NAME

Catalyst::Manual::Tutorial::03_MoreCatalystBasics - Catalyst Tutorial - Chapter 3: More Catalyst Application Development Basics

OVERVIEW

This is **Chapter 3 of 10** for the Catalyst tutorial.

Tutorial Overview

1. Introduction
2. Catalyst Basics
3. **03_More Catalyst Basics**
4. Basic CRUD
5. Authentication
6. Authorization
7. Debugging

DESCRIPTION

This chapter of the tutorial builds on the work done in Chapter 2 to explore some features that are more typical of "real world" web applications. From this chapter of the tutorial onward, we will be building a simple book database application. Although the application will be too limited to be of use to anyone, it should provide a basic environment where we can explore a variety of features used in virtually all web 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.

Please take a look at "STARTING WITH THE TUTORIAL VIRTUAL MACHINE" in Catalyst::Manual::Tutorial::01_Intro before doing the rest of this tutorial. Although the tutorial should work correctly under most any recent version of Perl running on any operating system, the tutorial has been written using the virtual machine that is available for download. The entire tutorial has been tested to be sure it runs correctly in this environment, so it is the most trouble-free way to get started with Catalyst.

CREATE A NEW APPLICATION

The remainder of the tutorial will build an application called `MyApp`. First use the Catalyst `catalyst.pl` script to initialize the framework for the `MyApp` application (make sure you aren't still inside the directory of the `Hello` application from the previous chapter of the tutorial or in a directory that already has a "MyApp" subdirectory):

```
    $ catalyst.pl MyApp
    created "MyApp"
    created "MyApp/script"
    created "MyApp/lib"
    created "MyApp/root"
    ...
    created "MyApp/script/myapp_create.pl"
    Change to application directory and Run "perl Makefile.PL" to make sure your install is complete
```

And change the "MyApp" directory the helper created:

```
    $ cd MyApp
```

This creates a similar skeletal structure to what we saw in Chapter 2 of the tutorial, except with `MyApp` and `myapp` substituted for `Hello` and `hello`. (As noted in Chapter 2, omit the ".pl" from the command if you are using Strawberry Perl.)

EDIT THE LIST OF CATALYST PLUGINS

One of the greatest benefits of Catalyst is that it has such a large library of base classes and plugins available that you can use to easily add functionality to your application. Plugins are used to seamlessly integrate existing Perl modules into the overall Catalyst framework. In general, they do this by adding additional methods to the `context` object (generally written as `$c`) that Catalyst passes to every component throughout the framework.

Take a look at the file `lib/MyApp.pm` that the helper created above. By default, Catalyst enables three plugins/flags:

- `-Debug` Flag

  Enables the Catalyst debug output you saw when we started the `script/myapp_server.pl` development server earlier. You can remove this item when you place your application into production.

  To be technically correct, it turns out that `-Debug` is not a plugin, but a *flag*. Although most of the items specified on the `use Catalyst` line of your application class will be plugins, Catalyst supports a limited number of flag options (of these, `-Debug` is the most common). See the documentation for https://metacpan.org/module/Catalyst|Catalyst.pm to get details on other flags (currently `-Engine`, `-Home`, `-Log`, and `-Stats`).

  If you prefer, there are several other ways to enable debug output:

  - the `$c->debug` method on the `$c` Catalyst context object
  - the `-d` option on the `script/myapp_server.pl` script
  - the `CATALYST_DEBUG=1` environment variable (or `CATALYST_DEBUG=0` to temporarily disable debug output)

  TIP: Depending on your needs, it can be helpful to permanently remove `-Debug` from `lib/MyApp.pm` and then use the `-d` option to `script/myapp_server.pl` to re-enable it when needed. We will not be using that approach in the tutorial, but feel free to make use of it in your own projects.

- Catalyst::Plugin::ConfigLoader

  `ConfigLoader` provides an automatic way to load configurable parameters for your application from a central Config::General file (versus having the values hard-coded inside your Perl modules). Config::General uses syntax very similar to Apache configuration files. We will see how to use this feature of Catalyst during the authentication and authorization sections (Chapter 5 and Chapter 6).

  IMPORTANT NOTE: If you are using a version of Catalyst::Devel prior to version 1.06, be aware that Catalyst changed the default format from YAML to the more straightforward `Config::General` style. This tutorial uses the newer `myapp.conf` file for `Config::General`. However, Catalyst supports both formats and will automatically use either `myapp.conf` or `myapp.yml` (or any other format supported by Catalyst::Plugin::ConfigLoader and Config::Any). If you are using a version of Catalyst::Devel prior to 1.06, you can convert to the newer format by simply creating the `myapp.conf` file manually and deleting `myapp.yml`. The default contents of the `myapp.conf` you create should only consist of one line:

  ```
  name MyApp
  ```

  TIP: This script can be useful for converting between configuration formats:

  ```
  perl -Ilib -e 'use MyApp; use Config::General;
      Config::General->new->save_file("myapp.conf", MyApp->config);'
  ```

- Catalyst::Plugin::Static::Simple

  `Static::Simple` provides an easy way to serve static content, such as images and CSS files, from the development server.

For our application, we want to add one new plugin to the mix. To do this, edit `lib/MyApp.pm` (this file is generally referred to as your *application class*) and delete the lines with:

```
    use Catalyst qw/
        -Debug
        ConfigLoader
        Static::Simple
    /;
```

Then replace it with:

```
    # Load plugins
    use Catalyst qw/
        -Debug
        ConfigLoader
        Static::Simple

        StackTrace
    /;
```

Note: Recent versions of `Catalyst::Devel` have used a variety of techniques to load these plugins/flags. For example, you might see the following:

```
    __PACKAGE__->setup(qw/-Debug ConfigLoader Static::Simple/);
```

Don't let these variations confuse you -- they all accomplish the same result.

This tells Catalyst to start using one additional plugin, Catalyst::Plugin::StackTrace, to add a stack trace near the top of the standard Catalyst "debug screen" (the screen Catalyst sends to your browser when an error occurs). Be aware that StackTrace output appears in your browser, not in the console window from which you're running your application, which is where logging output usually goes.

Make sure when adding new plugins you also include them as a new dependency within the Makefile.PL file. For example, after adding the StackTrace plugin the Makefile.PL should include the following line:

```
    requires 'Catalyst::Plugin::StackTrace';
```

Notes:

- `__PACKAGE__` is just a shorthand way of referencing the name of the package where it is used. Therefore, in `MyApp.pm`, `__PACKAGE__` is equivalent to `MyApp`.
- You will want to disable StackTrace before you put your application into production, but it can be helpful during development.
- When specifying plugins, you can omit `Catalyst::Plugin::` from the name. Additionally, you can spread the plugin names across multiple lines as shown here or place them all on one line.

