NAME

Catalyst::Manual::Intro - Introduction to Catalyst

DESCRIPTION

This is a brief introduction to Catalyst. It explains the most important features of how Catalyst works and shows how to get a simple application up and running quickly. For an introduction (without code) to Catalyst itself, and why you should be using it, see Catalyst::Manual::About. For a systematic step-by-step introduction to writing an application with Catalyst, see Catalyst::Manual::Tutorial.

What is Catalyst?

Catalyst is an elegant web application framework, extremely flexible yet extremely simple. It's similar to Ruby on Rails, Spring (Java), and Maypole, upon which it was originally based. Its most important design philosophy is to provide easy access to all the tools you need to develop web applications, with few restrictions on how you need to use these tools. However, this does mean that it is always possible to do things in a different way. Other web frameworks are *initially* simpler to use, but achieve this by locking the programmer into a single set of tools. Catalyst's emphasis on flexibility means that you have to think more to use it. We view this as a feature. For example, this leads to Catalyst being better suited to system integration tasks than other web frameworks.

MVC

Catalyst follows the Model-View-Controller (MVC) design pattern, allowing you to easily separate concerns, like content, presentation, and flow control, into separate modules. This separation allows you to modify code that handles one concern without affecting code that handles the others. Catalyst promotes the re-use of existing Perl modules that already handle common web application concerns well.

Here's how the Model, View, and Controller map to those concerns, with examples of well-known Perl modules you may want to use for each.

- **Model**

  Access and modify content (data). DBIx::Class, Class::DBI, Xapian, Net::LDAP...

- **View**

  Present content to the user. Template Toolkit, Mason, HTML::Template...

- **Controller**

  Control the whole request phase, check parameters, dispatch actions, flow control. This is the meat of where Catalyst works.

If you're unfamiliar with MVC and design patterns, you may want to check out the original book on the subject, *Design Patterns*, by Gamma, Helm, Johnson, and Vlissides, also known as the Gang of Four (GoF). Many, many web application frameworks are based on MVC, which is becoming a popular design paradigm for the world wide web.

Flexibility

Catalyst is much more flexible than many other frameworks. Rest assured you can use your favorite Perl modules with Catalyst.

- **Multiple Models, Views, and Controllers**

  To build a Catalyst application, you handle each type of concern inside special modules called "Components". Often this code will be very simple, just calling out to Perl modules like those listed above under "MVC". Catalyst handles these components in a very flexible way. Use as many Models, Views, and Controllers as you like, using as many different Perl modules as you like, all in the same application. Want to manipulate multiple databases, and retrieve some data via LDAP? No problem. Want to present data from the same Model using Template Toolkit and PDF::Template? Easy.

- **Reuseable Components**

  Not only does Catalyst promote the re-use of already existing Perl modules, it also allows you to re-use your Catalyst components in multiple Catalyst applications.

- **Unrestrained URL-to-Action Dispatching**

  Catalyst allows you to dispatch any URLs to any application "Actions", even through regular expressions! Unlike most other frameworks, it doesn't require mod_rewrite or class and method names in URLs.

With Catalyst you register your actions and address them directly. For example:

```
    sub hello : Local {
        my ( $self, $context ) = @_;
        $context->response->body('Hello World!');
    }
```

Now http://localhost:3000/hello prints "Hello World!".

Note that actions with the `:Local` attribute are equivalent to using a `:Path('action_name')` attribute, so our action could be equivalently:

```
    sub hi : Path('hello') {
        my ( $self, $context ) = @_;
        $context->response->body('Hello World!');
    }
```

- **Support for CGI, mod_perl, Apache::Request, FastCGI**

  Use Catalyst::Engine::Apache or Catalyst::Engine::CGI. Another interesting engine is Catalyst::Engine::HTTP::Prefork - available from CPAN separately - which will turn the built server into a fully fledged production ready server (although you'll probably want to run it behind a front end proxy if you end up using it).

- PSGI Support

  Starting with Catalyst version 5.9 Catalyst ships with PSGI integration for even more powerful and flexible testing and deployment options. See Catalyst::PSGI for details.

Simplicity

The best part is that Catalyst implements all this flexibility in a very simple way.

- **Building Block Interface**

  Components interoperate very smoothly. For example, Catalyst automatically makes a "Context" object available to every component. Via the context, you can access the request object, share data between components, and control the flow of your application. Building a Catalyst application feels a lot like snapping together toy building blocks, and everything just works.

- **Component Auto-Discovery**

  No need to `use` all of your components. Catalyst automatically finds and loads them.

- **Pre-Built Components for Popular Modules**

  See Catalyst::Model::DBIC::Schema for DBIx::Class, or Catalyst::View::TT for Template Toolkit.

- **Built-in Test Framework**

  Catalyst comes with a built-in, lightweight http server and test framework, making it easy to test applications from the web browser, and the command line.

- **Helper Scripts**

Catalyst provides helper scripts to quickly generate running starter code for components and unit tests. Install Catalyst::Devel and see Catalyst::Helper.

