Ausgabe 1/2013 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Synchrone Operationen
sind überholt

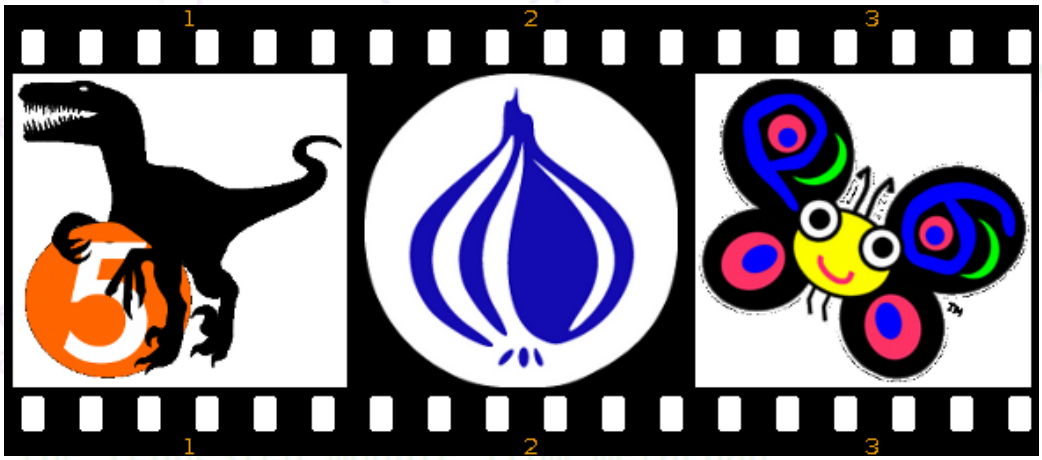
Mojolicious Tutorial
Teil 2

Nr 25

Berlin.pm Production presents

Hauptrollen

* Velo Ci. Raptor
* "The Onion"
* Camelia



Premiere: 13.-15. März 2013

URL: http://act.yapc.eu/gpw2013

VORWORT

Jubiläen und andere Events

Sie halten die "Silberausgabe" in den Händen! Kaum zu glauben, aber es ist tatsächlich schon die 25. Ausgabe des Perl-Magazins. Seit Februar 2007 erscheint das Magazin alle drei Monate, anfangs noch mit einem unprofessionellen Design (oder wie Martin Seibert sagte: "suboptimal") und ab der 3. Ausgabe mit dem jetzigen Layout.

Zur Feier des Tages gibt es die alten Ausgaben (3/2007 - 4/2012) - soweit noch verfügbar - für nur 3 EUR das Heft. Vielen Dank an die treuen Leser!

Das Jahr 2013 hat gerade erst angefangen nachdem wir nur knapp dem Weltuntergang entkommen sind. Da möchte ich einen kleinen Ausblick darüber geben, welche Events uns in diesem Jahr sonst noch bevorstehen. Mitte März (13.-15.) findet in Berlin der 15. Deutsche Perl-Workshop statt. Schon bei Redaktionsschluss dieser Ausgabe waren etliche Vorträge angenommen, die sich sehr spannend anhören. Wir unterstützen dieses Event natürlich wieder als Sponsor.

Direkt im Anschluss (16./17. März) gibt es den Chemnitzer Linuxtag, auf dem es auch einen Perl-Stand geben wird. Auch hier werden wir anzutreffen sein. Im Juni gibt es in Berlin die Linuxtage, wobei es hier vermutlich keinen Perl-Stand geben wird.

Ansonsten gibt es wahrscheinlich auch wieder in Kiel und Augsburg kleinere Linuxtage sowie die Linuxtag-Serie in Österreich (Wien, Graz, ...). Und auch die FrOSCon steht wieder auf dem Plan.

Nach einem Jahr Pause findet im November wieder die OpenRheinRuhr in Oberhausen statt. Ich versuche, auch dort wieder einen kleinen Stand zu haben.

Wir versuchen auf mehreren Veranstaltungen anwesend zu sein, aber mit Anreise und Standbetreuung ist es doch recht zeitaufwändig. Deshalb wäre es schön, wenn sich viele Helfer für die Veranstaltungen finden würden. Die "alten Hasen" helfen gerne bei der Organisation eines Stands und dem ganzen Material. Solche Events sind immer eine gute Gelegenheit, auf Perl-Projekte und die aktuellen Änderungen an Perl selbst aufmerksam zu machen.

Nicht aufgelistet habe ich die ganzen "normalen" Perl-Events wie die Stammtische der einzelnen Perlmonger-Gruppen. Ich kann Ihnen nur empfehlen, mal bei einer solchen Gruppe vorbeizuschauen und sich mit anderen über alle möglichen Themen - nicht nur Perl - auszutauschen. Da kommen häufig ganz interessante Ideen zu Tage. Die jüngste deutschsprachige Perlmonger-Gruppe ist in Karlsruhe zu finden. Im Januar 2013 fand das erste Treffen statt.

Damit ist der Südwesten von Deutschland ganz gut abgedeckt: Frankfurt, Darmstadt, Kaiserslautern, Karlsruhe, Stuttgart und Ulm.

Renée Bäcker

Die Codebeispiele können mit dem Code

6824l8g

von der Webseite www.foo-magazin.de heruntergeladen werden!

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue25

IMPRESSUM

Herausgeber:	Perl-Services.de Renée Bäcker Bergfeldstr. 23 D - 64560 Riedstadt
Redaktion:	Renée Bäcker
Anzeigen:	Renée Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	print24 (Marke der unitedprint.com Deutschland GmbH) Friedrich-List-Straße 3 D - 01445 Radebeul
ISSN Print:	1864-7537
ISSN Online:	1864-7545
Feedback:	feedback@perl-magazin.de

INHALTSVERZEICHNIS

ALLGEMEINES

Hier werden kurz die Autoren vorgestellt, die zu dieser Ausgabe beigetragen haben.

Renée Bäcker

Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv - als Moderator bei Perl-Community.de, Organisator des kleinen Frankfurt Perl-Community Workshops, Grant Manager bei der TPF und bei der Perl-Marketing-Gruppe.

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem ruhigen Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber auch schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten, sowie elektronischer und handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er an Kephra, einem Texteditor in Perl. Er war auch am Aufbau der Wikipedia-Kategorie: "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

Breno G. de Oliveira

Breno ist ein Perl-Programmierer aus Rio de Janeiro / Brasilien. Als "Leiter" der Brasilianischen Perlmongers ist er auch Verantwortlich für die jährliche YAPC::Brazil-Konferenz. 2012 hat er für seine Verdienste um die Südamerikanische Perl-Community den White Camel Award bekommen.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996. Websites:
http://www.thomas-fahle.de
http://Perl-HowTo.de

Ulli Horlacher

Ulli "Framstag" Horlacher (43) arbeitet als UNIX-Admin und -Programmierer am Rechenzentrum der Universität Stuttgart. Seine Lieblingssprache entdeckte er vor 20 Jahren mit Perl 3 unter VMS. Internet, Serverbetriebssysteme und Serverdienste sind seine Arbeitsschwerpunkte. Gelegentlich hält er auch (Perl-)Kurse an der Universität Stuttgart. Weitere Perl-UNIX-Software von ihm ist unter http://fex.rus.uni-stuttgart.de/fstools/ zu finden. Seine Freizeit verbringt er meistens mit Fahrrad- bzw. Tandemfahren und dem Bau von innovativer Illuminationshardware: http://tandem-fahren.de/Mitglieder/Framstag/LED/

Sawyer X

Sawyer X benutzt seit mittlerweile fast 10 Jahren Perl. Er arbeitet als Perl-Entwickler und Systemadministrator. Er ist Verantwortlich für die Organisation der Tel Aviv Perlmongers Treffen. Mittlerweile gehört er zum Kernteam der Dancer-Entwickler.

Renée Bäcker, Breno G. de Oliveira

Ist Dein Code Perl-5.18-sicher? - Mehr zum Thema Hashes

In den P5P-News dieser Ausgabe wurde es erwähnt: Das Hashing in Perl 5.18 wird anders sein als bisher und wer sich auf ein undokumentiertes Feature verlässt, hat eventuell ein Problem. Der Großteil dieses Artikels basiert auf einem Blogeintrag von Breno G. de Oliveira unter http://onionstand.blogspot.com.br/, mit dessen Erlaubnis die Codebeispiele übernommen werden durften.

Mit dem Hashing wird die Reihenfolge von Hash-Einträgen bestimmt. Schon in Perl 5.8.1 wurde diese Reihenfolge "zufälliger". Wie vermutlich alle wissen, ist die Reihenfolge von Hash-Einträgen nicht vorhersagbar. Ende 2011 wurde in vielen Programmiersprachen eine Schwachstelle gefunden, die durch die fehlende Zufälligkeit der Sortierung eine DoS-Attacke möglich machte. In Perl gab es diese Schwachstelle aber nicht bzw. wurde schon 2003 gefixt.

Yves Orton hat sich zu dem Thema weitere Gedanken gemacht und Perl so angepasst, dass sich die Sortierung von Perl-Prozess zu Perl-Prozess ändert. Bisher hat Perl einen Hash-Seed verwendet, der bei der Kompilierung von Perl gesetzt wurde. Alle Hashes haben diesen Wert benutzt um die Hash-Werte zu berechnen. Wenn es Kollisionen gab, wurden die Hash-Werte neu berechnet. Das hat natürlich auch Auswirkungen auf den Speicherverbrauch und die Performanz.

Nach den Änderungen von Orton erzeugt jeder Prozess einen neuen, zufälligen Seed. Das macht Perl sicherer, hat aber den Nebeneffekt, dass sich die Reihenfolge der Einträge häufiger ändert.

So ganz zufällig ist die Reihenfolge aber auch in den bisherigen Perl-Versionen nicht. Einfach mal

```
$ perl -E 'local $,=q[, ];
  $hash{$_} = $_ for 1..15; say keys %hash'
```

mehrmals aufrufen. Es wird jedes Mal die gleiche Reihenfolge ausgegeben - siehe Listing 1.

Diese Reihenfolge dürfte aber bei den meisten Lesern eine andere sein.

Es gibt genügend Code, der sich auf diese Reihenfolge verlässt.

```
use feature 'say';

my %hash = map { $_ => $_ } 3 .. 10;
my @keys = keys %hash;

say "yes" if $keys[0] == 6;
__END__
$ perl buggy.pl
yes
$ perl buggy.pl
yes
$ perl buggy.pl
yes
$ perl buggy.pl
yes
$ perl buggy.pl
yes
```

Auch wenn seit Perl 5.8.1 in der Dokumentation steht, dass die Reihenfolge der Einträge zufällig ist, haben sich manche Entwickler auf diese zwar zufällige, aber doch konstante, Reihenfolge verlassen, weil es auf ihrer Maschine so funktioniert hat.

```
$ perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
6, 11, 3, 7, 9, 12, 2, 15, 14, 8, 1, 4, 13, 10, 5
$ perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
6, 11, 3, 7, 9, 12, 2, 15, 14, 8, 1, 4, 13, 10, 5
$ perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
6, 11, 3, 7, 9, 12, 2, 15, 14, 8, 1, 4, 13, 10, 5
$ perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
6, 11, 3, 7, 9, 12, 2, 15, 14, 8, 1, 4, 13, 10, 5
```

Listing 1

Erst durch das Hinzufügen weiterer Hash-Einträge wurde die Reihenfolge neu berechnet (siehe Listing 2).

Selbst wenn der neu hinzugefügte Wert wieder gelöscht wird, wird die ursprüngliche Reihenfolge nicht wiederhergestellt (siehe Listing 3).

Das kann immer passieren, auch wenn die Variable wiederverwendet wird:

```
sub init { ( 1=>1, 2=>2, 3=>3, 4=>4, 5=>5 ) }

my %hash = init();
say "original: " . join ', ' => keys %hash;
$hash{$_} = $_ for 6..100;

%hash = init(); # restores original values
say "original? " . join ', ' => keys %hash;
```

Diese Ausgabe gibt es mit meinem Perl 5.14.3:

```
original: 4, 1, 3, 2, 5
original? 2, 1, 3, 4, 5
```

Das zeigt anschaulich, dass das Problem jetzt schon im Code lauert. Aber Dank der Änderungen im Perl-Kern, wird der Code noch schneller zum Problem (siehe Listing 4).

Fällt der Unterschied zu den Beispielen am Anfang auf? Jetzt ist die Ausgabe von Lauf zu Lauf unterschiedlich. Das ist nicht nur aus der Sicht der Sicherheit wichtig und gut, sondern offenbart auch problematischen Code, weil der jetzt schneller auf die Nase fällt.

Man sollte also seinen Code prüfen ob er anfällig ist. Hier zeigt sich wieder ein großer Vorteil von einer gelebten Kultur des Testens. Unittests sollten anfälligen Code schnell ausfindig machen.

Wenn man nicht so viele Tests hat, kann man nach den Funktionsaufrufen von `keys()`, `values()` oder `each()` suchen und sicherstellen, dass die Reihenfolge der Elemente nicht wichtig ist. Folgendes ist aber absolut ok:

```
my @keys   = keys %hash;
my @values = values %hash;
say "hash key $keys[3] is $values[3]";
```

Es ist in der Dokumentation von Perl (siehe `perldoc -f values`) festgeschrieben, dass sich `keys` und `values` auf die gleiche Reihenfolge bezieht, wenn am Hash nichts verändert wurde.

Folgendes ist aber problematisch:

```
if ($keys[0] eq 'some_key') {
   ...
}
```

Es gibt keinen Weg mit dem man die Reihenfolge der Liste festlegen kann. Der Code könnte bisher funktioniert haben. Das wäre aber reiner Zufall, weil es gerade auf der Maschine des Entwicklers so passt.

Problematischen Code kann man nicht so einfach entdecken, wenn er nicht direkt verwendet wird. Z.B., weil der Code, der den Hash verändert, in ein Modul ausgelagert ist:

```
my ($name, $age, $rate) =
 Some::Module->new->get_list( 'some_user' );
```

```
$ perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash; $hash{99} = 99;
   say keys %hash'
6, 11, 3, 7, 9, 12, 2, 15, 14, 8, 1, 4, 13, 10, 5
11, 7, 2, 99, 1, 13, 6, 3, 9, 12, 14, 15, 8, 4, 10, 5
```

Listing 2

```
$ perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash; $hash{99} = 99;
  delete $hash{99}; say keys %hash'
6, 11, 3, 7, 9, 12, 2, 15, 14, 8, 1, 4, 13, 10, 5
11, 7, 2, 1, 13, 6, 3, 9, 12, 14, 15, 8, 4, 10, 5
```

Listing 3

```
> perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
  1, 5, 15, 12, 6, 4, 10, 9, 3, 13, 7, 14, 11, 2, 8
> perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
  5, 11, 7, 3, 15, 6, 12, 2, 13, 9, 8, 14, 10, 1, 4
> perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
  2, 15, 14, 13, 5, 1, 9, 10, 3, 11, 6, 8, 12, 4, 7
> perl -E 'local $,=q[, ]; $hash{$_} = $_ for 1..15; say keys %hash'
  8, 2, 14, 10, 1, 9, 4, 3, 6, 15, 5, 13, 7, 12, 11
```

Listing 4

Wobei die Funktion so aussieht:

```
sub get_list {
  my ($self, $username) = @_;
  return values $self->{data}{$username};
}
```

Fix den Code

Den anfälligen Code zu fixen ist in der Regel sehr einfach. Anstatt nur eine Funktion wie `keys()`, `values()` oder `each()` zu verwenden, sollte man die Werte sortieren. Also aus `... for keys %hash` einfach `... for sort keys %hash` machen.

Zusammenfassung: Wenn du Perl 5 verwendest und planst auf Perl 5.18 umzusteigen, solltest du folgendes tun:

- Installiere ein Perl > 5.17.6 mit `perlbrew`

- Teste alle Deine Module und Anwendungen mit dem neuen Perl

- Wenn irgendwas nicht funktioniert, was vorher funktioniert hat, verlässt du dich wahrscheinlich darauf, dass `keys()`, `values()` oder `each()` die Werte in einer bestimmten Reihenfolge liefern. Benutze `sort()` um die Reihenfolge von Werten festzulegen.

- Wenn ein Modul von CPAN, das Du benutzt, die Ursache ist, schreibe dem Autor oder der Autorin eine Mail.

Zum letzten Punkt: Unter *https://rt.perl.org/rt3/Public/Bug/Display.html?id=115908* ist ein RT-Ticket zu finden, in dem die Perl 5 Porters die Module auflisten, die bekanntermaßen dieses Problem haben. Diese Autoren wurden bereits angeschrieben.

Weitere Infos

Mit seinen Änderungen geht Orton sogar noch weiter, denn beim Kompilieren von Perl kann man angeben, welcher Hashing-Algorithmus verwendet werden soll:

In der Datei *hv.h* findet sich folgender Code:

```
/* Uncomment one of the following lines
   to use an alternative hash algorithm.
#define PERL_HASH_FUNC_SDBM
#define PERL_HASH_FUNC_DJB2
#define PERL_HASH_FUNC_SUPERFAST
#define PERL_HASH_FUNC_MURMUR3
#define PERL_HASH_FUNC_SIPHASH
#define PERL_HASH_FUNC_ONE_AT_A_TIME
*/
```

Hier werden die sechs möglichen Hashing-Algorithmen aufgeführt, die verwendet werden können. Die Vor- und Nachteile sowie Benchmarks werden hier nicht gezeigt. Als Standard wird der Murmur3-Algorithmus verwendet, der laut Yves Orton bei kurzen Strings genauso schnell ist wie der bisherige Algorithmus, bei langen String aber bis zu doppelt so schnell.