- If you want to see what the StackTrace error screen looks like, edit `lib/MyApp/Controller/Root.pm` and put a `die "Oops";` command in the `sub index :Path :Args(0)` method. Then start the development server and open http://localhost:3000/ in your browser. You should get a screen that starts with "Caught exception in MyApp::Controller::Root->index" with sections showing a stacktrace, information about the Request and Response objects, the stash (something we will learn about soon), and the applications configuration. **Just don't forget to remove the die before you continue the tutorial!** :-)

CREATE A CATALYST CONTROLLER

As discussed earlier, controllers are where you write methods that interact with user input. Typically, controller methods respond to `GET` and `POST` requests from the user's web browser.

Use the Catalyst `create` script to add a controller for book-related actions:

```
$ script/myapp_create.pl controller Books
 exists "/home/catalyst/MyApp/script/../lib/MyApp/Controller"
 exists "/home/catalyst/MyApp/script/../t"
created "/home/catalyst/MyApp/script/../lib/MyApp/Controller/Books.pm"
created "/home/catalyst/MyApp/script/../t/controller_Books.t"
```

Then edit `lib/MyApp/Controller/Books.pm` (as discussed in [Chapter 2](#) of the Tutorial, Catalyst has a separate directory under `lib/MyApp` for each of the three parts of MVC: `Model`, `View` and `Controller`) and add the following method to the controller:

```
=head2 list

Fetch all book objects and pass to books/list.tt2 in stash to be displayed

=cut

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) = @_;

    # 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]);
    # But, for now, use this code until we create the model later
    $c->stash(books => '');

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

TIP: See [Appendix 1](#) for tips on removing the leading spaces when cutting and pasting example code from POD-based documents.

Programmers experienced with object-oriented Perl should recognize `$self` as a reference to the object where this method was called. On the other hand, `$c` will be new to many Perl programmers who have not used Catalyst before. This is the "Catalyst Context object", and it is automatically passed as the second argument to all Catalyst action methods. It is used to pass information between components and provide access to Catalyst and plugin functionality.

Catalyst Controller actions are regular Perl methods, but they make use of attributes (the "`:Local`" next to the "`sub list`" in the code above) to provide additional information to the Catalyst dispatcher logic (note that there can be an optional space between the colon and the attribute name; you will see attributes written both ways). Most Catalyst Controllers use one of five action types:

- **:Private** -- Use `:Private` for methods that you want to make into an action, but you do not want Catalyst to directly expose the method to your users. Catalyst will not map `:Private` methods to a URI. Use them for various sorts of "special" methods (the `begin`, `auto`, etc. discussed below) or for methods you want to be able to `forward` or `detach` to. (If the method is a "plain old method" that you don't want to be an action at all, then just define the method without any attribute -- you can call it in your code, but the Catalyst dispatcher will ignore it. You will also have to manually include `$c` if you want access to the context object in the method vs. having Catalyst automatically include `$c` in the argument list for you if it's a full-fledged action.)

  There are five types of "special" built-in `:Private` actions: `begin`, `end`, `default`, `index`, and `auto`.

  - With `begin`, `end`, `default`, `index` private actions, only the most specific action of each type will be called. For example, if you define a `begin` action in your controller it will *override* a `begin` action in your application/root controller -- *only* the action in your controller will be called.
  - Unlike the other actions where only a single method is called for each request, *every* auto action along the chain of namespaces will be called. Each `auto` action will be called *from the application/root controller down through the most specific class*.
- **:Path** -- `:Path` actions let you map a method to an explicit URI path. For example, "`:Path('list')`" in `lib/MyApp/Controller/Books.pm` would match on the URL `http://localhost:3000/books/list`, but "`:Path('/list')`" would match on `http://localhost:3000/list` (because of the leading slash). You can use `:Args()` to specify how many arguments an action should accept. See "Action-types" in Catalyst::Manual::Intro for more information and examples.
- **:Local** -- `:Local` is merely a shorthand for "`:Path('_name_of_method_')`". For example, these are equivalent: "`sub create_book :Local {...}`" and "`sub create_book :Path('create_book') {...}`".
- **:Global** -- `:Global` is merely a shorthand for "`:Path('/_name_of_method_')`". For example, these are equivalent: "`sub create_book :Global {...}`" and "`sub create_book :Path('/create_book') {...}`".
- **:Chained** -- Newer Catalyst applications tend to use the Chained dispatch form of action types because of its power and flexibility. It allows a series of controller methods to be automatically dispatched when servicing a single user request. See Catalyst::Manual::Tutorial::04_BasicCRUD and Catalyst::DispatchType::Chained for more information on chained actions.

You should refer to "Action-types" in Catalyst::Manual::Intro for additional information and for coverage of some lesser-used action types not discussed here (`Regex` and `LocalRegex`).

CATALYST VIEWS

As mentioned in Chapter 2 of the tutorial, views are where you render output, typically for display in the user's web browser (but can generate other types of output such as PDF or JSON). The code in `lib/MyApp/View` selects the *type* of view to use, with the actual rendering template found in the `root` directory. As with virtually every aspect of Catalyst, options abound when it comes to the specific view technology you adopt inside your application. However, most Catalyst applications use the Template Toolkit, known as TT (for more information on TT, see http://www.template-toolkit.org). Other somewhat popular view technologies include Mason (http://www.masonhq.com and http://www.masonbook.com) and HTML::Template (http://html-template.sourceforge.net).

Create a Catalyst View

When using TT for the Catalyst view, the main helper script is Catalyst::Helper::View::TT. You may also come across references to Catalyst::Helper::View::TTSite, but its use is now deprecated.

For our book application, enter the following command to enable the `TT` style of view rendering:

```
$ script/myapp_create.pl view HTML TT
 exists "/home/catalyst/MyApp/script/../lib/MyApp/View"
 exists "/home/catalyst/MyApp/script/../t"
 created "/home/catalyst/MyApp/script/../lib/MyApp/View/HTML.pm"
 created "/home/catalyst/MyApp/script/../t/view_HTML.t"
```

This creates a view called `HTML` (the first argument) in a file called `HTML.pm` that uses Catalyst::View::TT (the second argument) as the "rendering engine".

It is now up to you to decide how you want to structure your view layout. For the tutorial, we will start with a very simple TT template to initially demonstrate the concepts, but quickly migrate to a more typical "wrapper page" type of configuration (where the "wrapper" controls the overall "look and feel" of your site from a single file or set of files).

Edit `lib/MyApp/View/HTML.pm` and you should see something similar to the following:

```
__PACKAGE__->config(
    TEMPLATE_EXTENSION => '.tt',
    render_die => 1,
);
```

And update it to match:

```
__PACKAGE__->config(
    # Change default TT extension
    TEMPLATE_EXTENSION => '.tt2',
    render_die => 1,
);
```

This changes the default extension for Template Toolkit from '.tt' to '.tt2'.

You can also configure components in your application class. For example, Edit `lib/MyApp.pm` and you should see the default configuration above the call to `_PACKAGE__->setup` (your defaults could be different depending on the version of Catalyst you are using):

```
__PACKAGE__->config(
    name => 'MyApp',
    # Disable deprecated behavior needed by old applications
    disable_component_resolution_regex_fallback => 1,
);
```

Change this to match the following (insert a new `__PACKAGE__->config` below the existing statement):

```
__PACKAGE__->config(
    name => 'MyApp',
    # Disable deprecated behavior needed by old applications
    disable_component_resolution_regex_fallback => 1,
);
__PACKAGE__->config(
    # Configure the view
    'View::HTML' => {
        #Set the location for TT files
        INCLUDE_PATH => [
            __PACKAGE__->path_to( 'root', 'src' ),
        ],
    },
);
```

This changes the base directory for your template files from `root` to `root/src`.

Please stick with the settings above for the duration of the tutorial, but feel free to use whatever options you desire in your applications (as with most things in Perl, there's more than one way to do it...).

Note: We will use `root/src` as the base directory for our template files, with a full naming convention of `root/src/_controller_name_/_action_name_.tt2`. Another popular option is to use `root/` as the base (with a full filename pattern of `root/_controller_name_/_action_name_.tt2`).

Create a TT Template Page

First create a directory for book-related TT templates:

```
    $ mkdir -p root/src/books
```

Then create `root/src/books/list.tt2` in your editor and enter:

```
    [% # This is a TT comment. -%]

    [%- # Provide a title -%]
    [% META title = 'Book List' -%]

    [% # Note That the '-' at the beginning or end of TT code  -%]
    [% # "chomps" the whitespace/newline at that end of the    -%]
    [% # output (use View Source in browser to see the effect) -%]

    [% # Some basic HTML with a loop to display books -%]
    <table>
    <tr><th>Title</th><th>Rating</th><th>Author(s)</th></tr>
    [% # Display each book in a table row %]
    [% FOREACH book IN books -%]
      <tr>
        <td>[% book.title %]</td>
        <td>[% book.rating %]</td>
        <td></td>
      </tr>
    [% END -%]
    </table>
```

As indicated by the inline comments above, the `META title` line uses TT's META feature to provide a title to the "wrapper" that we will create later (and essentially does nothing at the moment). Meanwhile, the `FOREACH` loop iterates through each `book` model object and prints the `title` and `rating` fields.

The `[%` and `%]` tags are used to delimit Template Toolkit code. TT supports a wide variety of directives for "calling" other files, looping, conditional logic, etc. In general, TT simplifies the usual range of Perl operators down to the single dot (".") operator. This applies to operations as diverse as method calls, hash lookups, and list index values (see Template::Manual::Variables for details and examples). In addition to the usual Template::Toolkit module Pod documentation, you can access the TT manual at https://metacpan.org/module/Template::Manual.

TIP: While you can build all sorts of complex logic into your TT templates, you should in general keep the "code" part of your templates as simple as possible. If you need more complex logic, create helper methods in your model that abstract out a set of code into a single call from your TT template. (Note that the same is true of your controller logic as well -- complex sections of code in your controllers should often be pulled out and placed into your model objects.) In Chapter 4 of the tutorial we will explore some extremely helpful and powerful features of DBIx::Class that allow you to pull code out of your views and controllers and place it where it rightfully belongs in a model class.

Test Run The Application

To test your work so far, first start the development server:

```
    $ script/myapp_server.pl -r
```

Then point your browser to http://localhost:3000 and you should still get the Catalyst welcome page. Next, change the URL in your browser to http://localhost:3000/books/list. If you have everything working so far, you should see a web page that displays nothing other than our column headers for "Title", "Rating", and "Author(s)" -- we will not see any books until we get the database and model working below.

If you run into problems getting your application to run correctly, it might be helpful to refer to some of the debugging techniques covered in the Debugging chapter of the tutorial.

CREATE A SQLITE DATABASE

In this step, we make a text file with the required SQL commands to create a database table and load some sample data. We will use SQLite (http://www.sqlite.org), a popular database that is lightweight and easy to use. Be sure to get at least version 3. Open `myapp01.sql` in your editor and enter:

```
    --
    -- Create a very simple database to hold book and author information
    --
    PRAGMA foreign_keys = ON;
    CREATE TABLE book (
            id          INTEGER PRIMARY KEY,
            title       TEXT ,
            rating      INTEGER
    );
    -- 'book_author' is a many-to-many join table between books & authors
    CREATE TABLE book_author (
            book_id     INTEGER REFERENCES book(id) ON DELETE CASCADE ON UPDATE CASCADE,
            author_id   INTEGER REFERENCES author(id) ON DELETE CASCADE ON UPDATE CASCADE,
            PRIMARY KEY (book_id, author_id)
    );
    CREATE TABLE author (
            id          INTEGER PRIMARY KEY,
            first_name  TEXT,
            last_name   TEXT
    );
```

```
---
--- Load some sample data
---
INSERT INTO book VALUES (1, 'CCSP SNRS Exam Certification Guide', 5);
INSERT INTO book VALUES (2, 'TCP/IP Illustrated, Volume 1', 5);
INSERT INTO book VALUES (3, 'Internetworking with TCP/IP Vol.1', 4);
INSERT INTO book VALUES (4, 'Perl Cookbook', 5);
INSERT INTO book VALUES (5, 'Designing with Web Standards', 5);
INSERT INTO author VALUES (1, 'Greg', 'Bastien');
INSERT INTO author VALUES (2, 'Sara', 'Nasseh');
INSERT INTO author VALUES (3, 'Christian', 'Degu');
INSERT INTO author VALUES (4, 'Richard', 'Stevens');
INSERT INTO author VALUES (5, 'Douglas', 'Comer');
INSERT INTO author VALUES (6, 'Tom', 'Christiansen');
INSERT INTO author VALUES (7, 'Nathan', 'Torkington');
INSERT INTO author VALUES (8, 'Jeffrey', 'Zeldman');
INSERT INTO book_author VALUES (1, 1);
INSERT INTO book_author VALUES (1, 2);
INSERT INTO book_author VALUES (1, 3);
INSERT INTO book_author VALUES (2, 4);
INSERT INTO book_author VALUES (3, 5);
INSERT INTO book_author VALUES (4, 6);
INSERT INTO book_author VALUES (4, 7);
INSERT INTO book_author VALUES (5, 8);
```

Then use the following command to build a `myapp.db` SQLite database:

```
$ sqlite3 myapp.db < myapp01.sql
```

If you need to create the database more than once, you probably want to issue the `rm myapp.db` command to delete the database before you use the `sqlite3 myapp.db < myapp01.sql` command.

Once the `myapp.db` database file has been created and initialized, you can use the SQLite command line environment to do a quick dump of the database contents:

```
$ sqlite3 myapp.db
SQLite version 3.7.3
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from book;
1|CCSP SNRS Exam Certification Guide|5
2|TCP/IP Illustrated, Volume 1|5
3|Internetworking with TCP/IP Vol.1|4
4|Perl Cookbook|5
5|Designing with Web Standards|5
sqlite> .q
$
```

Or:

```
$ sqlite3 myapp.db "select * from book"
1|CCSP SNRS Exam Certification Guide|5
2|TCP/IP Illustrated, Volume 1|5
3|Internetworking with TCP/IP Vol.1|4
4|Perl Cookbook|5
5|Designing with Web Standards|5
```

As with most other SQL tools, if you are using the full "interactive" environment you need to terminate your SQL commands with a ";" (it's not required if you do a single SQL statement on the command line). Use ".q" to exit from SQLite from the SQLite interactive mode and return to your OS command prompt.

Please note that here we have chosen to use 'singular' table names. This is because the default inflection code for older versions of DBIx::Class::Schema::Loader does NOT handle plurals. There has been much philosophical discussion on whether table names should be plural or singular. There is no one correct answer, as long as one makes a choice and remains consistent with it. If you prefer plural table names (e.g. you think that they are easier to read) then see the documentation in "naming" in DBIx::Class::Schema::Loader::Base (version 0.05 or greater).

For using other databases, such as PostgreSQL or MySQL, see Appendix 2.

DATABASE ACCESS WITH DBIx::Class

Catalyst can be used with virtually any form of datastore available via Perl. For example, Catalyst::Model::DBI can be used to access databases through the traditional Perl DBI interface or you can use a model to access files of any type on the filesystem. However, most Catalyst applications use some form of object-relational mapping (ORM) technology to create objects associated with tables in a relational database, and Matt Trout's DBIx::Class (abbreviated as "DBIC") is the usual choice (this tutorial will use DBIx::Class).

Although DBIx::Class has included support for a `create=dynamic` mode to automatically read the database structure every time the application starts, its use is no longer recommended. While it can make for "flashy" demos, the use of the `create=static` mode we use below can be implemented just as quickly and provides many advantages (such as the ability to add your own methods to the overall DBIC framework, a technique that we see in Chapter 4).

Create Static DBIx::Class Schema Files

Note: If you are not following along in the Tutorial Virtual Machine, please be sure that you have version 1.27 or higher of DBD::SQLite and version 0.39 or higher of Catalyst::Model::DBIC::Schema. (The Tutorial VM already has versions that are known to work.) You can get your currently installed version numbers with the following commands.

```
    $ perl -MCatalyst::Model::DBIC::Schema\ 999
    $ perl -MDBD::SQLite\ 999
```

Before you continue, make sure your `myapp.db` database file is in the application's topmost directory. Now use the model helper with the `create=static` option to read the database with DBIx::Class::Schema::Loader and automatically build the required files for us:

```
    $ script/myapp_create.pl model DB DBIC::Schema MyApp::Schema \
        create=static dbi:SQLite:myapp.db \
        on_connect_do="PRAGMA foreign_keys = ON"
     exists "/home/catalyst/MyApp/script/../lib/MyApp/Model"
     exists "/home/catalyst/MyApp/script/../t"
    Dumping manual schema for MyApp::Schema to directory /home/catalyst/MyApp/script/../lib ...
    Schema dump completed.
    created "/home/catalyst/MyApp/script/../lib/MyApp/Model/DB.pm"
    created "/home/catalyst/MyApp/script/../t/model_DB.t"
```

Please note the '\' above. Depending on your environment, you might be able to cut and paste the text as shown or need to remove the '\' character to that the command is all on a single line.

The `script/myapp_create.pl` command breaks down like this:

- `DB` is the name of the model class to be created by the helper in the `lib/MyApp/Model` directory.
- `DBIC::Schema` is the type of the model to create. This equates to Catalyst::Model::DBIC::Schema, the standard way to use a DBIC-based model inside of Catalyst.
- `MyApp::Schema` is the name of the DBIC schema file written to `lib/MyApp/Schema.pm`.
- `create=static` causes DBIx::Class::Schema::Loader to load the schema as it runs and then write that information out into `lib/MyApp/Schema.pm` and files under the `lib/MyApp/Schema` directory.
- `dbi:SQLite:myapp.db` is the standard DBI connect string for use with SQLite.
- And finally, the `on_connect_do` string requests that DBIx::Class::Schema::Loader create foreign key relationships for us (this is not needed for databases such as PostgreSQL and MySQL, but is required for SQLite). If you take a look at `lib/MyApp/Model/DB.pm`, you will see that the SQLite pragma is propagated to the Model, so that SQLite's recent (and optional) foreign key enforcement is enabled at the start of every database connection.

If you look in the `lib/MyApp/Schema.pm` file, you will find that it only contains a call to the `load_namespaces` method. You will also find that `lib/MyApp` contains a `Schema` subdirectory, which then has a subdirectory called "Result". This "Result" subdirectory then has files named according to each of the tables in our simple database (`Author.pm`, `BookAuthor.pm`, and `Book.pm`). These three files are called "Result Classes" (or "ResultSource Classes") in DBIx::Class nomenclature. Although the Result Class files are named after tables in our database, the classes correspond to the *row-level data* that is returned by DBIC (more on this later, especially in "EXPLORING THE POWER OF DBIC" in Catalyst::Manual::Tutorial::04_BasicCRUD).

The idea with the Result Source files created under `lib/MyApp/Schema/Result` by the `create=static` option is to only edit the files below the `# DO NOT MODIFY THIS OR ANYTHING ABOVE!` warning. If you place all of your changes below that point in the file, you can regenerate the automatically created information at the top of each file should your database structure get updated.

Also note the "flow" of the model information across the various files and directories. Catalyst will initially load the model from `lib/MyApp/Model/DB.pm`. This file contains a reference to `lib/MyApp/Schema.pm`, so that file is loaded next. Finally, the call to `load_namespaces` in `Schema.pm` will load each of the "Result Class" files from the `lib/MyApp/Schema/Result` subdirectory. The final outcome is that Catalyst will dynamically create three table-specific Catalyst models every time the application starts (you can see these three model files listed in the debug output generated when you launch the application).

Additionally, the `lib/MyApp/Schema.pm` model can easily be loaded outside of Catalyst, for example, in command-line utilities and/or cron jobs. `lib/MyApp/Model/DB.pm` provides a very thin "bridge" between Catalyst and this external database model. Once you see how we can add some powerful features to our DBIC model in Chapter 4, the elegance of this approach will start to become more obvious.

NOTE: Older versions of Catalyst::Model::DBIC::Schema use the deprecated DBIx::Class `load_classes` technique instead of the newer `load_namespaces`. For new applications, please try to use `load_namespaces` since it more easily supports a very useful DBIC technique called "ResultSet Classes." If you need to convert an existing application from "load_classes" to "load_namespaces," you can use this process to automate the migration, but first make sure you have version `0.39` of Catalyst::Model::DBIC::Schema and DBIx::Class::Schema::Loader version `0.05000` or later.

