Zusammenhang ist Alles oder wie Catalyst im Einsatz funktioniert

Simon Bertrang

4. Februar 2007

Inhaltsverzeichnis

Dieser Artikel wurde für den German Perl Workshop 2007 geschrieben.

Autor

Simon Bertrang <janus@errornet.de>,

Abstract

Mit Catalyst hat man ein mächtiges Webframework zur Hand das auf Perl basiert. Es gibt inzwischen viele Module und Plugins die einem mehr und mehr Funktionalität und Möglichkeiten bieten. Eine riesige Auswahl an Modulen die sich mit CPAN direkt und unkompliziert installieren lassen.
Die Anzahl realisierbaren Kombinationen ist so groß, dass einem die Wahl nicht immer leicht fällt.
Zu wissen welche Module auf jeden Fall funktionieren um schnell im Entwicklungsprozess vorwärts zu kommen ist essentiell.
Die meisten Vorträge die bisher über Catalyst gehalten wurden haben die Möglichkeiten alle aufgezeigt und Lust auf mehr gemacht, aber womit fängt man an und was für eine Konstellation funktioniert?
An genau dieser Stelle setzt dieser Vortrag an.
Er soll anhand eines Systems im praktischen Einsatz ein Beispiel für eine funktionierende Lösung zeigen.
Das sind grundlegend wichtige, technische Entscheidungen die man als Entwickler treffen muss oder schon getroffen haben sollte um für den Einsatz gewappnet zu sein.
Es wird am Beispiel einer einfachen Webanwendung eine Implementation gezeigt die sich bereits in produktiver Umgebung bewährt hat.
Da das ganze System auf Perl basiert ist es so portabel wie Perl selbst und seine Module und ist genauso auch auf Platformen lauffähig die ungleich der gezeigten sind.

1 Einführung

Die Idee für den Vortrag auf dem Deutschen Perl Workshop 2007[1] ist dabei entstanden, Catalyst[2] zu OpenBSD[3] zu portieren[4].
Im Laufe der Entwicklung einer Webanwendung bestand die Notwendigkeit Catalyst in die OpenBSD Ports Collection zu integrieren[5].
Nachdem alle benötigten Pakete fertig waren und die Entwicklungsphase begann, fiel auf dass es für die Kombination von Modulen - oder ähnlichen - noch keine Dokumentation dieser Art vorhanden ist.
Es gibt zwar Manuals[6], HowTos und Tutorials, viele davon sind aber so experimenteller Natur, dass sie sich als Beispiele in realen Anwendungen nicht immer eignen.

Mit diesem Vortrag wird versucht einen praktischen Einblick in Catalyst zu bieten und Lösungen die sich im produktiven Einsatz bewährt haben.
Im Folgenden wird Catalyst und einige Module im Zusammenspiel gezeigt.
Der beiliegende Programmcode ist bei der Anwendung der im Folgenden beschriebenen Schritte entstanden.

2 Voraussetzungen

Das Ziel ist es eine web-basierende, Benutzer-Platform zu schaffen.
Wichtige Punkte sind die Performance und Skalierbarkeit da die Anwendung auch komplexe Vorgängen verwalten können muss - auch wenn grade mal wieder mehr Benutzer zu gast sind.
Es gibt ein rollenbasiertes Berechtigungssystem um die einzelnen Funktionen voneinander trennen zu können.
Der eingesetzte Webserver spielt im Prinzip keine Rolle, da Catalyst sich problemlos an alle gängigen Schnittstellen anschliessen lässt.
Wir beschränken uns in den Beispielen zwar auf den mitgelieferten Perl-Webserver, in Produktion erprobt sind jedoch bereits CGI, FastCGI und mod_perl 1&2.
Als Datenbank kommt PostgreSQL[7] zum Einsatz, da Konsistenz in diesem Projekt eine große Rolle spielt und spezielle Datentypen benutzen werden. Ausserdem ist der Autor zu sehr an Transaktionssicherheit, serverseitige Funktionen und referenzielle Integrität gewöhnt.