Eine Konfigurationsoption für `Configure` gab es bei Redaktionsschluss noch nicht.

Dass verschiedene Hashing-Funktionen unterstützt werden, hat noch einen weiteren Vorteil: Wenn eine der Funktionen als unsicher gegenüber Attacken gilt und man diese Funktion verwendet, muss man einfach nur eine andere Funktion aktivieren und sein Perl neu kompilieren. Man muss also nicht auf einen Patch warten.

Hash::Util

Orton hat zusätzlich noch das Modul `Hash::Util` um vier Funktionen erweitert:

- `hash_value()` liefert den Hashwert (Integer) für einen String
- `bucket_info()` liefert Basisinfos über Hash-Buckets
- `bucket_stats()` liefert weitergehende Infos über Hash-Buckets
- `bucket_array()` zeigt, welche Schlüssel in welchem Bucket im Hash sind

Ein Bucket ist ein Container, in dem die Schlüssel gespeichert werden. Wenn man ein Telefonbuch als Hash sieht, dann ist z.B. das "Registerblatt 'A'" ein Bucket, in dem die Namen (Schlüssel) gespeichert werden.

```
use Data::Dumper;

BEGIN {
    die "Need Perl >= 5.17.6!"
        if $] < 5.017006;
}

use Hash::Util qw(
    bucket_info hash_value
    bucket_stats bucket_array
);

use feature 'say';

my %hash = map{ $_ => $_ } 2..3;
```

Als erstes schauen wir uns `bucket_info()` an:

```
my @info = bucket_info( \%hash );
say Dumper \@info;
```

In dem Array sind jetzt Informationen darüber zu finden, wie viele Schlüssel in dem Hash vorhanden sind. Das nächste Element ist die Anzahl der vorhandenen Buckets, darauf folgt die Anzahl der Buckets, in denen die Schlüssel zu finden sind. Danach folgt eine Auflistung wie viele Buckets es mit x Elementen gibt:

```
$VAR1 = [
          2,
          8,
          1,
          7,
          0,
          1
        ];
```

Hier gibt es zwei Schlüssel im Hash. Insgesamt gibt es acht Buckets, wobei nur ein Bucket mit Schlüsseln belegt ist. Also gibt es sieben Buckets mit 0 Schlüsseln, 0 Buckets mit einem Schlüssel und ein Bucket mit zwei Schlüsseln.

Gibt man den Hash im Skalaren Kontext aus

```
say %hash . "";
```

erscheint eine Ausgabe wie `1/8` - was gerade bei Perl-Einsteigern immer wieder für Verwirrung sorgt. Die erste Zahl ist dabei die Anzahl der verwendeten Buckets und die zweite Zahl die Anzahl der Buckets insgesamt im Hash.

Holt man sich noch Informationen über `bucket_array()`

```
my $array = bucket_array( \%hash );
say Dumper $array;
```

- hier eine Beispielausgabe -

```
$VAR1 = [
    2,
    [
        '3',
        '2'
    ],
    5
];
```

bekommt man eine genaue Aufstellung der Buckets und der darin enthaltenen Schlüssel. Wenn das Element eine Zahl ist, dann gibt das die Anzahl der freien Buckets an, eine Arrayreferenz zeigt die Schlüssel in dem Bucket. In diesem Beispiel gibt es erst zwei freie Buckets, dann ein Bucket mit den Schlüsseln "2" und "3" und danach wieder fünf freie Buckets.

Da man mit der Ausgabe sensitive Informationen über den Aufbau des Hashs preisgibt und damit eine DoS-Attacke vereinfacht, sollte man sie nur zum Debuggen verwenden.

```
my @stats = bucket_stats( \%hash );
say Dumper \@stats;
```

Mit `bucket_stats` bekommt man neben den schon bekannten Daten noch weitergehende Informationen zu den Buckets. Dazu zählt, wie viel Prozent der Buckets belegt sind und wie viel Prozent der Schlüssel Kollisionen verursacht haben. Auch die durchschnittliche Bucket-Länge gehört dazu.

```
$VAR1 = [
          2,
          8,
          1,
          '0.125',
          '0.5',
          '0.25',
          '0.661437827766148',
          7,
          0,
          1
        ];
```

Eine weitere Information, die man mit `Hash::Util` bekommen kann, ist der Hash-Wert für einen String.

```
my $hash_value = hash_value(
                   '$foo Perl-Magazin' );
say $hash_value;
# 247278595
```

Thomas Fahle

How To XXV

Überprüfung von Nummerncodes mit Algorithm::CheckDigits

Algorithm::CheckDigits - *Perl extension to generate and test check digits* - von Mathias Weidner vereinfacht die Überprüfung von Nummerncodes, wie Umsatzsteuer-Identifikationsnummern, ISBN, Betriebsnummern, Blutbeutel-Eurocodes und zahlreichen mehr. Dabei bietet Algorithm::CheckDigits eine konsistente Schnittstelle zu allen Berechnungsmethoden an.

Mit `CheckDigits()` wird das gewünschte Berechnungsverfahren gewählt und man erhält das passende Objekt zurück. Die Methode `is_valid()` prüft den Nummerncode, `basenumber()` liefert die Basiszahl des Nummerncodes und `checkdigit()` liefert die Prüfziffer für die Basiszahl zurück.

Alle verfügbaren Berechnungsmethoden auflisten

Eine Übersicht aller verfügbaren Berechnungsmethoden liefert `method_list`:

```
#!/usr/bin/perl
use strict;
use warnings;

use Algorithm::CheckDigits;

my @ml =
  Algorithm::CheckDigits->method_list();

foreach my $method ( @ml ) {
      print "$method\n";
}
```

Das Programm liefert die folgende Ausgabe (deutlich gekürzt):

```
2aus5
ahv_ch
amex
bahncard
betriebsnummer
blutbeutel
...
ustid_at
ustid_be
ustid_de
...
verhoeff
visa
wagonnr_br
```

Beispiel: Umsatzsteuer-Identifikationsnummer

Dieses Beispiel testet eine deutsche Umsatzsteuer-Identifikationsnummer (`ustid_de`).

```
#!/usr/bin/perl
use strict;
use warnings;

use Algorithm::CheckDigits;

my $ustid_de = CheckDigits('ustid_de');

#Umsatzsteuernummer: DE175903868
my $ustid = '175903868';

if ( $ustid_de->is_valid($ustid) ) {
  print "UmsatzsteuerID: $ustid ist okay.\n";
  my $cd = $ustid_de->checkdigit($ustid);
  my $bn = $ustid_de->basenumber($ustid);
  print "Checkdigit: $cd\n";
  print "Basiszahl:  $bn\n";

}
else {
  print
  "UmsatzsteuerID: $ustid ist NICHT okay.\n";
}
```

Das Programm liefert folgende Ausgabe:

```
UmsatzsteuerID: 175903868 ist okay.
Checkdigit: 8
Basiszahl: 17590386
```

Umsatzsteuer-Identifikationsnummern aus Österreich lassen sich mit der Methode `ustid_at` checken. Ebenso lassen sich Umsatzsteuer-Identifikationsnummern aus den Niederlanden mittels `ustid_nl` oder aus Belgien via `ustid_be` testen.

Beispiel: Internationale Standardbuchnummer (ISBN)

Das folgende Beispiel testet eine 10-stellige ISBN (`isbn`). 13-stellige ISBN lassen sich mit `isbn13` prüfen.

```
#!/usr/bin/perl
use strict;
use warnings;

use Algorithm::CheckDigits;

my $isbn10_cd = CheckDigits('isbn');

# Einführung in Perl
my $isbn = '386899145X';

if ( $isbn10_cd->is_valid($isbn) ) {
   print "ISBN-10: $isbn ist okay.\n";
   my $cd = $isbn10_cd->checkdigit($isbn);
   my $bn = $isbn10_cd->basenumber($isbn);
   print "Checkdigit: $cd\n";
   print "Basiszahl:  $bn\n";
}
else {
   print "ISBN-10: $isbn ist NICHT okay.\n";
}
```

Das Programm liefert folgende Ausgabe:

```
ISBN-10: 386899145X ist okay.
Checkdigit: X
Basiszahl: 386899145
```

Weitere Nummerncodes

Algorithm::CheckDigits stellt aktuell **127** Testverfahren nach dem gezeigtem Schema bereit. Ich empfehle einen Blick in die Dokumention: Der gesuchte Nummerncode ist mit hoher Wahrscheinlichkeit bereits implementiert.

Ullrich Horlacher

SEX: Stream EXchange mit F*EX

In $foo #09 (1/2009) wurde F*EX (Frams' Fast File EXchange) [1] vorgestellt, mit dem beliebig große Dateien an beliebige Empfänger verschickt werden können. Sender wie Empfänger benötigen dazu nur einen Webbrowser und ein E-Mail-Programm, egal welcher Art und egal auf welchem Betriebssystem. Die gesamte Intelligenz sitzt im perligen F*EX Server.

Wenn man auf Client-Seite UNIX einsetzt, kann man mit F*EX aber noch viel mehr machen:

- SEX: Stream EXchange
- streaming F*EX
- F*EX yourself
- anonymous F*EX

head2 SEX

Was unterscheidet einen Datenstrom (Stream) von einer Datei (File)? Im Gegensatz zu einer Datei hat ein Datenstrom keine bekannte Größe und keinen Namen und er kann nur sequentiell vorwärts gelesen werden.

Datenströme kennt der normale Anwender meist nur als Konsument von Audio- oder Videostreams. Auf UNIX ist es aber für jeden Benutzer möglich einfach selber beliebige Streams zu erzeugen mittels Pipe:

```
programm_1 | programm_2
```

Hierbei wird die Ausgabe von `programm_1` isochron (mit minimaler Pufferung) an `programm_2` weitergeleitet, das diese Daten als Eingabe ohne Dateizwischenspeicherung benutzt. Das spart Zeit und Speicherplatz und man muss keine temporären Hilfsdateien hinterher löschen. Pipes sind eine DER zentralen Eigenschaften von UNIX-Systemen.

Wenn man Pipes zwischen zwei unterschiedlichen Maschinen benutzen will, braucht man Hilfsprogramme wie *rsh* oder *ssh*, die Daten übertragen und für Authentifizierung des Benutzers sorgen.

Was aber, wenn Maschine 1 aufgrund von NAT, Firewall-Restriktionen, o.ä. keine direkte Verbindung zu Maschine 2 aufbauen kann? Dann hilft SEX weiter!

Selbst bei restriktiven Firewall-Einstellungen und auch bei NAT ist es üblich, dass ein System von sich aus HTTP-Verbindungen zu anderen Servern aufbauen kann, denn sonst "ist das Internet kaputt" (beliebte Benutzer-Fehlermeldung beim Support).

Jeder F*EX Server ist nicht nur ein HTTP Server, sondern hat auch Unterstützung für Streams eingebaut. Dabei wird die vorhandene Authentifizierung [3] benutzt. SEXing funktioniert erst mal nur an registrierte Benutzer.

Der Sender meldet seinen Stream beim F*EX Server mit `sexsend` [2] an und danach kann ihn der (registrierte) Empfänger dort mit `sexget` [2] abholen. Die Übertragung verläuft dabei im Gegensatz zu regulärem F*EX nicht nur ohne Zwischenspeicherung, sondern auch synchron, d.h. Sender und Empfänger müssen zur selben Zeit aktiv sein.

Anwendungsbeispiel: Der Administrator schickt in Echtzeit das *ipfilter*-Log an den Hiwi, ohne dass er diesem direkten Zugang zum sicherheitskritischen Server geben muss.

Sender:

```
root@flupp# tail -f /var/log/ipfilter.log |
   sexsend hiwi
```

Empfänger:

```
hiwi@bunny: sexget
```

Die Daten laufen dabei vom Server `flupp` zum F*EX Server und dann weiter zur Empfänger-Maschine `bunny`. Zur Erhöhung der Sicherheit kann die Übertragung auch SSL-verschlüsselt werden oder die Kommunikationspartner setzen ein beliebiges sonstiges Verschlüsselungsprogramm in der Pipe ein, wie z.B. *gpg* oder *blowfish*.

Welcher F*EX-Server verwendet wird und ob HTTP oder HTTPS verwendet wird, steht in der Datei `$HOME/.fex/id` oder in der Prozess Umgebungsvariable `FEXID`, die der Anwender initial setzen muss. Am einfachsten geht das mit `fexsend -I`, das alle notwendige Daten abfragt und in `$HOME/.fex/id` speichert, das dann von allen F*EX Client Programmen ausgewertet wird.

Ein weiteres Anwendungsbeispiel aus der Praxis: eine Kollegin musste den Webserver auf eine neue schnellere Maschine umziehen. Der eigentliche Webserver (Apache) war kein Problem, aber die 400 GB an Nutzerdaten. Der neue Webserver stand sowohl physisch als auch netzwerktoplogisch weit entfernt, so dass eine direkte Verbindung nicht möglich war.

Mittels SEX war die Übertragung aber einfach:

Senderseite:

```
tar cvf - /srv/www | sexsend webmaster
```

Empfängerseite:

```
sexget | tar xvf -
```

Weil Dateitransfer via SEX ein häufiger Anwendungsfall ist, gibt es das Zusatzprogramm `sexxx`, das das Ein- und Auspacken mittels `tar` gleich mitmacht. So verkürzt sich die Anwendung zu:

Senderseite:

```
sexxx /srv/www
```

Empfängerseite:

```
sexxx
```

Public SEX

Normalerweise muss sich bei SEX der Empfänger authentifizieren. Hat dieser aber keinen F*EX Account, so kann der Sender *public SEX* verwenden, wobei er sich dann authentifizieren muss [3]. Als Empfänger gibt er den Pseudobenutzer *public* an und bekommt von `sexsend` daraufhin eine URL angezeigt, unter der der Stream abgeholt werden kann, z.B. mit `wget` oder `curl`. Der Sender muss dem Empfänger nur noch diese Stream-URL mitteilen.

Anonymous SEX

Was aber, wenn weder Sender noch Empfänger auf ihren Accounts eine gültige F*EX-ID haben? Jeden Account extra zu registrieren kann schnell lästig werden, wenn man ein paar hundert Server zu betreuen hat. In dem Fall kann der F*EX-Administrator die IP-Adressen der betreffenden Server freischalten für *anonymous SEX*.

Sender wie Empfänger geben dazu `FEXURL/anonymous` als Zieladresse an. Beispiele für Sender:

```
 ... | sexsend fex/anonymous
 ... | sexsend
http://fex.rus.uni-stuttgart.de:8080/anonymous
 ... | sexsend
https://fex.rus.uni-stuttgart.de/anonymous
```

Bei allen 3 SEX-Arten kann man optional noch einen Stream-Namen mit angeben, um mehrere Übertragungen parallel durchführen zu können. Beispiel:

```
 ... | sexsend fex/anonymous quarktasche
 sexget fex/anonymous quarktasche | ...
```

*streaming F*EX*

Ein Nachteil von SEX sollte nicht verschwiegen werden: Es kann nach Verbindungsabbruch nicht wiederansetzen, weil der interne Lesezeiger im Datenstrom nicht frei positionierbar (seek) ist. Deshalb eignet sich SEX nur für stabile Netzwerke, z.B. innerhalb des LAN oder in besonders stabilen Hochgeschwindigkeitsnetzen wie BelWü [4].

Was aber, wenn diese Bedingung nicht erfüllt ist und man trotzdem VIELE Dateien verschicken möchte? Dann hilft streaming F*EX!

Ein reales Beispiel:

Ein deutsches Großforschungsinstitut musste 14.000 Dateien à 100 MB an eine australische Universität verschicken. Die Daten lagen auf einem System ohne Internetanschluss, aber mit Verbindung zum F*EX-Server, allerdings mit knappem lokalen Plattenplatz, der die Erzeugung eines temporären Transfercontainers nicht zuließ. Die Lösung lautete dann:

```
fexsend -a labdata.tar
  labfile* wombat@down.under.au
```

Mit diesem Befehl erzeugt der F*EX-Client `fexsend` [2] *on-the-fly* ein 1.2 TB großes Archiv labdata.tar, in dem alle Dateien stecken und überträgt dieses an den F*EX-Server. Letzterer braucht natürlich entsprechend Speicherplatz. Der Empfänger wombat@down.under.au bekommt automatisch eine Benachrichtigungs-E-Mail und kann sich dann die Forschungsdaten abholen, z.B. mit `fexget` [2], `wget` oder `curl`.

Verbindungsabbrüche beim Hoch- oder Herunterladen bleiben folgenlos, weil die F*EX-Clients automatisch wiederansetzen, ohne dass es zu Datenverlust oder Mehrfachsendungen kommt.