```
    $ # Re-run the helper to upgrade for you
    $ script/myapp_create.pl model DB DBIC::Schema MyApp::Schema \
        create=static naming=current use_namespaces=1 \
        dbi:SQLite:myapp.db \
        on_connect_do="PRAGMA foreign_keys = ON"
```

ENABLE THE MODEL IN THE CONTROLLER

Open `lib/MyApp/Controller/Books.pm` and un-comment the model code we left disabled earlier so that your version matches the following (un-comment the line containing `[$c->model('DB::Book')->all]` and delete the next 2 lines):

```
    =head2 list

    Fetch all book objects and pass to books/list.tt2 in stash to be displayed

    =cut

    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) = @_;

        # 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 (action methods respond to user input in
        # your controllers).
        $c->stash(template => 'books/list.tt2');
    }
```

TIP: You may see the `$c->model('DB::Book')` un-commented above written as `$c->model('DB')->resultset('Book')`. The two are equivalent. Either way, `$c->model` returns a DBIx::Class::ResultSet which handles queries against the database and iterating over the set of results that is returned.

We are using the `->all` to fetch all of the books. DBIC supports a wide variety of more advanced operations to easily do things like filtering and sorting the results. For example, the following could be used to sort the results by descending title:

```
    $c->model('DB::Book')->search({}, {order_by => 'title DESC'});
```

Some other examples are provided in "Complex WHERE clauses" in DBIx::Class::Manual::Cookbook, with additional information found at "search" in DBIx::Class::ResultSet, "Searching" in DBIx::Class::Manual::FAQ, DBIx::Class::Manual::Intro and Catalyst::Model::DBIC::Schema.

Test Run The Application

First, let's enable an environment variable that causes DBIx::Class to dump the SQL statements used to access the database. This is a helpful trick when you are trying to debug your database-oriented code. Press `Ctrl-C` to break out of the development server and enter:

```
    $ export DBIC_TRACE=1
    $ script/myapp_server.pl -r
```

This assumes you are using bash as your shell -- adjust accordingly if you are using a different shell (for example, under tcsh, use `setenv DBIC_TRACE 1`).

NOTE: You can also set this in your code using `$class->storage->debug(1);`. See DBIx::Class::Manual::Troubleshooting for details (including options to log to a file instead of displaying to the Catalyst development server log).

Then launch the Catalyst development server. The log output should display something like:

```
$ script/myapp_server.pl -r
[debug] Debug messages enabled
[debug] Statistics enabled
[debug] Loaded plugins:
.----------------------------------------------------------------------------.
| Catalyst::Plugin::ConfigLoader  0.30                                       |
| Catalyst::Plugin::StackTrace  0.11                                         |
'----------------------------------------------------------------------------'

[debug] Loaded dispatcher "Catalyst::Dispatcher"
[debug] Loaded engine "Catalyst::Engine"
[debug] Found home "/home/catalyst/MyApp"
[debug] Loaded Config "/home/catalyst/MyApp/myapp.conf"
[debug] Loaded components:
.-----------------------------------------------------------------+----------.
| Class                                                           | Type     |
+-----------------------------------------------------------------+----------+
| MyApp::Controller::Books                                        | instance |
| MyApp::Controller::Root                                         | instance |
| MyApp::Model::DB                                                | instance |
| MyApp::Model::DB::Author                                        | class    |
| MyApp::Model::DB::Book                                          | class    |
| MyApp::Model::DB::BookAuthor                                    | class    |
| MyApp::View::HTML                                               | instance |
'-----------------------------------------------------------------+----------'

[debug] Loaded Private actions:
.----------------------+--------------------------------------+--------------.
| Private              | Class                                | Method       |
+----------------------+--------------------------------------+--------------+
| /default             | MyApp::Controller::Root              | default      |
| /end                 | MyApp::Controller::Root              | end          |
| /index               | MyApp::Controller::Root              | index        |
| /books/index         | MyApp::Controller::Books             | index        |
| /books/list          | MyApp::Controller::Books             | list         |
'----------------------+--------------------------------------+--------------'

[debug] Loaded Path actions:
.-------------------------------------+--------------------------------------.
| Path                                | Private                              |
+-------------------------------------+--------------------------------------+
| /                                   | /default                             |
| /                                   | /index                               |
| /books                              | /books/index                         |
| /books/list                         | /books/list                          |
'-------------------------------------+--------------------------------------'