Quickstart

Here's how to install Catalyst and get a simple application up and running, using the helper scripts described above.

Install

Installation of Catalyst should be straightforward:

```
    # perl -MCPAN -e 'install Catalyst::Runtime'
    # perl -MCPAN -e 'install Catalyst::Devel'
    # perl -MCPAN -e 'install Catalyst::View::TT'
```

Setup

```
    $ catalyst.pl MyApp
    # output omitted
    $ cd MyApp
    $ script/myapp_create.pl controller Library::Login
```

Frank Speiser's Amazon EC2 Catalyst SDK

There are currently two flavors of publicly available Amazon Machine Images (AMI) that include all the elements you'd need to begin developing in a fully functional Catalyst environment within minutes. See Catalyst::Manual::Installation for more details.

Run

```
    $ script/myapp_server.pl
```

Now visit these locations with your favorite browser or user agent to see Catalyst in action:

(NOTE: Although we create a controller here, we don't actually use it. Both of these URLs should take you to the welcome page.)

http://localhost:3000/

http://localhost:3000/library/login/

How It Works

Let's see how Catalyst works, by taking a closer look at the components and other parts of a Catalyst application.

Components

Catalyst has an uncommonly flexible component system. You can define as many "Models", "Views", and "Controllers" as you like. As discussed previously, the general idea is that the View is responsible for the output of data to the user (typically via a web browser, but a View can also generate PDFs or e-mails, for example); the Model is responsible for providing data (typically from a relational database);

and the Controller is responsible for interacting with the user and deciding how user input determines what actions the application takes.

In the world of MVC, there are frequent discussions and disagreements about the nature of each element - whether certain types of logic belong in the Model or the Controller, etc. Catalyst's flexibility means that this decision is entirely up to you, the programmer; Catalyst doesn't enforce anything. See Catalyst::Manual::About for a general discussion of these issues.

Model, View and Controller components must inherit from Catalyst::Model, Catalyst::View and Catalyst::Controller, respectively. These, in turn, inherit from Catalyst::Component which provides a simple class structure and some common class methods like `config` and `new` (constructor).

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

    BEGIN { extends 'Catalyst::Controller' }

    __PACKAGE__->config( foo => 'bar' );

    1;
```

You don't have to `use` or otherwise register Models, Views, and Controllers. Catalyst automatically discovers and instantiates them when you call `setup` in the main application. All you need to do is put them in directories named for each Component type. You can use a short alias for each one.

- **MyApp/Model/**
- **MyApp/View/**
- **MyApp/Controller/**

Views

To show how to define views, we'll use an already-existing base class for the Template Toolkit, Catalyst::View::TT. All we need to do is inherit from this class:

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

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

    1;
```

(You can also generate this automatically by using the helper script:

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

where the first `TT` tells the script that the name of the view should be `TT`, and the second that it should be a Template Toolkit view.)

This gives us a process() method and we can now just do $c->forward('MyApp::View::TT') to render our templates. The base class makes process() implicit, so we don't have to say `$c->forward(qw/MyApp::View::TT process/)`.

```
    sub hello : Global {
        my ( $self, $c ) = @_;
        $c->stash->{template} = 'hello.tt';
    }

    sub end : Private {
        my ( $self, $c ) = @_;
        $c->forward( $c->view('TT') );
    }
```

You normally render templates at the end of a request, so it's a perfect use for the global `end` action.

In practice, however, you would use a default `end` action as supplied by Catalyst::Action::RenderView.

Also, be sure to put the template under the directory specified in `$c->config->{root}`, or you'll end up looking at the debug screen.

Models

Models are providers of data. This data could come from anywhere - a search engine index, a spreadsheet, the file system - but typically a Model represents a database table. The data source does not intrinsically have much to do with web applications or Catalyst - it could just as easily be used to write an offline report generator or a command-line tool.

To show how to define models, again we'll use an already-existing base class, this time for DBIx::Class: Catalyst::Model::DBIC::Schema. We'll also need DBIx::Class::Schema::Loader.

But first, we need a database.

```
    -- myapp.sql
    CREATE TABLE foo (
        id INTEGER PRIMARY KEY,
        data TEXT
    );

    CREATE TABLE bar (
        id INTEGER PRIMARY KEY,
        foo INTEGER REFERENCES foo,
        data TEXT
    );

    INSERT INTO foo (data) VALUES ('TEST!');

    % sqlite3 /tmp/myapp.db < myapp.sql
```

Now we can create a DBIC::Schema model for this database.

```
    script/myapp_create.pl model MyModel DBIC::Schema MySchema create=static 'dbi:SQLite:/tmp/myapp.db'
```

DBIx::Class::Schema::Loader can automatically load table layouts and relationships, and convert them into a static schema definition `MySchema`, which you can edit later.

Use the stash to pass data to your templates.

