NAME

Catalyst::Manual::Tutorial::05_Authentication - Catalyst Tutorial - Chapter 5: Authentication

OVERVIEW

This is **Chapter 5 of 10** for the Catalyst tutorial.

Tutorial Overview

1. Introduction
2. Catalyst Basics
3. More Catalyst Basics
4. Basic CRUD
5. **05_Authentication**
6. Authorization
7. Debugging
8. Testing
9. Advanced CRUD
10. Appendices

DESCRIPTION

Now that we finally have a simple yet functional application, we can focus on providing authentication (with authorization coming next in Chapter 6).

This chapter of the tutorial is divided into two main sections: 1) basic, cleartext authentication and 2) hash-based authentication.

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.

BASIC AUTHENTICATION

This section explores how to add authentication logic to a Catalyst application.

Add Users and Roles to the Database

First, we add both user and role information to the database (we will add the role information here although it will not be used until the authorization section, Chapter 6). Create a new SQL script file by opening `myapp02.sql` in your editor and insert:

```
    --
    -- Add users and role tables, along with a many-to-many join table
    --
    PRAGMA foreign_keys = ON;
    CREATE TABLE users (
            id            INTEGER PRIMARY KEY,
            username      TEXT,
            password      TEXT,
            email_address TEXT,
            first_name    TEXT,
            last_name     TEXT,
            active        INTEGER
    );
    CREATE TABLE role (
            id   INTEGER PRIMARY KEY,
            role TEXT
    );
    CREATE TABLE user_role (
            user_id INTEGER REFERENCES users(id) ON DELETE CASCADE ON UPDATE CASCADE,
            role_id INTEGER REFERENCES role(id) ON DELETE CASCADE ON UPDATE CASCADE,
            PRIMARY KEY (user_id, role_id)
    );
    --
    -- Load up some initial test data
    --
    INSERT INTO users VALUES (1, 'test01', 'mypass', 't01@na.com', 'Joe',  'Blow', 1);
    INSERT INTO users VALUES (2, 'test02', 'mypass', 't02@na.com', 'Jane', 'Doe',  1);
    INSERT INTO users VALUES (3, 'test03', 'mypass', 't03@na.com', 'No',   'Go',   0);
    INSERT INTO role VALUES (1, 'user');
    INSERT INTO role VALUES (2, 'admin');
    INSERT INTO user_role VALUES (1, 1);
    INSERT INTO user_role VALUES (1, 2);
    INSERT INTO user_role VALUES (2, 1);
    INSERT INTO user_role VALUES (3, 1);
```

Then load this into the `myapp.db` database with the following command:

```
    $ sqlite3 myapp.db < myapp02.sql
```

Add User and Role Information to DBIC Schema

Although we could manually edit the DBIC schema information to include the new tables added in the previous step, let's use the `create=static` option on the DBIC model helper to do most of the work for us:

```
$ 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"
$
$ ls lib/MyApp/Schema/Result
Author.pm  BookAuthor.pm  Book.pm  Role.pm  User.pm  UserRole.pm
```

Notice how the helper has added three new table-specific Result Source files to the `lib/MyApp/Schema/Result` directory. And, more importantly, even if there were changes to the existing result source files, those changes would have only been written above the `# DO NOT MODIFY THIS OR ANYTHING ABOVE!` comment and your hand-edited enhancements would have been preserved.

Speaking of "hand-edited enhancements," we should now add the `many_to_many` relationship information to the User Result Source file. As with the Book, BookAuthor, and Author files in Chapter 3, DBIx::Class::Schema::Loader has automatically created the `has_many` and `belongs_to` relationships for the new User, UserRole, and Role tables. However, as a convenience for mapping Users to their assigned roles (see Chapter 6), we will also manually add a `many_to_many` relationship. Edit `lib/MyApp/Schema/Result/User.pm` add the following information between the `# DO NOT MODIFY THIS OR ANYTHING ABOVE!` comment and the closing `1;`:

```
# many_to_many():
#   args:
#     1) Name of relationship, 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(roles => 'user_roles', 'role');
```

The code for this update is obviously very similar to the edits we made to the `Book` and `Author` classes created in Chapter 3 with one exception: we only defined the `many_to_many` relationship in one direction. Whereas we felt that we would want to map Authors to Books **AND** Books to Authors, here we are only adding the convenience `many_to_many` in the Users to Roles direction.

