NAME

Catalyst::Manual::Tutorial::04_BasicCRUD - Catalyst Tutorial - Chapter 4: Basic CRUD

OVERVIEW

This is **Chapter 4 of 10** for the Catalyst tutorial.

Tutorial Overview

1. Introduction
2. Catalyst Basics
3. More Catalyst Basics
4. **04_Basic CRUD**
5. Authentication
6. Authorization

7. Debugging
8. Testing
9. Advanced CRUD
10. Appendices

DESCRIPTION

This chapter of the tutorial builds on the fairly primitive application created in Chapter 3 to add basic support for Create, Read, Update, and Delete (CRUD) of `Book` objects. Note that the 'list' function in Chapter 3 already implements the Read portion of CRUD (although Read normally refers to reading a single object; you could implement full Read functionality using the techniques introduced below). This section will focus on the Create and Delete aspects of CRUD. More advanced capabilities, including full Update functionality, will be addressed in Chapter 9.

Although this chapter of the tutorial will show you how to build CRUD functionality yourself, another option is to use a "CRUD builder" type of tool to automate the process. You get less control, but it can be quick and easy. For example, see Catalyst::Plugin::AutoCRUD, CatalystX::CRUD, and CatalystX::CRUD::YUI.

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.

FORMLESS SUBMISSION

Our initial attempt at object creation will utilize the "URL arguments" feature of Catalyst (we will employ the more common form-based submission in the sections that follow).

Include a Create Action in the Books Controller

Edit `lib/MyApp/Controller/Books.pm` and enter the following method:

```
    =head2 url_create

    Create a book with the supplied title, rating, and author

    =cut

    sub url_create :Local {
        # In addition to self & context, get the title, rating, &
        # author_id args from the URL.  Note that Catalyst automatically
        # puts extra information after the "/<controller_name>/<action_name/"
        # into @_.  The args are separated  by the '/' char on the URL.
        my ($self, $c, $title, $rating, $author_id) = @_;

        # Call create() on the book model object. Pass the table
        # columns/field values we want to set as hash values
        my $book = $c->model('DB::Book')->create({
                title  => $title,
                rating => $rating
            });

        # Add a record to the join table for this book, mapping to
        # appropriate author
        $book->add_to_book_authors({author_id => $author_id});
        # Note: Above is a shortcut for this:
        # $book->create_related('book_authors', {author_id => $author_id});
```

```
    # Assign the Book object to the stash for display and set template
    $c->stash(book     => $book,
              template => 'books/create_done.tt2');

    # Disable caching for this page
    $c->response->header('Cache-Control' => 'no-cache');
}
```

Notice that Catalyst takes "extra slash-separated information" from the URL and passes it as arguments in `@_` (as long as the number of arguments is not "fixed" using an attribute like `:Args(0)`). The `url_create` action then uses a simple call to the DBIC `create` method to add the requested information to the database (with a separate call to `add_to_book_authors` to update the join table). As do virtually all controller methods (at least the ones that directly handle user input), it then sets the template that should handle this request.

Also note that we are explicitly setting a `no-cache` "Cache-Control" header to force browsers using the page to get a fresh copy every time. You could even move this to a `auto` method in `lib/MyApp/Controller/Root.pm` and it would automatically get applied to every page in the whole application via a single line of code (remember from Chapter 3, that every `auto` method gets run in the Controller hierarchy).

Include a Template for the 'url_create' Action:

Edit `root/src/books/create_done.tt2` and then enter:

```
[% # Use the TT Dumper plugin to Data::Dumper variables to the browser   -%]
[% # Not a good idea for production use, though. :-)  'Indent=1' is      -%]
[% # optional, but prevents "massive indenting" of deeply nested objects -%]
[% USE Dumper(Indent=1) -%]

[% # Set the page title.  META can 'go back' and set values in templates -%]
[% # that have been processed 'before' this template (here it's updating -%]
[% # the title in the root/src/wrapper.tt2 wrapper template).  Note that -%]
[% # META only works on simple/static strings (i.e. there is no variable -%]
[% # interpolation -- if you need dynamic/interpolated content in your   -%]
[% # title, set "$c->stash(title => $something)" in the controller).     -%]
[% META title = 'Book Created' %]

[% # Output information about the record that was added.  First title.   -%]
<p>Added book '[% book.title %]'

[% # Then, output the last name of the first author -%]
by '[% book.authors.first.last_name %]'

[% # Then, output the rating for the book that was added -%]
with a rating of [% book.rating %].</p>

[% # Provide a link back to the list page.  'c.uri_for' builds -%]
[% # a full URI; e.g., 'http://localhost:3000/books/list'      -%]
<p><a href="[% c.uri_for('/books/list') %]">Return to list</a></p>

[% # Try out the TT Dumper (for development only!) -%]
<pre>
Dump of the 'book' variable:
[% Dumper.dump(book) %]
</pre>
```

The TT `USE` directive allows access to a variety of plugin modules (TT plugins, that is, not Catalyst plugins) to add extra functionality to the base TT capabilities. Here, the plugin allows Data::Dumper

"pretty printing" of objects and variables. Other than that, the rest of the code should be familiar from the examples in Chapter 3.

