Ausgabe 2/2013 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Perl 6 Exceptions
Für den Ausnahmefall gerüstet

LOG::REPORT
Exceptions, Logging und Übersetzungen

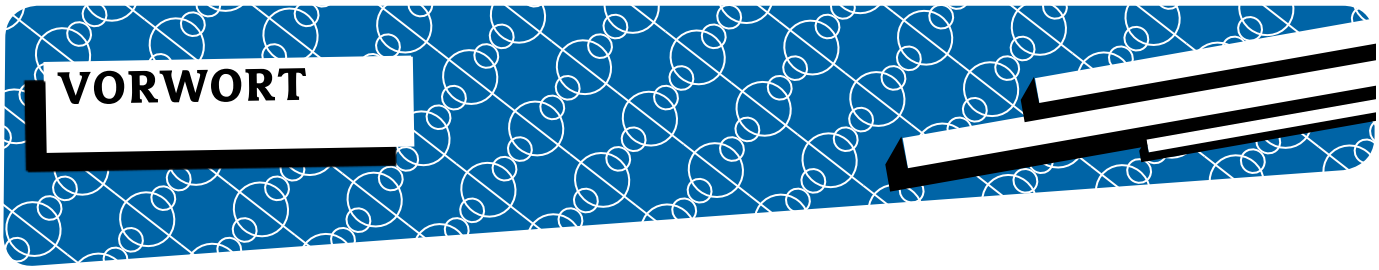
Mosaik-Bilder
Ein Geschenk für die Liebsten

Nr. 26

Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de



Perl Werbung

An dieser Stelle habe ich schon häufiger über "Werbung für Perl" gesprochen. Da es für mich ein wichtiges Thema ist, beschäftige ich mich auch in dieser Ausgabe wieder an dieser Stelle damit. In den letzten Wochen gab es ein paar Neuigkeiten.

Auf dem Deutschen Perl-Workshop in Berlin hat Richard Jelinek einen guten Vortrag mit dem Titel "Umrisse einer Perl-Strategie" gehalten. Darin ist er auch darauf eingegangen, wie das Ansehen und die Beliebtheit von Perl ist und wie man diese Dinge verbessern kann. Kurz darauf hat er Propaganda.pm (http://propaganda.pm) gegründet, die sich als Ziel gesetzt hat, Perl zur beliebtesten Programmiersprache zu machen. Ein ambitioniertes Ziel, aber die Mitglieder der Gruppe haben sich schon ein paar Gedanken gemacht, wie sie dieses Ziel erreichen wollen.

Als Reaktion auf Jelineks Vortrag kamen einige Kommentare die darauf hinwiesen, dass es auch zu wenige Anwendungen in Perl gibt und Perl-Programmierer(-innen) solche Anwendungen schreiben sollen - gerade für den Webbereich, der in der heutigen Zeit immer wichtiger wird. Der Einstieg in die Programmierung kommt häufig über die Entwicklung von Webanwendungen. Torsten Raudssus hat es so ausgedrückt: "Keiner hat PHP gewählt, alle haben Wordpress gewählt". Aber auf dem Markt gibt es häufig keine konkurrenzfähigen Perl-Anwendungen oder sie sind nicht bekannt genug.

Gabor Szabo hat eine Liste von Perl-basierten Open-Source-Anwendungen zusammengestellt: http://szabgab.com/perl-based-open-source-products.html . Wer noch weitere Anwendungen kennt, kann diese in den Kommentaren von Gabors Post auflisten.

So eine Anwendung zu schreiben kostet Zeit und damit auch Geld. Eine - für Perl-Projekte - neue Art der Finanzierung haben brian d foy und Jeffrey Thalhammer ausprobiert. Sie haben ein Crowdfunding-Projekt gestartet um ein neues Feature für *Pinto* (siehe $foo Ausgabe 21) von der Gemeinschaft finanzieren zu lassen. Dazu haben sie die in Perl geschriebene Plattform Crowdtilt gewählt. 6 Tage vor Ablauf der Zeitspanne, in der man seine Unterstützung für das Projekt kundtun konnte, wurde das Minimalziel von rund 4.100 USD erreicht. Vielleicht kann das ein Vorbild für weitere Perl-Projekte sein, da auch andere Open-Source-Projekte schon erfolgreich solche Crowdfunding-Runden hinter sich gebracht haben.

Eine Marketing-Möglichkeit für Perl-Projekte sind auch Stände und Vorträge auf Messen und Konferenzen. Wir versuchen einen Perl-Stand auf der FrOSCon (24./25. August) und der OpenRheinRuhr (09./10. November) und für die FrOSCon noch zusätzlich einen Projektraum zu bekommen. Für die Stände möchten wir Plakate erstellen, auf denen einige Projekte kurz vorgestellt werden. Welche Projekte sollen wir vorstellen? Bitte stimmen Sie unter *http://projekte.perl-magazin.de* ab!

Viel Spaß beim Lesen der neuen Ausgabe.

Renée Bäcker

Die Codebeispiele können mit dem Code

382mB4

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

Ab dieser Ausgabe werden alle weiterführenden Links auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue26.

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

MODULE

PERL

ANWENDUNG

NEWS

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

Katrin Bäcker

Zur Zeit Vollzeit-Mami zweier Söhne (5 Monate und 2 Jahre) und Ehefrau des Herausgebers dieses Magazins. Vor der Elternzeit aber auch in der IT-Branche tätig (als Wirtschaftsinformatikerin) - allerdings ohne Perl-Know-How.

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 Workshop und Mitglied im Orga-Team des deutschen Perl-Workshops.

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, der auch äußerlich versucht die Perlphilosophie umzusetzen. Er war darüber hinaus am Aufbau der Wikipedia-Kategorie "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

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/

Moritz Lenz

Moritz Lenz wurde 1984 in Nürnberg geboren. Seine Brötchen verdient er mit Perl 5, in der Freizeit entwickelt er Perl 6. Daneben sind seine Lieblingsthemen Kryptographie und Sicherheitsaspekte, reguläre Ausdrücke und Unicode.

Mark Overmeer

Mark Overmeer ist ein fleißiger Programmierer - mit einigen großen Perl Projekten. Auf CPAN findet man Mail::Box --for automatic email processing-- und XML::Compile --XML processing--. Außerdem arbeitet er an CPAN6, dem Nachfolger des CPAN Archivs. Mark hat einen Abschluss als Master in Computing Science und arbeitet zur Zeit als selbständiger Programmierer, Systemadministrator und Trainer. Er ist Geschäftsführer von NLUUG (früher bekannt als die Niederländische UNIX User Group), SPPN (Niederländische Perl Promotion Foundation) und Oophaga (CAcert Equipment). Und das neben vielen anderen Aktivitäten. Mehr Informationen unter http://solutions.overmeer.net.

Moritz Lenz

Für den Ausnahmefall gerüstet

Perl 6 bewahrt die einfache Benutzbarkeit von Perl 5 und untermauert dabei viele Eigenschaften mit einem soliden, erweiterbaren Modell.

So ist das auch bei Exceptions. Im einfachsten Fall schmeißt man Exceptions mit `die "Fehlermeldung";`, und fängt sie mit `try { code-der-sterben-kann() }`. Die Fehlermeldung landet in `$!`.

Soweit überrascht es den Perl 5-Kenner noch nicht, nur `eval BLOCK` heißt jetzt `try`. Wie immer kommt eine kleine syntaktische Besonderheit hinzu: Die geschweiften Klammern sind jetzt optional, `try code-der-sterben-kann()` geht auch.

Objekte, Objekte.

Um beim Abfangen von Fehlern mehr Informationen zur Verfügung zu haben, wirft man Objekte als Exceptions. Damit das reibungslos funktioniert, erben die Fehlerklassen von der eingebauten Klasse `Exception`. Es ist Konvention, Fehlerklassen in den `X::`-Namensraum zu stellen. Die Stringdarstellung eines Fehlerobjekts wird dabei durch die Methode `message` kontrolliert (siehe Listing 1).