We add the following to MyApp/Controller/Root.pm

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

        $c->stash->{item} = $c->model('MyModel::Foo')->find($id);
    }

    1;

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

        $c->stash->{template} ||= 'index.tt';
        $c->forward( $c->view('TT') );
    }
```

We then create a new template file "root/index.tt" containing:

```
    The Id's data is [% item.data %]
```

Models do not have to be part of your Catalyst application; you can always call an outside module that serves as your Model:

```
    # in a Controller
    sub list : Local {
      my ( $self, $c ) = @_;

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

      use Some::Outside::Database::Module;
      my @records = Some::Outside::Database::Module->search({
        artist => 'Led Zeppelin',
        });

      $c->stash->{records} = \@records;
    }
```

But by using a Model that is part of your Catalyst application, you gain several things: you don't have to `use` each component, Catalyst will find and load it automatically at compile-time; you can `forward` to the module, which can only be done to Catalyst components. Only Catalyst components can be fetched with `$c->model('SomeModel')`.

Happily, since 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`.

Within Catalyst, the common approach to writing a model for your application is wrapping a generic model (e.g. DBIx::Class::Schema, a bunch of XMLs, or anything really) with an object that contains configuration data, convenience methods, and so forth. Thus you will in effect have two models - a wrapper model that knows something about Catalyst and your web application, and a generic model that is totally independent of these needs.

Technically, within Catalyst a model is a **component** - an instance of the model's class belonging to the application. It is important to stress that the lifetime of these objects is per application, not per request.

While the model base class (Catalyst::Model) provides things like `config` to better integrate the model into the application, sometimes this is not enough, and the model requires access to `$c` itself.

Situations where this need might arise include:

- Interacting with another model
- Using per-request data to control behavior
- Using plugins from a Model (for example Catalyst::Plugin::Cache).

From a style perspective it's usually considered bad form to make your model "too smart" about things - it should worry about business logic and leave the integration details to the controllers. If, however, you find that it does not make sense at all to use an auxiliary controller around the model, and the model's need to access `$c` cannot be sidestepped, there exists a power tool called "ACCEPT_CONTEXT".

Controllers

Multiple controllers are a good way to separate logical domains of your application.

```
    package MyApp::Controller::Login;

    use base qw/Catalyst::Controller/;

    sub sign_in : Path("sign-in") { }
    sub new_password : Path("new-password") { }
    sub sign_out : Path("sign-out") { }

    package MyApp::Controller::Catalog;

    use base qw/Catalyst::Controller/;

    sub view : Local { }
    sub list : Local { }

    package MyApp::Controller::Cart;

    use base qw/Catalyst::Controller/;

    sub add : Local { }
    sub update : Local { }
    sub order : Local { }
```

Note that you can also supply attributes via the Controller's config so long as you have at least one attribute on a subref to be exported (:Action is commonly used for this) - for example the following is equivalent to the same controller above:

```
    package MyApp::Controller::Login;

    use base qw/Catalyst::Controller/;
    __PACKAGE__->config(
      actions => {
        'sign_in' => { Path => 'sign-in' },
        'new_password' => { Path => 'new-password' },
        'sign_out' => { Path => 'sign-out' },
      },
    );

    sub sign_in : Action { }
    sub new_password : Action { }
    sub sign_out : Action { }
```

ACCEPT_CONTEXT

Whenever you call $c->component("Foo") you get back an object - the instance of the model. If the component supports the `ACCEPT_CONTEXT` method instead of returning the model itself, the return value of `$model->ACCEPT_CONTEXT( $c )` will be used.

This means that whenever your model/view/controller needs to talk to `$c` it gets a chance to do this when it's needed.

A typical `ACCEPT_CONTEXT` method will either clone the model and return one with the context object set, or it will return a thin wrapper that contains `$c` and delegates to the per-application model object.

Generally it's a bad idea to expose the context object (`$c`) in your model or view code. Instead you use the `ACCEPT_CONTEXT` subroutine to grab the bits of the context object that you need, and provide accessors to them in the model. This ensures that `$c` is only in scope where it is needed which reduces maintenance and debugging headaches. So, if for example you needed two Catalyst::Model::DBIC::Schema models in the same Catalyst model code, you might do something like this:

```
  __PACKAGE__->mk_accessors(qw(model1_schema model2_schema));
  sub ACCEPT_CONTEXT {
      my ( $self, $c, @extra_arguments ) = @_;
      $self = bless({ %$self,
              model1_schema  => $c->model('Model1')->schema,
              model2_schema => $c->model('Model2')->schema
          }, ref($self));
      return $self;
  }
```

This effectively treats $self as a **prototype object** that gets a new parameter. `@extra_arguments` comes from any trailing arguments to `$c->component( $bah, @extra_arguments )` (or `$c->model(...)`, `$c->view(...)` etc).

In a subroutine in the model code, we can then do this:

```
  sub whatever {
      my ($self) = @_;
      my $schema1 = $self->model1_schema;
      my $schema2 = $self->model2_schema;
      ...
  }
```

Note that we still want the Catalyst models to be a thin wrapper around classes that will work independently of the Catalyst application to promote reusability of code. Here we might just want to grab the $c->model('DB')->schema so as to get the connection information from the Catalyst application's configuration for example.

The life time of this value is **per usage**, and not per request. To make this per request you can use the following technique:

Add a field to `$c`, like `my_model_instance`. Then write your `ACCEPT_CONTEXT` method to look like this:

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

      if ( my $per_request = $c->my_model_instance ) {
        return $per_request;
      } else {
        my $new_instance = bless { %$self, c => $c }, ref($self);
        Scalar::Util::weaken($new_instance->{c}); # or we have a circular reference
        $c->my_model_instance( $new_instance );
        return $new_instance;
      }
    }
```

For a similar technique to grab a new component instance on each request, see Catalyst::Component::InstancePerContext.

Application Class

In addition to the Model, View, and Controller components, there's a single class that represents your application itself. This is where you configure your application, load plugins, and extend Catalyst.

```
    package MyApp;

    use strict;
    use parent qw/Catalyst/;
    use Catalyst qw/-Debug ConfigLoader Static::Simple/;
    MyApp->config(
        name => 'My Application',

        # You can put anything else you want in here:
        my_configuration_variable => 'something',
    );
    1;
```

In older versions of Catalyst, the application class was where you put global actions. However, as of version 5.66, the recommended practice is to place such actions in a special Root controller (see "Actions", below), to avoid namespace collisions.

- **name**

  The name of your application.

Optionally, you can specify a **root** parameter for templates and static data. If omitted, Catalyst will try to auto-detect the directory's location. You can define as many parameters as you want for plugins or whatever you need. You can access them anywhere in your application via `$context->config->{$param_name}`.

Context

Catalyst automatically blesses a Context object into your application class and makes it available everywhere in your application. Use the Context to directly interact with Catalyst and glue your "Components" together. For example, if you need to use the Context from within a Template Toolkit template, it's already there:

```
    <h1>Welcome to [% c.config.name %]!</h1>
```

As illustrated in our URL-to-Action dispatching example, the Context is always the second method parameter, behind the Component object reference or class name itself. Previously we called it `$context` for clarity, but most Catalyst developers just call it `$c`:

```
    sub hello : Global {
        my ( $self, $c ) = @_;
        $c->res->body('Hello World!');
    }
```

The Context contains several important objects:

- Catalyst::Request

  ```
      $c->request
      $c->req # alias
  ```

  The request object contains all kinds of request-specific information, like query parameters, cookies, uploads, headers, and more.

  ```
      $c->req->params->{foo};
      $c->req->cookies->{sessionid};
      $c->req->headers->content_type;
      $c->req->base;
      $c->req->uri_with( { page = $pager->next_page } );
  ```

- Catalyst::Response

  ```
      $c->response
      $c->res # alias
  ```

  The response is like the request, but contains just response-specific information.

  ```
      $c->res->body('Hello World');
      $c->res->status(404);
      $c->res->redirect('http://oook.de');
  ```

- config

  ```
      $c->config
      $c->config->{root};
      $c->config->{name};
  ```

- Catalyst::Log

```
    $c->log
    $c->log->debug('Something happened');
    $c->log->info('Something you should know');
```

- **Stash**

```
    $c->stash
    $c->stash->{foo} = 'bar';
    $c->stash->{baz} = {baz => 'qox'};
    $c->stash->{fred} = [qw/wilma pebbles/];
```

  and so on.

The last of these, the stash, is a universal hash for sharing data among application components. For an example, we return to our 'hello' action:

```
    sub hello : Global {
        my ( $self, $c ) = @_;
        $c->stash->{message} = 'Hello World!';
        $c->forward('show_message');
    }

    sub show_message : Private {
        my ( $self, $c ) = @_;
        $c->res->body( $c->stash->{message} );
    }
```

Note that the stash should be used only for passing data in an individual request cycle; it gets cleared at a new request. If you need to maintain persistent data, use a session. See Catalyst::Plugin::Session for a comprehensive set of Catalyst-friendly session-handling tools.

Actions

You've already seen some examples of actions in this document: subroutines with `:Path` and `:Local` attributes attached. Here, we explain what actions are and how these attributes affect what's happening.

When Catalyst processes a webpage request, it looks for actions to take that will deal with the incoming request and produce a response such as a webpage. You create these actions for your application by writing subroutines within your controller and marking them with special attributes. The attributes, the namespace, and the function name determine when Catalyst will call the subroutine.

These action subroutines call certain functions to say what response the webserver will give to the web request. They can also tell Catalyst to run other actions on the request (one example of this is called forwarding the request; this is discussed later).

Action subroutines must have a special attribute on to show that they are actions - as well as marking when to call them, this shows that they take a specific set of arguments and behave in a specific way. At startup, Catalyst looks for all the actions in controllers, registers them and creates Catalyst::Action objects describing them. When requests come in, Catalyst chooses which actions should be called to handle the request.

