NAME

Catalyst::Manual::About - The philosophy of Catalyst

DESCRIPTION

This document is a basic introduction to the *why* of Catalyst. It does not teach you how to write Catalyst applications; for an introduction to that please see Catalyst::Manual::Intro. Rather, it explains the basics of what Catalyst is typically used for, and why you might want to use Catalyst to build your applications.

What is Catalyst? The short summary

Catalyst is a web application framework. This means that you use it to help build applications that run on the web, or that run using protocols used for the web. Catalyst is designed to make it easy to manage the various tasks you need to do to run an application on the web, either by doing them itself, or by letting you "plug in" existing Perl modules that do what you need. There are a number of things you typically do with a web application. For example:

- Interact with a web server

  If you're on the web, you're relying on a web server, a program that sends files over the web. There are a number of these, and your application has to do the right thing to make sure that your program works with the web server you're using. If you change your web server, you don't want to have to rewrite your entire application to work with the new one.

- Do something based on a URI

  It's typical for web applications to use URIs as a main way for users to interact with the rest of the application; various elements of the URI will indicate what the application needs to do. Thus, `http://www.mysite.com/add_record.cgi?name=John&title=President` will add a person named "John" whose title is "President" to your database, and `http://www.mysite.com/catalog/display/23` will go to a "display" of item 23 in your catalog, and `http://www.mysite.com/order_status/7582` will display the status of order 7582, and `http://www.mysite.com/add_comment/?page=8` will display a form to add a comment to page 8. Your application needs to have a regular way of processing these URIs so it knows what to do when such a request comes in.

- Interact with a data store

You probably use a database to keep track of your information. Your application needs to interact with your database, so you can create, edit, and retrieve your data.

- Handle forms

  When a user submits a form, you receive it, process it to make sure it's been filled in properly, and then do something based on the result--submit an order, update a record, send e-mail, or return to the form if there's an error.

- Display results

  If you have an application running on the web, people need to see things. You usually want your application displayed on a web browser, in which case you will probably be using a template system to help generate HTML code. But you might need other kinds of display, such as PDF files, or other forms of output, such as RSS feeds or e-mail.

- Manage users

  You might need the concept of a "user", someone who's allowed to use your system, and is allowed to do certain things only. Perhaps normal users can only view or modify their own information; administrative users can view or modify anything; normal users can only order items for their own account; normal users can view things but not modify them; order-processing users can send records to a different part of the system; and so forth. You need a way of ensuring that people are who they say they are, and that people only do the things they're allowed to do.

- Develop the application itself

  When you're writing or modifying the application, you want to have access to detailed logs of what it is doing. You want to be able to write tests to ensure that it does what it's supposed to, and that new changes don't break the existing code.

Catalyst makes it easy to do all of these tasks, and many more. It is extremely flexible in terms of what it allows you to do, and very fast. It has a large number of "components" and "plugins" that interact with existing Perl modules so that you can easily use them from within your application.

- Interact with a web server?

  Catalyst lets you use a number of different ones, and even comes with a built-in server for testing or local deployment.

- Do something based on a URI?

  Catalyst has extremely flexible systems for figuring out what to do based on a URI.

- Interact with a data store?

  Catalyst has many plugins for different databases and database frameworks, and for other non-database storage systems.

- Handle forms?

  Catalyst has plugins available for several form creation and validation systems that make it easy for the programmer to manage.

- Display results?

Catalyst has plugins available for a number of template modules and other output packages.

- Manage users?

  Catalyst has plugins that handle sessions, authentication, and authorization, in any way you need.

- Developing the application?

  Catalyst has detailed logging built-in, which you can configure as necessary, and supports the easy creation of new tests--some of which are automatically created when you begin writing a new application.

What isn't Catalyst?

Catalyst is not an out-of-the-box solution that allows you to set up a complete working e-commerce application in ten minutes. (There are, however, several systems built on top of Catalyst that can get you very close to a working app.)

Catalyst is designed for flexibility and power; to an extent, this comes at the expense of simplicity. Programmers have many options for almost everything they need to do, which means that any given need can be done in many ways, and finding the one that's right for you, and learning the right way to do it, can take time. TIMTOWDI works both ways.

Catalyst is not designed for end users, but for working programmers.

Web programming: The Olden Days

Perl has long been favored for web applications. There are a wide variety of ways to use Perl on the web, and things have changed over time. It's possible to handle everything with very raw Perl code:

```
    print "Content-type: text/html\n\n<center><h1>Hello
    World!</h1></center>";
```

for example, or

```
    my @query_elements = split(/&/, $ENV{'QUERY_STRING'});
    foreach my $element (@query_elements) {
        my ($name, $value) = split(/=/, $element);
        # do something with your parameters, or kill yourself
        # in frustration for having to program like this
    }
```

Much better than this is to use Lincoln Stein's great CGI module, which smoothly handles a wide variety of common tasks--parameter parsing, generating form elements from Perl data structures, printing http headers, escaping text, and very many more, all with your choice of functional or object-oriented style. While CGI was revolutionary and is still widely used, it has various drawbacks that make it unsuitable for larger applications: it is slow; your code with it generally combines application logic and display code; and it makes it very difficult to handle larger applications with complicated control flow.