Das liefert die Ausgabe:

```
Du kommst hier nicht rein!
(Mindestalter 18, du bist aber nur 16)
  in sub enter-disco at disco.pl:13
  in block  at disco.pl:20
```

Wenn man die Exception mit `try` abfängt, sind in `$!` alle wichtigen Informationen enthalten:

```
try enter-disco(16);
say "Probiere es in ",
    $!.min-age - $!.actual-age,
    " Jahren noch einmal.";
```

```
class X::Disco::TooYoung is Exception {
    has $.min-age;
    has $.actual-age;
    method message() {
        "Du kommst hier nicht rein!\n"
        ~ "(Mindestalter $.min-age, du bist aber nur $.actual-age)";
    }
}

sub enter-disco($your-age) {
    if $your-age < 18 {
        die X::Disco::TooYoung.new(
              actual-age => $your-age,
              min-age    => 18,
          );
      }
      say "Going to the clubs";
  }
 enter-disco(16);
```

Listing 1

Selektives Abfangen

Um nur bestimmte Exceptions abzufangen, kann man statt der Holzhammermethode `try` einen eigenen `CATCH`-Block schreiben. Wenn darin ein `when`-Block (bekannt aus `given/when` aus Perl 5.10) zur Exception passt, wird er ausgeführt und der äußere Block verlassen.

```
my (@accepted, @rejected);
for 17..19 -> $age {
    enter-disco($age);
    @accepted.push: $age;
    CATCH {
        when X::Disco::TooYoung {
            @rejected.push: $age;
        }
    }
}
say "Reingekommen: @accepted[]";
say "Abgewiesen:   @rejected[]";
```

liefert als Ausgabe:

```
Going to the clubs
Going to the clubs
Reingekommen: 18 19
Abgewiesen:   17
```

Sollte kein `when`-Block passen (und kein `default`-Block vorhanden sein), lässt der `CATCH`-Block die Exception durch. So stirbt der folgende Code, als gäbe es keinen `CATCH`-Block:

```
die "Nicht vom Typ X::Disco::TooYoung";
CATCH { when X::Disco::TooYoung { } }
```

Dieser Mechanismus ist zum Beispiel hilfreich, um temporäre und permanente Fehler bei Netzwerkzugriffen zu unterscheiden. Kontrollcode kann etwa temporäre Fehler abfangen und die Operation noch einmal versuchen, was bei permanenten Fehlern (z.B. fehlenden Berechtigungen um einen Socket zu öffnen) nicht sinnvoll ist.

Back to the roots

Zurück zum Basisfall `die "Fehlermeldung"` und zur vagen Äußerung, dass Fehlerklassen von der eingebauten Klasse `Exception` erben sollten. Bekommt `die` ein Argument, was nicht zur Klasse `Exception` passt, -- wie etwa ein String -- so landet in `$!` ein Objekt vom Typ `A::AdHoc`, der wiederum von `Exception` erbt. Unter dem Attribut `$!.payload` findet man das Objekt, das `die` übergeben wurde:

```
try die 42;
say $!.WHAT;            # (X::AdHoc);
say $!.payload;         # 42
say $!.payload.WHAT;    # (Int)
```

Damit hat man in `$!` sowohl Zugriff auf das ursprüngliche Objekt, das an `die` übergeben wurde, als auch auf die zusätzlichen Methoden, die `Exception` mitbringt. Dazu gehört `$!.rethrow`, was die Exception noch einmal wirft, und dabei so aussehen lässt, als komme sie aus dem vorherigen Kontext:

```
sub a { b(); }
sub b() { die "tot"; }
try a();
say "lebendig";
$!.rethrow;
```

Ausgabe:

```
lebendig
tot
  in sub b at rethrow.pl:2
  in sub a at rethrow.pl:1
  in block  at rethrow.pl:3
```

Außerdem bietet die Klasse `Exception` noch die Methode `backtrace`, deren Rückgabewert es erlaubt, den Weg der Exception durch den Quellcode zurückzuverfolgen.

eval

`eval` gibt es in Perl 6 immer noch. Es hat die Doppelfunktion, die es in Perl 5 hat, jedoch abgelegt, und führt nur noch Zeichenketten als Perl 6-Code aus. Exceptions fängt es nicht mehr ab. Um Fehler aus `eval` abzufangen, muss man jetzt `try eval '1+1'` schreiben.

Mit gutem Beispiel voran

Sowohl der Perl 6 Compiler Rakudo als auch viele eingebaute Funktionen werfen als Fehler jetzt Objekte mit reichen Zusatzinformationen.

So führt etwa der Code

```
sub a { }
sub a { }
```

Zum Fehlertext `Redeclaration of routine a`, bei dem man anhand des Typs `X::Redeclaration` erkennen kann, was schief gelaufen ist, `.symbol` sagt einem, dass das Symbol `a` mehrfach deklariert wurde, und `.what` liefert einem mit dem Wert `"routine"` genauere Informationen darüber, was für eine Art Symbol da mehrfach deklariert wurde. Durch Testen auf die Rolle `X::Comp` ist leicht feststellbar, dass es sich um einen Fehler zur Compilezeit handelt.

Mehr Kontext!

Häufig geben Programme zwar Fehlermeldungen aus, welche Datei nicht schreibbar war, verschweigen aber, warum das Programm probiert hat, diese Datei zu schreiben. In anderen Worten, sie geben nicht genug Kontext in Fehlermeldungen, um den Benutzer verstehen zu lassen, welchem Zweck die fehlgeschlagene Aktion dient.

Für den Programmierer können Stacktraces diese Art von Kontext liefern, aber den normalen Benutzer möchte man nicht damit belästigen. Um trotzdem sinnvollen Kontext auszugeben, und auch um die Flexibilität der Exceptions zu demonstrieren, sei hier eine Möglichkeit vorgestellt, einzelnen Subroutinen einen erklärenden Text mitzugeben.

```
# Deklaration des Traits 'user-facing',
   # mit dem Subroutinen markiert werden können
   multi sub trait_mod:<is>(Routine:D $r, :$user-facing!) {
       my role Described[$by] {
           method descr() { $by }
       }
       $r does Described[$user-facing];
   }

   # Testroutine: nicht extra markiert
   sub save-state() {
       save-config();
   }

   # Testroutine: markiert
   sub save-config() is user-facing('saving user config') {
       my $f = open :w, '/no/such/path';
   }

   try save-state();
   if $! {
       # Ausgabe der Fehlermeldung und extra Kontext:
       say "Error: $!.message()";
       for $!.backtrace.list {
           if .code.^can('descr') {
               say "  while ", $_.code.descr()
           }
       }
   }
```

Listing 2

Wenn eine Exception abgefangen wird, geht der Code durch alle aufrufenden Funktionen, und gibt nur die aus, die einen solchen erklärenden Text haben (siehe Listing 2).

Ausgabe:

```
Error: Unable to open filehandle
 from path '/no/such/path'
  while saving user config
```

Zum Vergleich: Ohne die extra Markierung und das Filtern sieht die Ausgabe so aus:

```
Unable to open filehandle from
     path '/no/such/path'
  in method open at
     src/gen/CORE.setting:7702
  in sub open at src/gen/CORE.setting:7953
  in sub save-config at dba-ex.pl:14
  in sub save-state at dba-ex.pl:10
  in block  at dba-ex.pl:17
```

Das System ist also flexibel genug, eigene Fehlerbehandlung und -diagnose zu erlauben. Trotzdem ist es im Basisfall einfach benutzbar.

Weiterführende Literatur

Exceptions sind in der Perl 6 Spezifikation in S04 *http://perlcabal.org/syn/S04.html* (allgemeine Funktion) und in *http://perlcabal.org/syn/S32/Exception.html* S32::Exception (Details der Exception-Klassen) beschrieben.

http://doc.perl6.org/ listet bei vielen eingebauten Fehlerklassen Beispiele auf, welcher Fehler zu dieser Exception führen kann.

Herbert Breunung

Perl 6 - Der Himmel für Programmierer - Update 5

In den letzten anderthalb Jahren (seit 3/2011) wuchs das Perl 6-Projekt weiter von der Breite in die Tiefe. Das bedeutet weniger Implementationen aber mehr Funktionalität denn je. Bevor es jedoch zu den Details geht, lohnt es sich, den Einfluss von Perl 6 auf Perl 5 zu betrachten.

Perl 5 lernt clever abgucken

Die größte Krötenwanderung an Ideen gab es zweifellos mit 5.10 und Moose. 5.16 enthält ebenso wie 5.18 lediglich vereinzelte, leichte Angleichungen, keine neuen vollen Feature aus Perl 6. Ein Beispiel wäre das veränderte Verhalten von `use`, welches den Parser komplett (so weit es geht) auf die gewünschte Sprachversion umstellt und nicht nur eventuell einige Funktionalität zuschaltet. Nur auf die Art wäre es möglich Perl 5 und 6 in einer Datei zu kombinieren.

Derzeit wichtig ist auch die erneut aufgegriffene Debatte um das Smartmatching (~~) [1]. Sie zeigt, wie schwer es tatsächlich ist, Funktionen in eine derart verschiedene Sprache zu portieren. In Perl 6 basiert die Auswahl der passenden Klausel auf Datentypen, welche der Perl 5-Compiler nicht kennt. Auch stellt sich durch die Diskussionen in der *p5p* deutlich heraus, dass etliche Möglichkeiten den Nutzern zu komplex sind, weswegen weitere Vereinfachungen hier in Perl 5 wohl folgen werden.