*F*EX yourself*

```
<TV-Verkaufsshow>
  Marianne, kennst du schon xx aus dem
    Hause F*EX?
  Nein, Michael, was kann man denn damit
    machen?
  Noch schneller und eleganter Dateien
    transferieren als mit fexsend!
  Nein, Michael, ist das denn moeglich!?
    Das ist ja UN-GLAUB-LICH!!
</TV-Verkaufsshow>
```

Soweit der kleine kommerzielle Einblender.

Jeder Administrator einer größeren Einrichtung hat viele Accounts auf den verschiedensten Servern, die unterschiedliche Namen und Rechte haben, u.a. auch administrative Accounts wie *root* oder *webmaster*. Da will man oft "schnell mal" Dateien kopieren.

`scp` und `sendfile` gehen nicht überall wegen Firewall-Einstellungen und Portsperren auf Routern, auf die man selber oft keinen Einfluss hat. Verschlimmert wird das Problem noch durch DHCP und NAT.

Mit `fexsend` [2] geht das schon mal ganz gut, aber noch nicht schnell und effizient genug. Deshalb steckt im `fexsend` auch das Programm `xx` (als symbolischer Link). Damit geht der Dateiaustausch zwischen eigenen Accounts sehr einfach wenn eine F*EX-ID vorhanden ist [3].

Beispiel Eintüten:

```
root@ptm1:~# xx /etc/cups
making tar transfer
  file /root/.fex/tmp/STDFEX :
cups/snmp.conf
cups/cupsd.conf
cups/client.conf
cups/printers.conf
cups/mime.types
cups/mime.convs
cups/ppd/VMware_Virtual_Printer.ppd
/root/.fex/tmp/STDFEX : 10 kB in 0 s
  (15 kB/s)
```

Beispiel Austüten siehe Listing 1.

Es kann beliebig oft ausgetütet werden, bis zum konfigurierten Verfallsdatum (expire).

Somit kann man sehr einfach Dateien schnell auf viele Accounts verteilen, auch über Firewall-Grenzen hinweg, wenn denn überall eine F*EX-ID vorhanden ist. Wenn dem nicht so ist, kann man auf *anonymous F*EX* zurückgreifen: Wie beim *anonymous SEX* wird als Empfänger der Pseudobenutzer *anonymous* verwendet, wozu es keine F*EX-ID braucht.

Auch hier steht wieder ein Spezial-Script (`afex`) zur Verfügung, das die Handhabung vereinfacht:

```
root@diaspora:/etc: xx
transferred: 10 kB (100%)
Files in transfer-container:

-rw-r----- root/lp         186 2011-09-21 18:43 cups/snmp.conf
-rw-r----- root/lp        4238 2011-09-21 18:43 cups/cupsd.conf
-rw-r--r-- root/root      2777 2011-09-21 18:43 cups/client.conf
-rw------- root/lp         111 2012-04-18 00:31 cups/printers.conf
-rw-r--r-- root/root      6536 2011-09-21 18:43 cups/mime.types
-rw-r--r-- root/root      4762 2011-09-21 18:43 cups/mime.convs
lrwxrwxrwx root/root      4299 2012-04-18 00:31 cups/ppd/VMware_Virtual_Printer.ppd

Extract these files? [Yn]
```

Listing 1

```
framstag@diaspora:/tmp: afex 300879
afex_300879.tar: 90 MB in 1 s (90229 kB/s)
Files in archive:
drwxr-xr-x root/root         0 2006-08-17 14:19 opt/tivoli/
drwxr-xr-x root/root         0 2006-08-17 14:19 opt/tivoli/tsm/
drwxr-xr-x root/root         0 2006-08-17 14:19 opt/tivoli/tsm/tivinv/
-r-xr-xr-x root/root         7 2006-08-17 14:19 opt/tivoli/tsm/tivinv/TSMBACL05030400.SYS
(...)
-r-xr-xr-x root/bin 2547335 2009-03-20 10:40 opt/tivoli/tsm/client/api/bin64/libApiTSM64.so
extract these files (Y/n)?
```

Listing 2

Eintüten:

```
root@obertux:/# afex /opt/tivoli
Making fex archive (afex_300879.tar):
Archive size: 90 MB
/opt/tivoli/tsm/
/opt/tivoli/tsm/tivinv/
/opt/tivoli/tsm/tivinv/TSMBACL05030400.SYS
(...)
/opt/tivoli/tsm/client/api/bin/libApiDS.so
afex_300879.tar: 90 MB in 1 s (87287 kB/s)
http://fex.rus.uni-stuttgart.de//afex_
                                  300879.tar
afex 300879
```

Austüten siehe Listing 2.

Steht auf dem Zielsystem kein `afex`-Programm zur Verfügung, kann auch die URL verwendet werden, die das Sende-`afex` ausgegeben hatte. Also z.B. so:

```
wget -O-
 http://fex.rus.uni-stuttgart.de//
                              afex_300879.tar
| tar xvf -
```

Im Gegensatz zu `xx` kann man bei `afex` aber auch das Dateipaket an Dritte weiterreichen, indem man ihnen die `afex`-ID oder -URL gibt.

Aus Sicherheitsgründen funktioniert *anonymous F*EX* wie *anonymous SEX* nur auf den Systemen, deren IP der F*EX-Administrator freigeschaltet hat. Typischerweise macht man dies für alle LAN-Adressen. Wenn man dann noch die F*EX-Client-Programme zentral installiert, können alle lokalen UNIX-Benutzer sofort (anonym) losfexen mit allen typischen F*EX-Vorteilen:

- Datentransfer über NAT- und Firewall-Grenzen hinweg
- Wiederaufsetzen beim zuletzt übertragenen Byte nach Verbindungsabbruch
- Automatisches Löschen nach Verfallsdatum

Zur Implementation

Wie der Server sind auch die Client-Programme [2] alle in Perl programmiert und damit nicht nur klein und schnell, sondern auch noch portabel. Es werden nur Module verwendet, die im Perl-Core enthalten sind. Die Clients sind also direkt lauffähig auf allen UNIX-Systemen. Der Server benötigt als einzige Abhängigkeit noch *xinetd* und einen *sendmail*-kompatiblen MTA wie z.B. *postfix*.

Der für SEX zuständige Servercode umfasst nur 5 kB, auf Clientseite sind es 13 kB.

Der eigentliche Streaming-Code auf Serverseite besteht dabei nur aus je einer Zeile:

Empfangen:

```
while (sysread($fifo,$_,$bs))
  { syswrite STDOUT,$_ }
```

Senden:

```
while (sysread(STDIN,$_,$bs))
  { syswrite $fifo,$_ }
```

Der restliche Code ist im Wesentlichen für Authentifizierung, Fehlerbehandlung und Logging da.

Bei "normalen" Webservern ist der eigentliche Webserver zuständig für Netzwerk-IO und HTTP Header, während das CGI(-Skript) nur noch den Inhalt der HTTP-Anfrage (HTTP-Body) abarbeitet. Nicht so beim F*EX-Server. Hier übergibt der Webserver `fexsrv` die Kontrolle vollständig an das CGI und läuft nicht mehr als vermittelnder Prozess mit. Das CGI muss also selber "HTTP sprechen". Diese andere Architektur wurde gewählt, weil damit ein deutlich größerer Datendurchsatz zu erreichen war. Eine erste Implementation mit Apache und dem Perl-Modul CGI erbrachte nur 20 MB/s, mit dem neuen `fexsrv` waren auf derselben Hardware dann 400 MB/s möglich.

Da der F*EX-Server nur Dateitransfer und Streaming machen muss, hielt sich die HTTP-Programmierung in Grenzen. Es musste also nicht das umfangreiche CGI.pm (250 kB) komplett nachprogrammiert werden, sondern nur wenige Routinen wie HTTP-Antwort und Parameterauswertung.

Probleme machten beim Dateitransfer vor allem die inkompatiblen und fehlerhaften Webbrowser. Es gibt z.B. keinen einzigen Webbrowser, der mit mehr als 4 GB POST-Daten umgehen kann. Die haben da ALLE einen 32-bit-Bug. Das betrifft aber eben nur die Benutzung mit Webbrowsern. Bei Verwendung eigener Clients treten diese Probleme nicht auf. Perl ist schon lange 64-bit-fähig.

Zudem ist ein Webbrowser eine Datensenke, der nur Daten abspeichern und nicht an beliebig andere Programme weitergeben kann (Pipelining).

Quellen / Anmerkungen

[1] http://fex.rus.uni-stuttgart.de:8080/

[2] http://fex.rus.uni-stuttgart.de/tools.html

[3] F*EX-ID ist entweder die Datei `$HOME/.fex/id` oder die Umgebungsvariable `FEXID`. Erzeugt wird sie mit `fexsend -I`.

[4] http://fex.belwue.de/

Renée Bäcker

Mojolicious Tutorial - Teil 2

In der vergangenen Ausgabe wurde gezeigt, wie man eine Mojolicious-Anwendung beginnt und wie man das Routing einrichtet. Außerdem wurde gezeigt, wie man mit Platzhaltern diese Routen dynamisch hält. Templates waren ein weiteres Thema: Wo liegen die Templates, wie sieht die Templatesprache bei Mojolicious aus. Als letztes wurde erklärt was der Stash ist und wie man diesen nutzt.

In dieser Ausgabe werden die ersten Aktionen programmiert. Auch ein ersts "Release" wird gemacht. Kurz zur Erinnerung: Wir wollen eine Anwendung entwickeln, bei der Läufer ihre Strecken und Laufzeiten eintragen können. Die Basisstruktur wurde mit Hilfe des `mojo`-Skripts erzeugt.

Schritt 1: Das erste Formular im Browser

Bislang sieht das Hauptmodul noch folgendermaßen aus:

```
package TrackRuns;
use Mojo::Base 'Mojolicious';

# This method will run once at server start
sub startup {
  my $self = shift;

  # Documentation browser under "/perldoc"
  $self->plugin('PODRenderer');

  # Router
  my $r = $self->routes;

  # change namespace of controllers
  $r->namespaces( ['Controller'] );
  $r->get('/')->to('example#welcome');
}

1;
```

Als erstes können wir das Plugin rauswerfen. Wir werden uns später noch mit Plugins beschäftigen. Als erstes werden wir die Route anpassen. Das Ziel der Route soll die Funktion `welcome` im Controller `Guest` sein, dort wird dann einfach eine Willkommensseite ausgegeben.

```
$r->get('/')->to('guest#welcome');
```

Des Weiteren brauchen wir eine Route zum Formular, in dem Benutzer ihre Trainingseinheit eintragen können. Diese Route soll zur Funktion `run` im Controller `User` führen.

```
$r->get('/user/run')->to('user#run');
```

Der Controller *User* sieht dann folgendermaßen aus:

```
package Controller::User;

use Mojo::Base 'Mojolicious::Controller';

sub run {
  my $self = shift;

  $self->render;
}

1;
```

Bis jetzt kein Hexenwerk. Wir machen (noch) nichts weiter als ein Template rendern zu lassen. Aus der letzten Ausgabe wissen wir, dass das Template unter *templates/user/run.html.ep* gesucht wird.

Jetzt haben wir das Hauptmodul angepasst, den Controller sowie das Template erstellt. Das ist die Gelegenheit, die Anwendung ein erstes Mal zu starten. Während der Entwicklung benutzen wir den Server `morbo`. Dazu rufen wir den Server folgendermaßen auf:

```
morbo script/track_runs
```

Im Browser können wir jetzt *http://localhost:3000/user/run* aufrufen und wir bekommen das Formular angezeigt (Abb. 1).

Wir können jetzt auch mal *http://localhost:3000/user/runs* aufrufen, da wir keine Route dafür definiert haben, kommt eine "404"-Seite (Abb. 2). Auf dieser sind alle Routen zu finden, die konfiguriert sind.

Eine weitere Seite, die während der Entwicklung sehr hilfreich sein kann, ist die Debug-Seite. Diese taucht immer dann auf wenn ein Fehler auftaucht. Um diese Seite mal zu Gesicht zu bekommen, reicht es, ein `die 'test';` in die Methode `run` einzubauen. Rufen wir jetzt *http://localhost:3000/user/run* auf, sehen wir die Debug-Seite (Abb. 3)

Im oberen Bereich der Seite bekommt man die Fehlermeldung und die Stelle gezeigt, an der der Fehler aufgetreten ist. Es ist sehr praktisch, dass man ein paar Zeilen vor und nach der fehlerhaften Codestelle angezeigt bekommt. So findet man den Fehler häufig viel schneller.

Unterhalb des Codes ist der Stacktrace zu finden, dieser ist besonders dann nützlich wenn ein Stück Code von mehreren Stellen aus aufgerufen wird.

Des weiteren bietet die Seite noch weitere Informationen über den Request -- welche Request-Methode, welche URL aufgerufen wurde mit welchen Parametern.

Auf der Suche nach Fehlern hilft es auch, möglichst viel zu loggen. Mojolicious liefert ein eigenes Logging-Modul mit und standardmäßig wird der Logger automatisch initialisiert und alle Meldungen werden in eine Logdatei geschrieben (siehe Listing 1).

Die Standardwerte sind wie folgt:

- Logdatei: log/development.log (wenn *log/* existiert)
- Level: debug (es gibt *debug*, *info*, *warn*, *error* und *fatal*)

Abb. 1: Das Formular zur Eingabe des Trainingslaufs

Abb. 2: Beim Aufruf einer nicht-existenten Route angezeigte 404-Seite

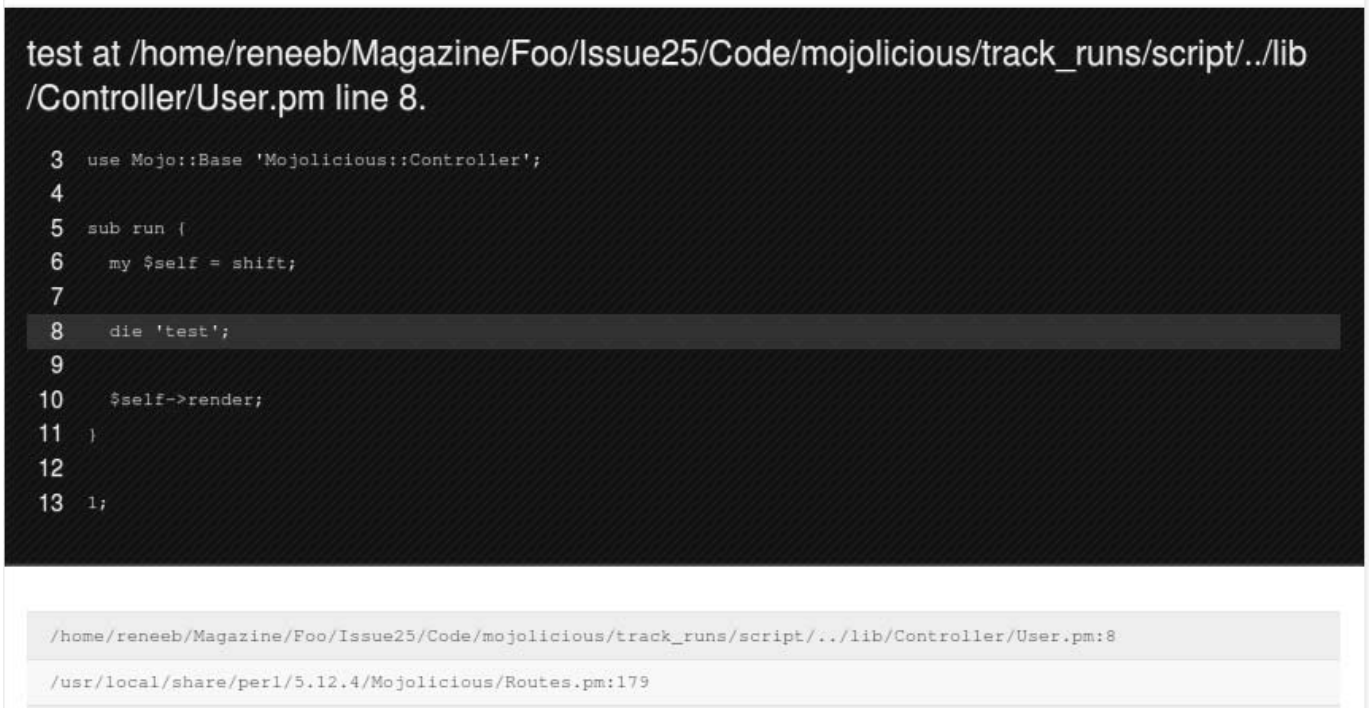


Abb. 3: Debug-Seite von Mojolicious

```
$ tail -f development.log
[Sat Jan 19 02:34:27 2013] [info] Listening at "http://*:3000".
[Sat Jan 19 02:34:35 2013] [debug] Your secret passphrase needs to be changed!!!
[Sat Jan 19 02:34:35 2013] [debug] GET /user/run (Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
   rv:18.0) Gecko/20100101 Firefox/18.0).
[Sat Jan 19 02:34:35 2013] [debug] Routing to controller "Controller::User" and action "run".
[Sat Jan 19 02:34:35 2013] [debug] Rendering template "user/run.html.ep".
[Sat Jan 19 02:34:35 2013] [debug] 200 OK (0.015258s, 65.539/s).
```

Listing 1

Sollen die Standardwerte geändert werden, kann man diese neu setzen:

```
$self->log(
    Mojo::Log->new(
        path  => '/my/dir/log.file',
        level => 'warn',
    );
);
```

Ist der Standard für die Logdatei gewünscht, aber ein anderes Minimumlevel soll gelten, kann man dieses Level einfach mit

```
$self->log->level( 'warn' );
```

anpassen.

