NAME

Catalyst::Manual::Cookbook - Cooking with Catalyst

DESCRIPTION

Yummy code like your mum used to bake!

RECIPES

Basics

These recipes cover some basic stuff that is worth knowing for Catalyst developers.

Delivering a Custom Error Page

By default, Catalyst will display its own error page whenever it encounters an error in your application. When running under `-Debug` mode, the error page is a useful screen including the error message and Data::Dump output of the relevant parts of the `$c` context object. When not in `-Debug`, users see a simple "Please come back later" screen.

To use a custom error page, use a special `end` method to short-circuit the error processing. The following is an example; you might want to adjust it further depending on the needs of your application (for example, any calls to `fillform` will probably need to go into this `end` method; see Catalyst::Plugin::FillInForm).

```
sub end : Private {
    my ( $self, $c ) = @_;
```

```
        if ( scalar @{ $c->error } ) {
            $c->stash->{errors}   = $c->error;
            for my $error ( @{ $c->error } ) {
                $c->log->error($error);
            }
            $c->stash->{template} = 'errors.tt';
            $c->forward('MyApp::View::TT');
            $c->clear_errors;
        }

        return 1 if $c->response->status =~ /^3\d\d$/;
        return 1 if $c->response->body;

        unless ( $c->response->content_type ) {
            $c->response->content_type('text/html; charset=utf-8');
        }

        $c->forward('MyApp::View::TT');
    }
```

You can manually set errors in your code to trigger this page by calling

```
    $c->error( 'You broke me!' );
```

Disable statistics

Just add this line to your application class if you don't want those nifty statistics in your debug messages.

```
    sub Catalyst::Log::info { }
```

Enable debug status in the environment

Normally you enable the debugging info by adding the `-Debug` flag to your `use Catalyst` statement . However, you can also enable it using environment variable, so you can (for example) get debug info without modifying your application scripts. Just set `CATALYST_DEBUG` or `<MYAPP>_DEBUG` to a true value.

Sessions

When you have your users identified, you will want to somehow remember that fact, to save them from having to identify themselves for every single page. One way to do this is to send the username and password parameters in every single page, but that's ugly, and won't work for static pages.

Sessions are a method of saving data related to some transaction, and giving the whole collection a single ID. This ID is then given to the user to return to us on every page they visit while logged in. The usual way to do this is using a browser cookie.

Catalyst uses two types of plugins to represent sessions:

State

A State module is used to keep track of the state of the session between the users browser, and your application.

A common example is the Cookie state module, which sends the browser a cookie containing the session ID. It will use default value for the cookie name and domain, so will "just work" when used.

Store

A Store module is used to hold all the data relating to your session, for example the users ID, or the items for their shopping cart. You can store data in memory (FastMmap), in a file (File) or in a database (DBI).

Authentication magic

If you have included the session modules in your application, the Authentication modules will automagically use your session to save and retrieve the user data for you.

Using a session

Once the session modules are loaded, the session is available as `$c->session`, and can be written to and read from as a simple hash reference.

EXAMPLE

```
package MyApp;
use Moose;
use namespace::autoclean;

use Catalyst  qw/
                      Session
                      Session::Store::FastMmap
                      Session::State::Cookie
                /;
extends 'Catalyst';
__PACKAGE__->setup;

package MyApp::Controller::Foo;
use Moose;
use namespace::autoclean;
BEGIN { extends 'Catalyst::Controller' };
## Write data into the session

sub add_item : Local {
   my ( $self, $c ) = @_;

   my $item_id = $c->req->params->{item};

   push @{ $c->session->{items} }, $item_id;

}

## A page later we retrieve the data from the session:

sub get_items : Local {
   my ( $self, $c ) = @_;

   $c->stash->{items_to_display} = $c->session->{items};

}
```

More information

http://search.cpan.org/dist/Catalyst-Plugin-Session

http://search.cpan.org/dist/Catalyst-Plugin-Session-State-Cookie

http://search.cpan.org/dist/Catalyst-Plugin-Session-State-URI

http://search.cpan.org/dist/Catalyst-Plugin-Session-Store-FastMmap

http://search.cpan.org/dist/Catalyst-Plugin-Session-Store-File

http://search.cpan.org/dist/Catalyst-Plugin-Session-Store-DBI

Configure your application

You configure your application with the `config` method in your application class. This can be hard-coded, or brought in from a separate configuration file.

Using Config::General

Config::General is a method for creating flexible and readable configuration files. It's a great way to keep your Catalyst application configuration in one easy-to-understand location.

Now create `myapp.conf` in your application home:

```
name     MyApp

# session; perldoc Catalyst::Plugin::Session::FastMmap
<Session>
  expires 3600
  rewrite 0
  storage /tmp/myapp.session
</Session>

# emails; perldoc Catalyst::Plugin::Email
# this passes options as an array :(
Mail SMTP
Mail localhost
```

This is equivalent to:

```
# configure base package
__PACKAGE__->config( name => MyApp );
# configure authentication
__PACKAGE__->config(
      'Plugin::Authentication' => {
          user_class => 'MyApp::Model::MyDB::Customer',
          ...
      },
_;
# configure sessions
__PACKAGE__->config(
  session => {
      expires => 3600,
      ...
  },
);
# configure email sending
__PACKAGE__->config( email => [qw/SMTP localhost/] );
```

Catalyst explains precedence of multiple sources for configuration values, how to access the values in your components, and many 'base' config variables used internally.

See also Config::General.

Skipping your VCS's directories

Catalyst uses Module::Pluggable to load Models, Views, and Controllers. Module::Pluggable will scan through all directories and load modules it finds. Sometimes you might want to skip some of these directories, for example when your version control system makes a subdirectory with meta-information in every version-controlled directory. While Catalyst skips subversion and CVS directories already, there are other source control systems. Here is the configuration you need to add their directories to the list to skip.

You can make Catalyst skip these directories using the Catalyst config:

```
# Configure the application
__PACKAGE__->config(
    name => 'MyApp',
    setup_components => { except => qr/SCCS/ },
);
```

See the Module::Pluggable manual page for more information on **except** and other options.

Users and Access Control

Most multiuser, and some single-user web applications require that users identify themselves, and the application is often required to define those roles. The recipes below describe some ways of doing this.

Authentication (logging in)

This is extensively covered in other documentation; see in particular Catalyst::Plugin::Authentication and the Authentication chapter of the Tutorial at Catalyst::Manual::Tutorial::06_Authorization.