(Occasionally, you might use the action objects directly, but in general, when we talk about actions, we're talking about the subroutines in your application that do things to process a request.)

You can choose one of several attributes for action subroutines; these specify which requests are processed by that subroutine. Catalyst will look at the URL it is processing, and the actions that it has found, and automatically call the actions it finds that match the circumstances of the request.

The URL (for example `http://localhost:3000/foo/bar`) consists of two parts, the base, describing how to connect to the server (`http://localhost:3000/` in this example) and the path, which the server uses to decide what to return (`foo/bar`). Please note that the trailing slash after the hostname[:port] always belongs to base and not to the path. Catalyst uses only the path part when trying to find actions to process.

Depending on the type of action used, the URLs may match a combination of the controller namespace, the arguments passed to the action attribute, and the name of the subroutine.

- **Controller namespaces**

  The namespace is a modified form of the component's class (package) name. This modified class name excludes the parts that have a pre-defined meaning in Catalyst ("MyApp::Controller" in the above example), replaces "::" with "/", and converts the name to lower case. See "Components" for a full explanation of the pre-defined meaning of Catalyst component class names.

- **Overriding the namespace**

  Note that `__PACKAGE__->config->(namespace => ... )` can be used to override the current namespace when matching. So:

  ```
      package MyApp::Controller::Example;
  ```

  would normally use 'example' as its namespace for matching, but if this is specially overridden with

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

  it matches using the namespace 'thing' instead.

- **Application-Wide Actions**

  MyApp::Controller::Root, as created by the catalyst.pl script, will typically contain actions which are called for the top level of the application (e.g. `http://localhost:3000/`):

  ```
      package MyApp::Controller::Root;
      use base 'Catalyst::Controller';

      # Sets the actions in this controller to be registered with no prefix
      # so they function identically to actions created in MyApp.pm

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

      sub default : Path  {
          my ( $self, $context ) = @_;
          $context->response->status(404);
          $context->response->body('404 not found');
      }

      1;
  ```

  The code

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

makes the controller act as if its namespace is empty. As you'll see below, an empty namespace makes many of the URL-matching attributes, such as :Path and :Local match at the start of the URL path (i.e. the application root).

Action types

Catalyst supports several types of actions. These mainly correspond to ways of matching a URL to an action subroutine. Internally, these matching types are implemented by Catalyst::DispatchType-derived classes; the documentation there can be helpful in seeing how they work.

They will all attempt to match the start of the path. The remainder of the path is passed as arguments.

- Namespace-prefixed (`:Local`)

  ```
  package MyApp::Controller::My::Controller;
  sub foo : Local { }
  ```

  Matches any URL beginning with> `http://localhost:3000/my/controller/foo`. The namespace and subroutine name together determine the path.

- Root-level (`:Global`)

  ```
  package MyApp::Controller::Foo;

  sub bar : Global {
      my ($self, $c) = @_;
      $c->res->body(
        $c->res->body('sub bar in Controller::Foo triggered on a request for '
                       . $c->req->uri));
  }
  ```

  1;

  Matches `http://localhost:3000/bar` - that is, the action is mapped directly to the method name, ignoring the controller namespace.

  `:Global` always matches from the application root: it is simply shorthand for `:Path('/methodname')`. `:Local` is shorthand for `:Path('methodname')`, which takes the controller namespace as described above.

  Usage of the `Global` handler is rare in all but very old Catalyst applications (e.g. before Catalyst 5.7). The use cases where `Global` used to make sense are now largely replaced by the `Chained` dispatch type, or by empty `Path` declarations on an controller action. `Global` is still included in Catalyst for backwards compatibility, although legitimate use-cases for it may still exist.

- Changing handler behaviour: eating arguments (`:Args`)

  `:Args` is not an action type per se, but an action modifier - it adds a match restriction to any action it's provided to, additionally requiring as many path parts as are specified for the action to be matched. For example, in MyApp::Controller::Foo,

```
sub bar :Local
```

would match any URL starting /foo/bar. To restrict this you can do

```
sub bar :Local :Args(1)
```

to only match URLs starting /foo/bar/* - with one additional path element required after 'bar'.

NOTE that adding `:Args(0)` and omitting `:Args` are **not** the same thing.

`:Args(0)` means that no arguments are taken. Thus, the URL and path must match precisely.

No `:Args` at all means that **any number** of arguments are taken. Thus, any URL that **starts with** the controller's path will match. Obviously, this means you cannot chain from an action that does not specify args, as the next action in the chain will be swallowed as an arg to the first!

- Literal match (`:Path`)

  `Path` actions match things starting with a precise specified path, and nothing else.