Try the 'url_create' Feature

Make sure the development server is running with the "-r" restart option:

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

Note that new path for `/books/url_create` appears in the startup debug output.

Next, use your browser to enter the following URL:

```
http://localhost:3000/books/url_create/TCPIP_Illustrated_Vol-2/5/4
```

Your browser should display "Added book 'TCPIP_Illustrated_Vol-2' by 'Stevens' with a rating of 5." along with a dump of the new book model object as it was returned by DBIC. You should also see the following DBIC debug messages displayed in the development server log messages if you have DBIC_TRACE set:

```
INSERT INTO book (rating, title) VALUES (?, ?): `5', `TCPIP_Illustrated_Vol-2'
INSERT INTO book_author (author_id, book_id) VALUES (?, ?): `4', `6'
```

The `INSERT` statements are obviously adding the book and linking it to the existing record for Richard Stevens. The `SELECT` statement results from DBIC automatically fetching the book for the `Dumper.dump(book)`.

If you then click the "Return to list" link, you should find that there are now six books shown (if necessary, Shift+Reload or Ctrl+Reload your browser at the `/books/list` page). You should now see the six DBIC debug messages similar to the following (where N=1-6):

```
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 = ? ): 'N'
```

CONVERT TO A CHAINED ACTION

Although the example above uses the same `Local` action type for the method that we saw in the previous chapter of the tutorial, there is an alternate approach that allows us to be more specific while also paving the way for more advanced capabilities. Change the method declaration for `url_create` in `lib/MyApp/Controller/Books.pm` you entered above to match the following:

```
sub url_create :Chained('/') :PathPart('books/url_create') :Args(3) {
    # In addition to self & context, get the title, rating, &
    # author_id args from the URL.  Note that Catalyst automatically
    # puts the first 3 arguments worth of extra information after the
    # "/<controller_name>/<action_name/" into @_ because we specified
    # "Args(3)".  The args are separated  by the '/' char on the URL.
    my ($self, $c, $title, $rating, $author_id) = @_;

    ...
```

This converts the method to take advantage of the Chained action/dispatch type. Chaining lets you have a single URL automatically dispatch to several controller methods, each of which can have precise control over the number of arguments that it will receive. A chain can essentially be thought of having three parts -- a beginning, a middle, and an end. The bullets below summarize the key points behind each of these parts of a chain:

- Beginning
  - **Use "`:Chained('/')`" to start a chain**
  - Get arguments through `CaptureArgs()`
  - Specify the path to match with `PathPart()`
- Middle
  - Link to previous part of the chain with `:Chained('_name_')`
  - Get arguments through `CaptureArgs()`
  - Specify the path to match with `PathPart()`
- End
  - Link to previous part of the chain with `:Chained('_name_')`
  - **Do NOT get arguments through "`CaptureArgs()`," use "`Args()`" instead to end a chain**
  - Specify the path to match with `PathPart()`

In our `url_create` method above, we have combined all three parts into a single method: `:Chained('/')` to start the chain, `:PathPart('books/url_create')` to specify the base URL to match, and `:Args(3)` to capture exactly three arguments and to end the chain.

As we will see shortly, a chain can consist of as many "links" as you wish, with each part capturing some arguments and doing some work along the way. We will continue to use the Chained action type in this chapter of the tutorial and explore slightly more advanced capabilities with the base method and delete feature below. But Chained dispatch is capable of far more. For additional information, see "Action types" in Catalyst::Manual::Intro, Catalyst::DispatchType::Chained, and the 2006 Advent calendar entry on the subject: http://www.catalystframework.org/calendar/2006/10.

Try the Chained Action

If you look back at the development server startup logs from your initial version of the `url_create` method (the one using the `:Local` attribute), you will notice that it produced output similar to the following:

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

When the development server restarts after our conversion to Chained dispatch, the debug output should change to something along the lines of the following:

```
[debug] Loaded Path actions:
.-------------------------------------+--------------------------------------.
| Path                                | Private                              |
+-------------------------------------+--------------------------------------+
```

```
    | /                                  | /default                           |
    | /                                  | /index                             |
    | /books                             | /books/index                       |
    | /books/list                        | /books/list                        |
    '------------------------------------+------------------------------------'

    [debug] Loaded Chained actions:
    .------------------------------------+------------------------------------.
    | Path Spec                          | Private                            |
    +------------------------------------+------------------------------------+
    | /books/url_create/*/*/*            | /books/url_create                  |
    '------------------------------------+------------------------------------'
```

`url_create` has disappeared from the "Loaded Path actions" section but it now shows up under the newly created "Loaded Chained actions" section. And the "/*/*/*" portion clearly shows our requirement for three arguments.

As with our non-chained version of `url_create`, use your browser to enter the following URL:

```
    http://localhost:3000/books/url_create/TCPIP_Illustrated_Vol-2/5/4
```

You should see the same "Added book 'TCPIP_Illustrated_Vol-2' by 'Stevens' with a rating of 5." along with a dump of the new book model object. Click the "Return to list" link, and you should find that there are now seven books shown (two copies of *TCPIP_Illustrated_Vol-2*).