Pass-through login (and other actions)

An easy way of having assorted actions that occur during the processing of a request that are orthogonal to its actual purpose - logins, silent commands etc. Provide actions for these, but when they're required for something else fill e.g. a form variable __login and have a sub begin like so:

```
    sub begin : Private {
      my ($self, $c) = @_;
      foreach my $action (qw/login docommand foo bar whatever/) {
        if ($c->req->params->{"__${action}"}) {
          $c->forward($action);
        }
      }
    }
```

Authentication/Authorization

This is done in several steps:

Verification

> Getting the user to identify themselves, by giving you some piece of information known only to you and the user. Then you can assume that the user is who they say they are. This is called **credential verification**.

Authorization

> Making sure the user only accesses functions you want them to access. This is done by checking the verified user's data against your internal list of groups, or allowed persons for the current page.

Modules

The Catalyst Authentication system is made up of many interacting modules, to give you the most flexibility possible.

Credential verifiers

A Credential module tables the user input, and passes it to a Store, or some other system, for verification. Typically, a user object is created by either this module or the Store and made accessible by a `$c->user` call.

Examples:

```
Password - Simple username/password checking.
HTTPD    - Checks using basic HTTP auth.
TypeKey  - Check using the typekey system.
```

Storage backends

A Storage backend contains the actual data representing the users. It is queried by the credential verifiers. Updating the store is not done within this system; you will need to do it yourself.

Examples:

```
DBIC     - Storage using a database via DBIx::Class.
Minimal  - Storage using a simple hash (for testing).
```

User objects

A User object is created by either the storage backend or the credential verifier, and is filled with the retrieved user information.

Examples:

```
Hash     - A simple hash of keys and values.
```

ACL authorization

ACL stands for Access Control List. The ACL plugin allows you to regulate access on a path-by-path basis, by listing which users, or roles, have access to which paths.

Roles authorization

Authorization by roles is for assigning users to groups, which can then be assigned to ACLs, or just checked when needed.

Logging in

When you have chosen your modules, all you need to do is call the `$c->authenticate` method. If called with no parameters, it will try to find suitable parameters, such as **username** and **password**, or you can pass it these values.

Checking roles

Role checking is done by using the `$c->check_user_roles` method. This will check using the currently logged-in user (via `$c->user`). You pass it the name of a role to check, and it returns true if the user is a member.

EXAMPLE

```
package MyApp;
use Moose;
use namespace::autoclean;
extends qw/Catalyst/;
use Catalyst qw/
  Authentication
  Authorization::Roles
/;

__PACKAGE__->config(
   authentication => {
       default_realm => 'test',
       realms => {
           test => {
               credential => {
                   class          => 'Password',
                   password_field => 'password',
                   password_type  => 'self_check',
               },
               store => {
                   class => 'Htpasswd',
                   file => 'htpasswd',
               },
           },
       },
   },
);

package MyApp::Controller::Root;
use Moose;
use namespace::autoclean;

BEGIN { extends 'Catalyst::Controller' }

__PACKAGE__->config(namespace => '');

sub login : Local {
   my ($self, $c) = @_;

   if ( my $user = $c->req->params->{user}
       and my $password = $c->req->param->{password} )
   {
       if ( $c->authenticate( username => $user, password => $password ) ) {
            $c->res->body( "hello " . $c->user->name );
       } else {
          # login incorrect
```

```
        }
    }
    else {
        # invalid form input
    }
}

sub restricted : Local {
    my ( $self, $c ) = @_;

    $c->detach("unauthorized")
      unless $c->check_user_roles( "admin" );

    # do something restricted here
}
```

Using authentication in a testing environment

Ideally, to write tests for authentication/authorization code one would first set up a test database with known data, then use Test::WWW::Mechanize::Catalyst to simulate a user logging in. Unfortunately this can be rather awkward, which is why it's a good thing that the authentication framework is so flexible.

Instead of using a test database, one can simply change the authentication store to something a bit easier to deal with in a testing environment. Additionally, this has the advantage of not modifying one's database, which can be problematic if one forgets to use the testing instead of production database.

Alternatively, if you want to authenticate real users, but not have to worry about their passwords, you can use Catalyst::Authentication::Credential::Testing to force all users to authenticate with a global password.

More information

Catalyst::Plugin::Authentication has a longer explanation.

Authorization

Introduction

Authorization is the step that comes after authentication. Authentication establishes that the user agent is really representing the user we think it's representing, and then authorization determines what this user is allowed to do.

Role Based Access Control

Under role based access control each user is allowed to perform any number of roles. For example, at a zoo no one but specially trained personnel can enter the moose cage (Mynd you, møøse bites kan be pretty nasti!). For example:

```
package Zoo::Controller::MooseCage;

sub feed_moose : Local {
    my ( $self, $c ) = @_;

    $c->model( "Moose" )->eat( $c->req->params->{food} );
}
```

With this action, anyone can just come into the moose cage and feed the moose, which is a very dangerous thing. We need to restrict this action, so that only a qualified moose feeder can perform that action.

The Authorization::Roles plugin lets us perform role based access control checks. Let's load it:

```
use parent qw/Catalyst/;
use Catalyst qw/
                Authentication
                Authorization::Roles
               /;
```

And now our action should look like this:

```
    sub feed_moose : Local {
        my ( $self, $c ) = @_;

        if ( $c->check_roles( "moose_feeder" ) ) {
            $c->model( "Moose" )->eat( $c->req->params->{food} );
        } else {
            $c->stash->{error} = "unauthorized";
        }
    }
```

This checks `$c->user`, and only if the user has **all** the roles in the list, a true value is returned.

`check_roles` has a sister method, `assert_roles`, which throws an exception if any roles are missing.

Some roles that might actually make sense in, say, a forum application:

- administrator
- moderator

each with a distinct task (system administration versus content administration).

Access Control Lists

Checking for roles all the time can be tedious and error prone.

The Authorization::ACL plugin lets us declare where we'd like checks to be done automatically for us.

For example, we may want to completely block out anyone who isn't a `moose_feeder` from the entire `MooseCage` controller:

```
    Zoo->deny_access_unless( "/moose_cage", [qw/moose_feeder/] );
```

The role list behaves in the same way as `check_roles`. However, the ACL plugin isn't limited to just interacting with the Roles plugin. We can use a code reference instead. For example, to allow either moose trainers or moose feeders into the moose cage, we can create a more complex check:

```
    Zoo->deny_access_unless( "/moose_cage", sub {
        my $c = shift;
        $c->check_roles( "moose_trainer" ) || $c->check_roles( "moose_feeder" );
    });
```

The more specific a role, the earlier it will be checked. Let's say moose feeders are now restricted to only the `feed_moose` action, while moose trainers get access everywhere:

```
    Zoo->deny_access_unless( "/moose_cage", [qw/moose_trainer/] );
    Zoo->allow_access_if( "/moose_cage/feed_moose", [qw/moose_feeder/]);
```

When the `feed_moose` action is accessed the second check will be made. If the user is a `moose_feeder`, then access will be immediately granted. Otherwise, the next rule in line will be tested - the one checking for a `moose_trainer`. If this rule is not satisfied, access will be immediately denied.