Die Sprache

Für Außenstehende ist es oftmals eine Provokation, dass allein an der Sprachdefinition so lange gearbeitet wird. Hierzu ist der Post unter [2] aufschlussreich. Die beinahe zeitgleich gegründete Sprache Fortress, mit der Sun einen Fortran-Nachfolger für die Java-VM schaffen wollte, und auf mehreren Gebieten innovative Wege zu gehen versuchte, wurde kurz nach der Übernahme durch Oracle eingestellt. In dem verlinkten Text bedauert der Chefentwickler natürlich, dass dem Projekt nicht mehr Zeit beschieden war, da seine Gruppe erst im Verlauf der zehn Jahre der Entwicklung Wissen ansammeln konnte, mit dem viele anfängliche Entscheidungen anders getroffen worden wären. Dies schrieb Guy Steele, der bereits *Java*, *Scheme* und *Common Lisp* mitentwarf und eine professionelle Gruppe leitete, die in Vollzeit ihre Vorgaben verfolgen durfte. Wenn man zusätzlich bedenkt, dass das Ziel von Perl 6 noch etwas breiter gefasst ist, als das von Fortress, lässt sich erahnen, dass es tatsächlich solcher Zeiträume bedarf, um eine bis in Details ausgefeilte Sprache zu entwerfen. Perl 6 trägt hierfür einen wesentlich höheren Anspruch als Perl 5. Deswegen wurden fast nur noch kaum aufzählenswerte Details verbessert. Die Handhabung von Pfaden und Dateinamen ragt als einzelnes Gebiet hervor, in dem sich mehr änderte als anderswo.

Dokumentation

Trotzdem verbesserte sich die Dokumentation spürbar. Moritz Lenz schuf eine systematische Übersicht zu fast allen Operatoren, Klassen und Methoden. Diese liegt unter [3] und kann bei installiertem *Rakudo* a la *perldoc* mit *perl6doc* gelesen werden. Konrad "GlitchMr" Borowski schrieb dazu auch ein Plugin für die Suchmaschine DuckDuckGo, welche Anfragen wie "perl6 print" direkt zu beantworten weiß, mit einem Link zu mehr Informationen.

Das Werkzeug *grok*, welches dazu dient Perl 6-Docs Perl 5-Nutzern anzuzeigen, bekam davon allerdings noch nichts mit.

Es gibt auch ein neues Wiki [4] und einige schöne Artikel im eigenen Adventskalender [5]. Die hypertextfreundliche Dokumentation Tablets [6] aus meiner Feder wurde lediglich aktualisiert, dient aber weiterhin nur als Übersicht. Das Buch [7] ist immer noch lesenswert, wurde aber nicht weiter geschrieben. Eine leichte Einführung in Sprache und Implementationen bietet das Gespräch mit Moritz Lenz [8].

Implementationen

Rakudo macht kontinuierlich Fortschritte. Dies schließt neue Funktionalitäten, eine mehr als zehnfach Beschleunigung [9] und einige neue Mitstreiter ein. Mittlerweile startet Rakudo schneller als ein Perl 5, welches sich die wesentlichen Möglichkeiten von Perl 6 per Modul dazulädt. Einzelheiten zum aktuellen Feature-Stand können der Übersicht unter [10] entnommen werden. Eigentlich diente sie als Anzeigetafel des Wettrennens zwischen *Rakudo* und *Niecza*, das *Niecza* zuletzt knapp angeführt hat. Leider entschloss sich Frontmann Stefan O'Rear dazu, sich aus der Entwicklung des bisher schnellsten Compilers zurückzuziehen, wodurch *Rakudo* aufholen konnte. Die wichtigste und erfreulichste Nachricht auf dem Gebiet ist aber, dass der JVM-Port von *NQP* schon weit fortgeschritten ist. Deshalb kann das in *NQP* geschriebene *Rakudo* wahrscheinlich noch dieses Jahr auf dieser populären Platform laufen. Eine Portierung auf *.Net* ist ebenfalls geplant. Die Hauptarbeit hierzu leistet der stets emsige Jonathan Worthington [11].

Parrot wird selbstverständlich nicht aufgegeben, gerade jetzt wo eine einfache, aber langersehnte Unterstützung für nebenläufige Programmierung ankam. Es muss ehrlicherweise aber angefügt werden, dass sich *Parrot* in letzter Zeit langsamer bewegt. Wie anfangs beschrieben, konzentrieren sich die Kräfte immer weiter darauf, eine hochwertige und voll ausgestattete Implementation zu bekommen. Die Anzahl der mit *panda* installierbaren Module wuchs seit dem letzten Update auf 162, also fast das Vierfache.

Es scheint sich auch die Stimmung gegenüber Perl 6 mancherorts aufzuhellen, wozu auch der "Perl Reunification Summit" (vom 17. bis 19. August 2012 - direkt vor der YAPC::EU in Frankfurt) einen Beitrag leisten konnte [12]. Dort trafen sich das Gros der Entwickler und einige andere wichtige Perl-Menschen, um in Perl, an der französischen Grenze, sich Aug in Aug auszutauschen. Wer virtuell dabei sein möchte, kann den Planeten 6 lesen [13] und die seit einiger Zeit wieder erhältlichen wöchentlichen Zusammenfassungen unter [14].

Links

[1] http://byte-me.org/perl-5-porters-weekly-august-20-august-26-2012/#6
[2] http://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up
[3] http://doc.perl6.org
[4] http://wiki.perl6.org/
[5] https://perl6advent.wordpress.com/
[6] http://tablets.perl6.org/
[7] http://github.com/perl6/book
[8] http://www.i-programmer.info/professional-programmer/i-programmer/5002-perl-6-and-parrot-in-conversation-with-moritz-lenz.html
[9] http://pmthium.com/2012/09/a-rakudo-performance/
[10] http://www.perl6.org/compilers/features
[11] http://6guts.wordpress.com/
[12] http://blogs.perl.org/users/gabor_szabo/2013/02/perl-reunification-summit-2012.html
[13] http://planetsix.perl.org/
[14] http://glitchmr.github.io/

Renée Bäcker

Mojolicious Tutorial Teil 3 - Du kommst hier nicht rein!

Wir sind mittlerweile beim dritten Schritt zu unserer Anwendung angekommen. Wir wollen eine Anwendung entwickeln, bei der Läufer ihre Strecken und Laufzeiten eintragen können. Die Basisstruktur wurde mit Hilfe des mojo-Skripts erzeugt, in der ersten Ausgabe des Tutorials gab es die allgemeine Einführung in Mojolicious inklusive Routing, Templates und Stashes.

In der vergangenen 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 dieser Ausgabe wird das Formular zum Hinzufügen von Trainingseinheiten 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 dieser Ausgabe Thema sein.

Von VIPs und Zaungästen

Viele Webseiten haben einen Bereich der jedem zugänglich ist und einen Bereich oder Funktionalitäten, die nur angemeldeten Nutzern zur Verfügung stehen. Um das auch in unserer Anwendung umsetzen zu können, brauchen wir eine Authentifizierung von Nutzern. Dazu stehen verschiedene Mechanismen zur Verfügung: Die HTTP-Authentifizierung, Authentifizierung über andere Webanwendungen wie Twitter oder Facebook oder auch eine eigene Nutzerdatenbank.

In unserer Anwendung sollen gewisse Funktionalitäten wie das Hinzufügen von Trainingsläufen nur für angemeldete Benutzer erreichbar sein. Der Einfachheit halber sehen wir eine kleine SQLite-Datenbank als gegeben an, in der die User schon eingetragen sind. Ein Formular für Registrierungen zu erstellen sei der geneigten Leserin überlassen.

Dass bestimmte Teile der Anwendung nur für registrierte Benutzer sichtbar sind, kann man auf verschiedenen Wegen erreichen. Alle diese Wege greifen in das Routing ein.

over

Der erste Weg ist die Verwendung von Bedingungen für Routen. Damit kann man die Erreichbarkeit von bestimmten URLs z.B. davon abhängig machen, dass ein bestimmtes Datum auf dem Kalender angezeigt wird oder bestimmte Seiten nur für Mobilgeräte anzeigen.

Mit `over` werden die Bedingungen bestimmten Routen zugewiesen. Die Bedingungen selbst werden mit `add_condition` hinzugefügt:

```
$r->add_condition(
    business_hours => sub {
        my @time = localtime;
        return 1 if $time[2] >= 8 &&
            $time[2] < 18;
        return 0;
    },
);
```

Damit werden Seiten nur zugänglich gemacht wenn man zwischen 8 und 18 Uhr versucht darauf zuzugreifen. Bis jetzt ist es aber einfach nur eine Bedingung und es wird noch nicht gesagt, für welche Routen diese Bedingung gelten soll.

Dies wird dann über `over` erledigt:

```
$r->get( '/office' )->over(
    business_hours => 1
)->to( 'guest#office' );
```

Versucht man jetzt ab 18 Uhr versucht auf die Office-Seite zu kommen, bekommt man die 404-Seite angezeigt. Die Seite existiert dann einfach nicht.

Übertragen auf die Beschränkung des Zugriffs auf eingeloggte Benutzer sieht das dann folgendermaßen aus:

```
$r->add_condition(
    authenticated => sub {
        # Cookie mit gültiger Session
        # vorhanden
    },
);

$r->route('/user/run')
  ->via([qw/GET POST/])
  ->over( authenticated => 1 )
  ->to('user#run');
```

Man sollte bei der Verwendung von `over` beachten, dass die Route nicht gecached wird, da das Caching der Bedingungen zu aufwändig wäre. Außerdem kann man nicht reagieren wenn die Bedingung nicht zutrifft. Man kann also nicht-eingeloggte Benutzer auf die Login-Seite weiterleiten.

