- NAME
- DESCRIPTION
- BEST PRACTICES
  - Quick Checklist
  - Namespaces
  - Can it be a simple module?
  - Using Moose roles to apply method modifiers
  - Inheritance and overriding methods
  - Tests and documentation
  - Maintenance
  - The context object
- CONFIGURATION
  - Attributes
  - Component Configuration
- IMPLEMENTATION
  - Action classes
  - Component base classes
  - Controllers
  - Models and Views
  - Plugins
  - Factory components with COMPONENT()
  - Applying roles to parts of the framework
- SEE ALSO
- AUTHORS
- COPYRIGHT

NAME

Catalyst::Manual::ExtendingCatalyst - Extending The Framework

DESCRIPTION

This document will provide you with access points, techniques and best practices to extend the Catalyst framework, or to find more elegant ways to abstract and use your own code.

The design of Catalyst is such that the framework itself should not get in your way. There are many entry points to alter or extend Catalyst's behaviour, and this can be confusing. This document is written to help you understand the possibilities, current practices and their consequences.

Please read the "BEST PRACTICES" section before deciding on a design, especially if you plan to release your code to CPAN. The Catalyst developer and user communities, which **you are part of**, will benefit most if we all work together and coordinate.

If you are unsure on an implementation or have an idea you would like to have RFC'ed, it surely is a good idea to send your questions and suggestions to the Catalyst mailing list (See "SUPPORT" in Catalyst) and/or come to the `#catalyst` channel on the `irc.perl.org` network. You might also want to refer to those places for research to see if a module doing what you're trying to implement already exists. This might give you a solution to your problem or a basis for starting.

BEST PRACTICES

During Catalyst's early days, it was common to write plugins to provide functionality application wide. Since then, Catalyst has become a lot more flexible and powerful. It soon became a best practice to use some other form of abstraction or interface, to keep the scope of its influence as close as possible to where it belongs.

For those in a hurry, here's a quick checklist of some fundamental points. If you are going to read the whole thing anyway, you can jump forward to "Namespaces".

Quick Checklist

Use the `CatalystX::*` namespace if you can!

> If your extension isn't a Model, View, Controller, Plugin, Engine, or Log, it's best to leave it out of the `Catalyst::` namespace. Use <CatalystX::> instead.

Don't make it a plugin unless you have to!

> A plugin should be careful since it's overriding Catalyst internals. If your plugin doesn't really need to muck with the internals, make it a base Controller or Model.
>
> Also, if you think you really need a plugin, please instead consider using a Moose::Role.

There's a community. Use it!

> There are many experienced developers in the Catalyst community, there's always the IRC channel and the mailing list to discuss things.

Add tests and documentation!

> This gives a stable basis for contribution, and even more importantly, builds trust. The easiest way is a test application. See Catalyst::Manual::Tutorial::Testing for more information.

Namespaces

While some core extensions (engines, plugins, etc.) have to be placed in the `Catalyst::*` namespace, the Catalyst core would like to ask developers to use the `CatalystX::*` namespace if possible.

Please **do not** invent components which are outside the well known `Model`, `View`, `Controller` or `Plugin` namespaces!

When you try to put a base class for a `Model`, `View` or `Controller` directly under your `MyApp` directory as, for example, `MyApp::Controller::Foo`, you will have the problem that Catalyst will try to load that base class as a component of your application. The solution is simple: Use another namespace. Common ones are `MyApp::Base::Controller::*` or `MyApp::ControllerBase::*` as examples.

Can it be a simple module?

Sometimes you want to use functionality in your application that doesn't require the framework at all. Remember that Catalyst is just Perl and you always can just `use` a module. If you have application specific code that doesn't need the framework, there is no problem in putting it in your `MyApp::*` namespace. Just don't put it in `Model`, `Controller` or `View`, because that would make Catalyst try to load them as components.

Writing a generic component that only works with Catalyst is wasteful of your time. Try writing a plain perl module, and then a small bit of glue that integrates it with Catalyst. See Catalyst::Model::DBIC::Schema for a module that takes the approach. The advantage here is that your "Catalyst" DBIC schema works perfectly outside of Catalyst, making testing (and command-line scripts) a breeze. The actual Catalyst Model is just a few lines of glue that makes working with the schema convenient.

If you want the thinnest interface possible, take a look at Catalyst::Model::Adaptor.