Refactor to Use a 'base' Method to Start the Chains

Let's make a quick update to our initial Chained action to show a little more of the power of chaining. First, open `lib/MyApp/Controller/Books.pm` in your editor and add the following method:

```
    =head2 base

    Can place common logic to start chained dispatch here

    =cut

    sub base :Chained('/') :PathPart('books') :CaptureArgs(0) {
        my ($self, $c) = @_;

        # Store the ResultSet in stash so it's available for other methods
        $c->stash(resultset => $c->model('DB::Book'));

        # Print a message to the debug log
        $c->log->debug('*** INSIDE BASE METHOD ***');
    }
```

Here we print a log message and store the DBIC ResultSet in `$c->stash->{resultset}` so that it's automatically available for other actions that chain off `base`. If your controller always needs a book ID as its first argument, you could have the base method capture that argument (with `:CaptureArgs(1)`) and use it to pull the book object with `->find($id)` and leave it in the stash for later parts of your chains to then act upon. Because we have several actions that don't need to retrieve a book (such as the `url_create` we are working with now), we will instead add that functionality to a common `object` action shortly.

As for `url_create`, let's modify it to first dispatch to `base`. Open up `lib/MyApp/Controller/Books.pm` and edit the declaration for `url_create` to match the following:

```
sub url_create :Chained('base') :PathPart('url_create') :Args(3) {
```

Once you save `lib/MyApp/Controller/Books.pm`, notice that the development server will restart and our "Loaded Chained actions" section will changed slightly:

```
[debug] Loaded Chained actions:
.-------------------------------------+--------------------------------------.
| Path Spec                           | Private                              |
+-------------------------------------+--------------------------------------+
| /books/url_create/*/*/*             | /books/base (0)                      |
|                                     | => /books/url_create                 |
'-------------------------------------+--------------------------------------'
```

The "Path Spec" is the same, but now it maps to two Private actions as we would expect. The `base` method is being triggered by the `/books` part of the URL. However, the processing then continues to the `url_create` method because this method "chained" off `base` and specified `:PathPart('url_create')` (note that we could have omitted the "PathPart" here because it matches the name of the method, but we will include it to make the logic as explicit as possible).

Once again, enter the following URL into your browser:

```
http://localhost:3000/books/url_create/TCPIP_Illustrated_Vol-2/5/4
```

The same "Added book 'TCPIP_Illustrated_Vol-2' by 'Stevens' with a rating of 5." message and a dump of the new book object should appear. Also notice the extra "INSIDE BASE METHOD" debug message in the development server output from the `base` method. Click the "Return to list" link, and you should find that there are now eight books shown. (You may have a larger number of books if you repeated any of the "create" actions more than once. Don't worry about it as long as the number of books is appropriate for the number of times you added new books... there should be the original five books added via `myapp01.sql` plus one additional book for each time you ran one of the url_create variations above.)

MANUALLY BUILDING A CREATE FORM

Although the `url_create` action in the previous step does begin to reveal the power and flexibility of both Catalyst and DBIC, it's obviously not a very realistic example of how users should be expected to enter data. This section begins to address that concern (but just barely, see Chapter 9 for better options for handling web-based forms).

Add Method to Display The Form

Edit `lib/MyApp/Controller/Books.pm` and add the following method:

```
=head2 form_create

Display form to collect information for book to create

=cut

sub form_create :Chained('base') :PathPart('form_create') :Args(0) {
    my ($self, $c) = @_;

    # Set the TT template to use
```

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

This action simply invokes a view containing a form to create a book.

Add a Template for the Form

Open `root/src/books/form_create.tt2` in your editor and enter:

```
[% META title = 'Manual Form Book Create' -%]

<form method="post" action="[% c.uri_for('form_create_do') %]">
<table>
  <tr><td>Title:</td><td><input type="text" name="title"></td></tr>
  <tr><td>Rating:</td><td><input type="text" name="rating"></td></tr>
  <tr><td>Author ID:</td><td><input type="text" name="author_id"></td></tr>
</table>
<input type="submit" name="Submit" value="Submit">
</form>
```

Note that we have specified the target of the form data as `form_create_do`, the method created in the section that follows.

Add a Method to Process Form Values and Update Database

Edit `lib/MyApp/Controller/Books.pm` and add the following method to save the form information to the database:

```
=head2 form_create_do

Take information from form and add to database

=cut

sub form_create_do :Chained('base') :PathPart('form_create_do') :Args(0) {
    my ($self, $c) = @_;

    # Retrieve the values from the form
    my $title     = $c->request->params->{title}     || 'N/A';
    my $rating    = $c->request->params->{rating}    || 'N/A';
    my $author_id = $c->request->params->{author_id} || '1';

    # Create the book
    my $book = $c->model('DB::Book')->create({
            title   => $title,
            rating  => $rating,
        });
    # Handle relationship with author
    $book->add_to_book_authors({author_id => $author_id});
    # Note: Above is a shortcut for this:
    # $book->create_related('book_authors', {author_id => $author_id});

    # Store new model object in stash and set template
    $c->stash(book     => $book,
              template => 'books/create_done.tt2');
}
```