Rules applied to the same path will be checked in the order they were added.

Lastly, handling access denial events is done by creating an `access_denied` private action:

```
    sub access_denied : Private {
        my ( $self, $c, $action ) = @_;
    }
```

This action works much like auto, in that it is inherited across namespaces (not like object oriented code). This means that the `access_denied` action which is **nearest** to the action which was blocked will be triggered.

If this action does not exist, an error will be thrown, which you can clean up in your `end` private action instead.

Also, it's important to note that if you restrict access to "/" then `end`, `default`, etc. will also be restricted.

```
MyApp->acl_allow_root_internals;
```

will create rules that permit access to `end`, `begin`, and `auto` in the root of your app (but not in any other controller).

Models

Models are where application data belongs. Catalyst is extremely flexible with the kind of models that it can use. The recipes here are just the start.

Using existing DBIC (etc.) classes with Catalyst

Many people have existing Model classes that they would like to use with Catalyst (or, conversely, they want to write Catalyst models that can be used outside of Catalyst, e.g. in a cron job). It's trivial to write a simple component in Catalyst that slurps in an outside Model:

```
package MyApp::Model::DB;

use base qw/Catalyst::Model::DBIC::Schema/;

__PACKAGE__->config(
    schema_class => 'Some::DBIC::Schema',
    connect_info => ['dbi:SQLite:foo.db', '', '', {AutoCommit=>1}],
);

1;
```

and that's it! Now `Some::DBIC::Schema` is part of your Cat app as `MyApp::Model::DB`.

DBIx::Class as a Catalyst Model

See Catalyst::Model::DBIC::Schema.

Create accessors to preload static data once per server instance

When you have data that you want to load just once from the model at startup, instead of for each request, use mk_group_accessors to create accessors and tie them to resultsets in your package that inherits from DBIx::Class::Schema:

```
package My::Schema;
use base qw/DBIx::Class::Schema/;
__PACKAGE__->register_class('RESULTSOURCEMONIKER',
                            'My::Schema::RESULTSOURCE');
__PACKAGE__->mk_group_accessors('simple' =>
                            qw(ACCESSORNAME1 ACCESSORNAME2 ACCESSORNAMEn));

sub connection {
    my ($self, @rest) = @_;
    $self->next::method(@rest);
    # $self is now a live My::Schema object, complete with DB connection

    $self->ACCESSORNAME1([ $self->resultset('RESULTSOURCEMONIKER')->all ]);
    $self->ACCESSORNAME2([ $self->resultset('RESULTSOURCEMONIKER')->search({ COLUMN => { '<' => '30' } })->all ]);
    $self->ACCESSORNAMEn([ $self->resultset('RESULTSOURCEMONIKER')->find(1) ]);
}

1;
```

and now in the controller, you can now access any of these without a per-request fetch:

```
$c->stash->{something} = $c->model('My::Schema')->schema->ACCESSORNAME;
```

XMLRPC

Unlike SOAP, XMLRPC is a very simple (and elegant) web-services protocol, exchanging small XML messages like these:

Request:

```
POST /api HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Accept: text/xml
Accept: multipart/*
Host: 127.0.0.1:3000
User-Agent: SOAP::Lite/Perl/0.60
Content-Length: 192
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
    <methodName>add</methodName>
    <params>
        <param><value><int>1</int></value></param>
        <param><value><int>2</int></value></param>
    </params>
</methodCall>
```

Response:

```
Connection: close
Date: Tue, 20 Dec 2005 07:45:55 GMT
Content-Length: 133
Content-Type: text/xml
Status: 200
X-Catalyst: 5.70

<?xml version="1.0" encoding="us-ascii"?>
<methodResponse>
    <params>
        <param><value><int>3</int></value></param>
    </params>
</methodResponse>
```

Now follow these few steps to implement the application:

1. Install Catalyst (5.61 or later), Catalyst::Plugin::XMLRPC (0.06 or later) and SOAP::Lite (for XMLRPCsh.pl).

2. Create an application framework:

```
% catalyst.pl MyApp
...
% cd MyApp
```

3. Add the XMLRPC plugin to MyApp.pm

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

4. Add an API controller

```
% ./script/myapp_create.pl controller API
```

5. Add a XMLRPC redispatch method and an add method with Remote attribute to lib/MyApp/Controller/API.pm

```
    sub default :Path {
        my ( $self, $c ) = @_;
        $c->xmlrpc;
    }

    sub add : Remote {
        my ( $self, $c, $a, $b ) = @_;
        return $a + $b;
    }
```

The default action is the entry point for each XMLRPC request. It will redispatch every request to methods with Remote attribute in the same class.

The `add` method is not a traditional action; it has no private or public path. Only the XMLRPC dispatcher knows it exists.

6. That's it! You have built your first web service. Let's test it with XMLRPCsh.pl (part of SOAP::Lite):

```
    % ./script/myapp_server.pl
    ...
    % XMLRPCsh.pl http://127.0.0.1:3000/api
    Usage: method[(parameters)]
    > add( 1, 2 )
    --- XMLRPC RESULT ---
    '3'
```

Tip

Your return data type is usually auto-detected, but you can easily enforce a specific one.

```
    sub add : Remote {
        my ( $self, $c, $a, $b ) = @_;
        return RPC::XML::int->new( $a + $b );
    }
```

Views

Views pertain to the display of your application. As with models, Catalyst is uncommonly flexible. The recipes below are just a start.

Catalyst::View::TT

One of the first things you probably want to do when starting a new Catalyst application is set up your View. Catalyst doesn't care how you display your data; you can choose to generate HTML, PDF files, or plain text if you wanted.

Most Catalyst applications use a template system to generate their HTML, and though there are several template systems available, Template Toolkit is probably the most popular.

Once again, the Catalyst developers have done all the hard work, and made things easy for the rest of us. Catalyst::View::TT provides the interface to Template Toolkit, and provides Helpers which let us set it up that much more easily.

Creating your View

Catalyst::View::TT provides two different helpers for us to use: TT and TTSite.

TT

Create a basic Template Toolkit View using the provided helper script:

```
    script/myapp_create.pl view TT TT
```

This will create lib/MyApp/View/MyView.pm, which is going to be pretty empty to start. However, it sets everything up that you need to get started. You can now define which template you want and forward to your view. For instance:

```
    sub hello : Local {
        my ( $self, $c ) = @_;

        $c->stash->{template} = 'hello.tt';

        $c->forward( $c->view('TT') );
    }
```

In practice you wouldn't do the forwarding manually, but would use Catalyst::Action::RenderView.