Using Moose roles to apply method modifiers

Rather than having a complex set of base classes which you have to mixin via multiple inheritance, if your functionality is well structured, then it's possible to use the composability of Moose roles, and method modifiers to hook onto to provide functionality.

These can be applied to your models/views/controllers, and your application class, and shipped to CPAN. Please see Catalyst::Manual::CatalystAndMoose for specific information about using Roles in combination with Catalyst, and Moose::Manual::Roles for more information about roles in general.

Inheritance and overriding methods

When overriding a method, keep in mind that some day additional arguments may be provided to the method, if the last parameter is not a flat list. It is thus better to override a method by shifting the invocant off of `@_` and assign the rest of the used arguments, so you can pass your complete arguments to the original method via `@_`:

```
use MRO::Compat; ...

sub foo {
  my $self = shift;
  my ($bar, $baz) = @_; # ...  return
  $self->next::method(@_);
}
```

If you would do the common

```
my ($self, $foo, $bar) = @_;
```

you'd have to use a much uglier construct to ensure that all arguments will be passed along and the method is future proof:

```
$self->next::method(@_[ 1 .. $#_ ]);
```

Tests and documentation

When you release your module to the CPAN, proper documentation and at least a basic test suite (which means more than pod or even just `use_ok`, sorry) gives people a good base to contribute to the module. It also shows that you care for your users. If you would like your module to become a recommended addition, these things will prove invaluable.

If you're just getting started, try using CatalystX::Starter to generate some example tests for your module.

Maintenance

In planning to release a module to the community (Catalyst or CPAN and Perl), you should consider if you have the resources to keep it up to date, including fixing bugs and accepting contributions.

If you're not sure about this, you can always ask in the proper Catalyst or Perl channels if someone else might be interested in the project, and would jump in as co-maintainer.

A public repository can further ease interaction with the community. Even read only access enables people to provide you with patches to your current development version. subversion, SVN and SVK, are broadly preferred in the Catalyst community.

If you're developing a Catalyst extension, please consider asking the core team for space in Catalyst's own subversion repository. You can get in touch about this via IRC or the Catalyst developers mailing list.

The context object

Sometimes you want to get a hold of the context object in a component that was created on startup time, where no context existed yet. Often this is about the model reading something out of the stash or other context information (current language, for example).

If you use the context object in your component you have tied it to an existing request. This means that you might get into problems when you try to use the component (e.g. the model - the most common case) outside of Catalyst, for example in cronjobs.

A stable solution to this problem is to design the Catalyst model separately from the underlying model logic. Let's take Catalyst::Model::DBIC::Schema as an example. You can create a schema outside of Catalyst that knows nothing about the web. This kind of design ensures encapsulation and makes development and maintenance a whole lot easier. The you use the aforementioned model to tie your schema to your application. This gives you a `MyApp::DBIC` (the name is of course just an example) model as well as `MyApp::DBIC::TableName` models to access your result sources directly.

By creating such a thin layer between the actual model and the Catalyst application, the schema itself is not at all tied to any application and the layer in-between can access the model's API using information from the context object.

A Catalyst component accesses the context object at request time with "ACCEPT_CONTEXT($c, @args)" in Catalyst::Component.

CONFIGURATION

The application has to interact with the extension with some configuration. There is of course again more than one way to do it.

Attributes

You can specify any valid Perl attribute on Catalyst actions you like. (See "Syntax of Attribute Lists" in attributes for a description of what is valid.) These will be available on the `Catalyst::Action` instance via its `attributes` accessor. To give an example, this action:

```
  sub foo : Local Bar('Baz') {
      my ($self, $c) = @_;
      my $attributes = $self->action_for('foo')->attributes;
      $c->res->body($attributes->{Bar}[0] );
  }
```

will set the response body to `Baz`. The values always come in an array reference. As you can see, you can use attributes to configure your actions. You can specify or alter these attributes via "Component Configuration", or even react on them as soon as Catalyst encounters them by providing your own component base class.

Component Configuration

At creation time, the class configuration of your component (the one available via `$self->config`) will be merged with possible configuration settings from the applications configuration (either directly or via config file). This is done by Catalyst, and the correctly merged configuration is passed to your component's constructor (i.e. the new method).

Ergo, if you define an accessor for each configuration value that your component takes, then the value will be automatically stored in the controller object's hash reference, and available from the accessor.