Eigene Debugging-Ausgaben kann man dann wie folgt erzeugen:

```
$self->log->debug(
  'Why is this not working?');
$self->log->info('FYI: it happened again.');
$self->log->warn(
  'This might be a problem.');
$self->log->error('Garden variety error.');
$self->log->fatal('Boom!');
```

Schritt 2: Release early, release often

Eine erste Funktion ist fertig. Also wird es Zeit, eine erste Version online zu stellen. Wie man eine Mojolicious-Anwendung bei einem der Cloud-Anbieter veröffentlicht, wird in der nächsten Ausgabe gezeigt. In dieser Ausgabe bedeutet "veröffentlichen" einfach nur ein etwas anderer Umgang mit der Anwendung.

Der erste Unterschied liegt schon im verwendeten Webserver. In der letzten Ausgabe wurde schon auf den Unterschied zwischen `hypnotoad` und `morbo` eingegangen. Die ganzen Features die während der Entwicklung ganz nützlich sind, brauchen wir in einer Produktivumgebung nicht. Dafür bietet hypnotoad zum Beispiel die Möglichkeit des "hot deployments".

Also starten wir die Anwendung

```
hypnotoad script/track_run
```

Erinnern wir uns kurz an die Entwicklungsumgebung: Rufen wir jetzt eine nicht-existierende Datei oder Route auf, bekommen wir die gleichen Fehlerseiten wir schon bei der Entwicklung. Das gleiche wenn ein Fehler auftritt. Da mit diesen Debug-Seiten aber zu viele Informationen (z.B. auch die Konfiguration) an den Endbenutzer -- und damit möglicherweise auch an "Böse Buben" -- gelangen, sollte man die Seiten anpassen.

Hier kommt dann ein "Trick" zum Tragen: Unterschiedliche Modi

Zu Beginn der Entwicklung befindet man sich im Modus "development". Aus diesem Grund heißt die Logdatei auch "development.log". Tritt ein Fehler auf oder wird eine fehlende Route aufgerufen, rendert Mojolicious das Template *exception.development.html.ep* bzw. *not_found.development.html.ep*, die mit Mojolicious mit ausgeliefert werden.

Einfach die Templates anpassen ist nicht die richtige Lösung, denn dann kann man die Vorteile der Seiten während der Entwicklung nicht nutzen. Viel besser ist es, von development auf production umzustellen und dementsprechend *exception.production.html.ep* bzw. *not_found.production.html.ep* erstellen.

Die Umstellung des Modus' erfolgt über die Umgebungsvariable `MOJO_MODE`.

```
MOJO_MODE=production
```

Da man jetzt keine Informationen von den Benutzern bekommen kann was denn schiefgegangen ist, sollte man Vorkehrungen treffen dass man informiert wird. Was man da machen kann, wird im letzten Abschnitt gezeigt.

Bei der Verwendung von `hypnotoad` wird der Modus -- falls nicht anders eingestellt -- automatisch auf "production" gesetzt. Ein weiterer Vorteil von `hypnotoad`. Allerdings sollte man an die Umstellung des Modus' denken wenn ein anderer Server wie z.B. `starman` oder Apache verwendet.

Alternativ zur Umgebungsvariablen kann man den Modus auch in der `startup`-Methode festlegen:

```
sub startup {
  my $self = shift;
  $self->mode('production');
  # ... mehr code
}
```

Man kann sogar für jeden Modus eine eigene Logik hinterlegen. Dazu kann man pro Modus eine Subroutine definieren:

```
sub production_mode {
  # ...
}

sub development_mode {
  # ...
}
```

Diese Funktionen werden noch vor der `startup`-Methode ausgeführt. Das zeigt auch den Nachteil beim Setzen des Modus' in der `startup`-Methode.

Schritt 3: Parameter, Konfiguration, Cookies und Mojolicious-Session

Die Liveumgebung läuft also auch. Zeit, sich an den Ausbau der Anwendung zu setzen und neue Funktionalitäten zu implementieren. In diesem Abschnitt werden wir das Speichern der Benutzereingaben umsetzen.

Als erstes passen wir die Route zum Formular an, denn jetzt sollen nicht nur GET-Requests abgearbeitet werden, sondern auch POST-Requests -- das Formular wird mittels POST abgeschickt.

```
$r->get('/user/run')->to('user#run');
```

wird abgeändert in

```
$r->route('/user/run')
  ->via( [qw/GET POST/] )
  ->to('user#run');
```

Als nächstes muss im Controller die Methode angepasst werden um die Parameter zu verarbeiten.

Parameterverarbeitung

Bevor es weiter geht, schauen wir uns mal an, wie man bei Mojolicious an die Parameter kommt. Man kann im Prinzip drei Arten von Parametern unterscheiden:

1. GET-Parameter/URL-Parameter
2. POST-Parameter
3. Routen-Parameter

Die ersten beiden Arten sollten klar sein was das ist. Als Routen-Parameter bezeichne ich die Parameter, durch das Routing gesetzt werden. Ein kurzer Rückblick auf die letzte Ausgabe von $foo. Dort wurden die Platzhalter in Routen erläutert. Es gibt z.B. diese Route:

```
'/:id'
```

und in der Anwendung (Ausschnitt aus einer `Mojolicious::Lite`-Anwendung) sieht das dann so aus:

```
any '/:id'  => sub {
    my $self = shift;

    # more code
};
```

Wenn jetzt der Pfad `/123` aufgerufen wird, wird der Routen-Parameter `id` auf 123 gesetzt.

In den Code-Downloads für diese Ausgabe ist ein kleines Testskript enthalten, das in Listing 2 dargestellt ist.

Auf das Testen von Mojolicious-Anwendungen wird in der übernächsten Ausgabe näher eingegangen. Daher an dieser Stelle nur die wichtigsten Sachen.

```
$t->get_ok( '/3?level=debug' );

$t->post_form_ok(
  '/4?level=debug' =>
  { name => 'foojolicious' }
);
```

```
use Test::More tests => 4;
use Test::Mojo;
use Mojolicious::Lite;
use Data::Dumper;

any '/:id'  => sub {
    my $self = shift;

    my $params      = $self->req->params;
    my $body_params = $self->req->body_params;
    my $get_params  = $self->req->query_params;

    my %data;
    for my $name (qw/id level name/) {
     $data{$name}  = $self->param($name) if $self->param($name);
    }

    diag( Dumper [ $params->to_hash, $body_params->to_hash, $get_params->to_hash, \%data ] );
    $self->render( message => 'test', dump => Dumper [ $params->to_hash,
                                                       $body_params->to_hash,
                                                       $get_params->to_hash,
                                                     \%data ] );
};

my $t = Test::Mojo->new;

$t->get_ok( '/3?level=debug' )->content_like( qr/test$/ );
$t->post_form_ok( '/4?level=debug' => { name => 'foojolicious' } )
    ->content_like( qr/test$/ );

__DATA__
@@ id.html.ep
<%= $dump %><%= $message %>
```

Listing 2

Bei `get_ok` wird ein einfacher GET-Request ausgelöst. Der Routen-Parameter *id* wird auf 3 gesetzt und es gibt noch den GET-Parameter *level*, der auf debug gesetzt wird. Bei `post_form_ok` wird das Absenden eines Formulars mittels POST-Request simuliert. Neben dem Routen-Parameter *id* und dem URL-Parameter *level* wird jetzt noch der POST-Parameter *name* verwendet.

In der Route werden dann verschiedene Wege gezeigt, wie man an Parameter kommt:

```
my $params =
  $self->req->params;
my $body_params =
  $self->req->body_params;
my $get_params  =
  $self->req->query_params;

my %data;
for my $name (qw/id level name/) {
 $data{$name}  = $self->param($name)
   if $self->param($name);
}
```

Die `diag`-Ausgabe des Testskripts für den `post_form_ok`-Aufruf sieht so aus:

```
# $VAR1 = [
#           {
#             'level' => 'debug',
#             'name' => 'foojolicious'
#           },
#           {
#             'name' => 'foojolicious'
#           },
#           {
#             'level' => 'debug'
#           },
#           {
#             'level' => 'debug',
#             'name' => 'foojolicious',
#             'id' => '4'
#           }
#         ];
```

Hieran kann man gut erkennen, mit welcher Methode man an welche Parameter kommt:

- `params` aus `Mojo::Message::Request`
  alle GET- und POST-Parameter des Requests

- `body_params` aus `Mojo::Message`
  Alle POST-Parameter -- wenn `x-application-urlencoded`, `application/x-www-form-urlencoded`

bzw. `multipart/form-data` gesetzt ist. In einem ersten Test wurde in dem Testskript `post_ok` statt `post_form_ok` verwendet. Da konnten die Parameter nicht ausgelesen werden.

• `query_params` aus `Mojo::Message::Request`
alle GET-Parameter des Requests

• `param` aus `Mojolicious::Controller`
Zugriff auf GET-/POST- und Routen-Parameter

Man hat also die Möglichkeit, sehr genau zu bestimmen welche Parameter man auslesen kann. Denn interessant wird das Ganze, wenn bei den unterschiedlichen Parameter-Arten die gleichen Namen verwendet werden:

```
$t->post_form_ok(
  '/4?level=debug' =>
  {
    level => 'foojolicious',
    id => 8
  }
);
```

Hier sieht die `diag`-Ausgabe so aus:

```
# $VAR1 = [
#           {
#             'level' => [
#                          'foojolicious',
#                          'debug'
#                        ],
#             'id' => '8'
#           },
#           {
#             'level' => 'foojolicious',
#             'id' => '8'
#           },
#           {
#             'level' => 'debug'
#           },
#           {
#             'level' => 'foojolicious',
#             'id' => '4'
#           }
#         ];
```

Bei `$self->req->params` werden die URL- und POST-Parameter zusammengeführt. Deswegen tauchen hier beide *level*-Angaben auf. Bei `param` des Controllers findet man nur noch das *level* aus den POST-Parametern und die *id* aus den Routen-Parametern.

Weiter geht's...

Nach dem Exkurs in die Parameterverarbeitung in Mojolicious schauen wir uns die `run`-Methode im Controller an. Wenn die Request-Methode *POST* ist, sollen die Daten verarbeitet werden, ansonsten einfach das Formular ausgeben.

```
sub run {
  my $self = shift;

  if ( $self->req->method eq 'POST' ) {
    # get request parameters and save
    # run
  }

  $self->render;
}
```

Mit `$self->req` bekommen wir ein `Mojo::Message::Request`-Objekt, mit dem alle möglichen Informationen über den Request abgefragt werden können. Einige Methoden dieser Klasse bzw. deren Objekte werden wir auch im weiteren Verlauf der Programmierung benötigen.

Die erste Methode des Request-Objekts, die wir nutzen werden, ist `params`, mit der man ein Objekt von `Mojo::Parameters` bekommt. Wurde ein POST-Request abgesetzt, nutzen wir die `to_hash`-Methode der von `Mojo::Parameters`, damit wir alles in einer Hashreferenz haben und nicht jeden Parameter einzeln abfragen müssen.

```
my $params =
    $self->req->params->to_hash || {};
```

Zu beachten ist dabei, dass `params` und `param` die Daten cached. Deshalb sollten diese Methoden erst aufgerufen werden wenn alle Daten des Request empfangen wurden.

Wenn alles klappt, soll auf eine Übersichtsseite umgeleitet werden. Also benutzen wir die Methode `redirect_to`.

Warum jetzt nicht in zwei getrennten Methoden? Ich bevorzuge es, alles in eine Methode zu packen, damit die zusammengehörigen Einheiten beieinander stehen. Zusätzlich soll ja das gleiche Template ausgegeben werden wenn Fehler auftreten.

Die Methode sieht jetzt wie folgt aus:

```
sub run {
  my $self = shift;

  if ( $self->req->method eq 'POST' ) {
    # get request parameters and save
    # run
    my $params =
        $self->req->params->to_hash || {};
    my $error = $self->_save_run($params);

    if ( !$error ) {
      $self->redirect_to('/user/runs/');
      return;
    }
  }

  $self->render;
}
```

Dann werfen wir jetzt noch einen kleinen Blick in die Methode `_save_run`:

```
sub _save_run {
  my ($self, $params) = @_;

  # some code

  my $path = File::Spec->catfile(
    $self->config->{run_dir},
    $cleaned_user,
    $cleaned_date . '.txt',
  );

  # more code
}
```

Hier ist der Aufruf `$self->config` zu finden. Das bringt uns zum Thema Konfiguration. Ohne Plugin kann man die Konfiguration nicht in Dateien auslagern, wenn man mit `config` arbeitet, arbeitet man mit einer "internen" Hashreferenz. Die Konfigurations-Plugins füllen diese Hashreferenz. Eines dieser Konfigurations-Plugins wird im nächsten Abschnitt vorgestellt.

Nachdem die Daten gespeichert sind, soll eine Eingabe für den User nicht mehr notwendig sein -- der Benutzername. Dazu wird der eingegebene Name in einem Cookie gespeichert und bei der Anzeige des Formulars wieder ausgelesen.

Das Erstellen des Cookies sieht folgendermaßen aus:

```
$self->cookie( schluessel => 'wert' );
```

das Auslesen dementsprechend

```
my $value = $self->cookie( 'schluessel' );
```

Wer Daten in einem signierten Cookie speichern möchte, kann `session` nutzen. Die Daten in der Session werden mit `Mojo::JSON` serialisiert und Base64-kodiert abgespeichert. Das bedeutet aber nicht, dass die Daten verschlüsselt sind. Jeder der das Cookie sehen kann, kann -- wenn er weiß wie die Session-Cookies funktionieren -- die Daten sehen. Damit die Daten nicht unerwünschterweise verändert werden, werden diese mit einer HMAC-SHA1-Signatur versehen.

```
$self->session( schluessel => 'wert' );
my $value = $self->session( 'schluessel' );
```

Die maximale Größe der Daten liegt bei 4 Kilobyte, so dass ggf. die Daten auf mehrere Cookies aufgeteilt werden müssen.

Wer die Logdateien von `morbo` oder `hypnotoad` bisher angeschaut hat, dem ist möglicherweise diese Meldung aufgefallen:

```
[debug] Your secret passphrase needs to be
  changed!!!
```

Mit diesem Schlüssel wird der Hash der Signatur berechnet. Deswegen sollte man diesen Schlüssel nicht öffentlich machen. Einen eigenen Schlüssel kann man mittels

```
# set new passphrase
$self->app->secret( 'secret' );
```

setzen.

Soll das Cookie nur bei HTTPS-Verbindungen übertragen werden, muss das *secure*-Flag gesetzt werden.

Schritt 4: Plugins - Ein Einstieg

Auch wenn Mojolicious schon sehr viele Funktionalitäten mitliefert, braucht man häufiger Zusatzfunktionen. So sollen die Daten dieser Anwendung in einer Datenbank gespeichert und nicht länger im Dateisystem. Wie schon im zweiten Abschnitt gesagt wurde, soll man informiert werden wenn im Produktivsystem ein Fehler auftritt.

All diese Funktionalitäten müssen über Plugins in die Anwendung gebracht werden. Jetzt kommen auch die ersten Abhängigkeiten ins Spiel, nachdem Mojolicious selbst ja keinerlei Abhängigkeiten hat.

Dieser Abschnitt führt ein paar nützliche Plugins ein:

- ::YamlConfig
- ::TagHelpers
- ::I18N
- ::MailException

Mojolicious::Plugin::YamlConfig

Konfigurationen innerhalb des Skripts zu halten ist unpraktisch, weil man für Konfigurationsänderungen das Skript an sich anfassen muss. So etwas endet nicht gerade selten in Fehlern. Besser ist es, eigenständige Konfigurationsdateien zu erstellen.

Für solche Dateien gibt es die unterschiedlichsten Formate: Von Perl-Datenstrukturen über YAML bis hin zu JSON. Für diese Formate gibt es entsprechende Plugins für Mojolicious.

```
sub startup {
  # ...

  $self->plugin(
      YamlConfig => {
          file      => '/etc/myapp.conf',
          stash_key => 'conf',
          class     => 'YAML::XS'
      },
  );

  # ...
}
```

Damit stehen im Controller die Konfigurationsdaten auch über `$self->config` zur Verfügung.

Mojolicious::Plugin::TagHelpers

Ein paar Plugins gibt es schon bei der Standardinstallation von Mojolicious. Dazu zählt neben dem Plugin zur Verarbeitung von Konfigurationsdateien im JSON-Format auch das Plugin *TagHelpers*. Das Plugin hilft, Templates übersichtlich zu gestalten, denn es bietet ein paar Funktionen zum Erzeugen von Formularfeldern.

Schauen wir uns das Template für das Formular vom Anfang nochmal an (siehe Listing 3).