3 Die Module

Um alle Aufgaben schnell und effektiv erledigen zu können, benutzen wir eine Reihe an Modulen zu denen im einzelnen kurz die Funktion beschrieben wird.
In einigen Fällen sind die Namen der Module so selbstredend, dass auf eine eingehendere Beschreibung zu Gunsten der Kompaktheit des Artikels verzichtet wurde. Weiterhin ist es empfehlenswert die Dokumentation der genannten Module in näheren Augenschein zu nehmen.

3.1 Catalyst

Hiermit wird uns alles Grundlegende geliefert was für die Webanwendung notwendig ist. Damit haben wir die Möglichkeit alle GET/POST-Parameter zu verarbeiten, es lassen sich Um- und Weiterleitungen programmieren und alles Weitere was im Laufe eines Projektes an Anforderungen aufkommt.
Der Name der Anwendung ist - entsprechend dem Artikel - GPW07::Sim81 und wird wie folgt angelegt:

```
$ catalyst.pl GPW07::Sim81
created "GPW07-Sim81"
created "GPW07-Sim81/script"
...
created "GPW07-Sim81/script/gpw07_sim81_test.pl"
created "GPW07-Sim81/script/gpw07_sim81_create.pl"
$ cd GPW07-Sim81
$
```

GPW07-Sim81 ist das Verzeichnis in dem sich die Anwendung befindet.

4 Die Dynamik

Catalyst selbst ist "nur" das Framework - wir benötigen aber mehr. Deswegen greifen wir auf das riesen Reportoire an fertigen Modulen zurück und sammeln so die notwendige, weitergehende Grundfunktionalität zusammen.
Das passiert indem wir uns das Grundmodul der Anwendung (lib/GPW07/Sim81.pm) vornehmen und erweitern:

```
13 use Catalyst qw/
14         -Debug
15         ConfigLoader
16         Static::Simple
17         FormValidator
18         Session
19         Session::Store::File
20         Session::State::Cookie
21         Authentication
22         Authentication::Store::DBIC
23         Authentication::Credential::Password
24         Authorization::Roles
25         /;
```

4.1 Catalyst::View::TT

Ein großer Teil der Anwendung wird durch die genutzte Template-Sprache bestimmt. Hier hat sich seit langem und in vielen Projekten Template-Toolkit[9] bewährt.
Gegenüber anderen Template-Sprachen erlaubt es eine vollständige Integration von Model, View und Controller mit Catalyst::View::TT[8].
Die Unterstützung dafür aktivieren wir durch die Generierung der "TT"-View:

```
$ ./script/gpw07_sim81_create.pl view TT TT
 exists "$PWD/script/../lib/GPW07/Sim81/View"
 exists "$PWD/script/../t"
created "$PWD/script/../lib/GPW07/Sim81/View/TT.pm"
created "$PWD/script/../t/view_TT.t"
$
```

4.2 Catalyst::Plugin::ConfigLoader

Die vorgenommenen Einstellungen sollen in extra Konfigurationsdateien[10] gespeichert werden - für eine bessere Handhabung, Anpassbar- und Übersichtlichkeit.
Das benutzte Format kann frei gewählt werden, der Einfachheit halber wird YAML[11] aber beibehalten.
Im "Basis-Modul" findet die verantwortliche Zeile:

```
15          ConfigLoader
```

Es muss nichts getan werden, das Plugin wurde schon vom "catalyst.pl"-Aufruf erstellt, wir müssen also nur noch unsere Einstellungen in "gpw07_sim81.yml" eintragen.

4.3 Catalyst::Plugin::Static::Simple

Normalerweise liefern wir mit Catalyst dynamische Inhalte aus.
Für den Fall dass wir aber doch statische Inhalte (Perl-Webserver) verarbeiten müssen ist dieses Plugin[12] gedacht:

```
16          Static::Simple
```

Wenn sich die Anwendung erst einmal in Produktion befindet, wo ein "richtiger" Webserver für die statischen Inhalte zuständig ist, kann man getrost auf dieses Plugin verzichten. Hier braucht es zunächst keinerlei Änderung; wird beim Erstellen generiert.