The `config` accessor always only contains the original class configuration and you **MUST NEVER** call $self->config to get your component configuration, as the data there is likely to be a subset of the correct config.

For example:

```
package MyApp
use Moose;

extends 'Catalyst';

...

__PACKAGE__->config(
  'Controller::Foo' => { some_value => 'bar' },
);

...

package MyApp::Controller::Foo;
use Moose;
use namespace::autoclean;
BEGIN { extends 'Catalyst::Controller' };

has some_value ( is => 'ro', required => 1 );

sub some_method {
    my $self = shift;
    return "the value of 'some_value' is " . $self->some_value;
}

...

my $controller = $c->controller('Foo');
warn $controller->some_value;
warn $controller->some_method;
```

IMPLEMENTATION

This part contains the technical details of various implementation methods. Please read the "BEST PRACTICES" before you start your implementation, if you haven't already.

Action classes

Usually, your action objects are of the class Catalyst::Action. You can override this with the `ActionClass` attribute to influence execution and/or dispatching of the action. A widely used example of this is Catalyst::Action::RenderView, which is used in every newly created Catalyst application in your root controller:

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

Usually, you want to override the `execute` and/or the `match` method. The execute method of the action will naturally call the methods code. You can surround this by overriding the method in a subclass:

```
package Catalyst::Action::MyFoo;
use Moose;
use namespace::autoclean;
use MRO::Compat;
extends 'Catalyst::Action';

sub execute {
    my $self = shift;
    my ($controller, $c, @args) = @_;
    # put your 'before' code here
    my $r = $self->next::method(@_);
    # put your 'after' code here
    return $r;
}
1;
```

We are using MRO::Compat to ensure that you have the next::method call, from Class::C3 (in older perls), or natively (if you are using perl 5.10) to re-dispatch to the original `execute` method in the Catalyst::Action class.

The Catalyst dispatcher handles an incoming request and, depending upon the dispatch type, will call the appropriate target or chain. From time to time it asks the actions themselves, or through the controller, if they would match the current request. That's what the `match` method does. So by overriding this, you can change on what the action will match and add new matching criteria.

For example, the action class below will make the action only match on Mondays:

```
package Catalyst::Action::OnlyMondays;
use Moose;
use namespace::autoclean;
use MRO::Compat;
extends 'Catalyst::Action';

sub match {
    my $self = shift;
    return 0 if ( localtime(time) )[6] == 1;
    return $self->next::method(@_);
 }
1;
```

And this is how we'd use it:

```
sub foo: Local ActionClass('OnlyMondays') {
    my ($self, $c) = @_;
    $c->res->body('I feel motivated!');
}
```

If you are using action classes often or have some specific base classes that you want to specify more conveniently, you can implement a component base class providing an attribute handler.

It is not possible to use multiple action classes at once, however Catalyst::Controller::ActionRole allows you to apply Moose Roles to actions.

For further information on action classes and roles, please refer to Catalyst::Action and Catalyst::Manual::Actions.

Component base classes

Many Catalyst::Plugin that were written in Catalyst's early days should really have been just controller base classes. With such a class, you could provide functionality scoped to a single controller, not polluting the global namespace in the context object.

You can provide regular Perl methods in a base class as well as actions which will be inherited to the subclass. Please refer to "Controllers" for an example of this.

You can introduce your own attributes by specifying a handler method in the controller base. For example, to use a `FullClass` attribute to specify a fully qualified action class name, you could use the following implementation. Note, however, that this functionality is already provided via the `+` prefix for action classes. A simple

```
sub foo : Local ActionClass('+MyApp::Action::Bar') { ... }
```

will use `MyApp::Action::Bar` as action class.

```
package MyApp::Base::Controller::FullClass;
use Moose;
use namespace::autoclean;
BEGIN { extends 'Catalyst::Controller'; }

sub _parse_FullClass_attr {
    my ($self, $app_class, $action_name, $value, $attrs) = @_;
    return( ActionClass => $value );
}
1;
```

Note that the full line of arguments is only provided for completeness sake. We could use this attribute in a subclass like any other Catalyst attribute:

```
package MyApp::Controller::Foo;
use Moose;
use namespace::autoclean;
BEGIN { extends 'MyApp::Base::Controller::FullClass'; }

sub foo : Local FullClass('MyApp::Action::Bar') { ... }

1;
```

Controllers

Many things can happen in controllers, and it often improves maintainability to abstract some of the code out into reusable base classes.