Test Out The Form

Notice that the server startup log reflects the two new chained methods that we added:

```
[debug] Loaded Chained actions:
.-------------------------------------+--------------------------------------.
| Path Spec                           | Private                              |
+-------------------------------------+--------------------------------------+
| /books/form_create                  | /books/base (0)                      |
|                                     | => /books/form_create                |
| /books/form_create_do               | /books/base (0)                      |
|                                     | => /books/form_create_do             |
| /books/url_create/*/*/*             | /books/base (0)                      |
|                                     | => /books/url_create                 |
'-------------------------------------+--------------------------------------'
```

Point your browser to http://localhost:3000/books/form_create and enter "TCP/IP Illustrated, Vol 3" for the title, a rating of 5, and an author ID of 4. You should then see the output of the same `create_done.tt2` template seen in earlier examples. Finally, click "Return to list" to view the full list of books.

Note: Having the user enter the primary key ID for the author is obviously crude; we will address this concern with a drop-down list and add validation to our forms in Chapter 9.

A SIMPLE DELETE FEATURE

Turning our attention to the Delete portion of CRUD, this section illustrates some basic techniques that can be used to remove information from the database.

Include a Delete Link in the List

Edit `root/src/books/list.tt2` and update it to match the following (two sections have changed: 1) the additional '<th>Links</th>' table header, and 2) the five lines for the Delete link near the bottom):

```
[% # 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><th>Links</th></tr>
[% # Display each book in a table row %]
[% FOREACH book IN books -%]
  <tr>
    <td>[% book.title %]</td>
    <td>[% book.rating %]</td>
    <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>
      <td>
        [% # Add a link to delete a book %]
        <a href="[%
          c.uri_for(c.controller.action_for('delete'), [book.id]) %]">Delete</a>
      </td>
    </tr>
  [% END -%]
  </table>
```

The additional code is obviously designed to add a new column to the right side of the table with a `Delete` "button" (for simplicity, links will be used instead of full HTML buttons; but, in practice, anything that modifies data should be handled with a form sending a POST request).

Also notice that we are using a more advanced form of `uri_for` than we have seen before. Here we use `$c->controller->action_for` to automatically generate a URI appropriate for that action based on the method we want to link to while inserting the `book.id` value into the appropriate place. Now, if you ever change `:PathPart('delete')` in your controller method to something like `:PathPart('kill')`, then your links will automatically update without any changes to your .tt2 template file. As long as the name of your method does not change (here, "delete"), then your links will still be correct. There are a few shortcuts and options when using `action_for()`:

- If you are referring to a method in the current controller, you can use `$self->action_for('_method_name_')`.
- If you are referring to a method in a different controller, you need to include that controller's name as an argument to `controller()`, as in `$c->controller('_controller_name_')->action_for('_method_name_')`.

Note: In practice you should **never** use a GET request to delete a record -- always use POST for actions that will modify data. We are doing it here for illustrative and simplicity purposes only.

Add a Common Method to Retrieve a Book for the Chain

As mentioned earlier, since we have a mixture of actions that operate on a single book ID and others that do not, we should not have `base` capture the book ID, find the corresponding book in the database and save it in the stash for later links in the chain. However, just because that logic does not belong in `base` doesn't mean that we can't create another location to centralize the book lookup code. In our case, we will create a method called `object` that will store the specific book in the stash. Chains that always operate on a single existing book can chain off this method, but methods such as `url_create` that don't operate on an existing book can chain directly off base.

To add the `object` method, edit `lib/MyApp/Controller/Books.pm` and add the following code:

```
    =head2 object

    Fetch the specified book object based on the book ID and store
    it in the stash

    =cut

    sub object :Chained('base') :PathPart('id') :CaptureArgs(1) {
```

```
    # $id = primary key of book to delete
    my ($self, $c, $id) = @_;

    # Find the book object and store it in the stash
    $c->stash(object => $c->stash->{resultset}->find($id));

    # Make sure the lookup was successful.  You would probably
    # want to do something like this in a real app:
    #   $c->detach('/error_404') if !$c->stash->{object};
    die "Book $id not found!" if !$c->stash->{object};

    # Print a message to the debug log
    $c->log->debug("*** INSIDE OBJECT METHOD for obj id=$id ***");
}
```

Now, any other method that chains off `object` will automatically have the appropriate book waiting for it in `$c->stash->{object}`.

Add a Delete Action to the Controller

Open `lib/MyApp/Controller/Books.pm` in your editor and add the following method:

```
=head2 delete

Delete a book

=cut

sub delete :Chained('object') :PathPart('delete') :Args(0) {
    my ($self, $c) = @_;

    # Use the book object saved by 'object' and delete it along
    # with related 'book_author' entries
    $c->stash->{object}->delete;

    # Set a status message to be displayed at the top of the view
    $c->stash->{status_msg} = "Book deleted.";

    # Forward to the list action/method in this controller
    $c->forward('list');
}
```

This method first deletes the book object saved by the `object` method. However, it also removes the corresponding entry from the `book_author` table with a cascading delete.