Wir haben hier ein Feld für den Benutzernamen. Da kann man nicht viel machen, da der Benutzername beliebig aussehen darf. Für Datumsfelder gibt es aber eine Helferfunktion, die man im Template nutzen kann:

```
<%= datetime_field date
     => '2013-02-01T12:00:00',
id => 'date' %>
```

Damit wird folgendes HTML erzeugt:

```
<input id="date" name="date" type="datetime"
  value="2013-02-01T12:00:00" /><br />
```

Erstmal kein großes Hexenwerk, das kann man auch selbst per Hand hinschreiben. Es wird auch nicht automatisch validiert, ob die Eingaben in Ordnung sind, aber das Plugin kann bei anderen Feldern seine Stärke ausspielen:

```
%= select_field country =>
 [{Europe => [[Germany => 'de'], 'en']}]
```

Erzeugt folgenden HTML-Code:

```
<select name="country">
  <optgroup label="Europe">
    <option value="de">Germany</option>
    <option value="en">en</option>
  </optgroup>
</select>
```

So wird mit wenig Code im Template viel HTML erzeugt.

Damit man diese Hilfsfunktionen aber nutzen kann, muss das Plugin in der `startup`-Methode geladen werden:

```
<form action="/user/run" method="post">
  <label for="username">Läufer</label>
  <input type="text" name="username" id="username" /><br />
  <label for="date">Datum</label>
  <input type="text" name="date" id="date" /><br />
  <label for="distance">Streckenlänge</label>
  <input type="text" name="distance" id="distance" /><br />
  <label for="time">Zeit</label>
  <input type="text" name="time" id="time" /><br />
  <button type="submit" value="Speichern">Speichern</button>
</form>
```

Listing 3

```
sub startup {
  # ...

  $self->plugin( 'TagHelpers' );

  # ...
}
```

Mojolicious::Plugin::I18N

Wer seine Anwendung nicht nur für einen kleinen Anwenderkreis geschrieben hat, sondern auch internationale Besucher bedienen möchte, sollte die Seite mehrsprachig machen. Eine große Hilfe dabei ist das Plugin `Mojolicious::Plugin::I18N`.

Los geht es mit der Aktivierung des Plugins:

```
sub startup {
  # ...

  $self->plugin( 'I18N' );

  # ...
}
```

Im zweiten Schritt müssen die Templates umgestellt werden. Mit der Aktivierung des Plugins steht sowohl im Controller als auch im Template die Helferfunktion `l()` zur Verfügung. Nach den Änderungen sieht das Template so aus wie in Listing 4 dargestellt (Ausschnitt):

Hinweis: Cosimo Streppone weist in seinem Blog darauf hin, dass z.B. bei Template Toolkit der Aufruf im Template leicht anders aussieht:

```
[% c.l('<your string here>') %]
```

Man beachte das `c.` vor dem Aufruf der Helferfunktion. Eventuell sieht es bei anderen Template-Engines wieder anders aus.

Im nächsten Schritt müssen die Übersetzungsmodule erstellt werden. Diese müssen standardmäßig im I18N-Namesraum der Anwendung liegen, hier als `TrackRuns::I18N::`. In den Modulen wird ein Hash `%Lexicon` definiert, der die Übersetzungen enthält:

```
package TrackRuns::I18N::de;

use strict;
use warnings;

use base qw(TrackRuns::I18N);

use utf8;

our %Lexicon = (
    Runner => 'Läufer',
    Date   => 'Datum',
    'Length of course' => 'Streckenlänge',
    Time   => 'Zeit',
    Save   => 'Speichern',
);


1;
```

Wie leicht zu erraten ist, muss für jede unterstützte Sprache ein solches Modul erstellt werden. Die Standardsprache ist erstmal *en*. Möchte man die Standardsprache ändern, kann man bei der Aktivierung des Plugins diese angeben:

```
$self->plugin(
  'I18N' => {
    default => 'de',
  }
);
```

Ist ein String nicht in der Übersetzungstabelle vorhanden, wird der an `l()` übergebene String wieder zurückgeliefert.

Noch ein Wort dazu, wie Mojolicious weiß, welche Sprachdatei herangezogen werden soll:

Ohne besondere Angabe in der Anwendung wird der *Accept-Language*-Header des Browsers ausgewertet. Ist dort keine Angabe zu finden oder existieren keine Übersetzungsmodule für diese Sprachen, wird die Standardsprache genommen.

Eine beliebte Methode ist es auch, in den URLs die gewünschte Sprache zu nennen. Auch das kann man für das Plugin einstellen:

```
$self->plugin(
  'I18N' => {
    support_url_langs => ['de','en'],
  }
);
```

```
<label for="username"><%= l('Runner') %>:</label>
<input type="text" name="username" id="username" value="<%= $username %>" /><br />
<label for="date"><%= l('Date') %>:</label>
<%= datetime_field date => '2013-02-01T12:00:00', id => 'date' %><br />
```

Wird jetzt

```
http://localhost:3000/en/user/run
```

aufgerufen, wird auf jeden Fall die englische Version ausgeliefert, egal was der *Accept-Language*-Header aussagt. Man muss dafür auch keine Route anpassen. Das funktioniert automatisch. Wird aber z.B.

```
http://localhost:3000/ru/user/run
```

aufgerufen und die Übersetzungsdatei für "ru" existiert nicht, wird die Route nicht gefunden und man landet auf der "Not found"-Seite.

Man kann die Sprache auch an Hand des Hostnamens setzen, so dass *http://perl-magazin.de* die deutschsprachige Version ausliefert und *http://perl-magazin.com* die Englischsprachige:

```
$self->plugin(
  'I18N' => {
    support_hosts => {
      'http://perl-magazin.de'  => 'de',
      'http://perl-magazin.com' => 'en',
    },
  }
);
```

Die beiden letztgenannten Features stehen erst in neueren Versionen zur Verfügung. Wer Mojolicious-Versionen < 3.0 einsetzt, hat das Plugin automatisch mitgeliefert bekommen. In 3.0 wurde es in eine eigenständige Distribution ausgelagert.

Mojolicious::Plugin::MailException

Dieses Plugin ist besonders für Produktivumgebungen wichtig, wenn die Fehlerseiten keinerlei Informationen liefern und wenn man selbst nicht mitbekommt, welche Fehler auftauchen.

```
$self->plugin( 'MailException' => {
    from    => 'exception@address.tld',
    to      => 'recipient@error.tld',
    subject => 'an error occured',
    send    => sub {
        my ($mail,$exception) = @_;

        my $message = $exception->message;
        return if $message =~
                    m{Can't locate};

        $mail->send(
            smtp =>
            AuthUser => 'User',
            AuthPass => 'Password',
        );
    },
});
```

Die Parameter `from`, `to` und `subject` dürften selbsterklärend sein. `send` ist optional. Wenn diese Option nicht gesetzt wird, wird die `send`-Methode von `MIME::Lite` genommen, ohne weitere Parameter. Wer also den Versand in die Hand nehmen möchte oder die Mail noch anpassen möchte, sollte eine Subroutinenreferenz übergeben.

In diesem Fall werden Fehler wegen nicht zu ladender Module nicht verschickt.

Hier wäre eine bessere Integration mit `Mojolicious::Plugin::Mail` zu wünschen.

Ausblick

In dieser Ausgabe wurden die ersten praktischen Schritte auf dem Weg zur Anwendung gegangen: Die Anwendung wurde konfiguriert ein erstes "Release" wurde vorgenommen und dabei vom Entwicklungsmodus in den Produktivmodus gewechselt. Abschließend wurden die ersten Plugins vorgestellt und verwendet.

In der nächsten Ausgabe wird das Formular nur für registrierte Benutzer zugänglich gemacht. Außerdem steigen wir tiefer in die Plugins ein und es wird gezeigt, wie man eigene Plugins für Mojolicious schreiben kann. Momentan ist es ja in Mode, Anwendungen bei dotCloud oder heroku zu veröffentlichen. Wie man das macht, wird ebenfalls in der nächsten Ausgabe Thema sein.

Swiss Perl Workshop 2013

Meet Perl hackers from Switzerland and other countries

•

Learn from and be inspired by talks

•

Chat and socialise during breaks and attendees dinner

We are proud to announce the first Perl Workshop in Switzerland.

Date: 22. March 2013

Venue: Bern

Language: English

Register today

•

Submit your talk

•

Become a Sponsor

www.perl-workshop.ch

Sawyer X

Synchrone Operationen sind überholt

Asynchrone Ereignisse verstehen

Der beste Weg zu erklären, warum synchroner Code abschreckend sein kann, ist, ein Beispiel aus dem realen Leben zu verwenden. Ein einziger Tag im Leben kann so viele Aktionen beinhalten, die uns zusammenzucken und knurren lassen. Man nehme beispielsweise den Versuch, ein Essen vorzubereiten.

Stellen wir uns vor, wir kochen. Man würde nie darauf warten, dass das Wasser kocht, bevor man die Kartoffeln vorbereitet. Genauso wenig würde man darauf warten, dass die Kartoffeln fertig sind, bevor man den Salat zubereitet.

Asychrones Programmieren bedeutet, dass mehrere Ereignisse zur selben Zeit stattfinden können. Es ermöglicht, dass mehrere Dinge erledigt werden können, während man darauf wartet, dass andere Dinge passieren.

Das fundamentale Element der asynchronen Programmierung ist der Callback. Schauen wir uns also zuerst einen Callback an und wenden uns anschließend einigen Beispiele asynchroner Programmierung zu.

Wir werden `AnyEvent` für diesen Artikel verwenden. Dieselben Gesetzmäßigkeiten existieren in allen anderen asynchronen Strukturen.

Callback-Einführung

Da mehrere Events zur gleichen Zeit stattfinden, muss die Anwendung genau wie das Würzen laufen. Um das zum Laufen zu bringen, müssen wir beim Starten eines Events Referenzen zu dem Code hinzufügen, der laufen soll, wenn es fertig ist oder einen anderen Meilenstein erreicht. Da das Ereignis dann selbst "weiß", wie es weitergeht, kann es starten und im Hintergrund arbeiten während der Rest des Programms andere Dinge fortführt.

Wir verwenden eine Technik, die einigen nicht vertraut ist: **callbacks**. Um Sie auf den aktuellen Stand zu bringen, lassen sich Callbacks zusammenfassend so erklären: Callbacks sind nur Referenzen zu Subroutinen. Diese Subroutinen können definiert werden, indem Namen verwendet werden oder sie können anonym sein. Diese Subroutinen können durch ihre Referenz anstatt ihres Namens aufgerufen werden.

```
# Callbacks zu Subroutinen mit Namen
sub func { ... }
my $func_reference = \&func;
$func_reference->(@arguments);

# Callback zu anonymen Subroutinen
my $func_reference = sub { ... };
$func_reference->(@arguments);
```

Wenn wir `sub` verwenden, um eine Referenz zu einer Subroutine zu erstellen, können wir den Callback direkt als Parameter weitergeben ohne ihn vorher zu speichern:

```
sub some_cb_handler {
    my $callback = shift;
    $callback->("hello");
}

# Wir gehen in den Callback ohne ihn
# jemals einen Namen gegeben zu haben
# oder ihn global verfügbar gemacht zu
# haben.
some_cb_handler( sub {
    my $greeting = shift;
    say "$greeting, world!";
} );
```

Vom Input lesen

Man hat eine Anwendung, die von einem Handle (der eine Dateibeschreibung, ein Socket, oder ein Standardinput sein kann) lesen muss, aber man weiß nicht, wann er fertig ist und gelesen werden kann. In einer synchronen Applikation würde man warten bis er verfügbar sein würde. Möglicherweise würde man in der Zwischenzeit `sleep` aufrufen. Aber in der heutigen Zeit würde man nicht einfach warten. Wir sind beschäftigte Leute und wir haben Dinge zu tun!

```
sub alert_action {
    my $action = shift;
    say "New action found: $action";
}

my $io_watcher = AnyEvent->io(
    fh => $fh,
    poll => 'r',
    cb => sub {
        # wir können jetzt lesen!
        my $input = <$fh>;
        if (
          $input =~ /^New action: (\w+)/ ) {
            alert_action($1);
        }
    },
);

 # mit etwas anderem fortfahren
```

Wie funktioniert das? Indem man die `AnyEvent io` Methode aufruft, erstellt man einen neuen Beobachter, der ein Datei-Handle prüft, um neue Ereignisse zu lesen. Wenn er etwas zu lesen bekommt, wird er den Referenzcode aufrufen, den man zur Verfügung gestellt hat. Beides, d.h. die Prüfung und der Aufruf der Subroutine, wird im Hintergrund passieren.

Er hält uns nicht auf, da wir ihm alle notwendigen Informationen gegeben haben: welcher Datei-Handle aufgerufen werden soll, welche Art von Ereignis wir wollen und was in dem Fall zu tun ist. Auf diese Weise können wir mit anderem Code fortfahren und die Aufgabe im Hintergrund laufen lassen ohne uns zu ärgern.

Den Beobachter am Leben lassen

Es gibt ein Problem, das hier noch nicht erwähnt wurde. Der Code ist gut, außer dass sich die Anwendung beendet, nur weil sie das Ende der Datei erreicht hat. Wir wollen die Anwendung aber weiterlaufen lassen, sodass unser Beobachter weiterarbeitet. Wie kann man das erreichen? Mit Zustandsvariablen!

Zustandsvariablen sind Variablen die einen Zustand darstellen, der darauf wartet, wahr zu werden. Wie eine Katze, die darauf wartet, dass man es sich mit einem Laptop bequem gemacht hat. Wenn diese Variable wahr wird, kommt die Katze herüber, legt sich auf den Schoß und unterbricht die Arbeit.

```
my $done = AnyEvent->condvar;
my $watcher = AnyEvent->io(
    fh => $fh,
    poll => 'r',
    cb => sub {
        my $input = <$fh>;
        ...
        if ( $input =
           ~ /^End of processing file/ ) {
            $done->send;
        }
    },
);

...
$done->recv;
say "All done!";
```

Dieses Mal haben wir eine Zustandsvariable erstellt, die für den Beobachter verfügbar ist. Der Beobachter führt seine Arbeit zuverlässig fort. Nur, wenn er eine Zeile findet, die das Ende der Datei anzeigt, wird es `send` aufrufen, den Zustand auf wahr setzen und effektiv sagen: "Das ist es, wir sind fertig".

Wenn jemand `recv` an der Zustandsvariable aufruft, wird er warten bis etwas anderes im Hintergrund (wie unser Beobachter) `send` aufruft und dann weiter ausgeführt.

Das bedeutet, dass die Zeile "All done!" geschrieben werden wird, sobald unser Worker mit dem Lesen der Zeile fertig ist.

Eine andere Konsequenz des Verhaltens der Zustandsvariable ist die Möglichkeit, eine unendliche Schleife zu erstellen, indem man eine Zustandsvariable erstellt, `recv` aufruft und nichts hat, was `send` aufruft. Das sieht wie folgt aus:

```
my $cv = AnyEvent->condvar;
$cv->recv;

# or in short
AnyEvent->condvar->recv;
```

Da die Anwendung jetzt darauf wartet, dass die Zustandsvariable wahr wird, wird sie sich nicht beenden. Da nichts `send` aufrufen kann, bedeutet das, dass die Anwendung auf unbestimmte Zeit offen bleibt. Am meisten wird das für Daemons genutzt, die immer laufen sollen.

Zeit zum Kochen

Das letzte Element in `AnyEvent`, das wir uns ansehen, ist ein Timer. Timer sind Ereignisse (jegliche Art von Ereignissen), die zu einem Zeitpunkt ausgeführt werden. Es kann in ein paar Minuten von jetzt an geschehen oder zu einer bestimmten Stunde. Es kann einmal passieren oder sich ein paar Mal wiederholen oder immer wiederholen.

```
my $timer = AnyEvent->timer(
    after => 3.5,
    interval => 5,
    cb => sub {
        say "Ping? Pong!";
    },
);
```

Dies definiert, dass der Timer 3,5 Sekunden warten wird und dann die Subroutine alle fünf Sekunden aufrufen wird. Ziemlich einfach. Schauen wir uns ein paar Timer an (siehe Listing 1.)

Was wir hier, haben ist nicht das beste Beispiel um ein Essen zuzubereiten. Aber es ist ein Beispiel, das mehrere Timer zeigt. Der erste Timer (`$t1`) alarmiert uns alle sieben Minuten über unseren Fortschritt. In der Zwischenzeit greift der zweite Timer eine Aktion auf um sie alle 10 Minuten auszuführen und tut das. Sobald keine Aktionen mehr zur Verfügung stehen, teilt er der Zustandsvariable mit, dass er fertig ist. Er tut das einfach indem er die Subroutine zurückgibt und gleichzeitig `send` aufruft.

Nachdem wir unseren Timer erstellt haben, setzen wir ein `recv` auf die Zustandsvariable, die besagt "den Rest der Anwendung nicht weiter ausführen bis wir informiert werden, dass alle Timer ihre Arbeit beendet haben". Er wird zu diesem Zeitpunkt warten bis `send` aufgerufen wird (ohne dabei die Timer zu blockieren). Danach wird er fortführen und schließlich sagen, dass das Essen fertig ist. Da es das Ende der Anwendung ist, werden die Timer beendet und die Anwendung geschlossen.