TTSite

Although the TT helper does create a functional, working view, you may find yourself having to create the same template files and changing the same options every time you create a new application. The TTSite helper saves us even more time by creating the basic templates and setting some common options for us.

Once again, you can use the helper script:

```
    script/myapp_create.pl view TT TTSite
```

This time, the helper sets several options for us in the generated View.

```
    __PACKAGE__->config({
        CATALYST_VAR => 'Catalyst',
        INCLUDE_PATH => [
            MyApp->path_to( 'root', 'src' ),
            MyApp->path_to( 'root', 'lib' )
        ],
        PRE_PROCESS  => 'config/main',
        WRAPPER      => 'site/wrapper',
        ERROR        => 'error.tt2',
        TIMER        => 0
    });
```

- INCLUDE_PATH defines the directories that Template Toolkit should search for the template files.
- PRE_PROCESS is used to process configuration options which are common to every template file.
- WRAPPER is a file which is processed with each template, usually used to easily provide a common header and footer for every page.

In addition to setting these options, the TTSite helper also created the template and config files for us! In the 'root' directory, you'll notice two new directories: src and lib.

Several configuration files in root/lib/config are called by PRE_PROCESS.

The files in root/lib/site are the site-wide templates, called by WRAPPER, and display the html framework, control the layout, and provide the templates for the header and footer of your page. Using the template organization provided makes it much easier to standardize pages and make changes when they are (inevitably) needed.

The template files that you will create for your application will go into root/src, and you don't need to worry about putting the <html> or <head> sections; just put in the content. The WRAPPER will the rest of the page around your template for you.

$c->stash

Of course, having the template system include the header and footer for you isn't all that we want our templates to do. We need to be able to put data into our templates, and have it appear where and how we want it, right? That's where the stash comes in.

In our controllers, we can add data to the stash, and then access it from the template. For instance:

```
sub hello : Local {
    my ( $self, $c ) = @_;

    $c->stash->{name} = 'Adam';

    $c->stash->{template} = 'hello.tt';

    $c->forward( $c->view('TT') );
}
```

Then, in hello.tt:

```
<strong>Hello, [% name %]!</strong>
```

When you view this page, it will display "Hello, Adam!"

All of the information in your stash is available, by its name/key, in your templates. And your data don't have to be plain, old, boring scalars. You can pass array references and hash references, too.

In your controller:

```
sub hello : Local {
    my ( $self, $c ) = @_;

    $c->stash->{names} = [ 'Adam', 'Dave', 'John' ];

    $c->stash->{template} = 'hello.tt';

    $c->forward( $c->view('TT') );
}
```

In hello.tt:

```
[% FOREACH name IN names %]
    <strong>Hello, [% name %]!</strong><br />
[% END %]
```

This allowed us to loop through each item in the arrayref, and display a line for each name that we have.

This is the most basic usage, but Template Toolkit is quite powerful, and allows you to truly keep your presentation logic separate from the rest of your application.

$c->uri_for()

One of my favorite things about Catalyst is the ability to move an application around without having to worry that everything is going to break. One of the areas that used to be a problem was with the http links in your template files. For example, suppose you have an application installed at http://www.domain.com/Calendar. The links point to "/Calendar", "/Calendar/2005", "/Calendar/2005/10", etc. If you move the application to be at http://www.mydomain.com/Tools/Calendar, then all of those links will suddenly break.

That's where $c->uri_for() comes in. This function will merge its parameters with either the base location for the app, or its current namespace. Let's take a look at a couple of examples.

In your template, you can use the following:

```
<a href="[% c.uri_for('/login') %]">Login Here</a>
```

Although the parameter starts with a forward slash, this is relative to the application root, not the webserver root. This is important to remember. So, if your application is installed at http://www.domain.com/Calendar, then the link would be http://www.mydomain.com/Calendar/Login. If you move your application to a different domain or path, then that link will still be correct.

Likewise,

```
    <a href="[% c.uri_for('2005','10', '24') %]">October, 24 2005</a>
```

The first parameter does NOT have a forward slash, and so it will be relative to the current namespace. If the application is installed at http://www.domain.com/Calendar. and if the template is called from MyApp::Controller::Display, then the link would become http://www.domain.com/Calendar/Display/2005/10/24.

If you want to link to a parent uri of your current namespace you can prefix the arguments with multiple '../':

```
    <a href="[% c.uri_for('../../view', stashed_object.id) %]">User view</a>
```

Once again, this allows you to move your application around without having to worry about broken links. But there's something else, as well. Since the links are generated by uri_for, you can use the same template file by several different controllers, and each controller will get the links that its supposed to. Since we believe in Don't Repeat Yourself, this is particularly helpful if you have common elements in your site that you want to keep in one file.

Further Reading:

http://search.cpan.org/perldoc?Catalyst

http://search.cpan.org/perldoc?Catalyst%3A%3AView%3A%3ATT

http://search.cpan.org/perldoc?Template

Adding RSS feeds

Adding RSS feeds to your Catalyst applications is simple. We'll see two different approaches here, but the basic premise is that you forward to the normal view action first to get the objects, then handle the output differently.

Using XML::Feed

Assuming we have a `view` action that populates 'entries' with some DBIx::Class iterator, the code would look something like this:

```
    sub rss : Local {
        my ($self,$c) = @_;
        $c->forward('view'); # get the entries

        my $feed = XML::Feed->new('RSS');
        $feed->title( $c->config->{name} . ' RSS Feed' );
        $feed->link( $c->req->base ); # link to the site.
        $feed->description('Catalyst advent calendar'); Some description

        # Process the entries
        while( my $entry = $c->stash->{entries}->next ) {
            my $feed_entry = XML::Feed::Entry->new('RSS');
            $feed_entry->title($entry->title);
            $feed_entry->link( $c->uri_for($entry->link) );
            $feed_entry->issued( DateTime->from_epoch(epoch => $entry->created) );
            $feed->add_entry($feed_entry);
        }
        $c->res->body( $feed->as_xml );
    }
```

With this approach you're pretty sure to get something that validates.

Note that for both of the above approaches, you'll need to set the content type like this:

```
    $c->res->content_type('application/rss+xml');
```

Final words

You could generalize the second variant easily by replacing 'RSS' with a variable, so you can generate Atom feeds with the same code.

Now, go ahead and make RSS feeds for all your stuff. The world *needs* updates on your goldfish!