[info] MyApp powered by Catalyst 5.80020
HTTP::Server::PSGI: Accepting connections at http://0:3000
```

NOTE: Be sure you run the `script/myapp_server.pl` command from the 'base' directory of your application, not inside the `script` directory itself or it will not be able to locate the `myapp.db` database file. You can use a fully qualified or a relative path to locate the database file, but we did not specify that when we ran the model helper earlier.

Some things you should note in the output above:

- Catalyst::Model::DBIC::Schema dynamically created three model classes, one to represent each of the three tables in our database (`MyApp::Model::DB::Author`, `MyApp::Model::DB::BookAuthor`, and `MyApp::Model::DB::Book`).
- The "list" action in our Books controller showed up with a path of `/books/list`.

Point your browser to http://localhost:3000 and you should still get the Catalyst welcome page.

Next, to view the book list, change the URL in your browser to http://localhost:3000/books/list. You should get a list of the five books loaded by the `myapp01.sql` script above without any formatting. The rating for each book should appear on each row, but the "Author(s)" column will still be blank (we will fill that in later).

Also notice in the output of the `script/myapp_server.pl` that DBIx::Class used the following SQL to retrieve the data:

```
    SELECT me.id, me.title, me.rating FROM book me
```

because we enabled DBIC_TRACE.

You now have the beginnings of a simple but workable web application. Continue on to future sections and we will develop the application more fully.

CREATE A WRAPPER FOR THE VIEW

When using TT, you can (and should) create a wrapper that will literally wrap content around each of your templates. This is certainly useful as you have one main source for changing things that will appear across your entire site/application instead of having to edit many individual files.

Configure HTML.pm For The Wrapper

In order to create a wrapper, you must first edit your TT view and tell it where to find your wrapper file.

Edit your TT view in `lib/MyApp/View/HTML.pm` and change it to match the following:

```
    __PACKAGE__->config(
        # Change default TT extension
        TEMPLATE_EXTENSION => '.tt2',
        # Set the location for TT files
        INCLUDE_PATH => [
                MyApp->path_to( 'root', 'src' ),
            ],
        # Set to 1 for detailed timer stats in your HTML as comments
        TIMER              => 0,
        # This is your wrapper template located in the 'root/src'
        WRAPPER => 'wrapper.tt2',
    );
```

Create the Wrapper Template File and Stylesheet

Next you need to set up your wrapper template. Basically, you'll want to take the overall layout of your site and put it into this file. For the tutorial, open `root/src/wrapper.tt2` and input the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" [%#
    %]"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>[% template.title or "My Catalyst App!" %]</title>
<link rel="stylesheet" href="[% c.uri_for('/static/css/main.css') %]" />
</head>

<body>
<div id="outer">
<div id="header">
    [%# Your logo could go here -%]
    <img src="[% c.uri_for('/static/images/btn_88x31_powered.png') %]" />
    [%# Insert the page title -%]
    <h1>[% template.title or site.title %]</h1>
</div>

<div id="bodyblock">
<div id="menu">
    Navigation:
    <ul>
        <li><a href="[% c.uri_for('/books/list') %]">Home</a></li>
        <li><a href="[% c.uri_for('/')
            %]" title="Catalyst Welcome Page">Welcome</a></li>
    </ul>
</div><!-- end menu -->

<div id="content">
    [%# Status and error messages %]
    <span class="message">[% status_msg %]</span>
    <span class="error">[% error_msg %]</span>
    [%# This is where TT will stick all of your template's contents. -%]
    [% content %]
</div><!-- end content -->
</div><!-- end bodyblock -->

<div id="footer">Copyright (c) your name goes here</div>
</div><!-- end outer -->

</body>
</html>
```

Notice the status and error message sections in the code above:

```
<span class="status">[% status_msg %]</span>
<span class="error">[% error_msg %]</span>
```

If we set either message in the Catalyst stash (e.g., `$c->stash->{status_msg} = 'Request was successful!'`) it will be displayed whenever any view used by that request is rendered. The `message` and `error` CSS styles can be customized to suit your needs in the `root/static/css/main.css` file we create below.

Notes:

- The Catalyst stash only lasts for a single HTTP request. If you need to retain information across requests you can use Catalyst::Plugin::Session (we will use Catalyst sessions in the Authentication chapter of the tutorial).
- Although it is beyond the scope of this tutorial, you may wish to use a JavaScript or AJAX tool such as jQuery (http://www.jquery.com) or Dojo (http://www.dojotoolkit.org).

Create A Basic Stylesheet

First create a central location for stylesheets under the static directory:

```
    $ mkdir root/static/css
```

Then open the file `root/static/css/main.css` (the file referenced in the stylesheet href link of our wrapper above) and add the following content:

```
    #header {
        text-align: center;
    }
    #header h1 {
        margin: 0;
    }
    #header img {
        float: right;
    }
    #footer {
        text-align: center;
        font-style: italic;
        padding-top: 20px;
    }
    #menu {
        font-weight: bold;
        background-color: #ddd;
    }
    #menu ul {
        list-style: none;
        float: left;
        margin: 0;
        padding: 0 0 50% 5px;
        font-weight: normal;
        background-color: #ddd;
        width: 100px;
    }
    #content {
        margin-left: 120px;
    }
    .message {
        color: #390;
    }
    .error {
        color: #f00;
    }
```

You may wish to check out a "CSS Framework" like Emastic (http://code.google.com/p/emastic/) as a way to quickly provide lots of high-quality CSS functionality.

Test Run The Application

Hit "Reload" in your web browser and you should now see a formatted version of our basic book list. (Again, the development server should have automatically restarted when you made changes to `lib/MyApp/View/HTML.pm`. If you are not using the "-r" option, you will need to hit `Ctrl-C` and manually restart it. Also note that the development server does *NOT* need to restart for changes to the TT and static files we created and edited in the `root` directory -- those updates are handled on a per-request basis.)

Although our wrapper and stylesheet are obviously very simple, you should see how it allows us to control the overall look of an entire website from two central files. To add new pages to the site, just provide a template that fills in the `content` section of our wrapper template -- the wrapper will provide the overall feel of the page.

Updating the Generated DBIx::Class Result Class Files

If you take a look at the Schema files automatically generated by DBIx::Class::Schema::Loader, you will see that it has already defined `has_many` and `belongs_to` relationships on each side of our foreign keys. For example, take a look at `lib/MyApp/Schema/Result/Book.pm` and notice the following code:

```
=head1 RELATIONS

=head2 book_authors

Type: has_many

Related object: L<MyApp::Schema::Result::BookAuthor>

=cut

__PACKAGE__->has_many(
  "book_authors",
  "MyApp::Schema::Result::BookAuthor",
  { "foreign.book_id" => "self.id" },
  { cascade_copy => 0, cascade_delete => 0 },
);
```

Each `Book` "has_many" `book_authors`, where `BookAuthor` is the many-to-many table that allows each Book to have multiple Authors, and each Author to have multiple books. The arguments to `has_many` are:

- `book_authors` - The name for this relationship. DBIC will create an accessor on the `Books` DBIC Row object with this name.
- `MyApp::Schema::Result::BookAuthor` - The name of the DBIC model class referenced by this `has_many` relationship.
- `foreign.book_id` - `book_id` is the name of the foreign key column in the *foreign* table that points back to this table.
- `self.id` - `id` is the name of the column in *this* table that is referenced by the foreign key.