Möchte man mehrere Bedingungen für eine Route festlegen, kann man diese in ein einzelnes `over` packen.

```
$r->route('/customer/ticket/create')
  ->via([qw/GET POST/])
  ->over(
      authenticated  => 1,
      business_hours => 1,
  )
  ->to('user#run');
```

Diese Bedingungen sind *UND*-Verknüpft. Es gibt leider keinen Weg, um diese Bedingungen in `over` mit *ODER* zu verknüpfen - das muss man in der Bedingung in `add_condition` erledigen.

Die Beispiele zeigen aber auch schön den Vorteil von Bedingungen und `over`: Man kann die Bedingungen einmal hinzufügen und dann in beliebigen Kombinationen verwenden. Man kann diese auch gut in ein eigenes Modul auslagern.

under und bridge

Diese beiden Befehle sind das gleiche. Wer `under` benutzt, ruft indirekt auch wieder `bridge` auf. In diesem Artikel wird mit `bridge` gearbeitet, aber die Befehle funktionieren genauso mit `under`.

Mit `bridge` wird quasi eine Zwischenroute erzeugt, die auf jeden Fall matcht und eine zusätzliche Dispatch-Runde erfordert. Man kann das sehr gut nutzen, um mehrere Routen mit gleichem "Beginn" kürzer zu definieren:

```
my $foo = $r->bridge( '/foo' )
            ->to( 'foo#index' );
$foo->route( '/order')->to( '#order' );
$foo->route( '/:issue' )->to( '#issue' );
```

`/foo` direkt kann nicht direkt aufgerufen werden, da für diesen Pfad nur eine `bridge` hinterlegt ist. Auf dieser `bridge` bauen aber die Routen `/foo/order` und `/foo/:issue` auf. Ruft man im Browser `/foo/order` auf, wird zu erst die Methode `index` im Controller `Foo` aufgerufen. Liefert diese einen wahren Wert zurück, wird der zusätzliche Dispatch auf *order* gemacht und die Methode `order` wird ausgeführt.

Liefert `index` aber einen unwahren Wert, bekommt man im Browser nur die 404-Seite zu sehen.

Damit können wir auch nicht-eingeloggte Nutzer von bestimmten Seiten ausschließen. Alles unter `/user` soll zum geschützten Bereich gehören.

```
my $user = $public->bridge('/user')
                  ->to('user#auth');

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

Und die Methode `auth` des Controllers `User` prüft, ob der Benutzer eingeloggt ist. Ist dies nicht der Fall, wird auf die Login-Seite weitergeleitet.

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

  # pruefe ob Cookie vorhanden ist
  return 1 if $self->session('SessionID');

  # leite um auf login seite
  $self->redirect_to( '/login' );
  return;
}
```

Authentifzierung

Jetzt haben wir den Zugriff auf einige Seiten eingeschränkt, aber wirklich sehen kann die Seiten im Moment gar keiner. Es fehlt noch die Authentifizierung der Benutzer. In diesem Artikel werden vier verschiedene Wege gezeigt, wie eine Authentifizierung aussehen könnte.

In der Hauptklasse wird - wie oben beschrieben - eingerichtet, welche Seiten nur einem bestimmten Nutzerkreis zugänglich sein soll:

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

   my $routes = $self->routes;

   my $intern = $r->bridge( '/internal' )
                  ->to( 'auth#check_auth' );
   $intern->route( '/page' )
          ->to( 'guest#internal' );
}
```

Die einfachste Benutzerverwaltung ist das Hinterlegen der Daten in der Konfiguration der Anwendung.

```
user:
  reneeb: passwort
  user2: test
```

Dann sieht die Prüfmethode wie folgt aus:

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

    my $config = $self->config;

    my $user = $self->param( 'user' );
    my $pass = $self->param( 'password' );

    return 1 if
        exists $config->{$user} &&
        $config->{$user} eq $pass;

    return;
}
```

Bei einigen Webseiten sind Bereiche auch über die sogenannten HTTP-BasicAuthentifzierung geschützt. In der `startup`-Methode muss dazu das *BasicAuth*-Plugin geladen werden:

```
$self->plugin( 'basic_auth' );
```

und dann kann die Prüfmethode folgendermaßen aussehen:

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

    return 1 if $self->basic_auth(
        internal => sub {
            return 1 if "@_" eq 'foo test'
        });

    return;
}
```

Für viele Anwendungsfälle ist es gar nicht notwendig, die Benutzerdaten selbst zu verwalten. Man kann es auch so einrichten, dass sich die Benutzer bei einer anderen Anwendung einloggen und man dann nur noch das Ergebnis - also eingeloggt oder nicht - weiterverwendet. Häufig wird das mit Facebook, Twitter oder Github gemacht. Diese drei Dienste können mit dem Plugin `Mojolicious::Plugin::Web::Auth` angebunden werden:

In der `startup`-Methode:

```
$self->plugin(
     'Mojolicious::Plugin::Web::Auth',
  module      => 'GitHub',
  key         => 'your_github_key',
  secret      => 'your_github_secret',
  on_finished => sub {
      my ($c, $token, $info) = @_;
      $c->session('token' => $token);
      $c->session('info'  => $info);

      return $c->redirect_to('index');
  };
);

$r->any( [qw/get post/] )->( '/login' )->to(
  cb => sub {
    my ($c) = @_;
    if (uc $c->req->method eq 'POST') {
        return $c->redirect_to(
           sprintf "/auth/%s/authenticate",
                "Github",
        );
    }
    $self->render( 'login' );
});
```

Hat sich der Benutzer erfolgreich bei Github angemeldet, wird in dem Session-Cookie das Token gespeichert.

Dieser Abschnitt hat gezeigt, dass es sehr vielfältige Möglichkeiten gibt, Benutzern Zugang zu bestimmten Bereichen zu gewähren. Die große Anzahl an bestehenden Plugins erleichtert es, andere Webdienste für die Authentifizierung einzubinden.

Plugins

Bei der Authentifizierung kamen einige Plugins zum Einsatz. In diesem Abschnitt wird darauf eingegangen, wie man selbst Mojolicious-Plugins erstellen kann. Es ist kein Hexenwerk und einfache Plugins sind in wenigen Minuten erstellt. Ich unterteile Plugins ganz gerne in zwei Kategorien. Zum einen sind das Plugins, die ein paar Hilfsmethoden zur Verfügung stellen. Diese unterstützen beim Programmieren. Die zweite Kategorie sind Plugins, die quasi eine Subanwendung bereitstellen. Ein Beispiel hierfür ist `Mojolicious::Plugin::SQLiteViewerLite`. Hier muss man nichts programmieren und mit nur wenig Konfiguration bekommt man einen ganzen Teil einer Anwendung geliefert. Das Plugin erstellt selbst Routen und Controller.

Als Beispiel der ersten Kategorie wird hier ein Plugin erstellt, mit dem ein Tweet über die Trainingseinheit erstellt wird. Dazu brauchen wir einfach nur das Helferlein `tweet`, so dass wir in der Anwendung einfach

```
$self->tweet(
    $message,
);
```

sagen können. Das Plugin besteht nur aus einem einzelnen Modul - siehe Listing 1.

Das Modul erbt von `Mojolicious::Plugin` und verwendet `Net::Twitter::Lite::WithAPI1_1`. Die Methode `register` muss jedes Plugin implementieren und diese wird zur Startzeit nach dem Laden des Moduls ausgeführt.

```
package Mojolicious::Plugin::Tweet;

use strict;
use warnings;

use Mojo::Base qw(Mojolicious::Plugin);
use Net::Twitter::Lite::WithAPIv1_1;

our $VERSION = 0.01;

sub register {
    my ($plugin, $mojo, $param) = @_;

    my $twitter = Net::Twitter::Lite::WithAPIv1_1->new(
        consumer_key        => '...',
        consumer_secret     => '...',
        access_token        => '...',
        access_token_secret => '...',
        legacy_lists_api    => 1,
    );

    $mojo->helper(
        'twitter' => sub {
            $twitter;
        }
    );

    $mojo->helper(
        'tweet' => sub {
            my $self = shift;
            my $msg  = shift;

            eval{
                $twitter->update( $msg );
                $self->stash(
                    notify => 1,
                );
            } or $self->stash(
              error => $@
            );
        }
    );
}

1;
```

Listing 1

In `register` werden dann die Hilfsmethoden über `helper` erstellt und stehen ab sofort der Anwendung zur Verfügung.

Ein aufwändigeres Plugin wird in der nächsten Ausgabe erstellt.

In die Cloud spielen

Es gibt zwar ein Modul, mit dem das deployen auf heroku mit nur einem Kommando funktioniert - `Mojolicious::Command::deploy::heroku` - aber ich möchte hier etwas näher auf heroku eingehen.

Um eine Anwendung bei heroku hosten zu können, benötigt man einen Account. Dieser ist schnell angelegt - man muss nur seine E-Mailadresse hinterlegen. Nach dem Klick auf den Link in der Bestätigungsmail kann man ein Passwort vergeben.

Heroku bietet zum - kostenlosen - Basisaccount noch viele Zusatzpakete an, so dass es mehr sogenannte Dynos, eine Datenbank oder ElasticSearch gibt. Dynos sind die Elemente, die den Code ausführen und die Antwort weiterleiten. Für die Testanwendung hier wird kein Add-On zugekauft, wir versuchen mit den Bordmitteln auszukommen.

Bevor man die eigene App veröffentlicht, braucht man noch das sogenannte *Toolbelt*. Damit bekommt man einen heroku-Client, über den man die eigene Anwendung auf heroku verwalten kann.