4.4 Catalyst::Plugin::FormValidator

Inhalte von Webformularen zu überprüfen kann schnell anstrengend werden.
Damit das garnicht erst passiert[13], benutzen wir von vorneherein Data::FormValidator[14], welches uns erlaubt Formular-Profile zu erstellen und diese gegen Benutzereingaben zu validieren.
Auch hier wieder der Eintrag im "Basis-Modul":

```
17          FormValidator
```

4.5 Catalyst::Plugin::Session

Da HTTP keine Zustände im Protokoll abbilden kann, wir aber eine Interaktive Webanwendung erstellen, nehmen wir das Session-Plugin[15] zur Hilfe:

```
18          Session
```

4.6 Catalyst::Plugin::Session::Store::File

Der Inhalt der Sessions soll auf dem Server in einer Datei gespeichert werden[16].
Je nach Bedarf stehen auch andere Backends zur Verfügung, aber in den meisten Fällen reicht diese bewährte Lösung aus:

```
19          Session::Store::File
```

4.7 Catalyst::Plugin::Session::State::Cookie

Der Client soll die Session in einem Browser-Cookie speichern[17]. Im Gegensatz zu URL-basierenden Sessions lassen sich so versehentliche Weitergaben von Links vermeiden und die Templates bleiben einfacher:

```
20          Session::State::Cookie
```

4.8 Catalyst::Action::RenderView

Eine Standard-Aktion zum automatischen Laden von Templates entsprechend dem Namen der aufgerufenen Funktion[18].
Wird beim Erstellen auch mitgeneriert:

```
16          Static::Simple
17          FormValidator
18          Session
19          Session::Store::File
20          Session::State::Cookie
```

5 Die Skalierbarkeit

Für eine möglichst effiziente Datenverwaltung wurden einige Entscheidungen getroffen:

5.1 DBD::Pg

Die von uns genutzte Schnittstelle zur Datenbank[19].

5.2 DBIx::Class

Datenmodelle liegen als DBIx::Class[20] Definitionen vor.
Hier können wir auch "Constraints" und mehr definieren die wiederum genauso durch Funktionen in der Datenbank abgebildet sein können.

5.3 Catalyst::Model::DBIC::Schema

Das Modul welches die Datenmodelle in Catalyst[21] zur Verfügung stellt.

6 Die Sicherheit

... ist immer ein wichtiges Thema. CPAN sei Dank, gibt es auch hier schon fertige Module die müheslige Arbeit einsparen können:

```
21          Authentication
22          Authentication::Store::DBIC
23          Authentication::Credential::Password
24          Authorization::Roles
25          /;
```

6.1 Catalyst::Plugin::Authentication

Um Benutzer zu erkennen[22] brauchen wir dieses Plugin:

```
21          Authentication
```

Die Konfiguration nehmen wir in unserer YAML-Datei vor:

```
15  authentication:
16      dbic:
17          user_class: DB::User
18          user_field: name
19          password_field: password
20          password_type: hashed
21          password_hash_type: SHA-1
```

Sie beinhaltet der Einfachheit halber schon die Einstellungen für die noch folgenden Authentication-Module.

6.2 Catalyst::Plugin::Authentication::Store::DBIC

... wird gebraucht um die Benutzer- und Rolleninformationen in der Datenbank zu speichern[23].

```
22          Authentication::Store::DBIC
```

6.3 Catalyst::Plugin::Authentication::Credential::Password

Wir benutzen die "ganz normalen" Möglichkeiten um verschlüsselte Passworte in der Datenbank zu hinterlegen[24]:

```
23          Authentication::Credential::Password
```

6.4 Catalyst::Plugin::Authorization::Roles

Da die Benutzer im System unterschiedliche Zugrifssrechte haben sollen, greifen wir bei der Authorisation auf das Rollen-Plugin[25] zurück:

```
24          Authorization::Roles
```

Wie zuvor schon, benötigt die Konfiguration eine weitere Änderung:

```
23  authorization:
24      dbic:
25          role_class: DB::Role
26          role_field: name
27          role_rel: users
28          user_role_user_field: user
```