Then, rather than forwarding to a "delete done" page as we did with the earlier create example, it simply sets the `status_msg` to display a notification to the user as the normal list view is rendered.

The `delete` action uses the context `forward` method to return the user to the book list. The `detach` method could have also been used. Whereas `forward` *returns* to the original action once it is completed, `detach` does *not* return. Other than that, the two are equivalent.

Try the Delete Feature

Once you save the Books controller, the server should automatically restart. The `delete` method should now appear in the "Loaded Chained actions" section of the startup debug output:

```
[debug] Loaded Chained actions:
.-------------------------------------+--------------------------------------.
| Path Spec                           | Private                              |
+-------------------------------------+--------------------------------------+
| /books/id/*/delete                  | /books/base (0)                      |
|                                     | -> /books/object (1)                 |
|                                     | => /books/delete                     |
| /books/form_create                  | /books/base (0)                      |
|                                     | => /books/form_create                |
| /books/form_create_do               | /books/base (0)                      |
|                                     | => /books/form_create_do             |
| /books/url_create/*/*/*             | /books/base (0)                      |
|                                     | => /books/url_create                 |
'-------------------------------------+--------------------------------------'
```

Then point your browser to http://localhost:3000/books/list and click the "Delete" link next to the first "TCPIP_Illustrated_Vol-2". A green "Book deleted" status message should display at the top of the page, along with a list of the eight remaining books. You will also see the cascading delete operation via the DBIC_TRACE output:

```
SELECT me.id, me.title, me.rating FROM book me WHERE ( ( me.id = ? ) ): '6'
DELETE FROM book WHERE ( id = ? ): '6'
```

If you get the error `file error - books/delete.tt2: not found` then you probably forgot to uncomment the template line in `sub list` at the end of chapter 3.

Fixing a Dangerous URL

Note the URL in your browser once you have performed the deletion in the prior step -- it is still referencing the delete action:

```
http://localhost:3000/books/id/6/delete
```

What if the user were to press reload with this URL still active? In this case the redundant delete is harmless (although it does generate an exception screen, it doesn't perform any undesirable actions on the application or database), but in other cases this could clearly lead to trouble.

We can improve the logic by converting to a redirect. Unlike `$c->forward('list'))` or `$c->detach('list'))` that perform a server-side alteration in the flow of processing, a redirect is a client-side mechanism that causes the browser to issue an entirely new request. As a result, the URL in the browser is updated to match the destination of the redirection URL.

To convert the forward used in the previous section to a redirect, open `lib/MyApp/Controller/Books.pm` and edit the existing `sub delete` method to match:

```
=head2 delete

Delete a book

=cut

sub delete :Chained('object') :PathPart('delete') :Args(0) {
    my ($self, $c) = @_;
```

```
    # Use the book object saved by 'object' and delete it along
    # with related 'book_author' entries
    $c->stash->{object}->delete;

    # Set a status message to be displayed at the top of the view
    $c->stash->{status_msg} = "Book deleted.";

    # Redirect the user back to the list page.  Note the use
    # of $self->action_for as earlier in this section (BasicCRUD)
    $c->response->redirect($c->uri_for($self->action_for('list')));
}
```

Try the Delete and Redirect Logic

Point your browser to http://localhost:3000/books/list (don't just hit "Refresh" in your browser since we left the URL in an invalid state in the previous section!) and delete the first copy of the remaining two "TCPIP_Illustrated_Vol-2" books. The URL in your browser should return to the http://localhost:3000/books/list URL, so that is an improvement, but notice that *no green "Book deleted" status message is displayed*. Because the stash is reset on every request (and a redirect involves a second request), the `status_msg` is cleared before it can be displayed.

Using 'uri_for' to Pass Query Parameters

There are several ways to pass information across a redirect. One option is to use the `flash` technique that we will see in Chapter 5 of this tutorial; however, here we will pass the information via query parameters on the redirect itself. Open `lib/MyApp/Controller/Books.pm` and update the existing `sub delete` method to match the following:

```
=head2 delete

Delete a book

=cut

sub delete :Chained('object') :PathPart('delete') :Args(0) {
    my ($self, $c) = @_;

    # Use the book object saved by 'object' and delete it along
    # with related 'book_author' entries
    $c->stash->{object}->delete;

    # Redirect the user back to the list page with status msg as an arg
    $c->response->redirect($c->uri_for($self->action_for('list'),
        {status_msg => "Book deleted."}));
}
```

This modification simply leverages the ability of `uri_for` to include an arbitrary number of name/value pairs in a hash reference. Next, we need to update `root/src/wrapper.tt2` to handle `status_msg` as a query parameter:

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

```
    </div><!-- end content -->
    ...
```

Although the sample above only shows the `content` div, leave the rest of the file intact -- the only change we made to the `wrapper.tt2` was to add "`|| c.request.params.status_msg`" to the `<span class="message">` line. Note that we definitely want the "`| html`" TT filter here since it would be easy for users to modify the message on the URL and possibly inject harmful code into the application if we left that off.