Forcing the browser to download content

Sometimes you need your application to send content for download. For example, you can generate a comma-separated values (CSV) file for your users to download and import into their spreadsheet program.

Let's say you have an `Orders` controller which generates a CSV file in the `export` action (i.e., `http://localhost:3000/orders/export`):

```
    sub export : Local Args(0) {
        my ( $self, $c ) = @_;

        # In a real application, you'd generate this from the database
        my $csv = "1,5.99\n2,29.99\n3,3.99\n";

        $c->res->content_type('text/comma-separated-values');
        $c->res->body($csv);
    }
```

Normally the browser uses the last part of the URI to generate a filename for data it cannot display. In this case your browser would likely ask you to save a file named `export`.

Luckily you can have the browser download the content with a specific filename by setting the `Content-Disposition` header:

```
    my $filename = 'Important Orders.csv';
    $c->res->header('Content-Disposition', qq[attachment; filename="$filename"]);
```

Note the use of quotes around the filename; this ensures that any spaces in the filename are handled by the browser.

Put this right before calling `$c->res->body` and your browser will download a file named `Important Orders.csv` instead of `export`.

You can also use this to have the browser download content which it normally displays, such as JPEG images or even HTML. Just be sure to set the appropriate content type and disposition.

Controllers

Controllers are the main point of communication between the web server and your application. Here we explore some aspects of how they work.

Action Types

Introduction

A Catalyst application is driven by one or more Controller modules. There are a number of ways that Catalyst can decide which of the methods in your controller modules it should call. Controller methods are also called actions, because they determine how your catalyst application should (re-)act to any given URL. When the application is started up, catalyst looks at all your actions, and decides which URLs they map to.

Type attributes

Each action is a normal method in your controller, except that it has an attribute attached. These can be one of several types.

Assume our Controller module starts with the following package declaration:

```
package MyApp::Controller::Buckets;
```

and we are running our application on localhost, port 3000 (the test server default).

Path

A Path attribute also takes an argument, this can be either a relative or an absolute path. A relative path will be relative to the controller namespace, an absolute path will represent an exact matching URL.

```
sub my_handles : Path('handles') { .. }
```

becomes

```
http://localhost:3000/buckets/handles
```

and

```
sub my_handles : Path('/handles') { .. }
```

becomes

```
http://localhost:3000/handles
```

See also: Catalyst::DispatchType::Path

Local

When using a Local attribute, no parameters are needed, instead, the name of the action is matched in the URL. The namespaces created by the name of the controller package is always part of the URL.

```
sub my_handles : Local { .. }
```

becomes

```
http://localhost:3000/buckets/my_handles
```

Global

A Global attribute is similar to a Local attribute, except that the namespace of the controller is ignored, and matching starts at root.

```
sub my_handles : Global { .. }
```

becomes

```
http://localhost:3000/my_handles
```

Regex

By now you should have figured that a Regex attribute is just what it sounds like. This one takes a regular expression, and matches starting from root. These differ from the rest as they can match multiple URLs.

```
sub my_handles : Regex('^handles') { .. }
```

matches

```
http://localhost:3000/handles
```

and

```
http://localhost:3000/handles_and_other_parts
```

etc.

See also: Catalyst::DispatchType::Regex

LocalRegex

A LocalRegex is similar to a Regex, except it only matches below the current controller namespace.

```
sub my_handles : LocalRegex(^handles') { .. }
```

matches

```
http://localhost:3000/buckets/handles
```

and

```
http://localhost:3000/buckets/handles_and_other_parts
```

etc.

Chained

See Catalyst::DispatchType::Chained for a description of how the chained dispatch type works.

Private

Last but not least, there is the Private attribute, which allows you to create your own internal actions, which can be forwarded to, but won't be matched as URLs.

```
sub my_handles : Private { .. }
```

becomes nothing at all..

Catalyst also predefines some special Private actions, which you can override, these are:

default

The default action will be called, if no other matching action is found. If you don't have one of these in your namespace, or any sub part of your namespace, you'll get an error page instead. If you want to find out where it was the user was trying to go, you can look in the request object using `$c->req->path`.

```
sub default :Path { .. }
```

works for all unknown URLs, in this controller namespace, or every one if put directly into MyApp.pm.

index

The index action is called when someone tries to visit the exact namespace of your controller. If index, default and matching Path actions are defined, then index will be used instead of default and Path.

```
sub index :Path :Args(0) { .. }
```

becomes

```
http://localhost:3000/buckets
```

begin

The begin action is called at the beginning of every request involving this namespace directly, before other matching actions are called. It can be used to set up variables/data for this particular part of your app. A single begin action is called, its always the one most relevant to the current namespace.

```
sub begin : Private { .. }
```

is called once when

```
http://localhost:3000/bucket/(anything)?
```

is visited.

end

Like begin, this action is always called for the namespace it is in, after every other action has finished. It is commonly used to forward processing to the View component. A single end action is called, its always the one most relevant to the current namespace.

```
sub end : Private { .. }
```

is called once after any actions when

```
http://localhost:3000/bucket/(anything)?
```

is visited.

auto

Lastly, the auto action is magic in that **every** auto action in the chain of paths up to and including the ending namespace, will be called. (In contrast, only one of the begin/end/default actions will be called, the relevant one).

```
package MyApp::Controller::Root;
sub auto : Private { .. }
```

and

```
sub auto : Private { .. }
```

will both be called when visiting

```
http://localhost:3000/bucket/(anything)?
```

A word of warning

You can put root actions in your main MyApp.pm file, but this is deprecated, please put your actions into your Root controller.

Flowchart

A graphical flowchart of how the dispatcher works can be found on the wiki at http://dev.catalyst.perl.org/attachment/wiki/WikiStart/catalyst-flow.png.

DRY Controllers with Chained actions

Imagine that you would like the following paths in your application:

/cd/<ID>/track/<ID>

> Displays info on a particular track.
>
> In the case of a multi-volume CD, this is the track sequence.

/cd/<ID>/volume/<ID>/track/<ID>

> Displays info on a track on a specific volume.