```
wget -qO-
 https://toolbelt.heroku.com/install.sh | sh
```

Teil der Anwendung ist auc `Foreman`, mit dem man die Anwendung auch lokal laufen lassen kann.

Diesem Tool muss man noch seine Zugangsdaten bekannt machen und einen Public-SSH-Key hochladen (siehe Listing 2).

Werden mehrere Schlüssel gefunden, kann man den passenden aussuchen. Sollte noch kein Schlüssel existieren, kann man einen neu generieren. Jetzt kann man loslegen - siehe Listing 3.

Der Code wird in ein *git* Repository gepackt, da heroku mit git arbeitet. Nachdem der Code im Repository ist, kann die heroku-App erzeugt werden:

```
track_runs$ heroku create -s cedar \
  --buildpack \
  http://github.com/judofyr/perloku.git
```

`-s cedar` ist der Stack, auf dem die Anwendung bei heroku laufen wird. `cedar` ist dabei ein Ubuntu 10.04. Der `cedar`-Stack hat nativ keine Unterstützung für irgendwelche Spra-

```
reneeb@reneeb-K73BY:~$ heroku login
Enter your Heroku credentials.
Email: heroku@foo.de
Password (typing will be hidden):
Found existing public key: /home/foo/.ssh/id_rsa.pub
Uploading SSH public key /home/foo/.ssh/id_rsa.pub... done
Authentication successful.
```

Listing 2

```
track_runs$ git init
Initialized empty Git repository
    in /home/foo/track_runs/.git/
track_runs$ git add *
track_runs$ git commit -m "initial commit"
[master (root-commit) 16e19e3] initial commit
 9 files changed, 1748 insertions(+), 0 deletions(-)
 create mode 100644 lib/Controller/User.pm
 create mode 100644 lib/TrackRuns.pm
 create mode 100644 lib/TrackRuns/I18N/de.pm
 create mode 100644 lib/TrackRuns/I18N/en.pm
 create mode 100644 log/development.log
 create mode 100644 public/index.html
 create mode 100755 script/track_runs
 create mode 100644 t/basic.t
 create mode 100644 templates/user/run.html.ep
track_runs$
```

Listing 3

chen oder Frameworks. Diese kommt mit den sogenannten Buildpacks, für die Perl-Unterstützung existiert Perloku von Magnus Holm. Soll zu einem späteren Zeitpunkt der Buildpack geändert werden, kann man das konfigurieren:

```
heroku config:add BUILDPACK_URL=\
    git://github.com/.../new.git
```

Aber zurück zur Veröffentlichung unserer Mojolicious-Anwendung. Die Ausgabe des `create`-Befehls ist in Listing 4 zu sehen.

Die Anwendung ist bei heroku registriert und man sieht die URL, unter der sie später erreichbar ist. Dem git Repository wurde ein (neues) Remote-Repository hinzugefügt. Heroku braucht aber noch ein paar Dateien, die bisher im Repository fehlen. Zum einen ist das eine `Makefile.PL` in der die Abhängigkeiten aufgelistet sind:

```
use strict;
use warnings;

use ExtUtils::MakeMaker;

WriteMakefile(
  NAME          => 'TrackRun',
  VERSION       => '1.0',
  AUTHOR        => 'foo',
  EXE_FILES     => ['script/track_runs'],
  PREREQ_PM     => {
      'Mojolicious' => '3.0'
  },
  test => {TESTS => 't/*.t'}
);
```

und zum anderen ein ausführbares Shellskript mit dem Namen *Perloku*. In diesem wird der Server - z.B. der von `Mojolicious` mitgelieferte `hypnotoad`.

Jetzt muss nur noch in dieses Remote-Repository gepusht werden (siehe Listing 5). Dabei kann man auch beobachten, was auf heroku-Seite passiert. Dort wird zuerst das buildpack geholt, Perl eingestellt und die Abhängigkeiten installiert.

Dann rufen wir doch mal die URL auf und siehe da - nichts geht (siehe Abbildung 1)

Über den heroku-Client kommt man an die Logs der Anwendung. Vielleicht gibt das einen Hinweis darauf was schief läuft (siehe Listing 6)

Wir haben also vergessen ein Plugin in den Abhängigkeiten anzugeben. Auch `DateTime` und die anderen verwendeten Module müssen wir eintragen und die Änderungen `push`en. Wenn wir es danach nochmal probieren, sehen wir das Formular... So schnell ist unsere Anwendung in der Cloud.

brian d foy gibt in seinem Blog unter http://bit.ly/127Stzy noch weitere Tipps, wie man Probleme mit Modulinstallationen auf heroku untersuchen kann.

```
Creating mysterious-plains-9634... done, region is us
BUILDPACK_URL=http://github.com/judofyr/perloku.git
http://mysterious-plains-9634.herokuapp.com/ | git@heroku.com:mysterious-plains-9634.git
Git remote heroku added
```

Listing 4

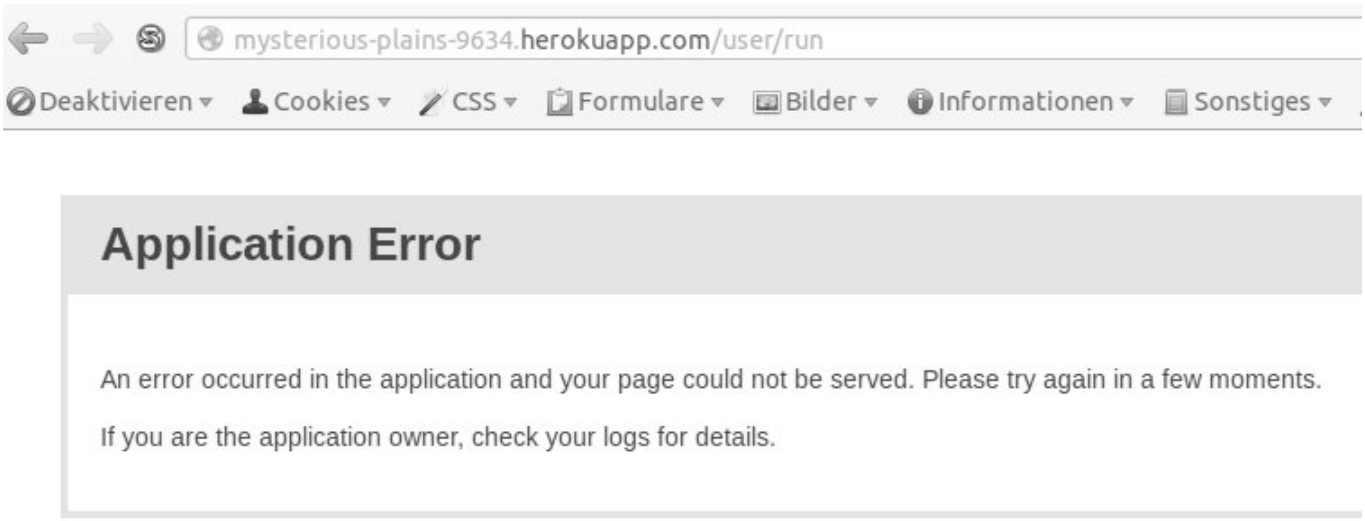


Abbildung 1: Der erste Versuch geht schief.

```
git push heroku master

Counting objects: 27, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (27/27), 9.65 KiB, done.
Total 27 (delta 2), reused 0 (delta 0)

-----> Fetching custom git buildpack... done
-----> Perloku app detected
-----> Vendoring Perl
       Using Perl 5.16.2
-----> Installing dependencies
       --> Working on /tmp/build_3bz8ruqmo5s5z
       Configuring /tmp/build_3bz8ruqmo5s5z ... OK
       ==> Found dependencies: Mojolicious
       --> Working on Mojolicious
       Fetching http://www.cpan.org/authors/id/S/SR/SRI/Mojolicious-3.97.tar.gz ... OK
       Configuring Mojolicious-3.97 ... OK
       Building Mojolicious-3.97 ... OK
       Successfully installed Mojolicious-3.97
       <== Installed dependencies for /tmp/build_3bz8ruqmo5s5z. Finishing.
       1 distribution installed
       Dependencies installed
-----> Discovering process types
       Procfile declares types   -> (none)
       Default types for Perloku -> web

-----> Compiled slug size: 12.7MB
-----> Launching... done, v5
       http://mysterious-plains-9634.herokuapp.com deployed to Heroku

To git@heroku.com:mysterious-plains-9634.git
 * [new branch]      master -> master
```

Listing 5