  `Path` actions without a leading forward slash match a specified path relative to their current namespace. This example matches URLs starting with `http://localhost:3000/my/controller/foo/bar`:

```
    package MyApp::Controller::My::Controller;
    sub bar : Path('foo/bar') { }
```

  `Path` actions **with** a leading slash ignore their namespace, and match from the start of the URL path. Example:

```
    package MyApp::Controller::My::Controller;
    sub bar : Path('/foo/bar') { }
```

  This matches URLs beginning with `http://localhost:3000/foo/bar`.

  Empty `Path` definitions match on the namespace only, exactly like `:Global`.

```
    package MyApp::Controller::My::Controller;
    sub bar : Path { }
```

  The above code matches `http://localhost:3000/my/controller`.

  Actions with the `:Local` attribute are similarly equivalent to `:Path('action_name')`:

```
    sub foo : Local { }
```

  is equivalent to

```
    sub foo : Path('foo') { }
```

- Pattern match (`:Regex` and `:LocalRegex`)

  Status: deprecated. Use Chained methods or other techniques. If you really depend on this, install the standalone Catalyst::DispatchType::Regex distribution.

  ```
      package MyApp::Controller::My::Controller;
      sub bar : Regex('^item(\d+)/order(\d+)$') { }
  ```

  This matches any URL that matches the pattern in the action key, e.g. `http://localhost:3000/item23/order42`. The '' around the regexp is optional, but perltidy likes it. :)

  `:Regex` matches act globally, i.e. without reference to the namespace from which they are called. So the above will **not** match `http://localhost:3000/my/controller/item23/order42` - use a `:LocalRegex` action instead.

  ```
      package MyApp::Controller::My::Controller;
      sub bar : LocalRegex('^widget(\d+)$') { }
  ```

  `:LocalRegex` actions act locally, i.e. the namespace is matched first. The above example would match urls like `http://localhost:3000/my/controller/widget23`.

  If you omit the "`^`" from either sort of regex, then it will match any depth from the base path:

  ```
      package MyApp::Controller::Catalog;
      sub bar : LocalRegex('widget(\d+)$') { }
  ```

  This differs from the previous example in that it will match `http://localhost:3000/my/controller/foo/widget23` - and a number of other paths.

  For both `:LocalRegex` and `:Regex` actions, if you use capturing parentheses to extract values within the matching URL, those values are available in the `$c->req->captures` array. In the above example, "widget23" would capture "23" in the above example, and `$c->req->captures->[0]` would be "23". If you want to pass arguments at the end of your URL, you must use regex action keys. See "URL Path Handling" below.

- Chained handlers (`:Chained`)

  Catalyst also provides a method to build and dispatch chains of actions, like

  ```
      sub catalog : Chained : CaptureArgs(1) {
          my ( $self, $c, $arg ) = @_;
          ...
      }

      sub item : Chained('catalog') : Args(1) {
          my ( $self, $c, $arg ) = @_;
          ...
      }
  ```

  to handle a `/catalog/*/item/*` path. Matching actions are called one after another - `catalog()` gets called and handed one path element, then `item()` gets called with another one. For further information about this dispatch type, please see Catalyst::DispatchType::Chained.

- **Private**

  ```
      sub foo : Private { }
  ```

  This will never match a URL - it provides a private action which can be called programmatically from within Catalyst, but is never called automatically due to the URL being requested.

  Catalyst's `:Private` attribute is exclusive and doesn't work with other attributes (so will not work combined with `:Path` or `:Chained` attributes, for instance).

  Private actions can only be executed explicitly from inside a Catalyst application. You might do this in your controllers by calling catalyst methods such as `forward` or `detach` to fire them:

  ```
      $c->forward('foo');
      # or
      $c->detach('foo');
  ```

  See "Flow Control" for a full explanation of how you can pass requests on to other actions. Note that, as discussed there, when forwarding from another component, you must use the absolute path to the method, so that a private `bar` method in your `MyApp::Controller::Catalog::Order::Process` controller must, if called from elsewhere, be reached with `$c->forward('/catalog/order/process/bar')`.

Note: After seeing these examples, you probably wonder what the point is of defining subroutine names for regex and path actions. However, every public action is also a private one with a path corresponding to its namespace and subroutine name, so you have one unified way of addressing components in your `forward`s.

Built-in special actions

If present, the special actions `index` , `auto` , `begin`, `end` and `default` are called at certain points in the request cycle.

In response to specific application states, Catalyst will automatically call these built-in actions in your application class:

- **default : Path**

  This is called when no other action matches. It could be used, for example, for displaying a generic frontpage for the main app, or an error page for individual controllers. **Note**: in older Catalyst applications you will see `default : Private` which is roughly speaking equivalent.

- **index : Path : Args (0)**

  `index` is much like `default` except that it takes no arguments and it is weighted slightly higher in the matching process. It is useful as a static entry point to a controller, e.g. to have a static welcome page. Note that it's also weighted higher than Path. Actually the sub name `index` can be called anything you want. The sub attributes are what determines the behaviour of the action. **Note**: in older Catalyst applications, you will see `index : Private` used, which is roughly speaking equivalent.