Hier ist die Ausgabe, die wir bekommen, wenn wir die Anwendung ausführen:

```
Current cooking state: Preparing
(do_step() called with "Cutting")
Current cooking state: Cutting
(do_step() called with "Simmering")
Current cooking state: Simmering
Current cooking state: Simmering
(do_step() called with "Cooking")
Current cooking state: Cooking
(do_step() called with "Seasoning")
Current cooking state: Seasoning
(do_step() called with "Serving")
Current cooking state: Serving
Current cooking state: Serving
Dinner is served!
```

```
my @steps = qw<Cutting Simmering Cooking Seasoning Serving>;
my $current_step = 'Preparing';

my $done = AnyEvent->condvar;
my $t1 = AnyEvent->timer(
    interval => 60 * 7,
    cb => sub {
        say "Current cooking state: $current_step";
    },
);

my $t2 = AnyEvent->timer(
    after => 2, # two seconds to wash hands before working!
    interval => 60 * 10, # assuming every action takes 10 minutes
    cb => sub {
        $current_step = shift @steps or return $done->send;
        do_step($current_step);
    },
);

$done->recv;
say "Dinner is served!";
```

Listing 1

Zustandsvariablen mit mehreren Aufrufen

Manchmal ist das Verhalten der Zustandsvariablen `send` und `recv` nicht flexibel genug um Instanzen zu bearbeiten, in denen man auf verschiedene Aufrufe warten muss.

Nehmen wir an wir haben eine Kalkulation durchzuführen, die abhängig von Resultaten verschiedener Datenbankabfragen ist. Bevor die SQL-Experten sich darauf stürzen, nehmen wir an, dass die Abfragen über verschiedene Datenbanken gemacht werden.

Eine Datenbankverbindung ist tatsächlich eine Netzwerkoperation, was bedeutet, dass sie blockiert. Das ist ein ideales Beispiel für asynchrones Programmieren. Man kann verschiedene Verbindungen und Abfragen gleichzeitig anstoßen statt nacheinander. Verwendet man Zustandsvariablen würde man möglicherweise versuchen drei Zustandsvariablen zu öffnen und darauf zu warten, dass jede einzelne wahr wird. Das würde nicht funktionieren, wenn man `recv` an einer Variablen zu einem Zeitpunkt aufruft.

Stattdessen können Zustandsvariablen einen `begin`- und `end`-Aufruf akzeptieren, um Mehrfachaufrufe zu kennzeichnen. Sobald ein `end` für jeden `begin`-Aufruf vorhanden ist, wird er zur `recv`-Methode zurückkehren.

```
my $cv = AnyEvent->condvar;
my $sum = 0;
foreach my $db (@dbs) {
    # beginning an event
    $cv->begin;

    $db->query( $query, sub {
        my $amount = shift;
        $sum += $amount;

        # finishing an event
        $cv->end;
    },
}

$cv->recv;
say "All database queries finished.";
```

Zusammenfassung

Nachdem wir uns einige Elemente von `AnyEvent` angeschaut haben, können wir eine kleine nützliche Anwendung erstellen. Wir fügen ein paar weitere Elemente wie `AnyEvent::HTTP`, `Regexp::Common` und `File::Basename` hinzu.

Angenommen, wir haben eine Datei, die viele Links beinhaltet und wir wollen jedes Bild herunterladen, das darin gelistet ist. Das sind zwei verschiedene Aktionen: (1) Lesen der Datei und (2) Herunterladen der Bilder. Wir werden auch einen Timer haben, der uns alle zwei Sekunden den Fortschritt anzeigt (siehe Listing 2).

Analysieren wir, was wir hier haben: Wir verwenden einige Module, die man wiedererkennen sollte. Wenn nicht, sollte man sie sich anschauen.

Als nächstes öffnen wir einen Datei-Handler. Dann setzen wir einen Beobachter für einige I/O-Operationen auf, indem wir die `io`-Methode von `AnyEvent` verwenden. Es benötigt den Datei-Handler, den wir benutzen werden und welche Art von Operationen wir durchführen werden (wir nutzen `r` zum Lesen) und einen Callback zur Ausführung. Der Callback ist die Hauptsache, deren Verständnis ein wenig Zeit braucht.

Jedes Mal, wenn wir eine Zeile lesen, die eine URL hat, rufen wir `begin` der Zustandsvariable auf. Wir geben eine HTTP-Anfrage für diese URL auf und sobald wir sie abgerufen und gespeichert haben, rufen wir den dazugehörigen Schlussaufruf auf. Wenn alle `begin`-Aufrufe beendet sind, wird er zur `recv` Methode zurückgehen, ganz wie das Aufrufen von `send`.

Wir haben auch einen Fortschrittstimer erstellt, der alle zwei Sekunden die Anzahl der Links, die wir gesendet haben, angibt. Er verwendet `now` von `AnyEvent`, was der empfohlene Weg ist, die Zeit aufzurufen, wenn man ein Ereignis in einer Schleife ausführt.

Der `recv`-Aufruf in dem `end` wird warten bis alle `begin`-Aufrufe geschlossen sind. Sobald wir die ganze Datei bearbeitet haben, wird es eine schöne Nachricht ausgeben und die Anwendung wird beendet.

Nur der Anfang ...

Sobald man sich mit dem asynchronen Programmieren vertraut gemacht hat, ist es, wie Scheren zu haben: Man läuft nur mit ihr! WICHTIG: Man sollte nicht mit Scheren rennen: ✂.

Testen Sie einen Ereignisrahmen und schauen Sie sich an, wie viel Spaß es macht. Perl hat mehr zu bieten, wie zum Beispiel `AnyEvent`, `POE`, `IO::Async`, `Reflex`, `IO::Lambda`, `Coro` und vieles mehr ...

```
use AnyEvent;
use AnyEvent::HTTP;
use Regexp::Common 'URI';
use File::Basename 'basename';
use autodie;

my $counter = 0;
my $cv = AnyEvent->condvar;
my $fh = open my $fh, '<', 'links.txt';
my $fhwatcher = AnyEvent->io(
    fh => $fh,
    poll => 'r',
    cb => sub {
        my $line = <$fh>;

        # ignoring lines that aren't HTTP URIs
        $line =~ /^$RE{URI}{HTTP}$/ or return;

        # call an HTTP request
        $cv->begin;
        http_get $line, sub {
            my $body = shift;
            my $filename = basename($line);

            syswrite $filename, $body ?
                $counter++ :
                or warn "Couldn't write to $filename: $!";

            $cv->end;
        };
    },
);

my $progress = AnyEvent->timer(
    after => 2, # giving it two seconds before starting
    interval => 2, # report every two seconds
    cb => sub {
        printf "[%s] Update: finished downloading $counter images.\n",
            scalar AnyEvent->now;
    },
);

$cv->recv;
close $fh;
say "Finished downloading all files";
```

Listing 2

Herbert Breunung

Rezension - Leidenschaft und Perl

Chad Fowler
Der leidenschaftliche Programmierer
mitp 2009
336 Seiten, Softcover
ISBN 978-3-8266-5885-3
$19,95

Udo Müller
Perl
mitp 2008
608 Seiten, Softcover
ISBN 978-3826617768
€34,95

Die erste Rezension schließt sich beinahe nahtlos an die letzte Folge an, als das kleine Einmaleins des Projektmanagements rezitiert wurde. Bevor man nämlich andere leitet, wäre es hilfreich, sich selbst zum Erfolg führen zu können. Darum geht es Chad Fowler in seinem handlichen Buch. Als zweiten Titel gibt es ein weniger beachtetes Perlbuch des Karlsruher Informatikprofessors Udo Müller.

Der leidenschaftliche Programmierer

Für die Käufer der *$foo* ist Programmieren wahrscheinlich etwas mehr, als nur Arbeit oder gelegentlicher Zeitvertreib. Daher könnte sie auch eine Anleitung interessieren, wie man ein souveräner Meister seines Faches wird. "Was muss man an seiner Haltung verändern?" und "Welche kleinen täglichen Schritte tragen einen letztlich zum Ziel?" sind Fragen denen sich Herr Fowler widmet. Vorzuweisen hat er dafür äußerlich nur ein abgebrochenes Musikstudium und eine eigne, kleine Firma, doch seine Begeisterung führte ihn zu den Computern und schließlich zu Ruby, für dessen Gemeinde er mehrere Konferenzen organisiert. Das erklärt auch das kurze Vorwort vom Rails-Schöpfer David Heinemeier Hansson.

In den Kapiteln finden sich dazu immer wieder bis zu fünfseitige, graue Boxen, die Beiträge anderer, namenhafter Programmierer enthalten. Das bereichert das Buch um weitere Sichtweisen. Eine Auflockerung hat es nicht nötig, da der Schreibstil recht einfach und sehr gut verständlich ist. Die kurzen Kapitel erzählen Geschichten und haben nie mehr der etwas kleiner als gewohnten Seiten als man mit Fingern zeigen könnte. Eindeutig ist es Lektüre für die S-Bahn-Fahrt zur Arbeit und auf eine sofortige Anwendung ausgelegt.

Dabei geht es neben der Aneignung wesentlicher Fähigkeiten und dem effektiven Abschluss von Arbeiten um die aktive Gestaltung der Karriere und Zurückgewinnung der Freude an der Arbeit. Die Anleitung zum unternehmerischen Denken wird vielleicht nicht jedem gefallen, dennoch ist es auch für uns Bewohner der alten Welt bekömmlicher, weil entspannter als ein durchschnittliches, amerikanisches Selbsthilfebuch.

Auch wenn es erst einmal nach einer Pointe klingt, hier von der Fibel der anonymen Programmierer zu reden - dieses Buch ist dort am nützlichsten, wo es dem Leser über die Momente hilft, in denen die Motivation knapp wird oder man sich blöd und unsicher fühlt, weil durch das Ausprobieren von etwas Neuem Unzulänglichkeiten zutage treten.

Ein Werk, das bisher fehlte und das Potential hat, ein Klassiker zu werden.

Obwohl*mitp* das Buch ins Deutsche übersetzen ließ und druckt, es stammt ursprünglich vom *pragmatic bookshelf* [1], wo einzig eine elektronische, aber englische Kopie bezogen werden kann. Um die guten Waren dieses kleinen Verlages

bemühen sich sichtlich Größen wie O'Reilly [2] oder *mitp*. Da dort allerdings die Hauptthemen Ruby und Java sind, wurde hier bisher nur das *Sieben Sprachen, Sieben Wochen* in Ausgabe 4/2011 rezensiert.

Perl

Diese schlichte Überschrift trägt ein hellgrau-blauer Einband, welcher hier mit mehreren Werken verglichen werden soll, wie etwa der bekannten *Einführung in Perl* von O'Reilly (3/2011), denn eine Einführung in Perl ist er ebenso.

Der erste Blick aufs Erscheinungsjahr (2008) erregt sicher reflexartig Bedenken, andererseits zeigte Ovid in der letzten Ausgabe, wie man ein heute immer noch relevantes Buch größtenteils über Perl 5.8 schreibt. Der feine Unterschied liegt jedoch darin, dass Curtis Poe 2012 eine praxisbewährte Stoffauswahl traf und Udo Müller einen Kurs ausbaute, den er als Professor der Wirtschaftsinformatik lehrte. Das moderne Perl war damals erst im Entstehen. Deshalb gibt es im *Web*, *GUI* und anderen, zu erwartenden Bereichen Hinweise, welche mit einer Neuauflage wohl geändert werden würden. Dennoch ist der Stoff insgesamt nicht zu veraltet wie etwa "Goto Perl" von Perlmeister Michael Schilli (1998). Darüber hinaus hat das Werk einige seltene Qualitäten, weswegen eine Betrachtung lohnt.

Anders als Randal L. Schwartz dosiert der Autor den Humor feiner, seine durchaus spitze Feder ist leiser. Er unterstützt auch den Lernenden mit umsichtigen und vorausschauenden Gedanken. Beim Lama und auch Alpaka konzentriert man sich weit mehr auf die Vermittlung des aktuellen Konzeptes. Sie bleiben auch bei den allgemeinen Grundlagen, wohingegen Prof. Müller als auch Prof. Plate (*Perl-Programmierer*, siehe Ausgabe 1/2012) selbst vor den selten verwendeten Signaturen nicht haltmachen. Letztere beide Bücher dokumentieren zudem wesentlich mehr Module, die bei Anwendungen nützlich sind. Beim Plate sind es sogar noch wesentlich mehr, da es doppelt so dick ist. Er taucht in viele Module tiefer ein und behandelt auch das seinem Fachbereich (Betriebsysteme) nahe liegende ausführlicher.
Trotz der Gunst der späteren Geburt (2010), ist der *Perl-Programmierer* kaum aktueller. Plate ist außerdem eher in der Hackerkultur verwurzelt, Müller geht es dann doch mehr um die sorgfältige Vermittlung der einfacheren Dinge. Deshalb gibt es hier zu jedem Kapitel Übungen und deren Lösung, was es beim Plate nicht gibt. Dieser verteilt freimütig ungewöhnlich viel Zusatzmaterial auf seiner Netzseite, Müller hingegen kann mit einer beigelegten CD punkten. Sie ist spartanisch gestaltet und enthält die benötigten Module, eine Referenz und sämtliche Beispiele. Wenigstens diese lassen sich mit Suchanfragen elektronisch durchforsten, was allerdings nicht an den Komfort reicht, den *O'Reilly* durch seinen Online-Dienst Safari bereitstellt. Seine Titel sind unbestritten die aktuellsten.

Dafür ist das Buch von *mitp* sehr günstig im Netz zu bekommen und für einen Neuling, der sich nebenbei noch das freie, aber sehr knapp gehaltene *Modern Perl* zieht, ein feiner und gründlicher Kurs.

Ausblick

Die Suche nach dem perfekten Perlbuch wird mit den verschiedenen Materialien des Gabor Szabo, Träger der Auszeichnung weißes Kamel, fortgesetzt. Er bietet auf *http://perl5maven.com* freie Artikel, Videokurse und drei E-Bücher, welche in den nächsten drei folgenden Ausgaben beäugt werden. Auch die auf *http://perlybook.org/perltuts* werden bald folgen, sowie das Bündel des Linux-Magazins. Als Papierbuch ist *The Art of Community* vom Moderator des LUG-Radio Jono Bacon geplant. Denn nach den Vorschlägen zur Planung der Karriere und eher kommerzieller Projekte fehlt noch die gekonnte Gestaltung freier Projekte und freiwilliger Gemeinschaften.

Links

[1] *http://pragprog.com/book/cfcar2/the-passionate-programmer*
[2] *http://www.oreilly.de/pragmatic/*

Renée Bäcker

Rezension - SQL Performance Explained

SQL Performance Explained
Markus Winand, Mai 2012
204 Seiten, broschiert
ISBN 978-3-9503078-1-8, €29,95

Die Geschwindigkeit von Datenbankabfragen ist immer wieder ein Problem. Gerade bei großen oder verknüpften Tabellen kann die Geschwindigkeit der Abfragen die Geschwindigkeit der gesamten Anwendung beeinflussen. Auf 204 Seiten geht Markus Winand darauf ein, wie ein geschickter Index Abfragen beschleunigen kann oder ein ungeschickt eingerichteter Index auch Performanznachteile mit sich bringt.

Der Aufbau und die Arbeitsweise eines Index werden hier sehr genau unter die Lupe genommen.

Nach einer allgemeinen Einführung in den B-Tree-Index (andere Indextypen werden nur vereinzelt am Rande angesprochen) werden in den einzelnen Kapiteln die verschiedenen Bestandteile eines SQL-Befehls betrachtet. Von der `WHERE`-Klausel über die `JOIN`-Operation bis hin zu Sortierungen und Gruppierungen. Auch innerhalb der Kapitel geht es vom Allgemeinen zum Speziellen. So startet das Kapitel über die `WHERE`-Klausel mit einfachen Beispielen und endet beim Spezialfall `LIKE`.

Überhaupt nimmt dieses Kapitel den Großteil des Buches ein. Mit einigen Illustrationen und Ausführungsplänen wird gezeigt, wie die Datenbank zu den Suchergebnissen kommt und wo der Haken liegt.

Wichtige Informationen werden explizit in einem Kasten dargestellt. Hier gibt es einen kleinen Kritikpunkt: Der Unterschied zwischen *Beachte*-, *Wichtig*- und *Tipp*-Kästen wird nur über die Überschrift und nicht zusätzlich über ein Symbol angezeigt. Das verhindert beim Nachschlagen nach dem ersten Durchlesen, dass man z.B. an den *Wichtig*-Kästen leichter hängen bleibt.

An einigen Punkten gibt es Denksportaufgaben, deren Lösungen aber leider nicht im Buch zu finden sind. Das würde aber gerade Einsteigern helfen, sich selbst zu prüfen.

Das Buch betrachtet nicht nur ein Datenbanksystem, sondern mit MySQL, PostgreSQL, SQL Server und Oracle gleich mehrere. Dadurch bekommt man einen guten Einblick, wie die eigene Datenbank arbeitet, aber auch wie andere funktionieren. Am Ende des Buches gibt es für jedes System ein eigenes Kapitel in dem genau beschrieben wird, wie man Performanzangaben bekommt und wie sie zu deuten sind.

Was das Buch abrunden würde, ist eine Art Rezept, wie man von einer blanken Tabelle zu den wichtigen Indizes kommt und ab wann zusätzliche Indizes keinen Nutzen mehr bringen. Man kann es sich teilweise aus den Problembeschreibungen in den einzelnen Kapiteln herleiten. Eine Übersicht am Ende des Buches fehlt leider, wäre aber durchaus hilfreich.