```
track_runs$ heroku logs -n 4
2013-05-07T22:29:55.450692+00:00 heroku[web.1]: State changed from crashed to starting
2013-05-07T22:29:59.327118+00:00 heroku[web.1]: Starting process with command `./Perloku`
2013-05-07T22:30:01.354096+00:00 app[web.1]:
                                      Plugin "I18N" missing, maybe you need to install it?
2013-05-07T22:30:02.760244+00:00 heroku[web.1]: Process exited with status 2
2013-05-07T22:30:02.775399+00:00 heroku[web.1]: State changed from starting to crashed
```

Listing 6

Noch ein paar Anmerkungen zu heroku: Bei den kostenlosen Accounts speichert heroku die Daten im RAM. Eine Datenbank oder Redis gehören zu den Addons und müssen extra bezahlt werden. Wenn längere Zeit keine Requests für die Anwendung kommen, werden die Prozesse heruntergefahren. Dadurch gehen die Daten im RAM verloren.

Sobald Requests für die Anwendung eintreffen, starten die Prozesse wieder.

MODULE

Mark Overmeer

Exceptions, Logging und Übersetzungen

Perl ist auf viele Arten eine beeindruckende Sprache. Gibt es eine Programmiersprache mit mehr Syntax? Wahrscheinlich nicht: es gibt immer mehrere Wege, etwas zu tun, während andere Programmiersprachen dem Entwickler ein vorgegebenes Konstrukt aufzwingen. Aber obwohl Perl bei einfacher Syntax extrem stark ist, fehlen doch manche höhere Features zur Entwicklung von größeren Anwendungen. So haben wir zum Beispiel keine ausgeprägte Infrastruktur zum Logging oder für Exceptions im Core.

Aber auch wenn es diese Features nicht im Core gibt, haben wir dennoch zahlreiche Alternativen dafür im CPAN. Und genau das ist das Problem. Wenn eine Anwendung verschiedene Bibliotheken verwenden will, haben diese (ziemlich sicher) inkompatible Konzepte für Exceptions und Logging. Das schränkt die operationale Fähigkeit des Codes ein. Dabei ist einer der Gründe, warum es so viele Implementierungen für die selbe Aufgabe im CPAN gibt, dass die verwendeten Frameworks inkompatibel sind.

Dieser Artikel versucht, den Leser davon zu überzeugen, dass die Infrastruktur für Exceptions, für Logging, und für die Übersetzung eng miteinander verwandt ist. Sogar so eng, dass sie am Besten als ein einziges Framework implementiert werden sollten. Das Modul `Log::Report` tut genau das, und dieser Artikel versucht dich davon zu überzeugen, dass es ein praktisches Framework ist.

Was ist das Problem?

Wenn man eine neue Anwendung entwickelt, konzentriert man sich auf den Zweck des Programms und das Ziel, dass es zu erreichen gilt. Zu Beginn des Projektes ist oft noch nicht klar, welche unterstützenden Bibliotheken man benötigt, und wo die Anwendung im Laufe ihres Lebens eingesetzt wird. Man erkennt zu spät, dass das Programm größer skaliert werden muss; dass andere Leute es benutzen müssen (und Ausländer die Fehlermeldungen verstehen müssen); dass es regelmäßig laufen muss (Fehler gehen ins syslog); dass es in eine größere Anwendung integriert werden (und ein anderes Framework verstehen) muss.

Selbst wenn das zunächst kleine Programm ausgeklügelte Frameworks für Exceptions, Logging und für die Übersetzung benutzt, muss man immer noch Glück haben dass alle drei mit der unbekannten Zukunft des Programms zusammenspielen. Wenn nicht bleibt einem nur, die Anwendung neu zu schreiben. Und neu schreiben ist teuer.

Sterben oder nicht sterben?

Was könnte an dieser gewöhnlichen Zeile Code falsch sein?

```
open IN, '<:encoding(utf8)', $fn
    or die "cannot read $fn: $!\n";
```

Hier braucht man sich eigentlich keine Sorgen zu machen, wenn man ein interaktiv laufendes Programm hat. Aber in der Realität ist dieser Code eher unangebracht. Ein paar Beispiele:

1) Die Subroutine die diesen Fehler verursacht wurde von einer anderen Funktion aufgerufen, die vielleicht ein Problem mit dieser Datei erwartet hat. Daher stößt man vielleicht auf Code wie diesen:

```
my $data = eval { open_files @files };
if($@ =~ m/cannot read (.*?): (.*)/)
{   my ($errfn, $errtxt) = ($1, $2);
    ... error recovery ...
}
```

Die Kombination aus `eval` und `die()` ist ein sehr schwacher Mechanismus für Exceptions. Wenn das `open` fehlschlägt ist

sehr viel über die Umstände bekannt die von `die()` einfach ignoriert werden. Zum Beispiel:

- der Name der Quelldatei und Zeile
- der Timestamp des Events
- ein Dateinamen-Objekt (falls eins benutzt wird)
- der Fehlercode
- der nicht übersetzte Fehlercode (mit gettext)
- ein call stack
- die anderen Parameter des fehlgeschlagenen `open()`

Das Hauptprogramm könnte vielleicht noch eine unerwartete Verwendung für diese Daten haben, bekommt sie aber nicht mit.

2) Selbst uralte Systeme wissen, dass Fehler unterschiedliche Bedeutungen haben können. Das UNIX Syslog verlangt, dass die Schwere des Fehlers angegeben wird: emergency, alert, critical, error, usw... Das ist hilfreich, um kleine von großen Problemen zu unterscheiden, und die Events zu den jeweils besten Handlern zu eskalieren.

Perls drei Fehler-Level (print, warn, die) sind da eher schwach, und jeder erfindet seine eigenen Debug-Level.

```
print "saving session\n" if $verbose;
print "logout user $user\n" if DEBUG;
```

Das Modul Carp bietet einige zusätzliche Funktionen: carp, confess, croak und cluck. Sie zeigen den call stack, aber fügen keine zusätzlichen Fehler-Level hinzu. Ausserdem ist nicht immer klar ob carp oder warn die bessere Wahl ist.

Gibt `die()` die Situation hinreichend wieder? Alert könnte die bessere Wahl sein, wenn die Situation reparierbar aussieht. Critical könnte für notwendige Konfigurationsdateien passen, die einen Webserver vom Starten abhalten.

3) Wusstet ihr, dass es es ein Character-Set-Problem mit `die $fn` gibt? Dateinamen sind eine Sequenz von Bytes, die vom Betriebssystem empfangen werden. Auf einigen Betriebssystemen weißt Du vielleicht, welches Character-Set für Dateinamen benutzt wird. Auf Windows ist es UTF-16. Auf UNIX/Linux kann das Character-Set für jedes Dateisystem und sogar für jeden Teil des Pfades im Dateinamen unterschiedlich sein. In jedem Fall codiert oder decodiert niemand den Dateinamen in Perls internes UTF-8!

Wenn Du also einen Dateinamen in einer Fehlermeldung benutzt, wird der mit anderem Text verbunden. Das kann verursachen, dass die Bytes des Dateinamens (eine utf8-Sequenz die vom Betriebssystem empfangen wurde, aber nicht als Perls internes utf8 gekennzeichnet ist) noch einmal nach utf8 codiert wird, als ob sie latin1 waeren. Ups.

4) Falls die Fehlermeldungen übersetzt werden müssen, zum Beispiel weil die Anwendung internationale Benutzer hat, muss man dazu Übersetzungstabellen anlegen. Normalerweise nimmt man dafür die (GNU) *gettext* Bibliothek.

Daraus entsteht Code wie dieser:

```
die sprintf(gettext(
  "cannot read file %s: %s\n"), $fn, $!);
```

Die Übersetzungstabellen brauchen strings die statisch sind. Die Variablen dürfen noch nicht im Text stehen. Das hier ist also falsch:

```
die gettext("cannot read file $fn: $!\n");
 # No!
```

Hässlich und mit `eval/die()` schlecht zu verarbeiten. Leider ist Perl eine der wenigen Open Source-Anwendungen, die Mehrsprachigkeit geradezu feindlich gesinnt ist.

Und dann versuch mal, einige der Fehlermeldungen dazu zu bringen, auf Chinesisch an die Webseite zu gehen, während andere auf Deutsch im syslog landen. Vielleicht muss sogar die gleiche Meldung gleichzeitig an beide Kanäle gehen!

Fassen wir zusammen: Fehlerreporting für kleine, interaktiv laufende Programme ist einfach. Aber für große Programme, internationale Anwendungen und für Daemons brauchen wir mehr als das. Die Module, die sich dazu auf CPAN finden, ignorieren entweder diese Problematik, oder sie verwenden Frameworks, die häufig zu Inkompatibilität führen.

Das Log::Report Konzept

Die Distribution `Log::Report` ist das einzige Framework, das alle Features in einem bietet:

- Übersetzung
- Exceptions
- Logging

Zum Beispiel:

```
# tranditional use of translations
die sprintf(gettext("
  cannot read file %s: %s\n"), $fn, $!);