Here is some example code, showing how to do this with chained controllers:

```
package CD::Controller;
use base qw/Catalyst::Controller/;

sub root : Chained('/') PathPart('/cd') CaptureArgs(1) {
    my ($self, $c, $cd_id) = @_;
    $c->stash->{cd_id} = $cd_id;
    $c->stash->{cd} = $self->model('CD')->find_by_id($cd_id);
}

sub trackinfo : Chained('track') PathPart('') Args(0) RenderView {
    my ($self, $c) = @_;
}

package CD::Controller::ByTrackSeq;
use base qw/CD::Controller/;

sub track : Chained('root') PathPart('track') CaptureArgs(1) {
    my ($self, $c, $track_seq) = @_;
    $c->stash->{track} = $self->stash->{cd}->find_track_by_seq($track_seq);
}

package CD::Controller::ByTrackVolNo;
use base qw/CD::Controller/;

sub volume : Chained('root') PathPart('volume') CaptureArgs(1) {
    my ($self, $c, $volume) = @_;
    $c->stash->{volume} = $volume;
}

sub track : Chained('volume') PathPart('track') CaptureArgs(1) {
    my ($self, $c, $track_no) = @_;
    $c->stash->{track} = $self->stash->{cd}->find_track_by_vol_and_track_no(
        $c->stash->{volume}, $track_no
    );
}
```

Note that adding other actions (i.e. chain endpoints) which operate on a track is simply a matter of adding a new sub to CD::Controller - no code is duplicated, even though there are two different methods of looking up a track.

This technique can be expanded as needed to fulfil your requirements - for example, if you inherit the first action of a chain from a base class, then mixing in a different base class can be used to duplicate an entire URL hierarchy at a different point within your application.

Component-based Subrequests

See Catalyst::Plugin::SubRequest.

File uploads

Single file upload with Catalyst

To implement uploads in Catalyst, you need to have a HTML form similar to this:

```
<form action="/upload" method="post" enctype="multipart/form-data">
  <input type="hidden" name="form_submit" value="yes">
  <input type="file" name="my_file">
  <input type="submit" value="Send">
</form>
```

It's very important not to forget `enctype="multipart/form-data"` in the form.

Catalyst Controller module 'upload' action:

```
sub upload : Global {
    my ($self, $c) = @_;

    if ( $c->request->parameters->{form_submit} eq 'yes' ) {

        if ( my $upload = $c->request->upload('my_file') ) {

            my $filename = $upload->filename;
            my $target   = "/tmp/upload/$filename";

            unless ( $upload->link_to($target) || $upload->copy_to($target) ) {
                die( "Failed to copy '$filename' to '$target': $!" );
            }
        }
    }

    $c->stash->{template} = 'file_upload.html';
}
```

Multiple file upload with Catalyst

Code for uploading multiple files from one form needs a few changes:

The form should have this basic structure:

```
<form action="/upload" method="post" enctype="multipart/form-data">
  <input type="hidden" name="form_submit" value="yes">
  <input type="file" name="file1" size="50"><br>
  <input type="file" name="file2" size="50"><br>
  <input type="file" name="file3" size="50"><br>
  <input type="submit" value="Send">
</form>
```

And in the controller:

```
sub upload : Local {
    my ($self, $c) = @_;

    if ( $c->request->parameters->{form_submit} eq 'yes' ) {

        for my $field ( $c->req->upload ) {

            my $upload   = $c->req->upload($field);
            my $filename = $upload->filename;
            my $target   = "/tmp/upload/$filename";

            unless ( $upload->link_to($target) || $upload->copy_to($target) ) {
                die( "Failed to copy '$filename' to '$target': $!" );
```

```
            }
        }
    }

    $c->stash->{template} = 'file_upload.html';
}
```

`for my $field ($c->req-upload)>` loops automatically over all file input fields and gets input names. After that is basic file saving code, just like in single file upload.

Notice: `die`ing might not be what you want to do, when an error occurs, but it works as an example. A better idea would be to store error `$!` in `$c->stash->{error}` and show a custom error template displaying this message.

For more information about uploads and usable methods look at Catalyst::Request::Upload and Catalyst::Request.

Forwarding with arguments

Sometimes you want to pass along arguments when forwarding to another action. As of version 5.30, arguments can be passed in the call to `forward`; in earlier versions, you can manually set the arguments in the Catalyst Request object:

```
# version 5.30 and later:
$c->forward('/wherever', [qw/arg1 arg2 arg3/]);

# pre-5.30
$c->req->args([qw/arg1 arg2 arg3/]);
$c->forward('/wherever');
```

(See the Catalyst::Manual::Intro Flow_Control section for more information on passing arguments via `forward`.)

Chained dispatch using base classes, and inner packages.

```
package MyApp::Controller::Base;
use base qw/Catalyst::Controller/;

sub key1 : Chained('/')
```

Extending RenderView (formerly DefaultEnd)

The recommended approach for an `end` action is to use Catalyst::Action::RenderView (taking the place of Catalyst::Plugin::DefaultEnd), which does what you usually need. However there are times when you need to add a bit to it, but don't want to write your own `end` action.

You can extend it like this:

To add something to an `end` action that is called before rendering (this is likely to be what you want), simply place it in the `end` method:

```
sub end : ActionClass('RenderView') {
  my ( $self, $c ) = @_;
  # do stuff here; the RenderView action is called afterwards
}
```

To add things to an `end` action that are called *after* rendering, you can set it up like this:

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

sub end : Private {
  my ( $self, $c ) = @_;
  $c->forward('render');
  # do stuff here
}
```

Serving static content

Serving static content in Catalyst used to be somewhat tricky; the use of Catalyst::Plugin::Static::Simple makes everything much easier. This plugin will automatically serve your static content during development, but allows you to easily switch to Apache (or other server) in a production environment.

Introduction to Static::Simple

Static::Simple is a plugin that will help to serve static content for your application. By default, it will serve most types of files, excluding some standard Template Toolkit extensions, out of your **root** file directory. All files are served by path, so if **images/me.jpg** is requested, then **root/images/me.jpg** is found and served.

Usage

Using the plugin is as simple as setting your use line in MyApp.pm to include:

```
use Catalyst qw/Static::Simple/;
```

and already files will be served.

Configuring

Static content is best served from a single directory within your root directory. Having many different directories such as `root/css` and `root/images` requires more code to manage, because you must separately identify each static directory--if you decide to add a `root/js` directory, you'll need to change your code to account for it. In contrast, keeping all static directories as subdirectories of a main `root/static` directory makes things much easier to manage. Here's an example of a typical root directory structure:

```
root/
root/content.tt
root/controller/stuff.tt
root/header.tt
root/static/
root/static/css/main.css
root/static/images/logo.jpg
root/static/js/code.js
```

All static content lives under `root/static`, with everything else being Template Toolkit files.

Include Path

You may of course want to change the default locations, and make Static::Simple look somewhere else, this is as easy as:

```
MyApp->config(
   static => {
       include_path => [
           MyApp->path_to('/'),
           '/path/to/my/files',
       ],
   },
 );
```

When you override include_path, it will not automatically append the normal root path, so you need to add it yourself if you still want it. These will be searched in order given, and the first matching file served.

Static directories

If you want to force some directories to be only static, you can set them using paths relative to the root dir, or regular expressions:

```
MyApp->config(
   static => {
       dirs => [
           'static',
```

```
            qr/^(images|css)/,
        ],
    },
);
```

File extensions

By default, the following extensions are not served (that is, they will be processed by Catalyst): **tmpl, tt, tt2, html, xhtml**. This list can be replaced easily:

```
MyApp->config(
       static => {
           ignore_extensions => [
               qw/tmpl tt tt2 html xhtml/
           ],
       },
);
```

Ignoring directories

Entire directories can be ignored. If used with include_path, directories relative to the include_path dirs will also be ignored:

```
MyApp->config( static => {
      ignore_dirs => [ qw/tmpl css/ ],
});
```

More information

http://search.cpan.org/dist/Catalyst-Plugin-Static-Simple/

Serving manually with the Static plugin with HTTP::Daemon (myapp_server.pl)

In some situations you might want to control things more directly, using Catalyst::Plugin::Static.

In your main application class (MyApp.pm), load the plugin:

```
use Catalyst qw/-Debug FormValidator Static OtherPlugin/;
```

You will also need to make sure your end method does *not* forward static content to the view, perhaps like this:

```
sub end : Private {
    my ( $self, $c ) = @_;

    $c->forward( 'MyApp::View::TT' )
      unless ( $c->res->body || !$c->stash->{template} );
}
```

This code will only forward to the view if a template has been previously defined by a controller and if there is not already data in `$c->res->body`.

Next, create a controller to handle requests for the /static path. Use the Helper to save time. This command will create a stub controller as `lib/MyApp/Controller/Static.pm`.

```
$ script/myapp_create.pl controller Static
```

Edit the file and add the following methods:

```
# serve all files under /static as static files
sub default : Path('/static') {
```

```
    my ( $self, $c ) = @_;

    # Optional, allow the browser to cache the content
    $c->res->headers->header( 'Cache-Control' => 'max-age=86400' );

    $c->serve_static; # from Catalyst::Plugin::Static
}

# also handle requests for /favicon.ico
sub favicon : Path('/favicon.ico') {
    my ( $self, $c ) = @_;

    $c->serve_static;
}
```

You can also define a different icon for the browser to use instead of favicon.ico by using this in your HTML header:

```
<link rel="icon" href="/static/myapp.ico" type="image/x-icon" />
```

Common problems with the Static plugin

The Static plugin makes use of the `shared-mime-info` package to automatically determine MIME types. This package is notoriously difficult to install, especially on win32 and OS X. For OS X the easiest path might be to install Fink, then use `apt-get install shared-mime-info`. Restart the server, and everything should be fine.

Make sure you are using the latest version (>= 0.16) for best results. If you are having errors serving CSS files, or if they get served as text/plain instead of text/css, you may have an outdated shared-mime-info version. You may also wish to simply use the following code in your Static controller:

```
if ($c->req->path =~ /css$/i) {
    $c->serve_static( "text/css" );
} else {
    $c->serve_static;
}
```

Serving Static Files with Apache

When using Apache, you can bypass Catalyst and any Static plugins/controllers controller by intercepting requests for the `root/static` path at the server level. All that is required is to define a DocumentRoot and add a separate Location block for your static content. Here is a complete config for this application under mod_perl 1.x:

```
<Perl>
    use lib qw(/var/www/MyApp/lib);
</Perl>
PerlModule MyApp

<VirtualHost *>
    ServerName myapp.example.com
    DocumentRoot /var/www/MyApp/root
    <Location />
        SetHandler perl-script
        PerlHandler MyApp
    </Location>
    <LocationMatch "/(static|favicon.ico)">
        SetHandler default-handler
    </LocationMatch>
</VirtualHost>
```

And here's a simpler example that'll get you started:

```
Alias /static/ "/my/static/files/"
<Location "/static">
    SetHandler none
</Location>
```

Caching

Catalyst makes it easy to employ several different types of caching to speed up your applications.

Cache Plugins

There are three wrapper plugins around common CPAN cache modules: Cache::FastMmap, Cache::FileCache, and Cache::Memcached. These can be used to cache the result of slow operations.

The Catalyst Advent Calendar uses the FileCache plugin to cache the rendered XHTML version of the source POD document. This is an ideal application for a cache because the source document changes infrequently but may be viewed many times.

```
    use Catalyst qw/Cache::FileCache/;

    ...

    use File::stat;
    sub render_pod : Local {
        my ( self, $c ) = @_;

        # the cache is keyed on the filename and the modification time
        # to check for updates to the file.
        my $file  = $c->path_to( 'root', '2005', '11.pod' );
        my $mtime = ( stat $file )->mtime;

        my $cached_pod = $c->cache->get("$file $mtime");
        if ( !$cached_pod ) {
            $cached_pod = do_slow_pod_rendering();
            # cache the result for 12 hours
            $c->cache->set( "$file $mtime", $cached_pod, '12h' );
        }
        $c->stash->{pod} = $cached_pod;
    }
```

We could actually cache the result forever, but using a value such as 12 hours allows old entries to be automatically expired when they are no longer needed.

Page Caching

Another method of caching is to cache the entire HTML page. While this is traditionally handled by a frontend proxy server like Squid, the Catalyst PageCache plugin makes it trivial to cache the entire output from frequently-used or slow actions.

Many sites have a busy content-filled front page that might look something like this. It probably takes a while to process, and will do the exact same thing for every single user who views the page.

```
    sub front_page : Path('/') {
        my ( $self, $c ) = @_;

        $c->forward( 'get_news_articles' );
        $c->forward( 'build_lots_of_boxes' );
        $c->forward( 'more_slow_stuff' );

        $c->stash->{template} = 'index.tt';
    }
```

We can add the PageCache plugin to speed things up.

```
    use Catalyst qw/Cache::FileCache PageCache/;

    sub front_page : Path ('/') {
        my ( $self, $c ) = @_;

        $c->cache_page( 300 );
```

```
        # same processing as above
    }
```

Now the entire output of the front page, from <html> to </html>, will be cached for 5 minutes. After 5 minutes, the next request will rebuild the page and it will be re-cached.

Note that the page cache is keyed on the page URI plus all parameters, so requests for / and /?foo=bar will result in different cache items. Also, only GET requests will be cached by the plugin.

You can even get that frontend Squid proxy to help out by enabling HTTP headers for the cached page.

```
    MyApp->config(
        page_cache => {
            set_http_headers => 1,
        },
    );
```

This would now set the following headers so proxies and browsers may cache the content themselves.