7 Die Logik

Es gibt zunächst ein paar wichtige Punkte die zum besseren Verständis des Codes beitragen, die hier kurz erklärt werden:

- Controller bestimmen den Namensraum in dem die Methoden ansprechbar sind. Das bedeutet dass der URL-Pfad zum Controller "User" mit "/user" anfängt.
- Funktionsattribute bestimmen das Verhalten beim Aufruf einer Methode. ":Private" ist - wie der Name schon sagt - privat und für interne und/oder spezielle Aufrufe zuständig. Als "normal" lässt sich ":Local" bezeichnen, welches einfach nur den Methodennamen an den Controller-Pfad anhängt.

Es gibt noch eine Reihe weiterer Funktionsattribute auf die an dieser Stelle nicht näher eingegangen wird, da sie innerhalb des Artikels keine weitere Relevanz haben. Dennoch lohnt es sich einen näheren Blick in das Manual zu werfen.

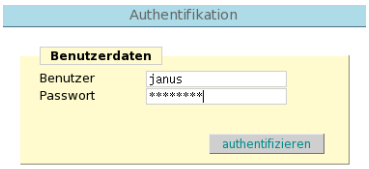
7.1 Login

Ganz am Anfang steht der Login. Da wir prinzipiell immer (bis auf wenige Ausnahmen) die Authentifikation des Benutzers wollen, sorgen wir dafür dass diese an höchster Stelle, immer überprüft wird. Das geht am einfachsten mit einer "auto"-Methode im "Root"-Controller:

```
53  sub auto : Private {
54          my ($self, $c) = @_;
55          my $path = $c->req->path;
56
57          # Standard-Berechtigungen
58          if ($path =~ /^(?:login|logout|static)/) {
```

```
59                  return 1;
60          }
61
62          # Login für nicht authentifizierte Benutzer erzwingen
63          unless ($c->user_exists()) {
64                  # Umleiten zur Login-Seite
65                  $c->res->redirect($c->uri_for('/login', $path));
66                  return 0;
67          }
68
69          # Benutzer gefunden und erlauben
70          return 1;
71  }
```

Wenn der Pfad mit “login”, “logout”, oder “static” beginnt erlauben wir immer Zugriff.
Für andere Pfade wird geprüft ob der Benutzer authentifiziert ist und schicken ihn anderenfalls auf die Login-Seite.
Der Login selbst ist auch schnell realisiert:




```
79  sub login : Local {
80          my ($self, $c, @path) = @_;
81
82          # Parameter holen und prüfen
83          if (my $user = $c->req->param("user")) {
84                  # Passwort holen
85                  my $password = $c->req->param("password");
86
87                  # Login durchführen
88                  if ($c->login($user, $password)) {
89                          # zum Ziel umleiten
90                          $c->res->redirect($c->uri_for('/', @path));
91                          return;
92                  } else {
93                          # Bei falschem Login 'user' im Template setzen
94                          $c->stash->{'user'} = $user;
95                  }
96          }
97  }
```

7.2 Logout

... ist auch einfach:

```
105 sub logout : Local {
106         my ($self, $c, @path) = @_;
107
108         # Wenn der Benutzer existiert
109         if ($c->user_exists()) {
```

```
110                 # ... ausloggen
111                 $c->logout();
112         }
113
114         # Und umleiten zur vorhergehenden Seite
115         $c->res->redirect($c->uri_for('/', @path));
116 }
```

Damit wäre ein wichtiger Teil erledigt und wir können uns um die eigentliche Anwendung kümmern.
Modelle und Mehr... Um die Inhalte der Datenbank anzuzeigen legen wir als erstes den Kontroller “User” an:

```
$ ./script/gpw07_sim81_create.pl controller User
  exists "$PWD/script/../lib/GPW07/Sim81/Controller"
  exists "$PWD/script/../t"
created "$PWD/script/../lib/GPW07/Sim81/Controller/User.pm"
created "$PWD/script/../t/controller_User.t"
$
```

