        "'test01' formless create");
    $ua1->title_is("Book Created", "Book created title");
    $ua1->content_contains("Added book 'TestTitle'", "Check title added OK");
    $ua1->content_contains("by 'Stevens'", "Check author added OK");
    $ua1->content_contains("with a rating of 2.", "Check rating added");
    # Try a regular expression to combine the previous 3 checks & account for whitespace
    $ua1->content_like(qr/Added book 'TestTitle'\s+by 'Stevens'\s+with a rating of 2./,
        "Regex check");

    # Make sure the new book shows in the list
    $ua1->get_ok("http://localhost/books/list", "'test01' book list");
    $ua1->title_is("Book List", "Check logged in and at book list");
    $ua1->content_contains("Book List", "Book List page test");
    $ua1->content_contains("TestTitle", "Look for 'TestTitle'");

    # Make sure the new book can be deleted
    # Get all the Delete links on the list page
    my @delLinks = $ua1->find_all_links(text => 'Delete');
    # Use the final link to delete the last book
    $ua1->get_ok($delLinks[$#delLinks]->url, 'Delete last book');
    # Check that delete worked
    $ua1->content_contains("Book List", "Book List page test");
    $ua1->content_like(qr/Deleted book \d+/, "Deleted book #");

    # User 'test02' should not be able to add a book
    $ua2->get_ok("http://localhost/books/url_create/TestTitle2/2/5", "'test02' add");
    $ua2->content_contains("Unauthorized!", "Check 'test02' cannot add");

    done_testing;
```

The `live_app.t` test cases uses copious comments to explain each step of the process. In addition to the techniques shown here, there are a variety of other methods available in Test::WWW::Mechanize::Catalyst (for example, regex-based matching). Consult Test::WWW::Mechanize::Catalyst, Test::WWW::Mechanize, WWW::Mechanize, and Test::More for more detail.

TIP: For *unit tests* vs. the "full application tests" approach used by Test::WWW::Mechanize::Catalyst, see Catalyst::Test.

Note: The test script does not test the `form_create` and `form_create_do` actions. That is left as an exercise for the reader (you should be able to complete that logic using the existing code as a template).

To run the new test script, use a command such as:

```
    $ CATALYST_DEBUG=0 prove -vwl t/live_app01.t
```

or

```
    $ DBIC_TRACE=0 CATALYST_DEBUG=0 prove -vwl t/live_app01.t
```

Experiment with the `DBIC_TRACE`, `CATALYST_DEBUG` and `-v` settings. If you find that there are errors, use the techniques discussed in the "Catalyst Debugging" section (Chapter 7) to isolate and fix any problems.

If you want to run the test case under the Perl interactive debugger, try a command such as:

```
$ DBIC_TRACE=0 CATALYST_DEBUG=0 perl -d -Ilib t/live_app01.t
```

Note that although this tutorial uses a single custom test case for simplicity, you may wish to break your tests into different files for better organization.

TIP: If you have a test case that fails, you will receive an error similar to the following:

```
#   Failed test 'Check we are NOT logged in'
#   in t/live_app01.t at line 31.
#     searched: "\x{0a}<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Tran"...
#   can't find: "You need to log in to use this application."
```

Unfortunately, this only shows us the first 50 characters of the HTML returned by the request -- not enough to determine where the problem lies. A simple technique that can be used in such situations is to temporarily insert a line similar to the following right after the failed test:

```
diag $ua1->content;
```

This will cause the full HTML returned by the request to be displayed.

Another approach to see the full HTML content at the failure point in a series of tests would be to insert a "`$DB::single=1;` right above the location of the failure and run the test under the Perl debugger (with `-d`) as shown above. Then you can use the debugger to explore the state of the application right before or after the failure.

SUPPORTING BOTH PRODUCTION AND TEST DATABASES

You may wish to leverage the techniques discussed in this tutorial to maintain both a "production database" for your live application and a "testing database" for your test cases. One advantage to Test::WWW::Mechanize::Catalyst is that it runs your full application; however, this can complicate things when you want to support multiple databases.

DATABASE CONFIG SWITCHING IN YOUR MODEL CLASS

One solution is to allow the database specification to be overridden with an environment variable. For example, open `lib/MyApp/Model/DB.pm` in your editor and change the `__PACKAGE__->config(...` declaration to resemble:

```
my $dsn = $ENV{MYAPP_DSN} ||= 'dbi:SQLite:myapp.db';
__PACKAGE__->config(
    schema_class => 'MyApp::Schema',

    connect_info => {
        dsn => $dsn,
        user => '',
        password => '',
        on_connect_do => q{PRAGMA foreign_keys = ON},
    }
);
```

Then, when you run your test case, you can use commands such as:

```
$ cp myapp.db myappTEST.db
$ CATALYST_DEBUG=0 MYAPP_DSN="dbi:SQLite:myappTEST.db" prove -vwl t/live_app01.t
```

This will modify the DSN only while the test case is running. If you launch your normal application without the `MYAPP_DSN` environment variable defined, it will default to the same `dbi:SQLite:myapp.db` as before.

DATABASE CONFIG SWITCHING USING MULTIPLE CONFIG FILES

Catalyst::Plugin::ConfigLoader has functionality to load multiple config files based on environment variables, allowing you to override your default (production) database connection settings during development (or vice versa).

Setting `$ENV{ MYAPP_CONFIG_LOCAL_SUFFIX }` to 'testing' in your test script results in loading of an additional config file named `myapp_testing.conf` after `myapp.conf` which will override any parameters in `myapp.conf`.

You should set the environment variable in the BEGIN block of your test script to make sure it's set before your Catalyst application is started.

The following is an example for a config and test script for a DBIx::Class model named MyDB and a controller named Foo:

myapp_testing.conf:

```
<Model::MyDB>
    <connect_info>
        dsn dbi:SQLite:myapp.db
    </connect_info>
</Model::MyDB>
```

t/controller_Foo.t:

```
use strict;
use warnings;
use Test::More;

BEGIN {
    $ENV{ MYAPP_CONFIG_LOCAL_SUFFIX } = 'testing';
}

eval "use Test::WWW::Mechanize::Catalyst 'MyApp'";
plan $@
    ? ( skip_all => 'Test::WWW::Mechanize::Catalyst required' )
    : ( tests => 2 );

ok( my $mech = Test::WWW::Mechanize::Catalyst->new, 'Created mech object' );

$mech->get_ok( 'http://localhost/foo' );
```

You can jump to the next chapter of the tutorial here: Advanced CRUD

AUTHOR

Kennedy Clark, hkclark@gmail.com

Feel free to contact the author for any errors or suggestions, but the best way to report issues is via the CPAN RT Bug system at https://rt.cpan.org/Public/Dist/Display.html?Name=Catalyst-Manual.