Try the Delete and Redirect With Query Param Logic

Point your browser to http://localhost:3000/books/list (you should now be able to safely hit "refresh" in your browser). Then delete the remaining copy of "TCPIP_Illustrated_Vol-2". The green "Book deleted" status message should return. But notice that you can now hit the "Reload" button in your browser and it just redisplays the book list (and it correctly shows it without the "Book deleted" message on redisplay).

NOTE: Be sure to check out Authentication where we use an improved technique that is better suited to your real world applications.

EXPLORING THE POWER OF DBIC

In this section we will explore some additional capabilities offered by DBIx::Class. Although these features have relatively little to do with Catalyst per se, you will almost certainly want to take advantage of them in your applications.

Add Datetime Columns to Our Existing Books Table

Let's add two columns to our existing `books` table to track when each book was added and when each book is updated:

```
    $ sqlite3 myapp.db
    sqlite> ALTER TABLE book ADD created TIMESTAMP;
    sqlite> ALTER TABLE book ADD updated TIMESTAMP;
    sqlite> UPDATE book SET created = DATETIME('NOW'), updated = DATETIME('NOW');
    sqlite> SELECT * FROM book;
    1|CCSP SNRS Exam Certification Guide|5|2010-02-16 04:15:45|2010-02-16 04:15:45
    2|TCP/IP Illustrated, Volume 1|5|2010-02-16 04:15:45|2010-02-16 04:15:45
    3|Internetworking with TCP/IP Vol.1|4|2010-02-16 04:15:45|2010-02-16 04:15:45
    4|Perl Cookbook|5|2010-02-16 04:15:45|2010-02-16 04:15:45
    5|Designing with Web Standards|5|2010-02-16 04:15:45|2010-02-16 04:15:45
    9|TCP/IP Illustrated, Vol 3|5|2010-02-16 04:15:45|2010-02-16 04:15:45
    sqlite> .quit
    $
```

Here are the commands without the surrounding sqlite3 prompt and output in case you want to cut and paste them as a single block (but still start sqlite3 before you paste these in):

```
    ALTER TABLE book ADD created TIMESTAMP;
    ALTER TABLE book ADD updated TIMESTAMP;
    UPDATE book SET created = DATETIME('NOW'), updated = DATETIME('NOW');
    SELECT * FROM book;
```

This will modify the `books` table to include the two new fields and populate those fields with the current time.

Update DBIx::Class to Automatically Handle the Datetime Columns

Next, we should re-run the DBIC helper to update the Result Classes with the new fields:

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

Notice that we modified our use of the helper slightly: we told it to include the DBIx::Class::TimeStamp in the `load_components` line of the Result Classes.

If you open `lib/MyApp/Schema/Result/Book.pm` in your editor you should see that the `created` and `updated` fields are now included in the call to `add_columns()`. However, also notice that the `many_to_many` relationships we manually added below the "`# DO NOT MODIFY...`" line were automatically preserved.

While we `lib/MyApp/Schema/Result/Book.pm` open, let's update it with some additional information to have DBIC automatically handle the updating of these two fields for us. Insert the following code at the bottom of the file (it **must** be **below** the "`# DO NOT MODIFY...`" line and **above** the `1;` on the last line):

```
#
# Enable automatic date handling
#
__PACKAGE__->add_columns(
    "created",
    { data_type => 'timestamp', set_on_create => 1 },
    "updated",
    { data_type => 'timestamp', set_on_create => 1, set_on_update => 1 },
);
```

This will override the definition for these fields that Schema::Loader placed at the top of the file. The `set_on_create` and `set_on_update` options will cause DBIx::Class to automatically update the timestamps in these columns whenever a row is created or modified.

Note that adding the lines above will cause the development server to automatically restart if you are running it with the "-r" option. In other words, the development server is smart enough to restart not only for code under the `MyApp/Controller/`, `MyApp/Model/`, and `MyApp/View/` directories, but also under other directions such as our "external DBIC model" in `MyApp/Schema/`. However, also note that it's smart enough to **not** restart when you edit your `.tt2` files under `root/`.

Then enter the following URL into your web browser:

```
http://localhost:3000/books/url_create/TCPIP_Illustrated_Vol-2/5/4
```

You should get the same "Book Created" screen we saw earlier. However, if you now use the sqlite3 command-line tool to dump the `books` table, you will see that the new book we added has an

appropriate date and time entered for it (see the last line in the listing below):

```
$ sqlite3 myapp.db "select * from book"
1|CCSP SNRS Exam Certification Guide|5|2010-02-16 04:15:45|2010-02-16 04:15:45
2|TCP/IP Illustrated, Volume 1|5|2010-02-16 04:15:45|2010-02-16 04:15:45
3|Internetworking with TCP/IP Vol.1|4|2010-02-16 04:15:45|2010-02-16 04:15:45
4|Perl Cookbook|5|2010-02-16 04:15:45|2010-02-16 04:15:45
5|Designing with Web Standards|5|2010-02-16 04:15:45|2010-02-16 04:15:45
9|TCP/IP Illustrated, Vol 3|5|2010-02-16 04:15:45|2010-02-16 04:15:45
10|TCPIP_Illustrated_Vol-2|5|2010-02-16 04:18:42|2010-02-16 04:18:42
```