7.3 Übersicht

Wir benutzen die ‘index’-Methode um eine einfache Übersicht zu geben... dazu holen wir uns den Inhalt unserer Modelle:


```
24         my $users = $c->model('User')->search(undef, {
25                 'order_by'      => 'name',
26         });
```

Nun haben wir in ‘$users’ einen sogenannten Iterator, den wir unserem Template zur Verfügung stellen indem wir ihn in ‘stash’ packen:

```
28         $c->stash->{'users'} = $users;
```

... und das war’s auch schon für den Code, nur noch ein Template in ‘root/user/index.html’ speichern:

```
1  [%
2  page = { title = 'Benutzerübersicht' };
3  PROCESS header.html;
4  %]
5  <fieldset><legend>Liste</legend>
6  <ul>
7  [% WHILE (user = users.next) %]
8  <li><a href="[% c.uri_for('/user/show', user.id) %]">[% user.name %]</a></li>
9  [% END %]
```

```
10 </ul>
11 [% PROCESS footer.html %]
```

Ein besonderer Augenmerk gilt hier dem Methodenaufruf von “uri_for()”. Mit dieser Helferfunktion - die sich auch in den Templates benutzen lässt - bekommt man ganz einfach und komfortabel eine passende URL innerhalb der Anwendung.
Und damit haben wir nun schonmal eine einfache Liste der Benutzer.

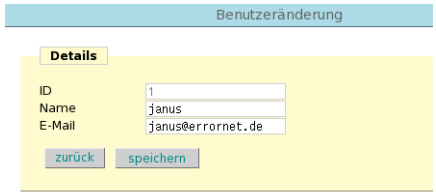
7.4 Ansicht

Um die Details eines Benutzers anzuzeigen wird eine ‘show’-Methode angelegt, die den Benutzer an Hand der ID sucht und an das Template übergibt:


```
96  sub show : Local {
97          my ($self, $c, $id) = @_;
98          my $user;
99
100         # Prüfen ob ID eine Zahl ist
101         unless ($id =~ m/^\d+$/) {
102                 # und anderenfalls zum index umleiten
103                 $c->response->redirect($c->uri_for(''));
104                 return;
105         }
106
107         # Benutzer an Hand der ID finden
108         unless (($user = $c->model('DB::User')->find($id))) {
109                 # oder wiederum umleiten zum index
110                 $c->response->redirect($c->uri_for(''));
111                 return;
112         }
113
114         # den gefundenen Benutzer an das Template übergeben
115         $c->stash->{'user'} = $user;
116 }
```

7.5 Änderung

Das einzige was uns jetzt noch fehlt sind Formulare... fügen wir dazu eine neue Methode ‘edit’ in den ‘User’-Controller ein:




```
37  sub edit : Local {
38          my ($self, $c, $id) = @_;
39          my $req = $c->request();
```

```
40          my $stash = $c->stash();
41          my $data;
42          my %form;
43          my $user;
44
45          # Prüfen ob 'id' numerisch ist
46          unless ($id =~ m/^\d+$/) {
47                  # ansonsten umleiten...
48                  $c->response->redirect($c->uri_for('index'));
49                  return;
50          }
51
52          # es werden Benutzername und Email benötigt
53          %form = (
54                  'required'
    => [qw(name email)],
55          );
56
57          # Benutzer an Hand der ID finden
58          unless (($user = $c->model('User')->find($id))) {
59                  # ansonsten umleiten
60                  $c->response->redirect($c->uri_for('index'));
61                  return;
62          }
63
64          # ein GET zeigt den Benutzer an
65          if ($req->method() eq 'GET') {
66                  $stash->{'user'} = $user;
67                  return;
68          }
69          # ein POST führt die Überprüfung aus
70          $c->form(%form);
71          # Formular prüfen
72          $data = $c->form->valid();
73
74          # solange nicht Alles ok ist...
75          unless ($c->form->success()) {
76                  # ... Daten wieder ausgeben
77                  $stash->{'user'} = $data;
78                  return;
79          }
80
81          # Daten aktualisieren
82          $user->set_columns($data);
83          # und Speichern
84          $user->update();
85
86          # Umleiten zur Anzeige
87          $c->response->redirect($c->uri_for('show', $id));
88  }
```