You can provide usual Perl methods that will be available via your controller object, or you can even define Catalyst actions which will be inherited by the subclasses. Consider this controller base class:

```
package MyApp::Base::Controller::ModelBase;
use Moose;
use namespace::autoclean;

BEGIN { extends 'Catalyst::Controller'; }

sub list : Chained('base') PathPart('') Args(0) {
    my ($self, $c) = @_;
    my $model = $c->model( $self->{model_name} );
    my $condition = $self->{model_search_condition} || {};
    my $attrs = $self->{model_search_attrs} || {};
    $c->stash(rs => $model->search($condition, $attrs);
}

sub load : Chained('base') PathPart('') CaptureArgs(1) {
    my ($self, $c, $id) = @_;
    my $model = $c->model( $self->{model_name} );
    $c->stash(row => $model->find($id));
}
1;
```

This example implements two simple actions. The `list` action chains to a (currently non-existent) `base` action and puts a result-set into the stash taking a configured `model_name` as well as a search condition and attributes. This action is a chained endpoint. The other action, called `load` is a chain midpoint that takes one argument. It takes the value as an ID and loads the row from the configured model. Please not that the above code is simplified for clarity. It misses error handling, input validation, and probably other things.

The class above is not very useful on its own, but we can combine it with some custom actions by sub-classing it:

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

BEGIN { extends 'MyApp::Base::Controller::ModelBase'; }

__PACKAGE__->config( model_name => 'DB::Foo',
                     model_search_condition=> { is_active => 1 },
                     model_search_attrs => { order_by => 'name' },
                 );

sub base : Chained PathPart('foo') CaptureArgs(0) { }

sub view : Chained('load') Args(0) {
    my ($self, $c) = @_;
    my $row = $c->stash->{row};
    $c->res->body(join ': ', $row->name,
    $row->description); }
1;
```

This class uses the formerly created controller as a base class. First, we see the configurations that were used in the parent class. Next comes the `base` action, where everything chains off of.

Note that inherited actions act like they were declared in your controller itself. You can therefor call them just by their name in `forward`s, `detaches` and `Chained(..)` specifications. This is an important part of what makes this technique so useful.

The new `view` action ties itself to the `load` action specified in the base class and outputs the loaded row's `name` and `description` columns. The controller `MyApp::Controller::Foo` now has these publicly available paths:

/foo

> Will call the controller's `base`, then the base classes `list` action.

/foo/$id/view

> First, the controller's `base` will be called, then it will `load` the row with the corresponding `$id`. After that, `view` will display some fields out of the object.

Models and Views

If the functionality you'd like to add is really a data-set that you want to manipulate, for example internal document types, images, files, it might be better suited as a model.

The same applies for views. If your code handles representation or deals with the applications interface and should be universally available, it could be a perfect candidate for a view.

Please implement a `process` method in your views. This method will be called by Catalyst if it is asked to forward to a component without a specified action. Note that `process` is **not a Catalyst action** but a simple Perl method.

You are also encouraged to implement a `render` method corresponding with the one in Catalyst::View::TT. This has proven invaluable, because people can use your view for much more fine-grained content generation.

Here is some example code for a fictional view:

```
    package Catalyst::View::MyView;
    use Moose;
    use namespace::autoclean;

    extends 'Catalyst::View';

    sub process {
        my ($self, $c) = @_;
        my $template = $c->stash->{template};
        my $content = $self->render($c, $template, $c->stash);
        $c->res->body( $content );
    }

