NAME

Catalyst::Manual::Tutorial::07_Debugging - Catalyst Tutorial - Chapter 7: Debugging

OVERVIEW

This is **Chapter 7 of 10** for the Catalyst tutorial.

DESCRIPTION

This chapter of the tutorial takes a brief look at the primary options available for troubleshooting Catalyst applications.

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.

Note that when it comes to debugging and troubleshooting, there are two camps:

- Fans of `log` and `print` statements embedded in the code.
- Fans of interactive debuggers.

Catalyst is able to easily accommodate both styles of debugging.

LOG STATEMENTS

Folks in the former group can use Catalyst's `$c->log` facility. (See Catalyst::Log for more detail.) For example, if you add the following code to a controller action method:

```
    $c->log->info("Starting the foreach loop here");

    $c->log->debug("Value of \$id is: ".$id);
```

Then the Catalyst development server will display your message along with the other debug output. To accomplish the same thing in a TT template view use:

```
    [% c.log.debug("This is a test log message") %]
```

As with many other logging facilities, a method is defined for each of the following "logging levels" (in increasing order of severity/importance):

```
    $c->log->debug
    $c->log->info
    $c->log->warn
    $c->log->error
    $c->log->fatal
```

You can also use Data::Dumper in both Catalyst code and in TT templates. For use in Catalyst code:

```
use Data::Dumper;
$c->log->debug("\$var is: ".Dumper($c->stash->{something}));
```

and TT templates:

```
[% USE Dumper ; Dumper.dump(c.stash.something) %].
```

NOTE: Whether you are a logging fanatic or not, we strongly recommend that you take advantage of Log::Log4perl or Log::Dispatch. It's easy to use Catalyst::Log with either of these and they will provide a huge amount of extra functionality that you will want in virtually every production application you run or support.

RUNNING CATALYST UNDER THE PERL DEBUGGER

Members of the interactive-debugger fan club will also be at home with Catalyst applications. One approach to this style of Perl debugging is to embed breakpoints in your code. For example, open `lib/MyApp/Controller/Books.pm` in your editor and add the `DB::single=1` line as follows inside the `list` method (I like to "left-justify" my debug statements so I don't forget to remove them, but you can obviously indent them if you prefer):

```
    sub list :Local {
        # Retrieve the usual Perl OO '$self' for this object. $c is the Catalyst
        # 'Context' that's used to 'glue together' the various components
        # that make up the application
        my ($self, $c) = @_;

    $DB::single=1;

        # Retrieve all of the book records as book model objects and store in the
        # stash where they can be accessed by the TT template
```

```
    $c->stash->{books} = [$c->model('DB::Book')->all];

    # Set the TT template to use.  You will almost always want to do this
    # in your action methods.
    $c->stash->{template} = 'books/list.tt2';
}
```

This causes the Perl Debugger to enter "single step mode" when this command is encountered (it has no effect when Perl is run without the `-d` flag).

NOTE: The `DB` here is the Perl Debugger, not the DB model.

If you haven't done it already, enable SQL logging as before:

```
$ export DBIC_TRACE=1
```

To now run the Catalyst development server under the Perl debugger, simply prepend `perl -d` to the front of `script/myapp_server.pl`:

```
$ perl -d script/myapp_server.pl
```

This will start the interactive debugger and produce output similar to:

```
$ perl -d script/myapp_server.pl

Loading DB routines from perl5db.pl version 1.3
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(script/myapp_server.pl:16):      my $debug         = 0;

  DB<1>
```

Press the `c` key and hit `Enter` to continue executing the Catalyst development server under the debugger. Although execution speed will be slightly slower than normal, you should soon see the usual Catalyst startup debug information.

Now point your browser to http://localhost:3000/books/list and log in. Once the breakpoint is encountered in the `MyApp::Controller::list` method, the console session running the development server will drop to the Perl debugger prompt:

```
MyApp::Controller::Books::list(/home/catalyst/MyApp/script/../lib/MyApp/Controller/Books.pm:48):
48:         $c->stash->{books} = [$c->model('DB::Book')->all];

  DB<1>
```

You now have the full Perl debugger at your disposal. First use the `next` feature by typing `n` to execute the `all` method on the Book model (`n` jumps over method/subroutine calls; you can also use `s` to `single-step` into methods/subroutines):

```
  DB<1> n
SELECT me.id, me.title, me.rating, me.created, me.updated FROM book me:
MyApp::Controller::Books::list(/home/catalyst/MyApp/script/../lib/MyApp/Controller/Books.pm:53):
53:         $c->stash->{template} = 'books/list.tt2';

  DB<1>
```

This takes you to the next line of code where the template name is set. Notice that because we enabled `DBIC_TRACE=1` earlier, SQL debug output also shows up in the development server debug information.

Next, list the methods available on our `Book` model:

```
  DB<1> m $c->model('DB::Book')
()
(0+
(bool
__result_class_accessor
__source_handle_accessor
_add_alias
__bool
_build_unique_query
_calculate_score
_collapse_cond
<lines removed for brevity>

  DB<2>
```

We can also play with the model directly:

```
  DB<2> x ($c->model('DB::Book')->all)[1]->title
SELECT me.id, me.title, me.rating, me.created, me.updated FROM book me:
0  'TCP/IP Illustrated, Volume 1'
```

This uses the Perl debugger `x` command to display the title of a book.