A variety of frameworks followed, of which the most widely used is probably CGI::Application, which encourages the development of modular code, with easy-to-understand control-flow handling, the use of plugins and templating systems, and the like. Other systems include AxKit, which is designed for use with XML running under mod_perl; Maypole--upon which Catalyst was originally based--designed for the easy development of powerful web databases; Jifty, which does a great deal of automation in helping to set up web sites with many complex features; and Ruby on Rails (see http://www.rubyonrails.org), written of course in Ruby and among the most popular web development systems. It is not the purpose of this document to criticize or even briefly evaluate these other frameworks; they may be useful for you and if so we encourage you to give them a try.

The MVC pattern

MVC, or Model-View-Controller, is a model currently favored for web applications. This design pattern is originally from the Smalltalk programming language. The basic idea is that the three main areas of an application--handling application flow (Controller), processing information (Model), and outputting the results (View)--are kept separate, so that it is possible to change or replace any one without affecting the others, and so that if you're interested in one particular aspect, you know where to find it.

Discussions of MVC often degenerate into nitpicky arguments about the history of the pattern, and exactly what "usually" or "should" go into the Controller or the Model. We have no interest in joining such a debate. In any case, Catalyst does not enforce any particular setup; you are free to put any sort of code in any part of your application, and this discussion, along with others elsewhere in the Catalyst documentation, are only suggestions based on what we think works well. In most Catalyst applications, each branch of MVC will be made of up of several Perl modules that can handle different needs in your application.

The purpose of the **Model** is to access and modify data. Typically the Model will interact with a relational database, but it's also common to use other data sources, such as the Xapian search engine or an LDAP server.

The purpose of the **View** is to present data to the user. Typical Views use a templating module to generate HTML code, using Template Toolkit, Mason, HTML::Template, or the like, but it's also possible to generate PDF output, send e-mail, etc., from a View. In Catalyst applications the View is usually a small module, just gluing some other module into Catalyst; the display logic is written within the template itself.

The **Controller** is Catalyst itself. When a request is made to Catalyst, it will be received by one of your Controller modules; this module will figure out what the user is trying to do, gather the necessary data from a Model, and send it to a View for display.

A simple example

The general idea is that you should be able to change things around without affecting the rest of your application. Let's look at a very simple example (keeping in mind that there are many ways of doing this, and what we're discussing is one possible way, not the only way). Suppose you have a record to display. It doesn't matter if it's a catalog entry, a library book, a music CD, a personnel record, or anything else, but let's pretend it's a catalog entry. A user is given a URL such as `http://www.mysite.com/catalog/display/2782`. Now what?

First, Catalyst figures out that you're using the "catalog" Controller (how Catalyst figures this out is entirely up to you; URL dispatching is *extremely* flexible in Catalyst). Then Catalyst determines that you want to use a `display` method in your "catalog" Controller. (There could be other `display` methods in other Controllers, too.) Somewhere in this process, it's possible that you'll have authentication and authorization routines to make sure that the user is registered and is allowed to display a record. The Controller's `display` method will then extract "2782" as the record you want to retrieve, and make a request to a Model for that record. The Controller will then look at what the Model returns: if there's no record, the Controller will ask the View to display an error message, otherwise it will hand the View the record and ask the View to display it. In either case, the View will then generate an HTML page, which Catalyst will send to the user's browser, using whatever web server you've configured.

How does this help you?

In many ways. Suppose you have a small catalog now, and you're using a lightweight database such as SQLite, or maybe just a text file. But eventually your site grows, and you need to upgrade to something more powerful--MySQL or Postgres, or even Oracle or DB2. If your Model is separate, you only have to change one thing, the Model; your Controller can expect that if it issues a query to the Model, it will get the right kind of result back.

What about the View? The idea is that your template is concerned almost entirely with display, so that you can hand it off to a designer who doesn't have to worry about how to write code. If you get all the data in the Controller and then pass it to the View, the template isn't responsible for any kind of data processing. And if you want to change your output, it's simple: just write a new View. If your Controller is already getting the data you need, you can pass it in the same way, and whether you display the results to a web browser, generate a PDF, or e-mail the results back to the user, the Controller hardly changes at all--it's up to the View.

And throughout the whole process, most of the tools you need are either part of Catalyst (the parameter-processing routines that extract "2782" from the URL, for example) or are easily plugged into it (the authentication routines, or the plugins for using Template Toolkit as your View).

Now, Catalyst doesn't enforce very much at all. Template Toolkit is a very powerful templating system, and you can connect to a database, issue queries, and act on them from within a TT-based View, if you want. You can handle paging (i.e. retrieving only a portion of the total records possible) in your Controller or your Model. In the above example, your Controller looked at the query result, determining whether to ask the View for a no-result error message, or for a result display; but it's perfectly possible to hand your query result directly to the View, and let your template decide what to do. It's up to you; Catalyst doesn't enforce anything.

In some cases there might be very good reasons to do things a certain way (issuing database queries from a template defeats the whole purpose of separation-of-concerns, and will drive your designer crazy), while in others it's just a matter of personal preference (perhaps your template, rather than your Controller, is the better place to decide what to display if you get an empty result). Catalyst just gives you the tools.

SEE ALSO

Catalyst, Catalyst::Manual::Intro

AUTHORS

Catalyst Contributors, see Catalyst.pm

COPYRIGHT

syntax highlighting: no syntax highlighting ▼