Notice in the debug log that the SQL DBIC generated has changed to incorporate the datetime logic:

```
INSERT INTO book ( created, rating, title, updated ) VALUES ( ?, ?, ?, ? ):
'2010-02-16 04:18:42', '5', 'TCPIP_Illustrated_Vol-2', '2010-02-16 04:18:42'
INSERT INTO book_author ( author_id, book_id ) VALUES ( ?, ? ): '4', '10'
```

Create a ResultSet Class

An often overlooked but extremely powerful features of DBIC is that it allows you to supply your own subclasses of `DBIx::Class::ResultSet`. This can be used to pull complex and unsightly "query code" out of your controllers and encapsulate it in a method of your ResultSet Class. These "canned queries" in your ResultSet Class can then be invoked via a single call, resulting in much cleaner and easier to read controller code (or View code, if that's where you want to call it).

To illustrate the concept with a fairly simple example, let's create a method that returns books added in the last 10 minutes. Start by making a directory where DBIx::Class will look for our ResultSet Class:

```
$ mkdir lib/MyApp/Schema/ResultSet
```

Then open `lib/MyApp/Schema/ResultSet/Book.pm` and enter the following:

```
package MyApp::Schema::ResultSet::Book;

use strict;
use warnings;
use base 'DBIx::Class::ResultSet';

=head2 created_after

A predefined search for recently added books

=cut

sub created_after {
    my ($self, $datetime) = @_;

    my $date_str = $self->result_source->schema->storage
                        ->datetime_parser->format_datetime($datetime);

    return $self->search({
        created => { '>' => $date_str }
    });
}

1;
```

Then add the following method to the `lib/MyApp/Controller/Books.pm`:

```
    =head2 list_recent

    List recently created books

    =cut

    sub list_recent :Chained('base') :PathPart('list_recent') :Args(1) {
        my ($self, $c, $mins) = @_;

        # Retrieve all of the book records as book model objects and store in the
        # stash where they can be accessed by the TT template, but only
        # retrieve books created within the last $min number of minutes
        $c->stash(books => [$c->model('DB::Book')
                                ->created_after(DateTime->now->subtract(minutes => $mins))]);

        # 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');
    }
```

Now try different values for the "minutes" argument (the final number value) using the URL `http://localhost:3000/books/list_recent/_#_` in your browser. For example, this would list all books added in the last fifteen minutes:

```
    http://localhost:3000/books/list_recent/15
```

Depending on how recently you added books, you might want to try a higher or lower value for the minutes.

Chaining ResultSets

One of the most helpful and powerful features in `DBIx::Class` is that it allows you to "chain together" a series of queries (note that this has nothing to do with the "Chained Dispatch" for Catalyst that we were discussing earlier). Because each ResultSet method returns another ResultSet, you can take an initial query and immediately feed that into a second query (and so on for as many queries you need). Note that no matter how many ResultSets you chain together, the database itself will not be hit until you use a method that attempts to access the data. And, because this technique carries over to the ResultSet Class feature we implemented in the previous section for our "canned search", we can combine the two capabilities. For example, let's add an action to our `Books` controller that lists books that are both recent *and* have "TCP" in the title. Open up `lib/MyApp/Controller/Books.pm` and add the following method:

```
    =head2 list_recent_tcp

    List recently created books

    =cut

    sub list_recent_tcp :Chained('base') :PathPart('list_recent_tcp') :Args(1) {
        my ($self, $c, $mins) = @_;
```

```
    # Retrieve all of the book records as book model objects and store in the
    # stash where they can be accessed by the TT template, but only
    # retrieve books created within the last $min number of minutes
    # AND that have 'TCP' in the title
    $c->stash(books => [
            $c->model('DB::Book')
                ->created_after(DateTime->now->subtract(minutes => $mins))
                ->search({title => {'like', '%TCP%'}})
        ]);

    # 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');
}
```

To try this out, enter the following URL into your browser:

```
http://localhost:3000/books/list_recent_tcp/100
```

And you should get a list of books added in the last 100 minutes that contain the string "TCP" in the title. However, if you look at all books within the last 100 minutes, you should get a longer list (again, you might have to adjust the number of minutes depending on how recently you added books to your database):

```
http://localhost:3000/books/list_recent/100
```

Take a look at the DBIC_TRACE output in the development server log for the first URL and you should see something similar to the following:

```
SELECT me.id, me.title, me.rating, me.created, me.updated FROM book me
WHERE ( ( title LIKE ? AND created > ? ) ): '%TCP%', '2010-02-16 02:49:32'
```

However, let's not pollute our controller code with this raw "TCP" query -- it would be cleaner to encapsulate that code in a method on our ResultSet Class. To do this, open `lib/MyApp/Schema/ResultSet/Book.pm` and add the following method:

```
=head2 title_like

A predefined search for books with a 'LIKE' search in the string

=cut

sub title_like {
    my ($self, $title_str) = @_;

    return $self->search({
        title => { 'like' => "%$title_str%" }
    });
}
```

We defined the search string as `$title_str` to make the method more flexible. Now update the `list_recent_tcp` method in `lib/MyApp/Controller/Books.pm` to match the following (we have replaced the