Next we inspect the `books` element of the Catalyst `stash` (the `4` argument to the `x` command limits the depth of the dump to 4 levels):

```
  DB<3> x 4 $c->stash->{books}
0  ARRAY(0xa8f3b7c)
   0  MyApp::Model::DB::Book=HASH(0xb8e702c)
      '_column_data' => HASH(0xb8e5e2c)
         'created' => '2009-05-08 10:19:46'
         'id' => 1
         'rating' => 5
         'title' => 'CCSP SNRS Exam Certification Guide'
         'updated' => '2009-05-08 10:19:46'
      '_in_storage' => 1
<lines removed for brevity>
```

Then enter the `c` command to continue processing until the next breakpoint is hit (or the application exits):

```
  DB<4> c
SELECT author.id, author.first_name, author.last_name FROM ...
```

Finally, press `Ctrl+C` to break out of the development server. Because we are running inside the Perl debugger, you will drop to the debugger prompt.

```
^CCatalyst::Engine::HTTP::run(/usr/local/share/perl/5.10.0/Catalyst/Engine/HTTP.pm:260):
260:            while ( accept( Remote, $daemon ) ) {

DB<4>
```

Finally, press `q` to exit the debugger and return to your OS shell prompt:

```
  DB<4> q
$
```

For more information on using the Perl debugger, please see `perldebug` and `perldebtut`. For those daring souls out there, you can dive down even deeper into the magical depths of this fine debugger by checking out `perldebguts`.

You can also type `h` or `h h` at the debugger prompt to view the built-in help screens.

For an excellent book covering all aspects of the Perl debugger, we highly recommend reading 'Pro Perl Debugging' by Richard Foley.

Oh yeah, before you forget, be sure to remove the `DB::single=1` line you added above in `lib/MyApp/Controller/Books.pm`.

DEBUGGING MODULES FROM CPAN

Although the techniques discussed above work well for code you are writing, what if you want to use print/log/warn messages or set breakpoints in code that you have installed from CPAN (or in module that ship with Perl)? One helpful approach is to place a copy of the module inside the `lib` directory of your Catalyst project. When Catalyst loads, it will load from inside your `lib` directory first, only turning to the global modules if a local copy cannot be found. You can then make modifications such as adding a `$DB::single=1` to the local copy of the module without risking the copy in the original location. This can also be a great way to "locally override" bugs in modules while you wait for a fix on CPAN.

Matt Trout has suggested the following shortcut to create a local copy of an installed module:

```
mkdir -p lib/Module; cp `perldoc -l Module::Name` lib/Module/
```

Note: If you are following along in Debian 6 or Ubuntu, you will need to install the `perl-doc` package to use the `perldoc` command. Use `sudo aptitude install perl-doc` to do that.

For example, you could make a copy of Catalyst::Plugin::Authentication with the following command:

```
mkdir -p lib/Catalyst/Plugin; cp \
    `perldoc -l Catalyst::Plugin::Authentication` lib/Catalyst/Plugin
```

You can then use the local copy inside your project to place logging messages and/or breakpoints for further study of that module.

Note: Matt has also suggested the following tips for Perl debugging:

- Check the version of an installed module:

```
    perl -M<mod_name> -e 'print "$<mod_name>::VERSION\n"'
```

  For example:

```
    $ perl -MCatalyst::Plugin::Authentication -e \
        'print $Catalyst::Plugin::Authentication::VERSION;'
    0.07
```

  and if you are using bash aliases:

```
    alias pmver="perl -le '\$m = shift; eval qq(require \$m) \
        or die qq(module \"\$m\" is not installed\\n); \
        print \$m->VERSION'"
```

- Check if a modules contains a given method:

```
    perl -MModule::Name -e 'print Module::Name->can("method");'
```

  For example:

```
    $ perl -MCatalyst::Plugin::Authentication -e \
        'print Catalyst::Plugin::Authentication->can("user");'
    CODE(0x9c8db2c)
```

  If the method exists, the Perl `can` method returns a coderef. Otherwise, it returns undef and nothing will be printed.

TT DEBUGGING

If you run into issues during the rendering of your template, it might be helpful to enable TT `DEBUG` options. You can do this in a Catalyst environment by adding a `DEBUG` line to the `__PACKAGE__->config>` declaration in `lib/MyApp/View/HTML.pm`:

```
__PACKAGE__->config({
    TEMPLATE_EXTENSION => '.tt2',
    DEBUG              => 'undef',
});
```

There are a variety of options you can use, such as 'undef', 'all', 'service', 'context', 'parser' and 'provider'. See Template::Constants for more information (remove the `DEBUG_` portion of the name shown in the TT docs and convert to lower case for use inside Catalyst).

NOTE: Please be sure to disable TT debug options before continuing with the tutorial (especially the 'undef' option -- leaving this enabled will conflict with several of the conventions used by this tutorial to leave some variables undefined on purpose).

Happy debugging.

You can jump to the next chapter of the tutorial here: Testing

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.