# in Log::Report
fault __x"cannot read {file}", file => $fn;
```

Ok, betrachten wir diese Zeile. `fault()` gehört zu einer ganzen Reihe solcher Funktionen: debug, trace, info, notice, warning, error, fault, alert und panic. Einige davon sind eher wie `warn()` und andere mehr wie `die()`. info und notice sind am nächsten an `print()`.

Die meisten dieser Funktionsnamen sind, mit kleinen Änderungen, direkt von *syslog(2)* ausgeliehen. Neu ist etwa `fault()`, das einen Fehler abbildet, der vom Betriebssystem kommt, während `error()` intern ausgelöst wurde. Wie man sieht muss `$!` nicht explizit angegeben werden. Es wird von `fault()` automatisch angehängt.

Übersetzung

Die Idee für den Übersetzer ist von `Locale::TextDomain` ausgeliehen worden. Aber statt `%s` wird `{file}` benutzt. Das `%s` wird für `printf()` gebraucht. Hier ist das `sprintf(gettext())` jedoch in eine neue Funktion eingebaut, die kurz `__x()` heißt. Zwei Unterstriche, gefolgt von einem 'x'.

Natürlich ist `__x()` ein komischer Name für eine Funktion. Genauso komisch ist die ähnliche Funktion '__' (zwei Unterstriche), die ebenfalls den Text Übersetzt, aber nicht versucht die Platzhalter zwischen den geschweiften Klammern einzufüllen. Ausserdem gib es noch ein paar andere solche Funktionen.

Das Standard-gettext ist für C designed worden. Es gibt aber keinen Grund, in Perl auf dem selben niedrigen Level zu bleiben. Daher trifft man hier auch auf diese Art von Übersetzung mit gettext:

```
# in the code
sprintf(gettext(
   "in %d minutes, read %s"), $min, $file)

# the Dutch translation table
msgid "in %d minutes, read %s"
msgstr "lees %2$s na %1$d minuten"
```

Je nach Sprache müssen die Parameter in einer anderen Reihenfolge im Text stehen. Die Übersetzer bekommen nicht automatisch eine Erläuterung für die Bedeutung des ersten und zweiten Parameters. Das führt häufig zu mehr Dokumentation im Programm, um den Übersetzern zu helfen, die Fehlermeldungen zu verstehen.

Durch die Syntax von `Log::Report` mit den geschweiften Klammern wird es einfacher, denn es sind viel weniger extra Zeilen mit Dokumentation nur für die Übersetzer notwendig.

```
# in the code
__x"in {minutes} minutes,
     read {filename}",
     minutes => $min, file => $file

# the Dutch translation table
msgid "in {minutes} minutes,
     read {filename}"
msgstr
    "lees {filename} na {minutes} minuten"
```

`Log::Report` geht sogar noch einen Schritt weiter als `Locale::TextDomain`: `__x()` *verzögert* die Übersetzung des Textes bis er benutzt wird. In `Log::Report` erzeugt `__x()` zunächst ein `Log::Report::Message` Objekt, das die zu übersetzende Zeichenkette und die Parameter enthält, die eingefügt werden sollen.

```
my $msg = __x"Hello, {who}!",
          who => 'World';
print $msg;           # "Hello, World!"

# it's a lazy object, so;
print ref $msg;       # Log::Report::Message
print $msg->toString; # "Hello, World!"
print $msg->msgid;    # "Hello, {who}!"
print $msg->valueOf('who');  # "World"
```

Die Zeichenkette die übersetzt werden soll wird *msgid* genannt. Sie ist nur ein Label in der Übersetzungstabelle, um das Gegenstück zu finden. Dennoch wird sie auch als Standard-Übersetzung genutzt, falls es für diese Sprache keine Tabelle gibt, oder die Übersetzung leer ist.

Exceptions

In diesem Beispiel steckt die `fault()` Funktion das Nachrichten-Objekt in ein `Log::Report::Exception`-Objekt. Dieses Objekt kennt den Stack Trace, die Fehlercodes `$!` und `$?` und noch einiges mehr. Beim späteren Ausgeben der Nachricht könnten diese Informationen hilfreich sein.

Die Exception wird in der Programmhierarchie nach oben geworfen (engl. *throw*) und landet beim Caller der Subrouti-

ne. Sie wird von einem (Log-) *Dispatcher* aufgefangen (engl. caught) und behandelt. `fault()` (sowie `error()` und `panic()`) sterben mit der Exception als Rückgabe. Sie sind fatal. Aber andere Funktionen wie etwa `info()` und `warning()` brechen nicht mit `die()` ab, sobald sie eine Exception (weiter nach oben) geworfen haben. Diese Exceptions landen beim Dispatcher, um dort verarbeitet zu werden.

So sieht die Struktur des Ganzen aus:

```
Log::Report::Exception object, contains
   - exception level (reason for message)
   - caller stack
   - $!, $?
   - Log::Report::Message object, contains
      - string to be translated (msgid)
      - parameters to be filled in
      - textdomain
      - $"
```

Logging

Wer weiss wo die Ausgaben und die Fehlermeldungen hin sollen? Wo ist der Überblick über die gesamte Anwendung? Natürlich im package main, und nicht in den Modulen! Im `Log::Report`-Konzept startet daher der Hauptteil der Anwendung die Dispatcher, die dann Exceptions (und damit die Nachrichten) behandeln. Die Module selbst sind ignorant.

Diese Anweisung im package *main* der Anwendung definiert, wie Exceptions in die Logdateien geschrieben werden sollen.

```
# send exceptions to syslog
dispatcher SYSLOG => 'my-log'
# explicit character conversions
, charset => 'iso-8859-1'
# overrule user's locale
, locale  => 'nl_NL'
# take priority INFO and higher
, accept  => 'INFO-';
```

Das Beispiel registriert einen zusätzlichen Dispatcher, um Exceptions zu verarbeiten. Wenn es mehr als einen Dispatcher gibt, bekommen *alle* Dispatcher *alle* Exceptions. Dieser Dispatcher hier ignoriert jedoch Exceptions mit Levels unterhalt von INFO. Er versucht, die Nachrichten ins Niederländische (nl_NL) zu übersetzen, und verwendet dafür den Zeichensatz latin1. Die Schwere der Exceptions wird in syslog-Equivalente übersetzt, die konfigurierbar sind. Das Schwere ERROR wird zum Beispiel zum syslog-Level LOG_ERR umgewandelt.

Wenn das Programm startet, gibt es immer einen Standard-Dispatcher. Der entspricht diesem hier:

```
dispatcher PERL => 'default'
  , accept => 'NOTICE-';
```

Wenn jetzt der Dispatcher SYSLOG von oben hinzugefügt wird, gibt es beide. Der PERL Dispatcher zeigt die Nachrichten in der nativen Sprache und Zeichensatz an, während der SYSLOG Dispatcher die gleichen Nachrichten ins Niederländische übersetzt.

Bei einem Daemon oder einer Webseite will man natürlich selten die Standard-Ausgaben. Nachdem der Daemon vollständig initialisiert wurde schliesst man daher am Besten den PERL Dispatcher:

```
# close by dispatcher name
dispatcher close => 'default';
```

Es ist hilfreich wenn Fehler sowohl auf den Bildschirm als auch ans syslog gehen, während der Daemon initialisiert wird. Danach aber nicht mehr. Das Framework kann deshalb einfach zwischen verschiedenen Anwendungsfällen wie stand-alone Daemon und Kommandozeilenapplikation hin- und herschalten.

• Exceptions abfangen

Exception-Frameworks bieten Wege, um Exceptions abzufangen, die von einem bestimmten Codeblock geworfen werden. So wie `eval()` und `die()` das in normalem Perl Code machen, gibt es in Exception-Frameworks `try()`, welches intern ebenfalls `eval()` verwendet.

Es gibt viele Module auf CPAN, die `try()` implementieren, wie zum Beispiel `Try::Tiny`. Auf die möchte ich hier aber nicht näher eingehen. `Log::Report` hat seine eigene Version von `try()`, die eng an `eval()` angelehnt ist:

```
try { error __x"Help!" };
if($@) { ... }  # there is an error
```

Achtung: Der Semikolon nach dem try-Block ist wichtig! Der Block ist ein Argument für die Funktion `try()`. Das Beispiel ist ein bisschen kompliziert. `__x()` produziert ein Nachrichten-Objekt, das von `error()` in eine Exception-Objekt eingepackt wird. Die Exception wird geworfen, und error() stirbt. Die geworfene Exception wird von try() abgefangen.

Tatsächlich ist `try()` als ein normaler Dispatcher implementiert: Alle vorhandenen Dispatcher werden deaktiviert, während der Block ausgeführt wird. `try()` bekommt alle auftretenden Exceptions.

Aber es geht noch verwirrender: `$@` wird auf das Dispatcher-Objekt gesetzt, das die `try()`-Mechanik implementiert. Und das gibt in boolschem Kontext false zurück! (Mehr Informationen dazu in `Log::Report::Dispatcher::Try`). Hier sind ein paar Beispiele, was man damit alles machen kann:

```
try { error };
if(my $e = $@->wasFatal)
{   # $e is a Log::Report::Exception object
    # recast exception, now not fatal
    $e->throw(reason => 'NOTICE');

    # select specific dispatcher
    $e->throw(to => 'syslog');
}
else # no fatal exception, when !$@
{   # but still may have caught
    # non-fatal exceptions
    $@->reportAll;