    sub render {
        my ($self, $c, $template, $args) = @_;
        # prepare content here
        return $content;
    }
    1;
```

Plugins

The first thing to say about plugins is that if you're not sure if your module should be a plugin, it probably shouldn't. It once was common to add features to Catalyst by writing plugins that provide accessors to said functionality. As Catalyst grew more popular, it became obvious that this qualifies as bad practice.

By designing your module as a Catalyst plugin, every method you implement, import or inherit will be available via your applications context object. A plugin pollutes the global namespace, and you should be only doing that when you really need to.

Often, developers design extensions as plugins because they need to get hold of the context object. Either to get at the stash or request/response objects are the widely spread reasons. It is, however, perfectly possible to implement a regular Catalyst component (read: model, view or controller) that receives the current context object via "ACCEPT_CONTEXT($c, @args)" in Catalyst::Component.

When is a plugin suited to your task? Your code needs to be a plugin to act upon or alter specific parts of Catalyst's request lifecycle. If your functionality needs to change some `prepare_*` or `finalize_*` stages, you won't get around a plugin.

Note, if you just want to hook into such a stage, and run code before, or after it, then it is recommended that you use Mooses method modifiers to do this.

Another valid target for a plugin architecture are things that **really** have to be globally available, like sessions or authentication.

Please do not release Catalyst extensions as plugins only to provide some functionality application wide. Design it as a controller base class or another better suited technique with a smaller scope, so that your code only influences those parts of the application where it is needed, and namespace clashes and conflicts are ruled out.

The implementation is pretty easy. Your plugin will be inserted in the application's inheritance list, above Catalyst itself. You can by this alter Catalyst's request lifecycle behaviour. Every method you declare, every import in your package will be available as method on the application and the context object. As an example, let's say you want Catalyst to warn you every time uri_for was called without an action object as the first parameter, for example to test that all your chained uris are generated from actions (a recommended best practice). You could do this with this simple implementation (excuse the lame class name, it's just an example):

```
    package Catalyst::Plugin::UriforUndefWarning;
    use strict;
    use Scalar::Util qw/blessed/;
    use MRO::Compat;

    sub uri_for {
        my $c = shift;
        my $uri = $c->next::method(@_);
        $c->log->warn( 'uri_for with non action: ', join(', ', @_), )
          if (!blessed($_[0]) || !$_[0]->isa('Catalyst::Action'));
        return $uri;
    }
```

```
1;
```

This would override Catalyst's `uri_for` method and emit a `warn` log entry containing the arguments to uri_for.

Please note this is not a practical example, as string URLs are fine for static content etc.

A simple example like this is actually better as a Moose role, for example:

```
package CatalystX::UriforUndefWarning;
use Moose::Role;
use namespace::autoclean;

after 'uri_for' => sub {
  my ($c, $arg) = @_;
  $c->log->warn( 'uri_for with non action: ', join(', ', @_), )
    if (!blessed($_[0]) || !$_[0]->isa('Catalyst::Action'));
  return $uri;
};
```

Note that Catalyst will load any Moose Roles in the plugin list, and apply them to your application class.

Factory components with COMPONENT()

Every component inheriting from Catalyst::Component contains a `COMPONENT` method. It is used on application startup by `setup_components` to instantiate the component object for the Catalyst application. By default, this will merge the components own `config`uration with the application wide overrides and call the class' `new` method to return the component object.

You can override this method and do and return whatever you want. However, you should use Class::C3 (via MRO::Compat) to forward to the original `COMPONENT` method to merge the configuration of your component.

Here is a stub `COMPONENT` method:

```
package CatalystX::Component::Foo;
use Moose;
use namespace::autoclean;

extends 'Catalyst::Component';

sub COMPONENT {
    my $class = shift;
    # Note: $app is like $c, but since the application isn't fully
    # initialized, we don't want to call it $c yet.  $config
    # is a hashref of config options possibly set on this component.
    my ($app, $config) = @_;

    # Do things here before instantiation
    $new = $class->next::method(@_);
    # Do things to object after instantiation
    return $new;
}
```

The arguments are the class name of the component, the class name of the application instantiating the component, and a hash reference with the controller's configuration.

You are free to re-bless the object, instantiate a whole other component or really do anything compatible with Catalyst's expectations on a component.

For more information, please see "COMPONENT($c,$arguments)" in Catalyst::Component.

Applying roles to parts of the framework

CatalystX::RoleApplicator will allow you to apply Roles to the following classes:

Request

Response

Engine

Dispatcher

Stats

These roles can add new methods to these classes, or wrap preexisting methods.

The namespace for roles like this is `Catalyst::TraitFor::XXX::YYYY`.

For an example of a CPAN component implemented in this manor, see Catalyst::TraitFor::Request::BrowserDetect.

SEE ALSO

Catalyst, Catalyst::Manual::Actions, Catalyst::Component

AUTHORS

Catalyst Contributors, see Catalyst.pm

COPYRIGHT

syntax highlighting: no syntax highlighting ▼