- **begin : Private**

Called at the beginning of a request, once the controller that will run has been identified, but before any URL-matching actions are called. Catalyst will call the `begin` function in the controller which contains the action matching the URL.

- **end : Private**

  Called at the end of a request, after all URL-matching actions are called. Catalyst will call the `end` function in the controller which contains the action matching the URL.

- **auto : Private**

  In addition to the normal built-in actions, you have a special action for making chains, `auto`. `auto` actions will be run after any `begin`, but before your URL-matching action is processed. Unlike the other built-ins, multiple `auto` actions can be called; they will be called in turn, starting with the application class and going through to the most specific class.

Built-in actions in controllers/autochaining

```
    package MyApp::Controller::Foo;
    sub begin : Private { }
    sub default : Path  { }
    sub end : Path  { }
```

You can define built-in actions within your controllers as well as on your application class. In other words, for each of the three built-in actions above, only one will be run in any request cycle. Thus, if `MyApp::Controller::Catalog::begin` exists, it will be run in place of `MyApp::begin` if you're in the `catalog` namespace, and `MyApp::Controller::Catalog::Order::begin` would override this in turn.

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

`auto`, however, doesn't override like this: providing they exist, `MyApp::Controller::Root::auto`, `MyApp::Controller::Catalog::auto` and `MyApp::Catalog::Order::auto` would be called in turn.

Here are some examples of the order in which the various built-ins would be called:

for a request for `/foo/foo`

```
    MyApp::Controller::Foo::auto
    MyApp::Controller::Foo::default # in the absence of MyApp::Controller::Foo::Foo
    MyApp::Controller::Foo::end
```

for a request for `/foo/bar/foo`

```
    MyApp::Controller::Foo::Bar::begin
    MyApp::Controller::Foo::auto
    MyApp::Controller::Foo::Bar::auto
    MyApp::Controller::Foo::Bar::default # for MyApp::Controller::Foo::Bar::foo
    MyApp::Controller::Foo::Bar::end
```

The `auto` action is also distinguished by the fact that you can break out of the processing chain by returning 0. If an `auto` action returns 0, any remaining actions will be skipped, except for `end`. So, for the request above, if the first auto returns false, the chain would look like this:

for a request for `/foo/bar/foo` where first `auto` returns false

```
MyApp::Controller::Foo::Bar::begin
MyApp::Controller::Foo::auto # returns false, skips some calls:
# MyApp::Controller::Foo::Bar::auto - never called
# MyApp::Controller::Foo::Bar::foo - never called
MyApp::Controller::Foo::Bar::end
```

You can also `die` in the auto action; in that case, the request will go straight to the finalize stage, without processing further actions. So in the above example, `MyApp::Controller::Foo::Bar::end` is skipped as well.

An example of why one might use `auto` is an authentication action: you could set up a `auto` action to handle authentication in your application class (which will always be called first), and if authentication fails, returning 0 would skip any remaining methods for that URL.

Note: Looking at it another way, `auto` actions have to return a true value to continue processing!

URL Path Handling

You can pass arguments as part of the URL path, separated with forward slashes (/). If the action is a Regex or LocalRegex, the '$' anchor must be used. For example, suppose you want to handle `/foo/$bar/$baz`, where `$bar` and `$baz` may vary:

```
sub foo : Regex('^foo$') { my ($self, $context, $bar, $baz) = @_; }
```

But what if you also defined actions for `/foo/boo` and `/foo/boo/hoo`?

```
sub boo : Path('foo/boo') { .. }
sub hoo : Path('foo/boo/hoo') { .. }
```

Catalyst matches actions in most specific to least specific order - that is, whatever matches the most pieces of the path wins:

```
/foo/boo/hoo
/foo/boo
/foo # might be /foo/bar/baz but won't be /foo/boo/hoo
```

So Catalyst would never mistakenly dispatch the first two URLs to the '^foo$' action.

If a Regex or LocalRegex action doesn't use the '$' anchor, the action will still match a URL containing arguments; however the arguments won't be available via `@_`, because the Regex will 'eat' them.

Beware! If you write two matchers, that match the same path, with the same specificity (that is, they match the same quantity of the path), there's no guarantee which will actually get called. Non-regex matchers get tried first, followed by regex ones, but if you have, for instance:

```
package MyApp::Controller::Root;

sub match1 :Path('/a/b') { }

package MyApp::Controller::A;
```

```
    sub b :Local { } # Matches /a/b
```

then Catalyst will call the one it finds first. In summary, Don't Do This.

Query Parameter Processing

Parameters passed in the URL query string are handled with methods in the Catalyst::Request class. The `param` method is functionally equivalent to the `param` method of `CGI.pm` and can be used in modules that require this.

```
    # http://localhost:3000/catalog/view/?category=hardware&page=3
    my $category = $c->req->param('category');
    my $current_page = $c->req->param('page') || 1;

    # multiple values for single parameter name
    my @values = $c->req->param('scrolling_list');

    # DFV requires a CGI.pm-like input hash
    my $results = Data::FormValidator->check($c->req->params, \%dfv_profile);
```

Flow Control

You control the application flow with the `forward` method, which accepts the key of an action to execute. This can be an action in the same or another Catalyst controller, or a Class name, optionally followed by a method name. After a `forward`, the control flow will return to the method from which the `forward` was issued.