Sehr interessant ist der Teil über *verschleierte Bedingungen*, die die Datenbankabfragen deutlich verlangsamen können. Auch räumt das Buch mit einigen Mythen auf, z.B. dass dynamisches SQL langsam ist.

Fazit

Das Buch sollte jeder Entwickler einmal gelesen haben, der mit Datenbanken arbeitet, die mehr als nur ein paar wenige Datensätze verwalten. In vielen Firmen gibt es zwar extra Datenbankadministratoren, mit denen Entwickler die Einrichtung von Indizes besprechen können, aber eigenes Wissen hilft ungemein dabei, die Anwendung schon richtig aufzubauen.

Die wenigen Kritikpunkte wirken sich nicht negativ auf die Nützlichkeit des Buches aus.

TPF News

25 Jahre Perl

Keine wirkliche Neuigkeit, aber trotzdem eine Erwähnung wert: Mark Keating hat einen lesenswerten Artikel zum 25. Geburtstag von Perl im Dezember 2012 geschrieben. Er stellt darin das Ökosystem rund um Perl vor: Die Community mit ihren verschiedenen Organisationen und Veranstaltungen, die verschiedenen Logos wie dem Kamel (O'Reilly), Zwiebel (Perl Foundation), Velociraptor (Perl 5) oder Camelia (Perl 6). Natürlich dürfen auch Ausführungen zu CPAN und Perl 5 Porters nicht fehlen.

Am Ende des Artikels listet Keating für jedes Jahr stichpunktartig ein paar wichtige Fakten für Perl auf.

TPF nicht teilnehmende Organisation beim Google Code-In

Trotz intensiver Vorbereitungen durch Paul Johnson und andere hat es die Perl Foundation in diesem Jahr nicht geschafft, als Organisation beim Google Code-In dabei zu sein. Schon beim Google Summer of Code war die Perl Foundation erstmals seit vielen Jahren nicht mit dabei. Auf Nachfrage hieß es von Seiten Googles, dass die Anzahl der Bewerbungen immer weiter steigt und man auch anderen Projekten die Chance der Teilnahme geben möchte.

Es ist zu hoffen, dass die Perl Foundation in den nächsten Jahren wieder mit von der Partie ist.

Abschluss des "Refactoring Perl 5 Debugger" Grants

Dieser Grant ist ein weiteres Beispiel für einen erfolgreichen Grant - auch wenn Shlomi Fish nicht alle seine selbstgesteckten Ziele erreicht hat. Er selbst ist der Meinung, dass der Code des Perl-Debuggers noch nicht so "schön" ist wie er sich das vorstellt. Dies ist auch der Tatsache geschuldet, dass Fish die Rückwärtskompatibilität bewahren wollte. Er wird aber auch nach Abschluss seines Grants weiter am Debugger arbeiten.

In seinem Abschlussbericht weist Shlomi Fish auch auf `Devel::Trepan` ist, ein von Rocky Bernstein geschriebener Debugger-Ersatz. Bernstein hat auch schon während des Grants immer wieder Fishs Entwicklungen kommentiert und so zu einer Verbesserung des Debuggers beigetragen.

Fish hat viele neue Tests für den Debugger hinzugefügt und den Code des Debuggers modularer gestaltet.

Fixing Perl 5 Core Bugs

In den vergangenen Monaten hat Dave Mitchell nur eingeschränkt an seinem Grant gearbeitet. Die meiste Zeit davon hat er an der Optimierung von mehreren aufeinander folgenden `my`-Anweisungen nach dem folgenden Muster:

```
for my $i (1..10_000_000) {
    my ($a,$b,$c);
    my $d;
    my (@e,%f);
    my $g;
    1;
}
```

Alle diese `my`s werden in einen neuen `OP` zusammengeführt, dem `PADRANGE`. Im Original wird für jede Variable einmal `padsv` ausgeführt.

Alien::Base

Nach Angaben von Joel Berger sind nur noch ein paar Kleinigkeiten zu erledigen bevor er Erfolg vermelden kann. Es gab in den letzten Monaten ein beta-Release und mehrere Entwicklerversionen auf dem Weg zu einem funktionierenden `Alien::Base`. Während einer "Projektnacht" der Chicago Perlmongers wurde an dem Modul gearbeitet. Eine gute Gelegenheit, dass auch andere Personen sich das Modul anschauen und Feedback geben.

Einen Nutzer hat das Modul auf jeden Fall schon: Toby Inkster, der ein `Alien::LibXML` kreiert hat. Dabei ist bei ihm ein Problem beim Auffinden von installieren Bibliotheken aufgefallen. Das zeigt auch, dass es für Module immer gut ist wenn auch schon im frühen Stadium Nutzer das Modul testen.

Während der Arbeiten hat Berger festgestellt, dass Strawberry Perl keinen Bash-Interpreter mit ausliefert und daher keine *configure*-Skripte ausführen kann. Aus diesem Grund kehrt er wieder zu einem alten Plan zurück und will vorkompilierte Bibliotheken für Windows bereitstellen.

Implementation of Macros in Rakudo

Carl Mäsak sieht sich aktuell bei der Hälfte des Grants angekommen. Rakudo hat jetzt Macros eingebaut - in einem frühen Stadium, aber sie funktionieren. Zu den erledigten Meilensteinen gehören die Makro-Deklaration und - Aufruf. Makros können genauso deklariert werden wie einfache Subroutinen. Dazu wurde das Schlüsselwort `macro` eingeführt. Ein kurzes Beispiel aus den Tests:

```
macro four { '4' }
my $foo = 100 + four;
is $foo, 104,
  "simple string returning macro (4)";
```

Makros unterstützen auch *quasiquoting* -- Quotes für Code, der einen Abstract Syntax Tree zurückliefert.

```
macro cupid {
   my $a = "I'm cupid!";

   quasi { $a }
}
  my $result = cupid;
  is $result, "I'm cupid!",
   "lexical lookup from quasi to macro
      works";
```

Improving Perl 5

Als erstes eine gute Nachricht: Der Grant wurde kurz vor Erscheinen der letzten Ausgabe von $foo verlängert. So hat Nicholas Clark weitere 400 Stunden zur Verfügung, in denen er sich um die Verbesserung von Perl 5 kümmern kann.

Im Gegensatz zu Dave Mitchell arbeitet Nicholas Clark nicht unbedingt an uralten Bugeinträgen von Perl, sondern kümmert sich ständig um Verbesserungen im Perl-Kern. So hat er sich die Änderungen am Hash-Algorithmus von Yves Orton (siehe auch Artikel über Hash Randomization in dieser Ausgabe) näher angeschaut und auch andere Sprachen betrachtet. Vor allem Bugeinträge in Python, Ruby und PHP über Hashing hat er sich zu Gemüte geführt und analysiert ob diese Fehler auch Perl betreffen können.

Ein paar Fehler an Yves' Implementierung hat Nick behoben und so sichergestellt, dass der Code auch unter HP-UX kompiliert.

Um das forking zu testen wurde `ulimit -u` benutzt. Damit wurde die Anzahl der User-Prozesse reduziert um ein Fehlschlagen des forkens zu provozieren. Auf einigen Plattformen wurde bei der Verwendung des Befehls aber ein Fehler ausgegeben, wodurch der komplette Test fehlschlug. Bei diesen Plattformen wird der Test jetzt übergangen.

Betriebssysteme, für die die Unterstützung für Perl in den letzten Monaten entfernt wurde:

- BeOS (die Perl-Portierung für Haiku, das als Ersatz für BeOS geschaffen wurde, wird aber weiterhin aktiv gepflegt)
- EPOC
- MPE/iX

Wenn jemand diese Betriebssysteme verwendet und eine Perl-Portierung für dieses braucht, sollte sich mit den Perl 5 Porters in Verbindung setzen wie er die Wartung der Portierung übernehmen kann ;-)

Spanish Localization of Perl Core Documentation

Über 6 Monate hinweg haben Enrique Nell und Joaquin Ferrero an der Übersetzung der Perl Dokumentation ins Spanische gearbeitet. Während dieser Zeit haben sie rund 65% der Kerndokumentation und 35% der sonstigen Dokumentation übersetzt.

Die Übersetzung wurde auf Basis der Dokumentation in Perl 5.16 vorgenommen. Aber nicht nur übersetzte Dokumente sind das Ergebnis des Grants, sondern auch Tools bzw. Verbesserungen an Tools, die auch anderen Übersetzungsteams helfen können. Eine kurze Beschreibung zu jedem Tool ist unter https://github.com/zipf/perldoc-es/tree/master/tools zu finden.

Eine übersetzte Version der Dokumentation wird besonders Perl-Einsteigern helfen.

Sponsoring MetaCPAN

Die Perl Foundation hat Equipment für MetaCPAN gekauft. MetaCPAN ist eine Alternative zu http://search.cpan.org, die ständig am wachsen ist und daher auch hin und wieder neue Hardware braucht.

Jahresbericht 2012

In einem 8-seitigen Bericht gibt die Perl Foundation einen Einblick in die geleistete Arbeit 2012 und die Gesamtsituation. In dem Rückblick werden die ausgerichteten Konferenzen in einem Schnelldurchlauf betrachtet, genauso wie der aktuelle "Zustand" der Community.

Der Schatzmeister der Perl Foundation gibt einen kurzen Einblick in die Ausgabensituation in 2012: Insgesamt wurden über 100.000 USD an Grants ausbezahlt. Das sind zum einen die Grants aus dem Perl 5 Spendentopf und zum Anderen die regulären Grants. Für die regulären Grants kann man sich quartalsweise bewerben. Mit diesen Grants werden (kleinere) Projekte gefördert, die für die Perl-Allgemeinheit von Nutzen sind. Die Fördersumme beträgt dabei zwischen 500 USD und 2.000 USD.

Aktueller Stand des Perl 5 Core Maintenance Fund

Dan Wright hat ein paar Zahlen zum aktuellen Stand des Perl 5 Core Maintenance Fund veröffentlicht. Demnach wurden rund 280.000 USD an Spenden eingenommen. Davon wurden schon rund 166.000 USD ausgegeben oder sind schon Grants zugewiesen. Das heißt, dass noch rund 116.000 USD zur Verfügung stehen.

In einem Google Doc listet Dan Wright auch auf, wann wie viel Geld gespendet wurde. In den letzten Monaten kamen dabei nur ein paar kleinere Spenden zusammen, während der größte Batzen vor gut einem Jahr eingegangen ist.

Die Empfänger der umfangreichsten Grants sind Dave Mitchell und Nicholas Clark, die am Perl-Interpreter selbst arbeiten. Shlomi Fish hat Änderungen am Perl 5 Debugger vorgenommen, Paul Johnson arbeitet an Verbesserungen von Devel::Cover und Jess Robinson verbessert die Cross-Compiling-Möglichkeiten von Perl.

Diese Beispiele zeigen, dass so ein Spendentopf sehr nützlich ist.

CPANNews XXV

Sub::Lazy

Vermutlich gibt es in jeder Anwendung Subroutinen-Aufrufe, deren Ergebnis in so manchem Durchlauf gar nicht benötigt wird. Meistens ist das kein großes Problem, weil die Subroutine schnell ausgeführt ist und das Ergebnis schnell berechnet ist. Manche Subroutinen laufen aber lang oder das Ergebnis verbraucht viel Speicher. Beides ist nicht erstrebenswert wenn die Berechnung am Ende umsonst war.

Dieses Problem wird mit dem Modul `Sub::Lazy` angegangen. Ziel des Moduls ist es, die Ausführung der Subroutine so lange herauszuzögern, bis das Ergebnis tatsächlich benötigt wird. Dazu wird der Subroutine das Attribut `:Lazy` vergeben (siehe auch $foo Ausgabe #5). Die Hauptarbeit hinter dem Modul macht das Modul `Data::Thunk`.

Aber nicht immer lohnt es sich, die Ausführung von Subroutinen zu verzögern. Zwei Punkte, an denen man Kandidaten für die Verwendung von `Sub::Lazy` erkennt:

• Es gibt keine Seiteneffekte wie z.B. das Verändern von Globalen Variablen. Auch Subroutinen mit Ein-/Ausgabe oder Systemaufrufen sind nicht geeignet.

• Sie werden nur im Skalaren Kontext aufgerufen. Das Modul erzeugt bei den Subs immer einen Skalaren Kontext. Wenn die verzögerte Subroutine eine Liste zurückliefert, wird das Ergebnis unter `Sub::Lazy` nicht das gewünschte Ergebnis sein.

Der nebenstehende Code zeigt die Verwendung des Moduls.

```
use Test::More;
use Sub::Lazy;

my $it_happens = 0;

sub double :Lazy {
   $it_happens++;  # side-effect

   my $n = shift;
   return $n * 2;
}

my $eight = double(4);

# The 'double' function hasn't been
# executed yet.
is($it_happens, 0);

# The correct answer was calculated.
is($eight, 8);

# The 'double' function was executed
# when necessary.
is($it_happens, 1);

done_testing;
```

Net::IPAddress::Filter

Es gibt vielschichtige Gründe, warum man in seiner Anwendung einen Filter für IP-Adressen einbauen möchte. Z.B., weil man den Zugriff auf bestimmte Daten nur für Benutzer erlauben möchte, die aus einem bestimmten Netzwerksegment kommen oder als OTRS-Postmaster-Filter, um schneller Spam zu erkennen. So lange man nur einzelne IP-Adressen filtern möchte, kann man das noch händisch machen mit einfachen Vergleichen:

```
use List::Util qw(first);
my $ip  = '10.0.0.2';
my @ips =
    qw(10.0.0.1 127.0.0.1 192.168.2.5);
say "found!" if first{ $ip eq $_ }@ips;
```

Wenn es um Bereiche geht, wird es schon wieder etwas schwieriger, wobei es bei Bereichen mit CIDR-Schreibweise noch relativ einfach ist:

```
my $ip = '10.0.0.2';
my $range = '10.0.0.0/8';

my ($range_ip,$bits) = split /\//, $range;
my $int_range      = get_int( $range_ip );
my $int_ip         = get_int( $ip );
my $shift          = 32 - $bits;

say "found!" if
  $int_ip >> $shift ==
                     $int_range >> $shift;

sub get_int {
  return
    unpack 'N',
      pack 'C4',
        split /\./, shift;
}
```

Was ist aber, wenn man alles zwischen *192.168.2.3* und *192.168.2.11* filtern möchte? An dieser Stelle wird auf den Code verzichtet, mit dem man auch das selbst machen kann. Auf CPAN gibt es jede Menge Module, die einen dabei unterstützen können. Eines davon ist `Net::IPAddress::Filter`, das erst im November ganz neu auf CPAN erschien. Man kann beliebige Adressbereiche hinzufügen, egal ob im CIDR-Format, eine einzelne IP-Adresse oder einfach Start- und End-Adresse.

```
my $filter = Net::IPAddress::Filter->new();

$filter->add_range('172.168.0.0/24');
$filter->add_range('192.168.1.1');
$filter->add_range(
  '10.0.0.10',
  '10.0.0.50',
);
```

Danach kann man mit der Methode `in_filter` prüfen, ob eine IP-Adresse innerhalb der gegebenen Bereiche liegt:

```
my $ip = '172.168.0.10';
say "found!" if $filter->in_filter( $ip );

my $ip = '172.198.0.10';
say "found!" if $filter->in_filter( $ip );

my $ip = '192.168.1.1';
say "found!" if $filter->in_filter( $ip );

my $ip = '192.168.1.2';
say "found!" if $filter->in_filter( $ip );

my $ip = '10.0.0.11';
say "found!" if $filter->in_filter( $ip );

my $ip = '10.0.0.52';
say "found!" if $filter->in_filter( $ip );
```

Das Modul bietet außerdem die Möglichkeit, Anmerkungen zu den Adressbereichen zu speichern und bei der Prüfung wieder abzurufen. Das kann äußerst hilfreich sein, wenn man die Treffer des Filters mitloggen möchte. Dann ist eine lesbare Ausgabe besser als einfach nur die Adressbereiche im Log zu haben.

```
$filter->add_range_with_value(
  'IANA-reserved range',
  '10.0.0.0',
  '10.255.255.255',
);

my $array_ref = $filter->get_matches(
  '10.128.0.0'
); # [ 'IANA-reserved range' ]
```

Einen Nachteil hat das Modul gegenüber anderen Modulen auf CPAN: Es kann (noch) nicht mit IPv6-Adressen umgehen. Das steht aber auf der Todo-Liste des Autors.

JavaScript::Console

Wer Webanwendungen entwickelt, wird mit Sicherheit die JavaScript-Konsole kennen. Für Firefox gibt es z.B. Firebug, das eine Konsole mitbringt. Mit `JavaScript::Console` kann man Debugging-Ausgaben auf genau diese Konsole schreiben. Das Ergebnis ist in der Abbildung 1 zu sehen.

Die Gruppierung (`group` bis `group_end`) ist ganz nützlich wenn eine größere Menge an Debugging-Ausgaben zu erwarten ist. Denn mit den Gruppen können viele Ausgaben ausgeblendet werden. Die Ausgaben funktionieren aber auch ohne das Gruppieren.