Note that we do not need to make any change to the `lib/MyApp/Schema.pm` schema file. It simply tells DBIC to load all of the Result Class and ResultSet Class files it finds below the `lib/MyApp/Schema` directory, so it will automatically pick up our new table information.

Sanity-Check of the Development Server Reload

We aren't ready to try out the authentication just yet; we only want to do a quick check to be sure our model loads correctly. Assuming that you are following along and using the "-r" option on `myapp_server.pl`, then the development server should automatically reload (if not, press `Ctrl-C` to break out of the server if it's running and then enter `script/myapp_server.pl` to start it). Look for the three new model objects in the startup debug output:

```
...
 .-----------------------------------------------------------------+----------.
| 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::Model::DB::Role                                           | class    |
| MyApp::Model::DB::User                                           | class    |
| MyApp::Model::DB::UserRole                                       | class    |
| MyApp::View::HTML                                                | instance |
'------------------------------------------------------------------+----------'
...
```

Again, notice that your "Result Class" classes have been "re-loaded" by Catalyst under `MyApp::Model`.

Include Authentication and Session Plugins

Edit `lib/MyApp.pm` and update it as follows (everything below `StackTrace` is new):

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

    StackTrace

    Authentication

    Session
    Session::Store::File
    Session::State::Cookie
/;
```

Note: As discussed in Chapter 3, different versions of `Catalyst::Devel` have used a variety of methods to load the plugins, but we are going to use the current Catalyst 5.9 practice of putting them on the `use Catalyst` line.

The `Authentication` plugin supports Authentication while the `Session` plugins are required to maintain state across multiple HTTP requests.

Note that the only required Authentication class is the main one. This is a change that occurred in version 0.09999_01 of the Authentication plugin. You **do not need** to specify a particular Authentication::Store or `Authentication::Credential` you want to use. Instead, indicate the Store and Credential you want to use in your application configuration (see below).

Make sure you include the additional plugins as new dependencies in the Makefile.PL file something like this:

```
requires 'Catalyst::Plugin::Authentication';
requires 'Catalyst::Plugin::Session';
requires 'Catalyst::Plugin::Session::Store::File';
requires 'Catalyst::Plugin::Session::State::Cookie';
```

Note that there are several options for Session::Store. Session::Store::Memcached is generally a good choice if you are on Unix. If you are running on Windows Session::Store::File is fine. Consult Session::Store and its subclasses for additional information and options (for example to use a database-backed session store).

Configure Authentication

There are a variety of ways to provide configuration information to Catalyst::Plugin::Authentication. Here we will use Catalyst::Authentication::Realm::SimpleDB because it automatically sets a reasonable set of defaults for us. (Note: the `SimpleDB` here has nothing to do with the SimpleDB offered in Amazon's web services offerings -- here we are only talking about a "simple" way to use your DB as an authentication backend.) Open `lib/MyApp.pm` and place the following text above the call to `__PACKAGE__->setup();`:

```
    # Configure SimpleDB Authentication
    __PACKAGE__->config(
        'Plugin::Authentication' => {
            default => {
                class           => 'SimpleDB',
                user_model      => 'DB::User',
                password_type   => 'clear',
            },
        },
    );
```

We could have placed this configuration in `myapp.conf`, but placing it in `lib/MyApp.pm` is probably a better place since it's not likely something that users of your application will want to change during deployment (or you could use a mixture: leave `class` and `user_model` defined in `lib/MyApp.pm` as we show above, but place `password_type` in `myapp.conf` to allow the type of password to be easily modified during deployment). We will stick with putting all of the authentication-related configuration in `lib/MyApp.pm` for the tutorial, but if you wish to use `myapp.conf`, just convert to the following code:

```
    <Plugin::Authentication>
        <default>
            password_type clear
            user_model    DB::User
            class         SimpleDB
        </default>
    </Plugin::Authentication>
```

TIP: Here is a short script that will dump the contents of `MyApp`-config> to Config::General format in `myapp.conf`:

```
    $ CATALYST_DEBUG=0 perl -Ilib -e 'use MyApp; use Config::General;
        Config::General->new->save_file("myapp.conf", MyApp->config);'
```

HOWEVER, if you try out the command above, be sure to delete the "myapp.conf" command. Otherwise, you will wind up with duplicate configurations.

NOTE: Because we are using SimpleDB along with a database layout that complies with its default assumptions: we don't need to specify the names of the columns where our username and password information is stored (hence, the "Simple" part of "SimpleDB"). That being said, SimpleDB lets you specify that type of information if you need to. Take a look at `Catalyst::Authentication::Realm::SimpleDB` for details.

Add Login and Logout Controllers

Use the Catalyst create script to create two stub controller files:

```
    $ script/myapp_create.pl controller Login
    $ script/myapp_create.pl controller Logout
```

You could easily use a single controller here. For example, you could have a `User` controller with both `login` and `logout` actions. Remember, Catalyst is designed to be very flexible, and leaves such matters up to you, the designer and programmer.

Then open `lib/MyApp/Controller/Login.pm`, and update the definition of `sub index` to match:

```
    =head2 index

    Login logic

    =cut

    sub index :Path :Args(0) {
        my ($self, $c) = @_;

        # Get the username and password from form
        my $username = $c->request->params->{username};
        my $password = $c->request->params->{password};

        # If the username and password values were found in form
        if ($username && $password) {
            # Attempt to log the user in
            if ($c->authenticate({ username => $username,
                                   password => $password  } )) {
                # If successful, then let them use the application
                $c->response->redirect($c->uri_for(
                    $c->controller('Books')->action_for('list')));
                return;
            } else {
                # Set an error message
                $c->stash(error_msg => "Bad username or password.");
            }
        } else {
            # Set an error message
            $c->stash(error_msg => "Empty username or password.")
                unless ($c->user_exists);
        }

        # If either of above don't work out, send to the login page
        $c->stash(template => 'login.tt2');
    }
```

This controller fetches the `username` and `password` values from the login form and attempts to authenticate the user. If successful, it redirects the user to the book list page. If the login fails, the user will stay at the login page and receive an error message. If the `username` and `password` values are not present in the form, the user will be taken to the empty login form.

Note that we could have used something like "`sub default :Path`", however, it is generally recommended (partly for historical reasons, and partly for code clarity) only to use `default` in `MyApp::Controller::Root`, and then mainly to generate the 404 not found page for the application.

Instead, we are using "`sub somename :Path :Args(0) {...}`" here to specifically match the URL `/login`. `Path` actions (aka, "literal actions") create URI matches relative to the namespace of the controller where they are defined. Although `Path` supports arguments that allow relative and absolute paths to be defined, here we use an empty `Path` definition to match on just the name of the controller itself. The method name, `index`, is arbitrary. We make the match even more specific with the `:Args(0)` action modifier -- this forces the match on *only* `/login`, not `/login/somethingelse`.

Next, update the corresponding method in `lib/MyApp/Controller/Logout.pm` to match:

```
    =head2 index

    Logout logic

    =cut

    sub index :Path :Args(0) {
        my ($self, $c) = @_;

        # Clear the user's state
        $c->logout;

        # Send the user to the starting point
        $c->response->redirect($c->uri_for('/'));
    }
```

Add a Login Form TT Template Page

Create a login form by opening `root/src/login.tt2` and inserting:

```
    [% META title = 'Login' %]

    <!-- Login form -->
    <form method="post" action="[% c.uri_for('/login') %]">
      <table>
        <tr>
          <td>Username:</td>
          <td><input type="text" name="username" size="40" /></td>
        </tr>
        <tr>
          <td>Password:</td>
          <td><input type="password" name="password" size="40" /></td>
        </tr>
        <tr>
          <td colspan="2"><input type="submit" name="submit" value="Submit" /></td>
        </tr>
      </table>
    </form>
```

Add Valid User Check

We need something that provides enforcement for the authentication mechanism -- a *global* mechanism that prevents users who have not passed authentication from reaching any pages except the login page. This is generally done via an `auto` action/method in `lib/MyApp/Controller/Root.pm`.

Edit the existing `lib/MyApp/Controller/Root.pm` class file and insert the following method:

```
=head2 auto

Check if there is a user and, if not, forward to login page

=cut

# Note that 'auto' runs after 'begin' but before your actions and that
# 'auto's "chain" (all from application path to most specific class are run)
# See the 'Actions' section of 'Catalyst::Manual::Intro' for more info.
sub auto :Private {
    my ($self, $c) = @_;

    # Allow unauthenticated users to reach the login page.  This
    # allows unauthenticated users to reach any action in the Login
    # controller.  To lock it down to a single action, we could use:
    #   if ($c->action eq $c->controller('Login')->action_for('index'))
    # to only allow unauthenticated access to the 'index' action we
    # added above.
    if ($c->controller eq $c->controller('Login')) {
        return 1;
    }

    # If a user doesn't exist, force login
    if (!$c->user_exists) {
        # Dump a log message to the development server debug output
        $c->log->debug('***Root::auto User not found, forwarding to /login');
        # Redirect the user to the login page
        $c->response->redirect($c->uri_for('/login'));
        # Return 0 to cancel 'post-auto' processing and prevent use of application
        return 0;
    }

    # User found, so return 1 to continue with processing after this 'auto'
    return 1;
}
```

As discussed in "CREATE A CATALYST CONTROLLER" in Catalyst::Manual::Tutorial::03_MoreCatalystBasics, every `auto` method from the application/root controller down to the most specific controller will be called. By placing the authentication enforcement code inside the `auto` method of `lib/MyApp/Controller/Root.pm` (or `lib/MyApp.pm`), it will be called for *every* request that is received by the entire application.

Displaying Content Only to Authenticated Users

Let's say you want to provide some information on the login page that changes depending on whether the user has authenticated yet. To do this, open `root/src/login.tt2` in your editor and add the following lines to the bottom of the file:

```
...
<p>
[%
   # This code illustrates how certain parts of the TT
   # template will only be shown to users who have logged in
%]
[% IF c.user_exists %]
    Please Note: You are already logged in as '[% c.user.username %]'.
    You can <a href="[% c.uri_for('/logout') %]">logout</a> here.
[% ELSE %]
```

```
    You need to log in to use this application.
[% END %]
[%#
   Note that this whole block is a comment because the "#" appears
   immediate after the "[%" (with no spaces in between).  Although it
   can be a handy way to temporarily "comment out" a whole block of
   TT code, it's probably a little too subtle for use in "normal"
   comments.
%]
</p>
```

Although most of the code is comments, the middle few lines provide a "you are already logged in" reminder if the user returns to the login page after they have already authenticated. For users who have not yet authenticated, a "You need to log in..." message is displayed (note the use of an IF-THEN-ELSE construct in TT).

Try Out Authentication

The development server should have reloaded each time we edited one of the Controllers in the previous section. Now try going to http://localhost:3000/books/list and you should be redirected to the login page, hitting Shift+Reload or Ctrl+Reload if necessary (the "You are already logged in" message should *not* appear -- if it does, click the `logout` button and try again). Note the `***Root::auto User not found...` debug message in the development server output. Enter username `test01` and password `mypass`, and you should be taken to the Book List page.

IMPORTANT NOTE: If you are having issues with authentication on Internet Explorer (or potentially other browsers), be sure to check the system clocks on both your server and client machines. Internet Explorer is very picky about timestamps for cookies. You can use the `ntpq -p` command on the Tutorial Virtual Machine to check time sync and/or use the following command to force a sync:

```
sudo ntpdate-debian
```

Or, depending on your firewall configuration, try it with "-u":

```
sudo ntpdate-debian -u
```

Note: NTP can be a little more finicky about firewalls because it uses UDP vs. the more common TCP that you see with most Internet protocols. Worse case, you might have to manually set the time on your development box instead of using NTP.

Open `root/src/books/list.tt2` and add the following lines to the bottom (below the closing </table> tag):