`->search` line with the `->title_like` line shown here -- the rest of the method should be the same):

```
    =head2 list_recent_tcp

    List recently created books

    =cut

    sub list_recent_tcp :Chained('base') :PathPart('list_recent_tcp') :Args(1) {
        my ($self, $c, $mins) = @_;

        # Retrieve all of the book records as book model objects and store in the
        # stash where they can be accessed by the TT template, but only
        # retrieve books created within the last $min number of minutes
        # AND that have 'TCP' in the title
        $c->stash(books => [
                $c->model('DB::Book')
                    ->created_after(DateTime->now->subtract(minutes => $mins))
                    ->title_like('TCP')
            ]);

        # 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');
    }
```

Try out the `list_recent_tcp` and `list_recent` URLs as we did above. They should work just the same, but our code is obviously cleaner and more modular, while also being more flexible at the same time.

Adding Methods to Result Classes

In the previous two sections we saw a good example of how we could use DBIx::Class ResultSet Classes to clean up our code for an entire query (for example, our "canned searches" that filtered the entire query). We can do a similar improvement when working with individual rows as well. Whereas the ResultSet construct is used in DBIC to correspond to an entire query, the Result Class construct is used to represent a row. Therefore, we can add row-specific "helper methods" to our Result Classes stored in `lib/MyApp/Schema/Result/`. For example, open `lib/MyApp/Schema/Result/Author.pm` and add the following method (as always, it must be above the closing "`1;`"):

```
    #
    # Row-level helper methods
    #
    sub full_name {
        my ($self) = @_;

        return $self->first_name . ' ' . $self->last_name;
    }
```

This will allow us to conveniently retrieve both the first and last name for an author in one shot. Now open `root/src/books/list.tt2` and change the definition of `tt_authors` from this:

```
    ...
      [% tt_authors = [ ];
         tt_authors.push(author.last_name) FOREACH author = book.authors %]
    ...
```

to:

```
...
  [% tt_authors = [ ];
     tt_authors.push(author.full_name) FOREACH author = book.authors %]
...
```

(Only `author.last_name` was changed to `author.full_name` -- the rest of the file should remain the same.)

Now go to the standard book list URL:

```
http://localhost:3000/books/list
```

The "Author(s)" column will now contain both the first and last name. And, because the concatenation logic was encapsulated inside our Result Class, it keeps the code inside our TT template nice and clean (remember, we want the templates to be as close to pure HTML markup as possible). Obviously, this capability becomes even more useful as you use it to remove even more complicated row-specific logic from your templates!

Moving Complicated View Code to the Model

The previous section illustrated how we could use a Result Class method to print the full names of the authors without adding any extra code to our view, but it still left us with a fairly ugly mess (see `root/src/books/list.tt2`):

```
...
<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 does not print -%]
  [% # a value, so nothing will be printed here.  But, if you have something -%]
  [% # in TT that does return a method and you don't want it printed, you    -%]
  [% # can: 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.full_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>
...
```

Let's combine some of the techniques used earlier in this section to clean this up. First, let's add a method to our Book Result Class to return the number of authors for a book. Open `lib/MyApp/Schema/Result/Book.pm` and add the following method:

```
=head2 author_count

Return the number of authors for the current book

=cut
```

```
    sub author_count {
        my ($self) = @_;

        # Use the 'many_to_many' relationship to fetch all of the authors for the current
        # and the 'count' method in DBIx::Class::ResultSet to get a SQL COUNT
        return $self->authors->count;
    }
```

Next, let's add a method to return a list of authors for a book to the same `lib/MyApp/Schema/Result/Book.pm` file:

```
    =head2 author_list

    Return a comma-separated list of authors for the current book

    =cut

    sub author_list {
        my ($self) = @_;

        # Loop through all authors for the current book, calling all the 'full_name'
        # Result Class method for each
        my @names;
        foreach my $author ($self->authors) {
            push(@names, $author->full_name);
        }

        return join(', ', @names);
    }
```

This method loops through each author, using the `full_name` Result Class method we added to `lib/MyApp/Schema/Result/Author.pm` in the prior section.

Using these two methods, we can simplify our TT code. Open `root/src/books/list.tt2` and update the "Author(s)" table cell to match the following:

```
    ...
    <td>
      [% # Print count and author list using Result Class methods -%]
      ([% book.author_count | html %]) [% book.author_list | html %]
    </td>
    ...
```

Although most of the code we removed comprised comments, the overall effect is dramatic... because our view code is so simple, we don't need huge comments to clue people in to the gist of our code. The view code is now self-documenting and readable enough that you could probably get by with no comments at all. All of the "complex" work is being done in our Result Class methods (and, because we have broken the code into nice, modular chunks, the Result Class code is hardly something you would call complex).

As we saw in this section, always strive to keep your view AND controller code as simple as possible by pulling code out into your model objects. Because DBIx::Class can be easily extended in so many ways, it's an excellent to way accomplish this objective. It will make your code cleaner, easier to write, less error-prone, and easier to debug and maintain.

Before you conclude this section, hit Refresh in your browser... the output should be the same even though the backend code has been trimmed down.

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

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.