See "has_many" in DBIx::Class::Relationship for additional information. Note that you might see a "hand coded" version of the `has_many` relationship above expressed as:

```
__PACKAGE__->has_many(
  "book_authors",
  "MyApp::Schema::Result::BookAuthor",
  "book_id",
);
```

Where the third argument is simply the name of the column in the foreign table. However, the hashref syntax used by DBIx::Class::Schema::Loader is more flexible (for example, it can handle "multi-column foreign keys").

Note: If you are using older versions of SQLite and related DBIC tools, you will need to manually define your `has_many` and `belongs_to` relationships. We recommend upgrading to the versions specified above. :-)

Have a look at `lib/MyApp/Schema/Result/BookAuthor.pm` and notice that there is a `belongs_to` relationship defined that acts as the "mirror image" to the `has_many` relationship we just looked at above:

```
    =head1 RELATIONS

    =head2 book

    Type: belongs_to

    Related object: L<MyApp::Schema::Result::Book>

    =cut

    __PACKAGE__->belongs_to(
      "book",
      "MyApp::Schema::Result::Book",
      { id => "book_id" },
      { join_type => "LEFT", on_delete => "CASCADE", on_update => "CASCADE" },
    );
```

The arguments are similar, but see "belongs_to" in DBIx::Class::Relationship for the details.

Although recent versions of SQLite and DBIx::Class::Schema::Loader automatically handle the `has_many` and `belongs_to` relationships, `many_to_many` relationship bridges (not technically a relationship) currently need to be manually inserted. To add a `many_to_many` relationship bridge, first edit `lib/MyApp/Schema/Result/Book.pm` and add the following text below the `# You can replace this text...` comment:

```
    # many_to_many():
    #   args:
    #     1) Name of relationship bridge, DBIC will create accessor with this name
    #     2) Name of has_many() relationship this many_to_many() is shortcut for
    #     3) Name of belongs_to() relationship in model class of has_many() above
    #   You must already have the has_many() defined to use a many_to_many().
    __PACKAGE__->many_to_many(authors => 'book_authors', 'author');
```

Note: Be careful to put this code *above* the `1;` at the end of the file. As with any Perl package, we need to end the last line with a statement that evaluates to `true`. This is customarily done with `1;` on a line by itself.

The `many_to_many` relationship bridge is optional, but it makes it easier to map a book to its collection of authors. Without it, we would have to "walk" through the `book_author` table as in `$book->book_author->first->author->last_name` (we will see examples on how to use DBIx::Class objects in your code soon, but note that because `$book->book_author` can return multiple authors, we have to use `first` to display a single author). `many_to_many` allows us to use the shorter `$book->author->first->last_name`. Note that you cannot define a `many_to_many` relationship bridge without also having the `has_many` relationship in place.

Then edit `lib/MyApp/Schema/Result/Author.pm` and add the reverse `many_to_many` relationship bridge for `Author` as follows (again, be careful to put in above the `1;` but below the `# DO NOT MODIFY THIS OR ANYTHING ABOVE!` comment):

```
    # many_to_many():
    #   args:
    #     1) Name of relationship bridge, DBIC will create accessor with this name
    #     2) Name of has_many() relationship this many_to_many() is shortcut for
    #     3) Name of belongs_to() relationship in model class of has_many() above
    #   You must already have the has_many() defined to use a many_to_many().
    __PACKAGE__->many_to_many(books => 'book_authors', 'book');
```

Run The Application

Run the Catalyst development server script with the `DBIC_TRACE` option (it might still be enabled from earlier in the tutorial, but here is an alternate way to specify the trace option just in case):

```
    $ DBIC_TRACE=1 script/myapp_server.pl -r
```

Make sure that the application loads correctly and that you see the three dynamically created model classes (one for each of the Result Classes we created).

Then hit the URL http://localhost:3000/books/list with your browser and be sure that the book list still displays correctly.

Note: You will not see the authors yet because the view isn't taking advantage of these relationships. Read on to the next section where we update the template to do that.

UPDATING THE VIEW

Let's add a new column to our book list page that takes advantage of the relationship information we manually added to our schema files in the previous section. Edit `root/src/books/list.tt2` and replace the "empty" table cell "<td></td>" with the following:

```
    ...
    <td>
      [% # NOTE: See Chapter 4 for a better way to do this!                      -%]
      [% # First initialize a TT variable to hold a list.  Then use a TT FOREACH -%]
      [% # loop in 'side effect notation' to load just the last names of the     -%]
      [% # authors into the list. Note that the 'push' TT vmethod doesn't return -%]
      [% # a value, so nothing will be printed here.  But, if you have something -%]
      [% # in TT that does return a value and you don't want it printed, you     -%]
      [% # 1) assign it to a bogus value, or                                     -%]
      [% # 2) use the CALL keyword to call it and discard the return value.      -%]
      [% tt_authors = [ ];
         tt_authors.push(author.last_name) FOREACH author = book.authors %]
      [% # Now use a TT 'virtual method' to display the author count in parens   -%]
      [% # Note the use of the TT filter "| html" to escape dangerous characters -%]
      ([% tt_authors.size | html %])
      [% # Use another TT vmethod to join & print the names & comma separators   -%]
      [% tt_authors.join(', ') | html %]
    </td>
    ...
```

IMPORTANT NOTE: Again, you should keep as much "logic code" as possible out of your views. This kind of logic belongs in your model (the same goes for controllers -- keep them as "thin" as possible and push all of the "complicated code" out to your model objects). Avoid code like you see in the previous example -- we are only using it here to show some extra features in TT until we get to the more advanced model features we will see in Chapter 4 (see "EXPLORING THE POWER OF DBIC" in Catalyst::Manual::Tutorial::04_BasicCRUD).

Then hit "Reload" in your browser (note that you don't need to reload the development server or use the `-r` option when updating TT templates) and you should now see the number of authors each book has along with a comma-separated list of the authors' last names. (If you didn't leave the development server running from the previous step, you will obviously need to start it before you can refresh your browser window.)