Wie am Code und der Abbildung zu erkennen, gibt es verschiedene Level an Ausgaben. Hier wird eine "normale" Ausgabe (`log`) und eine Warnung (`warn`) ausgegeben. Es gibt auch noch `debug`, `info` und `error`. Interessant ist auch die Funktion `div_by_id`, die jede Menge Informationen zu einem Element ausgibt.

```
use Mojolicious::Lite;
use JavaScript::Console;

my $console = JavaScript::Console->new();

get '/' => sub {
   my $self = shift;
   $console->group( 'start' );
   $console->log
     ( 'logging with JavaScript::Console' );
   $console->warn( 'this is for $foo #25' );
   $console->group_end;

    $self->render(
       'index',
       console_output => $console->output,
   );
};

app->start;

__DATA__
@@ index.html.ep
Please open a JavaScript console
<%= Mojo::ByteStream->new(
   $console_output ) %>
```

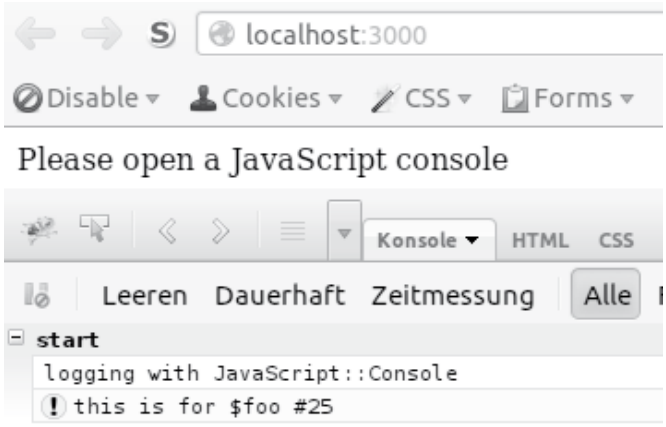


Abb. 1: Beispiel für JavaScript-Konsole-Debugging

Encode::BetaCode

Betacode ist ein Verfahren, altgriechische Text mit ASCII-Zeichen darzustellen. Ursprünglich wurde es für einen allgemeineren Einsatz entwickelt, aber Unicode löst Betacode in den meisten Fällen ab. Ein Nachteil von Unicode ist es, dass Zeichen mit unterschiedlicher Bedeutung als identisch angesehen werden. Ein Beispiel dafür ist `0374 GREEK NUMERAL SIGN` (griechische Buchstaben werden als Zahlen verwendet) und `02B9 MODIFIER LETTER PRIME`, aber das wird verwendet für Nachdruck, was etwas ganz anderes ist.

Doch zurück zu den altgriechischen Texten. Wenn dort die Zeichen □□ auftauchen und diese Zeichen als ASCII dargestellt werden müssen, kommt Betacode zum Einsatz. Dort werden die Zeichen als `TH=|` dargestellt.

```
use feature 'say';


use Encode::BetaCode qw(
   beta_decode beta_encode);

say beta_decode( 'greek', 'th=|' );
say beta_encode( 'greek', 'Perseus',
                        "\x{03C4}\x{1FC7}" );
```

JSON::Schema

Daten sollten validiert werden, bevor sie Verwendung finden, insbesondere wenn sie für den Datenaustausch gedacht sind. Für XML gibt es XML::Schema. JSON-Daten können mit JSON::Schema geprüft werden. In dem Schema werden die erwarteten Daten im JSON-Format beschrieben.

```
use feature 'say';
use JSON::Schema;
use JSON qw(from_json);

my $schema = q~{
    "description" : "Magazine",
    "type" : "object",
    "properties" : {
        "name" : {
            "title" : "Magazine's name",
            "type" : "string",
            "required" : true
        },
        "issue_nr" : {
            "title" : "Issue of Magazine",
            "type" : "integer",
            "required" : true
        }
    }
}~;

my @data = (
  q~{ "name" : "$foo", "issue_nr" : 25 }~,
  q~{ "id" : 2, "name" : "Person2" }~,
);

for my $data ( @data ) {
    say "validate $data";

    my $perl = from_json( $data );
    my $validator = JSON::Schema->new(
                                      $schema );

    my $result = $validator->validate(
                                        $perl );
    if ( $result ) {
        say "ok";
    }
    else {
       say "Fehler: $_" for $result->errors;
    }
}

$ perl json_schema.pl
validate { "name" : "$foo",
             "issue_nr" : 25 }
ok
validate { "id" : 2, "name" : "Person2" }
Fehler: $.issue_nr: is missing and it
  is required
```

Parallel::Jobs

Gerade wenn es um zeitaufwändige Aufgaben geht, ist man um Parallelisierung bemüht. Diese Aufgabe selbst zu lösen ist nicht unbedingt einfach, vor allem wenn man daran denken muss, auf `STDOUT` und `STDERR` Zugriff zu erhalten. Ein Modul, das das parallele Ausführen von Jobs erleichtert, ist `Parallel::Jobs`:

```
use Parallel::Jobs qw(start_job watch_jobs);

my $pid = Parallel::Jobs::start_job(
  'perl', '-E', 'sleep 3; say $$'
);

open my $fh, '>', '/tmp/stderr.log';
Parallel::Jobs::start_job(
    {
        stderr_handle => $fh,
    },
    'perl', '-E', 'sleep 3; warn "PID: $$"'
);
```

Perl5-Porters-News

Entwicklerversionen in veröffentlichten Perl-Versionen

Einige Module, die mit dem Perl-Kern ausgeliefert werden, werden direkt von den Perl 5 Porters geändert und erhalten dann eine Entwicklerversion. Bei ExtUtils::MakeMaker sind dabei Probleme aufgetaucht: `carton` konnte nicht das "richtige" Modul auf CPAN finden, so dass Jeffrey Thalhammer bei P5P angefragt hat, ob man in Zukunft nicht auf Entwicklerversion verzichten könne. Ricardo Signes sicherte zu, dass in Zukunft darauf geachtet wird, dass es möglichst keine Entwicklerversionen mehr im Perl-Kern gibt. Ganz verhindern lässt sich das aber nicht.

No-Taint-Unterstützung in Perl

Der Taint-Modus ist für Webentwickler sehr hilfreich, weil es die Paranoia vor bösen Benutzereingaben unterstützt. Durch die vielen Extraprüfungen im Perl-Kern gibt es aber einen Performanzverlust, selbst wenn kein Taint-Modus verwendet wird. Steffen Müller hat eine Compiler-Option eingebaut, mit der ein Perl gebaut werden kann, mit dem es keinen Taint-Modus mehr gibt: `-DNO_TAINT_SUPPORT`.

perl-5.16.2 veröffentlicht

Ricardo Signes hat die Version 5.16.2 von Perl veröffentlicht.

Was ist mit der "Small Core"-Idee passiert?

Jesse Vincent und Ricardo Signes haben in den letzten Jahren die Idee verbreitet, dass der Perl-Kern kleiner werden sollte und mehr Funktionalität in Module ausgelagert werden soll. Das vor dem Hintergrund, dass Perl nicht wirklich wartbar ist und mit den Modulen die Verantwortung auf viele Schultern verteilt wird.
Peter Rabbitson beklagt, dass diese Idee nicht gelebt wird, sondern weiterhin immer mehr Features in den Kern kommen.

Eine interessante Diskussion, die unter *http://markmail.org/message/u5kodpbmizbohrkx* nachgelesen werden kann.

Subroutinen-Signaturen

Signaturen für Subroutinen und Methoden sind ein begehrtes Feature, aber bisher ist nichts im Core enthalten. Peter Martini hat in diesem Jahr viel Zeit in einen Vorschlag gesteckt, der viel bei den Perl 5 Porters diskutiert wurde. Es gibt aber für viele Grenzfälle viele verschiedene Meinungen.

Und auch wegen der eben genannten Diskussion um einen schmaleren Kern, ist dieses Feature (noch) nicht in den Kern eingeflossen. Martini möchte an einer allgemeingültigeren API arbeiten, die das Entwickeln von Signaturen modularer macht.

Entfernen des rehash-Mechanismus in Perl 5.18

Statt des ständigen *rehashens* gibt es in Perl 5.18 einen zufälligen Hash seed pro Lauf. Mehr dazu gibt es in dieser Ausgabe in einem separaten Artikel. Diese Änderung hat nicht nur mehr Sicherheit zur Folge sondern auch geringere Kosten (Speicher und CPU), da der Hash nicht ständig neu berechnet wird.

Benchmarking Murmurhash3 vs. One-At-A-Time als Perls Hash-Funktion

Mit der Einführung des "Pro-Lauf-Hash-Seeds" hat Yves Orton auch zusätzliche Hash-Funktionen eingebaut, die beim Kompilieren von Perl aktiviert werden können. Orton hat zwei Hash-Funktionen gegeneinander antreten lassen: http://markmail.org/message/kufkc3esxdyzujlr

RFC: Einführung des prototype-Attributs

Peter Martini schlägt die Einführung des *prototype*-Attributs vor. Was aktuell die Prototypen bei Perl machen, wurde in der letzten Ausgabe (Winter 2012) vorgestellt. Der Nachteil der aktuellen Implementierung (`sub foo ($$$) {}`) ist der, dass die Prototypen erst an die Subroutine gebunden werden wenn die komplette Subroutine definiert wurde.

Mit einem *prototype*-Attribut könnten die Prototypen flexibler genutzt werden. Die Syntax würde wie folgt aussehen: `sub foo : prototype($$$) {}`.

http://markmail.org/message/vrqyk34zjf4otawy

Nichtdokumentiertes Verhalten von %ENV

Kent Fredric wurde von bestimmten Verhalten (in blead) von *%ENV* überrascht. Alle Werte werden - im Gegensatz zu "normalen" Hashes - stringifiziert. Er hat ein paar Patches bereitgestellt, mit denen man gewarnt wird sollte, man eine Referenz in *%ENV* speichern wollen.

Sicherheitshinweis: Storable

Ricardo Signes hat einen Sicherheitshinweis zum Modul `Storable` veröffentlicht. Das Modul ist im Perl-Kern enthalten. Er schreibt:

Im Laufe der Zeit gab es ein paar Mal Diskussionen über Storable als Angriffsweg. Wenn ein Benutzer dir Storable-Daten unterschieben kann, die du nicht erwartest, hat er gute Chancen, böse Dinge mit deinem Programm zu machen. Das wurde bei den Perl 5 Porters und auf YAPCs diskutiert, aber leider nie dokumentiert. Es wurde jetzt nachgeholt.

Dank an Brian Carlson von cPanel, der das P5P Sicherheitsteam darauf aufmerksam gemacht hat.

Sicherheitshinweis: Locale::Maketext

Ein weiterer Sicherheitshinweis von Brian Carlson betrifft `Locale::Maketext`. Dieses Modul wird ebenfalls mit dem Perl-Kern mitgeliefert. Ricardo Signes fasst das Problem folgendermaßen zusammen:

• In einem `[method,x,y,z]`-Template kann die Methode ein vollqualifizierter Name sein.

• Die Auflösung des Templates hat Metazeichen nicht richtig gequotet und damit Code-Injection über ein bösartiges Template ermöglicht.

Eine Aufgabe für dich? SvREFCNT_dec_NN()

Dave Mitchell hat ein neues Makro zum Perlinterpreter hinzugefügt: `SvREFCNT_dec_NN()`. Dieses Makro reduziert den Referenzzähler für Skalare und kann in Situationen verwendet werden, in denen der Skalar nicht Null sein kann. Das spart Code, Tests und einen Sprung im Codepfad. Es gibt 500 Vorkommen von SvREFCNT_dec im Code.

Wenn du Lust hast, mal in die Tiefen des Perl-Kerns einzutauchen und eine relativ einfache Aufgabe brauchst um damit anzufangen, dann könnte das etwas für dich sein.

Hash-Zuweisungen

Es gibt ein seltsames Verhalten wenn einem Hash eine ungerade Anzahl an Elementen zugewiesen wird. Ruslan Zakirov hat seine Arbeiten an Patches abgeschlossen. Damit wird das Verhalten in Perl 5.18 konsistenter:

- scalar(%h = (1,1,1,1)) liefert jetzt 4, nicht 2
- warnt bei ($s,%h) = (1,{}) genauso wie bei (%h) = ({})
- (%h = (1)) liefert im Listenkontext (1, $h{1}) statt (1)
- Rückgabe von (%h = (1,1,1)) im Listenkontext war falsch

CERT Perl Secure Coding Standard

Yves Orton fragt sich, ob jemand den CERT Perl Secure Coding Standard gesehen hat. In diesen Standards sind einige Dinge festgehalten, die man für die Entwicklung sicherer Perl-Anwendungen beachten sollte. Jeffrey Thalhammer antwortet, dass er hofft, dass Anfang diesen Jahres ein Perl::Critic-Modul veröffentlicht wird, das diese Richtlinien umsetzt.

perl5db Refactoring

Shlomi Fish hat seine Arbeiten am perl5db-Grant der Perl Foundation abgeschlossen und fragt nach Ideen für weitere Features des Perl-Debuggers. Einige Ideen wurden von Rocky Bernstein vorgebracht. Eine davon ist, dass der Debugger in mehrere Dateien aufgeteilt werden sollte und das REPL-Interface von der Kern-API getrennt werden sollte.

rand() benutzt nur 15 bits Entropie unter Windows

Was würdest du erwarten, wenn du dieses Programm laufen lässt?

```
my %nums;
my $cnt = 0;
foreach my $i (1 .. 1_000_000) {
    my $num = rand;
    $cnt++ if ($nums{$num});
    $nums{$num} = 1;
}
print "$cnt out of 1,000,000\n";
```

Wenn Du es unter Windows laufen lässt, könntest Du überrascht sein, dass es jedesmal `967232 out of 1,000,000` ausgibt. Das bedeutet, dass es nur 32.768 mögliche Fließkommazahlen zwischen 0 und 1 gibt, die `rand()` auf dieser Plattform generiert. Andere Plattformen scheinen nicht betroffen zu sein.

COW-Status

Der COW-Branch von Father Chrysostomos ist so weit fortgeschritten, dass alle Core-Tests bis auf einen durchlaufen. Er wird noch mehr Benchmarks laufen lassen und den Branch noch etwas aufräumen.

Pull request CPAN-1.99_51

Da die VMS-Unterstützung in CPAN.pm soweit verbessert wurde, dass es wirklich funktioniert, wurde die Version 1.99_51 in den Perl-Kern übernommen. Damit wird es möglich, CPANPLUS wieder aus dem Kern zu entfernen. Das wurde nur hinzugefügt, weil die Verwendung von CPAN.pm unter VMS nicht möglich war.

Termine

Februar 2013

02./03. FOSDEM
05. Treffen Stuttgart.pm
Treffen Frankfurt.pm
07. Treffen Dresden.pm
09. Bulgarischer Perl-Workshop
11. Treffen Ruhr.pm
13. Treffen Niederrhein.pm
18. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
24. Israelischer Perl-Workshop
26. Treffen Bielefeld.pm
27. Treffen Berlin.pm

März 2013

05. Treffen Stuttgart.pm
Treffen Frankfurt.pm
07. Treffen Dresden.pm
11. Treffen Ruhr.pm
13. Treffen Niederrhein.pm
13.-15. Deutscher Perl-Workshop
16./17. Chemnitzer Linuxtage
21. Treffen Darmstadt.pm
22. Schweizer Perl-Workshop
27. Treffen Berlin.pm

April 2013

02. Treffen Frankfurt.pm
Treffen Stuttgart.pm
04. Treffen Dresden.pm
08. Treffen Ruhr.pm
10. Treffen Niederrhein.pm
15. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
24. Treffen Bielefeld.pm
30. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss.

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

Kennen Sie weitere Termine, die in der nächsten Ausgabe von $foo veröffentlicht werden sollen? Dann schicken Sie bitte eine EMail an:

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

Unter Perl-Nachrichten.de sind deutschsprachige News rund um die Programmiersprache Perl zu finden. Jeder ist dazu eingeladen, solche Nachrichten auf der Webseite einzureichen.

http://www.perl-community.de

Perl-Community.de ist eine der größten deutschsprachigen Perl-Foren. Hier ist aber nicht nur ein Forum zu finden, sondern auch ein Wiki mit vielen Tipps und Tricks. Einige Teile der Perl-Dokumentation wurden ins Deutsche übersetzt. Auf der Linkseite von Perl-Community.de findet man viele Verweise auf nützliche Seiten.

http://www.perlmongers.de/
http://www.pm.org/

Das Online-Zuhause der Perl-Mongers. Hier findet man eine Übersicht mit allen Perlmonger-Gruppen weltweit. Außerdem kann man auch neue Gruppen gründen und bekommt Hilfe...

http://www.perlfoundation.org

Die Perl-Foundation nimmt eine führende Funktion in der Perl-Gemeinde ein: Es werden Zuwendungen für Leistungen zugunsten von Perl gegeben. So wird z.B. die Bezahlung der Perl6-Entwicklung über die Perl-Foundationen geleistet. Jedes Jahr werden Studenten beim "Google Summer of Code" betreut.

http://www.Perl.org

Auf Perl.org kann man die aktuelle Perl-Version downloaden. Ein Verzeichnis mit allen möglichen Mailinglisten, die mit Perl oder einem Modul zu tun haben, ist dort zu finden. Auch Links zu anderen Perl-bezogenen Seiten sind vorhanden.

Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de