Die wichtigen Punkt um mit Catalyst loslegen zu können sind damit erklärt. Ausgehend von der gesammelten Erfahrung reichen diese Grundlagen aus und ermöglichen ein schnelles Vorankommen.
Natürlich gibt es weitere Herausfoderungen denen sich der Autor in diesem Artikel nicht gewidmet hat, die er zu Übungszwecken aber gerne dem Leser überlässt.
An dieser Stelle ist ein wiederholter Hinweis auf die Dokumentation der genutzen Module angebracht.

Literatur

[1] German Perl Workshop - *German Perl Workshop Website.*
`http://www.perl-workshop.de`

[2] Catalyst - *The Elegant MVC Web Application Framework.*
`http://www.catalystframework.com`

[3] OpenBSD - *A free, multi-platform 4.4BSD-based UNIX-like operating system.*
`http://www.openbsd.org`

[4] OpenBSD Ports - *The OpenBSD packages and ports framework.*
`http://openbsd.org/faq/faq15.html`

[5] Catalyst on OpenBSD - *Journal entry i made when Catalyst got imported to OpenBSD CVS tree.*
`http://use.perl.org/ janus/journal/31542`

[6] Catalyst::Manual - *User guide and reference for Catalyst.*
`http://search.cpan.org/perldoc?Catalyst::Manual`

[7] PostgreSQL - *The world's most advanced open source database.*
`http://www.postgresql.org`

[8] Catalyst::View::TT - *Template View Class.*
`http://search.cpan.org/perldoc?Catalyst::View::TT`

[9] Template Toolkit - *a fast, powerful and extensible template processing system.*
`http://template-toolkit.org`

[10] Catalyst::Plugin::ConfigLoader - *Load config files of various types.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::ConfigLoader`

[11] YAML - *YAML Ain't Markup Language*
`http://yaml.org`

[12] Catalyst::Plugin::Static::Simple - *Make serving static pages painless.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Static::Simple`

[13] Catalyst::Plugin::FormValidator - *FormValidator for Catalyst.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::FormValidator`

[14] Data::FormValidator - *Validates user input (usually from an HTML form) based on input profile.*
`http://search.cpan.org/perldoc?Data::FormValidator`

[15] Catalyst::Plugin::Session - *Generic Session plugin - ties together server side storage and client side state required to maintain session data.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Session`

[16] Catalyst::Plugin::Session::Store::File - *File storage backend for session data.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Session::Store::File`

[17] Catalyst::Plugin::Session::State::Cookie - *Maintain session IDs using cookies.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Session::State::Cookie`

[18] Catalyst::Action::RenderView - *Sensible default end action.*
`http://search.cpan.org/perldoc?Catalyst::Action::RenderView`

[19] DBD::Pg - *PostgreSQL database driver for the DBI module.*
`http://search.cpan.org/perldoc?DBD::Pg`

[20] DBIx::Class - *Extensible and flexible object <-> relational mapper.*
`http://search.cpan.org/perldoc?DBIx::Class`

[21] Catalyst::Model::DBIC::Schema - *DBIx::Class::Schema Model Class.*
`http://search.cpan.org/perldoc?Catalyst::Model::DBIC::Schema`

[22] Catalyst::Plugin::Authentication - *Infrastructure plugin for the Catalyst authentication framework.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Authentication`

[23] Catalyst::Plugin::Authentication::Store::DBIC - *Authentication and authorization against a DBIx::Class or Class::DBI model.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Authentication::Store::DBIC`

[24] Catalyst::Plugin::Authentication::Credential::Password - *Authenticate a user with a password.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Authentication::Credential::Password`

[25] Catalyst::Plugin::Authorization::Roles - *Role based authorization for Catalyst based on Catalyst::Plugin::Authentication.*
`http://search.cpan.org/perldoc?Catalyst::Plugin::Authorization::Roles`