If you are still running the development server with `DBIC_TRACE` enabled, you should also now see five more `SELECT` statements in the debug output (one for each book as the authors are being retrieved by DBIx::Class):

```
    SELECT me.id, me.title, me.rating FROM book me:
    SELECT author.id, author.first_name, author.last_name FROM book_author me
    JOIN author author ON author.id = me.author_id WHERE ( me.book_id = ? ): '1'
    SELECT author.id, author.first_name, author.last_name FROM book_author me
    JOIN author author ON author.id = me.author_id WHERE ( me.book_id = ? ): '2'
    SELECT author.id, author.first_name, author.last_name FROM book_author me
    JOIN author author ON author.id = me.author_id WHERE ( me.book_id = ? ): '3'
    SELECT author.id, author.first_name, author.last_name FROM book_author me
    JOIN author author ON author.id = me.author_id WHERE ( me.book_id = ? ): '4'
    SELECT author.id, author.first_name, author.last_name FROM book_author me
    JOIN author author ON author.id = me.author_id WHERE ( me.book_id = ? ): '5'
```

Also note in `root/src/books/list.tt2` that we are using "| html", a type of TT filter, to escape characters such as < and > to &lt; and &gt; and avoid various types of dangerous hacks against your application. In a real application, you would probably want to put "| html" at the end of every field where a user has control over the information that can appear in that field (and can therefore inject markup or code if you don't "neutralize" those fields). In addition to "| html", Template Toolkit has a variety of other useful filters that can be found in the documentation for Template::Filters. (While we are on the topic of security and escaping of dangerous values, one of the advantages of using tools like DBIC for database access or HTML::FormFu for form management [see Chapter 9] is that they automatically handle most escaping for you and therefore dramatically increase the security of your app.)

RUNNING THE APPLICATION FROM THE COMMAND LINE

In some situations, it can be useful to run your application and display a page without using a browser. Catalyst lets you do this using the `script/myapp_test.pl` script. Just supply the URL you wish to display and it will run that request through the normal controller dispatch logic and use the appropriate view to render the output (obviously, complex pages may dump a lot of text to your terminal window). For example, if `Ctrl+C` out of the development server and then type:

```
    $ script/myapp_test.pl "/books/list"
```

You should get the same text as if you visited http://localhost:3000/books/list with the normal development server and asked your browser to view the page source. You can even pipe this HTML text output to a text-based browser using a command like:

```
$ script/myapp_test.pl "/books/list" | lynx -stdin
```

And you should see a fully rendered text-based view of your page. (If you are following along in Debian 6, type `sudo aptitude -y install lynx` to install lynx.) If you do start lynx, you can use the "Q" key to quit.

OPTIONAL INFORMATION

NOTE: The rest of this chapter of the tutorial is optional. You can skip to Chapter 4, Basic CRUD, if you wish.

Using 'RenderView' for the Default View

Once your controller logic has processed the request from a user, it forwards processing to your view in order to generate the appropriate response output. Catalyst uses Catalyst::Action::RenderView by default to automatically perform this operation. If you look in `lib/MyApp/Controller/Root.pm`, you should see the empty definition for the `sub end` method:

```
sub end : ActionClass('RenderView') {}
```

The following bullet points provide a quick overview of the `RenderView` process:

- `Root.pm` is designed to hold application-wide logic.
- At the end of a given user request, Catalyst will call the most specific `end` method that's appropriate. For example, if the controller for a request has an `end` method defined, it will be called. However, if the controller does not define a controller-specific `end` method, the "global" `end` method in `Root.pm` will be called.
- Because the definition includes an `ActionClass` attribute, the Catalyst::Action::RenderView logic will be executed **after** any code inside the definition of `sub end` is run. See Catalyst::Manual::Actions for more information on `ActionClass`.
- Because `sub end` is empty, this effectively just runs the default logic in `RenderView`. However, you can easily extend the `RenderView` logic by adding your own code inside the empty method body (`{}`) created by the Catalyst Helpers when we first ran the `catalyst.pl` to initialize our application. See Catalyst::Action::RenderView for more detailed information on how to extend `RenderView` in `sub end`.

RenderView's "dump_info" Feature

One of the nice features of `RenderView` is that it automatically allows you to add `dump_info=1` to the end of any URL for your application and it will force the display of the "exception dump" screen to the client browser. You can try this out by pointing your browser to this URL:

```
http://localhost:3000/books/list?dump_info=1
```

You should get a page with the following message at the top:

```
Caught exception in MyApp::Controller::Root->end "Forced debug -
Scrubbed output at /usr/share/perl5/Catalyst/Action/RenderView.pm line 46."
```

Along with a summary of your application's state at the end of the processing for that request. The "Stash" section should show a summarized version of the DBIC book model objects. If desired, you can adjust the summarization logic (called "scrubbing" logic) -- see Catalyst::Action::RenderView for details.

Note that you shouldn't need to worry about "normal clients" using this technique to "reverse engineer" your application -- `RenderView` only supports the `dump_info=1` feature when your application is running in `-Debug` mode (something you won't do once you have your application deployed in production).

Using The Default Template Name

By default, `Catalyst::View::TT` will look for a template that uses the same name as your controller action, allowing you to save the step of manually specifying the template name in each action. For example, this would allow us to remove the `$c->stash->{template} = 'books/list.tt2';` line of our `list` action in the Books controller. Open `lib/MyApp/Controller/Books.pm` in your editor and comment out this line to match the following (only the `$c->stash->{template}` line has changed):

```
=head2 list

Fetch all book objects and pass to books/list.tt2 in stash to be displayed

=cut

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) = @_;

    # 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 (actions methods respond to user input in
    # your controllers).
    #$c->stash(template => 'books/list.tt2');
}
```

You should now be able to access the http://localhost:3000/books/list URL as before.

NOTE: If you use the default template technique, you will **not** be able to use either the `$c->forward` or the `$c->detach` mechanisms (these are discussed in Chapter 2 and Chapter 9 of the Tutorial).

IMPORTANT: Make sure that you do **not** skip the following section before continuing to the next chapter 4 Basic CRUD.

Return To A Manually Specified Template

In order to be able to use `$c->forward` and `$c->detach` later in the tutorial, you should remove the comment from the statement in `sub list` in `lib/MyApp/Controller/Books.pm`:

```
$c->stash(template => 'books/list.tt2');
```

Then delete the `TEMPLATE_EXTENSION` line in `lib/MyApp/View/HTML.pm`.

Check the http://localhost:3000/books/list URL in your browser. It should look the same manner as with earlier sections.

You can jump to the next chapter of the tutorial here: Basic 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.

syntax highlighting: no syntax highlighting ▼