A `forward` is similar to a method call. The main differences are that it wraps the call in an `eval` to allow exception handling; it automatically passes along the context object (`$c` or `$context`); and it allows profiling of each call (displayed in the log with debugging enabled).

```
    sub hello : Global {
        my ( $self, $c ) = @_;
        $c->stash->{message} = 'Hello World!';
        $c->forward('check_message'); # $c is automatically included
    }

    sub check_message : Private {
        my ( $self, $c ) = @_;
        return unless $c->stash->{message};
        $c->forward('show_message');
    }

    sub show_message : Private {
        my ( $self, $c ) = @_;
        $c->res->body( $c->stash->{message} );
    }
```

A `forward` does not create a new request, so your request object (`$c->req`) will remain unchanged. This is a key difference between using `forward` and issuing a redirect.

You can pass new arguments to a `forward` by adding them in an anonymous array. In this case `$c->req->args` will be changed for the duration of the `forward` only; upon return, the original value of `$c->req->args` will be reset.

```
    sub hello : Global {
        my ( $self, $c ) = @_;
        $c->stash->{message} = 'Hello World!';
        $c->forward('check_message',[qw/test1/]);
        # now $c->req->args is back to what it was before
    }

    sub check_message : Action {
        my ( $self, $c, $first_argument ) = @_;
        my $also_first_argument = $c->req->args->[0]; # now = 'test1'
        # do something...
    }
```

As you can see from these examples, you can just use the method name as long as you are referring to methods in the same controller. If you want to forward to a method in another controller, or the main application, you will have to refer to the method by absolute path.

```
  $c->forward('/my/controller/action');
  $c->forward('/default'); # calls default in main application
```

You can also forward to classes and methods.

```
    sub hello : Global {
        my ( $self, $c ) = @_;
        $c->forward(qw/MyApp::View:Hello say_hello/);
    }

    sub bye : Global {
        my ( $self, $c ) = @_;
        $c->forward('MyApp::Model::Hello'); # no method: will try 'process'
    }

    package MyApp::View::Hello;

    sub say_hello {
        my ( $self, $c ) = @_;
        $c->res->body('Hello World!');
    }

    sub process {
        my ( $self, $c ) = @_;
        $c->res->body('Goodbye World!');
    }
```

This mechanism is used by Catalyst::Action::RenderView to forward to the `process` method in a view class.

It should be noted that whilst forward is useful, it is not the only way of calling other code in Catalyst. Forward just gives you stats in the debug screen, wraps the code you're calling in an exception handler and localises `$c->request->args`.

If you don't want or need these features then it's perfectly acceptable (and faster) to do something like this:

```
    sub hello : Global {
        my ( $self, $c ) = @_;
        $c->stash->{message} = 'Hello World!';
        $self->check_message( $c, 'test1' );
    }

    sub check_message {
        my ( $self, $c, $first_argument ) = @_;
        # do something...
    }
```

Note that `forward` returns to the calling action and continues processing after the action finishes. If you want all further processing in the calling action to stop, use `detach` instead, which will execute the `detach`ed action and not return to the calling sub. In both cases, Catalyst will automatically try to call process() if you omit the method.

Testing

Catalyst has a built-in http server for testing or local deployment. (Later, you can easily use a more powerful server, for example Apache/mod_perl or FastCGI, in a production environment.)

Start your application on the command line...

```
    script/myapp_server.pl
```

...then visit http://localhost:3000/ in a browser to view the output.

You can also do it all from the command line:

```
    script/myapp_test.pl http://localhost/
```

Catalyst has a number of tools for actual regression testing of applications. The helper scripts will automatically generate basic tests that can be extended as you develop your project. To write your own comprehensive test scripts, Test::WWW::Mechanize::Catalyst is an invaluable tool.

For more testing ideas, see Catalyst::Manual::Tutorial::08_Testing.

Have fun!

SEE ALSO

- Catalyst::Manual::About
- Catalyst::Manual::Tutorial
- Catalyst

SUPPORT

IRC:

```
    Join #catalyst on irc.perl.org.
    Join #catalyst-dev on irc.perl.org to help with development.
```

Mailing lists:

```
    http://lists.scsys.co.uk/mailman/listinfo/catalyst
    http://lists.scsys.co.uk/mailman/listinfo/catalyst-dev
```

Wiki:

```
    http://dev.catalystframework.org/wiki
```

FAQ:

```
    http://dev.catalystframework.org/wiki/faq
```

AUTHORS

Catalyst Contributors, see Catalyst.pm

COPYRIGHT

syntax highlighting: no syntax highlighting ▼