```
    Cache-Control: max-age=($expire_time - time)
    Expires: $expire_time
    Last-Modified: $cache_created_time
```

Template Caching

Template Toolkit provides support for caching compiled versions of your templates. To enable this in Catalyst, use the following configuration. TT will cache compiled templates keyed on the file mtime, so changes will still be automatically detected.

```
    package MyApp::View::TT;

    use strict;
    use warnings;
    use base 'Catalyst::View::TT';

    __PACKAGE__->config(
        COMPILE_DIR => '/tmp/template_cache',
    );

    1;
```

More Info

See the documentation for each cache plugin for more details and other available configuration options.

Catalyst::Plugin::Cache::FastMmap Catalyst::Plugin::Cache::FileCache Catalyst::Plugin::Cache::Memcached Catalyst::Plugin::PageCache http://search.cpan.org/dist/Template-Toolkit/lib/Template/Manual/Config.pod#Caching_and_Compiling_Options

Testing ⬆

Testing is an integral part of the web application development process. Tests make multi developer teams easier to coordinate, and they help ensure that there are no nasty surprises after upgrades or alterations.

Testing

Catalyst provides a convenient way of testing your application during development and before deployment in a real environment.

`Catalyst::Test` makes it possible to run the same tests both locally (without an external daemon) and against a remote server via HTTP.

Tests

Let's examine a skeleton application's `t/` directory:

```
mundus:~/MyApp chansen$ ls -l t/
total 24
-rw-r--r--  1 chansen  chansen   95 18 Dec 20:50 01app.t
-rw-r--r--  1 chansen  chansen  190 18 Dec 20:50 02pod.t
-rw-r--r--  1 chansen  chansen  213 18 Dec 20:50 03podcoverage.t
```

`01app.t`

Verifies that the application loads, compiles, and returns a successful response.

`02pod.t`

Verifies that all POD is free from errors. Only executed if the `TEST_POD` environment variable is true.

`03podcoverage.t`

Verifies that all methods/functions have POD coverage. Only executed if the `TEST_POD` environment variable is true.

Creating tests

```
mundus:~/MyApp chansen$ cat t/01app.t | perl -ne 'printf( "%2d  %s", $., $_ )'
1  use Test::More tests => 2;
2  BEGIN { use_ok( Catalyst::Test, 'MyApp' ) }
3
4  ok( request('/')->is_success );
```

The first line declares how many tests we are going to run, in this case two. The second line tests and loads our application in test mode. The fourth line verifies that our application returns a successful response.

`Catalyst::Test` exports two functions, `request` and `get`. Each can take three different arguments:

A string which is a relative or absolute URI.

```
request('/my/path');
request('http://www.host.com/my/path');
```

An instance of `URI`.

```
request( URI->new('http://www.host.com/my/path') );
```

An instance of `HTTP::Request`.

```
request( HTTP::Request->new( GET => 'http://www.host.com/my/path') );
```

`request` returns an instance of `HTTP::Response` and `get` returns the content (body) of the response.

Running tests locally

```
mundus:~/MyApp chansen$ CATALYST_DEBUG=0 TEST_POD=1 prove --lib lib/ t/
t/01app.....ok
t/02pod.....ok
t/03podcoverage....ok
All tests successful.
Files=3, Tests=4,  2 wallclock secs ( 1.60 cusr +  0.36 csys =  1.96 CPU)
```

`CATALYST_DEBUG=0` ensures that debugging is off; if it's enabled you will see debug logs between tests.

`TEST_POD=1` enables POD checking and coverage.

`prove` A command-line tool that makes it easy to run tests. You can find out more about it from the links below.

Running tests remotely

```
mundus:~/MyApp chansen$ CATALYST_SERVER=http://localhost:3000/ prove --lib lib/ t/01app.t
t/01app....ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.40 usr +  0.01 csys =  0.41 CPU)
```

`CATALYST_SERVER=http://localhost:3000/` is the absolute deployment URI of your application. In `CGI` or `FastCGI` it should be the host and path to the script.

`Test::WWW::Mechanize` and Catalyst

Be sure to check out `Test::WWW::Mechanize::Catalyst`. It makes it easy to test HTML, forms and links. A short example of usage:

```
use Test::More tests => 6;
BEGIN { use_ok( Test::WWW::Mechanize::Catalyst, 'MyApp' ) }

my $mech = Test::WWW::Mechanize::Catalyst->new;
$mech->get_ok("http://localhost/", 'Got index page');
$mech->title_like( qr/^MyApp on Catalyst/, 'Got right index title' );
ok( $mech->find_link( text_regex => qr/^Wiki/i ), 'Found link to Wiki' );
ok( $mech->find_link( text_regex => qr/^Mailing-List/i ), 'Found link to Mailing-List' );
ok( $mech->find_link( text_regex => qr/^IRC channel/i ), 'Found link to IRC channel' );
```

Further Reading

Catalyst::Test

> Catalyst::Test

Test::WWW::Mechanize::Catalyst

> http://search.cpan.org/dist/Test-WWW-Mechanize-Catalyst/lib/Test/WWW/Mechanize/Catalyst.pm

Test::WWW::Mechanize

> http://search.cpan.org/dist/Test-WWW-Mechanize/Mechanize.pm

WWW::Mechanize

> http://search.cpan.org/dist/WWW-Mechanize/lib/WWW/Mechanize.pm

LWP::UserAgent

> http://search.cpan.org/dist/libwww-perl/lib/LWP/UserAgent.pm

HTML::Form

> http://search.cpan.org/dist/libwww-perl/lib/HTML/Form.pm

HTTP::Message

> http://search.cpan.org/dist/libwww-perl/lib/HTTP/Message.pm

HTTP::Request

> http://search.cpan.org/dist/libwww-perl/lib/HTTP/Request.pm

HTTP::Request::Common

> http://search.cpan.org/dist/libwww-perl/lib/HTTP/Request/Common.pm

HTTP::Response

http://search.cpan.org/dist/libwww-perl/lib/HTTP/Response.pm

HTTP::Status

http://search.cpan.org/dist/libwww-perl/lib/HTTP/Status.pm

URI

http://search.cpan.org/dist/URI/URI.pm

Test::More

http://search.cpan.org/dist/Test-Simple/lib/Test/More.pm

Test::Pod

http://search.cpan.org/dist/Test-Pod/Pod.pm

Test::Pod::Coverage

http://search.cpan.org/dist/Test-Pod-Coverage/Coverage.pm

prove (Test::Harness)

http://search.cpan.org/dist/Test-Harness/bin/prove

More Information

http://search.cpan.org/perldoc?Catalyst::Plugin::Authorization::Roles http://search.cpan.org/perldoc?Catalyst::Plugin::Authorization::ACL

AUTHORS

Catalyst Contributors, see Catalyst.pm

COPYRIGHT