```
...
<p>
  <a href="[% c.uri_for('/login') %]">Login</a>
  <a href="[% c.uri_for(c.controller.action_for('form_create')) %]">Create</a>
</p>
```

Reload your browser and you should now see a "Login" and "Create" links at the bottom of the page (as mentioned earlier, you can update template files without a development server reload). Click the first link to return to the login page. This time you *should* see the "You are already logged in" message.

Finally, click the `You can logout here` link on the `/login` page. You should stay at the login page, but the message should change to "You need to log in to use this application."

USING PASSWORD HASHES

In this section we increase the security of our system by converting from cleartext passwords to SHA-1 password hashes that include a random "salt" value to make them extremely difficult to crack, even with dictionary and "rainbow table" attacks.

Note: This section is optional. You can skip it and the rest of the tutorial will function normally.

Be aware that even with the techniques shown in this section, the browser still transmits the passwords in cleartext to your application. We are just avoiding the *storage* of cleartext passwords in the database by using a salted SHA-1 hash. If you are concerned about cleartext passwords between the browser and your application, consider using SSL/TLS, made easy with modules such as Catalyst::Plugin:RequireSSL and Catalyst::ActionRole::RequireSSL.

Re-Run the DBIC::Schema Model Helper to Include DBIx::Class::PassphraseColumn

Let's re-run the model helper to have it include DBIx::Class::PassphraseColumn in all of the Result Classes it generates for us. Simply use the same command we saw in Chapters 3 and 4, but add `,PassphraseColumn` to the `components` argument:

```
    $ script/myapp_create.pl model DB DBIC::Schema MyApp::Schema \
        create=static components=TimeStamp,PassphraseColumn dbi:SQLite:myapp.db \
        on_connect_do="PRAGMA foreign_keys = ON"
```

If you then open one of the Result Classes, you will see that it includes PassphraseColumn in the `load_components` line. Take a look at `lib/MyApp/Schema/Result/User.pm` since that's the main class where we want to use hashed and salted passwords:

```
    __PACKAGE__->load_components("InflateColumn::DateTime", "TimeStamp", "PassphraseColumn");
```

Modify the "password" Column to Use PassphraseColumn

Open the file `lib/MyApp/Schema/Result/User.pm` and enter the following text below the "# DO NOT MODIFY THIS OR ANYTHING ABOVE!" line but above the closing "1;":

```
    # Have the 'password' column use a SHA-1 hash and 20-byte salt
    # with RFC 2307 encoding; Generate the 'check_password" method
    __PACKAGE__->add_columns(
        'password' => {
            passphrase       => 'rfc2307',
            passphrase_class => 'SaltedDigest',
            passphrase_args  => {
                algorithm   => 'SHA-1',
                salt_random => 20.
            },
            passphrase_check_method => 'check_password',
        },
    );
```

This redefines the automatically generated definition for the password fields at the top of the Result Class file to now use PassphraseColumn logic, storing passwords in RFC 2307 format (`passphrase` is set to `rfc2307`). `passphrase_class` can be set to the name of any `Authen::Passphrase::*` class, such as `SaltedDigest` to use Authen::Passphrase::SaltedDigest, or `BlowfishCrypt` to use Authen::Passphrase::BlowfishCrypt. `passphrase_args` is then used to customize the passphrase class you selected. Here we specified the digest algorithm to use as `SHA-1` and the size of the salt to use, but we could have also specified any other option the selected passphrase class supports.

Load Hashed Passwords in the Database

Next, let's create a quick script to load some hashed and salted passwords into the `password` column of our `users` table. Open the file `set_hashed_passwords.pl` in your editor and enter the following text:

```
    #!/usr/bin/perl

    use strict;
    use warnings;

    use MyApp::Schema;

    my $schema = MyApp::Schema->connect('dbi:SQLite:myapp.db');

    my @users = $schema->resultset('User')->all;

    foreach my $user (@users) {
        $user->password('mypass');
        $user->update;
    }
```

PassphraseColumn lets us simply call `$user-`check_password($password)> to see if the user has supplied the correct password, or, as we show above, call `$user-`update($new_password)> to update the hashed password stored for this user.

Then run the following command:

```
    $ DBIC_TRACE=1 perl -Ilib set_hashed_passwords.pl
```

We had to use the `-Ilib` argument to tell Perl to look under the `lib` directory for our `MyApp::Schema` model.

The DBIC_TRACE output should show that the update worked:

```
    $ DBIC_TRACE=1 perl -Ilib set_hashed_passwords.pl
    SELECT me.id, me.username, me.password, me.email_address,
    me.first_name, me.last_name, me.active FROM users me:
    UPDATE users SET password = ? WHERE ( id = ? ):
    '{SSHA}esgz64CpHMo8pMfgIIszP13ft23z/zio04aCwNdm0wc6MDeloMUH4g==', '1'
    UPDATE users SET password = ? WHERE ( id = ? ):
    '{SSHA}FpGhpCJus+Ea9ne4ww8404HH+hJKW/fW+bAv1v6FuRUy2G7I2aoTRQ==', '2'
    UPDATE users SET password = ? WHERE ( id = ? ):
    '{SSHA}ZyGlpiHls8qFBSbHr3r5t/iqcZE602XLMbkSVRRNl6rF8imv1abQVg==', '3'
```

But we can further confirm our actions by dumping the users table:

```
$ sqlite3 myapp.db "select * from users"
1|test01|{SSHA}esgz64CpHMo8pMfgIIszP13ft23z/zio04aCwNdm0wc6MDeloMUH4g==|t01@na.com|Joe|Blow|1
2|test02|{SSHA}FpGhpCJus+Ea9ne4ww8404HH+hJKW/fW+bAv1v6FuRUy2G7I2aoTRQ==|t02@na.com|Jane|Doe|1
3|test03|{SSHA}ZyGlpiHls8qFBSbHr3r5t/iqcZE602XLMbkSVRRNl6rF8imv1abQVg==|t03@na.com|No|Go|0
```

As you can see, the passwords are much harder to steal from the database (not only are the hashes stored, but every hash is different even though the passwords are the same because of the added "salt" value). Also note that this demonstrates how to use a DBIx::Class model outside of your web application -- a very useful feature in many situations.

Enable Hashed and Salted Passwords

Edit `lib/MyApp.pm` and update the config() section for `Plugin::Authentication` it to match the following text (the only change is to the `password_type` field):

```
# Configure SimpleDB Authentication
__PACKAGE__->config(
    'Plugin::Authentication' => {
        default => {
            class           => 'SimpleDB',
            user_model      => 'DB::User',
            password_type   => 'self_check',
        },
    },
);
```

The use of `self_check` will cause Catalyst::Plugin::Authentication::Store::DBIx::Class to call the `check_password` method we enabled on our `password` columns.

Try Out the Hashed Passwords

The development server should restart as soon as your save the `lib/MyApp.pm` file in the previous section. You should now be able to go to http://localhost:3000/books/list and login as before. When done, click the "logout" link on the login page (or point your browser at http://localhost:3000/logout).

USING THE SESSION FOR FLASH

As discussed in the previous chapter of the tutorial, `flash` allows you to set variables in a way that is very similar to `stash`, but it will remain set across multiple requests. Once the value is read, it is cleared (unless reset). Although `flash` has nothing to do with authentication, it does leverage the same session plugins. Now that those plugins are enabled, let's go back and update the "delete and redirect with query parameters" code seen at the end of the Basic CRUD chapter of the tutorial to take advantage of `flash`.

First, open `lib/MyApp/Controller/Books.pm` and modify `sub delete` to match the following (everything after the model search line of code has changed):

```
=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_authors' entries
        $c->stash->{object}->delete;

        # Use 'flash' to save information across requests until it's read
        $c->flash->{status_msg} = "Book deleted";

        # Redirect the user back to the list page
        $c->response->redirect($c->uri_for($self->action_for('list')));
    }
```

Next, open `root/src/wrapper.tt2` and update the TT code to pull from flash vs. the `status_msg` query parameter:

```
    ...
    <div id="content">
        [%# Status and error messages %]
        <span class="message">[% status_msg || c.flash.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 -->
    ...
```

Although the sample above only shows the `content` div, leave the rest of the file intact -- the only change we made to replace "|| c.request.params.status_msg" with "c.flash.status_msg" in the `<span class="message">` line.

Try Out Flash

Authenticate using the login screen and then point your browser to http://localhost:3000/books/url_create/Test/1/4 to create an extra several books. Click the "Return to list" link and delete one of the "Test" books you just added. The `flash` mechanism should retain our "Book deleted" status message across the redirect.

NOTE: While `flash` will save information across multiple requests, *it does get cleared the first time it is read*. In general, this is exactly what you want -- the `flash` message will get displayed on the next screen where it's appropriate, but it won't "keep showing up" after that first time (unless you reset it). Please refer to Catalyst::Plugin::Session for additional information.

Note: There is also a `flash-to-stash` feature that will automatically load the contents the contents of flash into stash, allowing us to use the more typical `c.flash.status_msg` in our TT template in lieu of the more verbose `status_msg || c.flash.status_msg` we used above. Consult Catalyst::Plugin::Session for additional information.

Switch To Catalyst::Plugin::StatusMessages

Although the query parameter technique we used in Chapter 4 and the `flash` approach we used above will work in most cases, they both have their drawbacks. The query parameters can leave the status message on the screen longer than it should (for example, if the user hits refresh). And `flash` can display the wrong message on the wrong screen (flash just shows the message on the next page for that user... if the user has multiple windows or tabs open, then the wrong one can get the status message).

Catalyst::Plugin::StatusMessage is designed to address these shortcomings. It stores the messages in the user's session (so they are available across multiple requests), but ties each status message to a random token. By passing this token across the redirect, we are no longer relying on a potentially ambiguous "next request" like we do with flash. And, because the message is deleted the first time it's displayed, the user can hit refresh and still only see the message a single time (even though the URL may continue to reference the token, it's only displayed the first time). The use of `StatusMessage` or a similar mechanism is recommended for all Catalyst applications.

To enable `StatusMessage`, first edit `lib/MyApp.pm` and add `StatusMessage` to the list of plugins:

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

        StackTrace

        Authentication

        Session
        Session::Store::File
        Session::State::Cookie

        StatusMessage
    /;
```

Then edit `lib/MyApp/Controller/Books.pm` and modify the `delete` action to match the following:

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

        # Saved the PK id for status_msg below
        my $id = $c->stash->{object}->id;

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

        # Redirect the user back to the list page
        $c->response->redirect($c->uri_for($self->action_for('list'),
            {mid => $c->set_status_msg("Deleted book $id")}));
    }
```

This uses the `set_status_msg` that the plugin added to `$c` to save the message under a random token. (If we wanted to save an error message, we could have used `set_error_msg`.) Because `set_status_msg` and `set_error_msg` both return the random token, we can assign that value to the "`mid`" query parameter via `uri_for` as shown above.

Next, we need to make sure that the list page will load display the message. The easiest way to do this is to take advantage of the chained dispatch we implemented in Chapter 4. Edit `lib/MyApp/Controller/Books.pm` again and update the `base` action to match:

```
    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 ***');

        # Load status messages
        $c->load_status_msgs;
    }
```

That way, anything that chains off `base` will automatically get any status or error messages loaded into the stash. Let's convert the `list` action to take advantage of this. Modify the method signature for `list` from:

```
    sub list :Local {
```

to:

```
    sub list :Chained('base') :PathPart('list') :Args(0) {
```

Finally, let's clean up the status/error message code in our wrapper template. Edit `root/src/wrapper.tt2` and change the "content" div to match the following:

```
    <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 -->
```

Now go to http://localhost:3000/books/list in your browser. Delete another of the "Test" books you added in the previous step. You should get redirection from the `delete` action back to the `list` action, but with a "mid=########" message ID query parameter. The screen should say "Deleted book #" (where # is the PK id of the book you removed). However, if you hit refresh in your browser, the status message is no longer displayed (even though the URL does still contain the message ID token, it is ignored -- thereby keeping the state of our status/error messages in sync with the users actions).

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

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 ▼