    # print warnings, info etc
    print $@->exceptions;
}
```

Exception-Klassen

`try()` fängt vielleicht verschiedene Fehler ab, die dann vielleicht auf verschiedene Arten behandelt werden müssen. Es gibt verschiedene Möglichkeiten, sie zu unterscheiden:

```
if($e->reason ne 'DEBUG') ...
if($e->message->msgid =~ /open/) ...
```

Andere Exception-Module (auch in anderen Sprachen) nutzen die Klassenhierarchie für Exceptions. In diesen Frameworks kann man Exception-Objekte wie 'Exception::Grammer::NoMatch' erzeugen. Dann kann die abgefangene Exception so gefiltert werden:

```
# not Log::Report
if($e->isa('Exception::Grammar')) ...
```

Der Nachteil dieser Strategie ist, dass viele Klassen erzeugt werden müssen. Häufig haben diese auch noch lange Namen. In `Log::Report` hingegen geht das so:

```
try { error __x"Auch!",
  _class => 'test,pain' };
if(my $e = $@->wasFatal)
{   if($e->inClass('pain')) ...
    ...
```

Log::Report laden

Jedes Modul muss `Log::Report` einbinden. Falls Übersetzungen benutzt werden sollen, muss eine *textdomain* angegeben werden. Die ist der Name der Übersetzungstabelle. Falls es keine Übersetzungstabelle gibt, kann man `__x()` und Co aber trotzdem benuzen. Die Übersetzungen können auch später hinzugefügt werden.

```
use Log::Report 'my-project';
```

Man kann sowohl für jeden Dispatcher als auch global einen run *mode* angeben. Wenn man den von NORMAL auf DEBUG ändert bekommt man viel mehr Informationen für jeden Fehler. Zum Beispiel erhält man den call stack, so wie bei `carp()` und `confess()`. Sobald die Anwendung fertig ist, kann man den Modus wieder zurück auf NORMAL stellen.

```
use Log::Report 'my-project',
  mode => 'DEBUG';
```

Oder für einen Dispatcher:

```
dispatcher mode => VERBOSE => 'ALL';
```

Nimm dir was Du brauchst:

Vielleicht werden gar nicht alle drei Komponenten des `Log::Report` Konzepts gebraucht:

```
# only interpolation
# (without translation tables found)
print __x"opening files: {files}\n",
   files => \@files;

# translation and interpolation
# (tables exist)
print __x"\t\tPlease pay: {price}\n",
   price => 3.14;

# exception without translation
error "cannot read file $file";

# logging without exceptions
# or translations
my $syslog =
   dispatcher SYSLOG => 'syslog', ...;
$syslog->log({}, ERROR => 'Hello, World!');

# logging without exceptions
$syslog->log({locale => 'de'},
   ERROR => __x"Hello, World!");
```

Tipps und Tricks

Hier sind noch ein paar Ideen, um einen vollständigeren Eindruck zu vermitteln:

• Übersetzbare Objekte verketten

Die Nachrichten-Objekte sind schlau. Sie verzögern ihre Serialisierung. Sogar im nachfolgenden Beispiel ist $msg immer noch ein Objekt, ohne dass es übersetzt wird.

```
my $msg =
  __x("Hello") . ", " . __x("World!");
```

In der Übersetzungstabelle sind "Hello" und "World!" einzeln aufgelistet.

• Behandlung von Whitespace

Auch `__x()` ist schlau, vor allem wenn es um Whitespace geht. Die folgenden Zeilen sind äquivalent.

```
print __x"\t\tPlease Login:\n";
print "\t\t" . __x("Please Login:") . "\n";
```

Hier wird `print()` das Nachrichten-Objekt serialisieren. In beiden Fällen wird die Übersetzungstabelle "Please Login:" anzeigen.

• Formattierung in den Übersetzungen

Als eine Erweiterung der `Locale::TextDomain`-Features kann man ein bisschen Formattierung in den Sprachvariablen unterbringen:

```
print __x"to pay: {price%.2f} USD",
  price => $p;
```

Die Übersetzungstabelle kann dabei sogar für bestimmte Sprachen Währungen runden.

```
"te betalen: {price%d} EUR"
```

• Arrays darstellen

Es ist auch möglich, Arrays yu interpolieren. Wenn die Übersetzung auf einem der Werte basiert, kann man `__xn()` verwenden:

```
print __xn"opening file {file}"
    , "opening {nr} files: {files}"
    , scalar @files
    , file  => $files[0]
    , files => \@files, nr => scalar @files;
```

Der dritte Parameter entscheidet, ob der erste oder zweite übersetzte String benutzt wird. In diesem Beispiel ist 'files' ein ARRAY, das mit $" zwischen den Elementen gejoined wird. Man kann aber auch einen _join-Paramter übergeben.

Das gleiche, aber einfacher:

```
print __xn"opening file {files}"
        , "opening {_count} files: {files}"
        # <-- scalar context, becomes _count
        , @files
        # join of 1 == 1
        , files => \@files;
```

• Template::Toolkit

In Log::Report::Extract::Templates/DETAILS wird beschrieben, wie man Log::Report benutzen kann, um für TT2 zu übersetzen.

```
[% loc("hi {n}", n => name) %]
[% loc("msgid|plural", count,
   key => value, ...) %]
[% INCLUDE
     title = loc("search results")
 %]
```

Man braucht nur vier Zeilen Code zum Programm hinzuzufügen, um es zum Laufen zu kriegen.

• Ups!

Das hier klappt nicht:

```
print __x'hello';   # crash
```

Weißt Du, warum nicht? Die einfachen Anführungszeichen sind der alte Namespache-Trenner, der später zu '::' umbenannt wurde.

• die()/warn()/carp() kapseln

Wenn man Code mit try() aufruft, der nicht die Exceptions von Log::Report benutzt,dann werden die Nachrichten automatisch in Exception-Objekte umgewandelt.

```
try { confess "help!" }
if(my $e = $@->wasFatal) { ... }
```

Hier enthält das `Log::Report::Exception`-Objekt den Text "help!" als Nachricht (aber nicht als Objekt), den decodierten Stack-Trace und `$!`.

Fazit

`Log::Report` bietet noch viel mehr als hier gezeigt wird. Es kann dabei helfen, Übersetzungs-Tabellen aufzubauen und Lexika zu verwalten. Mehr als hier das Modul zu bewerben hoffe ich aber vor allem, dass ich den Leser davon überzeugt

habe, dass die Integration von Übersetzungen, Logging und Exceptions in ein Framework Sinn macht.

Dieses Modul wird in ein paar großen Projekten genutzt. Eines davon ist `XML::Compile`. Gute Programme enthalten einen ganzen Haufen Texte, die irgendwann mal den User erreichen müssen. Dabei ist ein abstraktes Framework mit einfacher Syntax wirklich hilfreich.

Renée Bäcker

Ein Geschenk für die Liebsten

Was schenke ich meiner Frau zu Weihnachten? Ein schönes Bild, zusammengestellt aus lauter Bildern von Urlauben und anderen Gelegenheiten. So ein Mosaik sieht nett aus, macht aber viel Arbeit. Eigentlich! Als Perl-Programmierer bin ich faul und versuche möglichst viel zu automatisieren. Außerdem können die Skripte auch für andere Zwecke - z.B. für Werbung auf Veranstaltungen - genutzt werden.

In diesem Artikel soll ein Plakat erstellt werden, auf dem der Schriftzug "PERL" mit den Gravataren der CPAN-Autoren gefüllt werden soll.

Als erstes müssen die Gravatare der CPAN-Autoren heruntergeladen werden. Die Informationen dazu findet man bei MetaCPAN und das Abholen der Bilder wird mit Mojo erledigt. Für die Abfrage der Gravatar-URLs wird das Modul `MetaCPAN::API` verwendet, da das Modul die URL zur Abfrage der API zusammenbaut und das zurückgelieferte JSON automatisch parst. Damit wird Programmierern schon viel Arbeit erspart.

Zuerst wird ein Objekt von `MetaCPAN::API` benötigt.

```
use MetaCPAN::API

my $mcpan = MetaCPAN::API->new;
```

Als nächstes wird MetaCPAN nach den Gravatar-URLs gefragt. Das ist der Teil, der ein wenig Kenntnis über den Aufbau des MetaCPAN-Index benötigt. Alternativ kann man sich im MetaCPAN-Explorer unter *http://explorer.metacpan.org/* die Beispiele anschauen und dann ein wenig rumprobieren bis man die gewünschten Informationen gefunden hat.

Die Gravatar-URL ist in der Autoreninformation im Feld *gravatar_url* gespeichert. Das ist das Feld, das wir benötigen. Damit sieht die Suche wie folgt aus:

```
my $hits = $mcpan->author(
    search => {
        fields => 'author.gravatar_url',
        size   => 5_000,
    },
);
```

Die maximale Anzahl an Treffern ist bei der MetaCPAN-API auf 5000 Einträge begrenzt. Wer mehr Treffer erwartet oder benötigt, sollte die *scrolling*-API nutzen. MetaCPAN benutzt einen ElasticSearch-Index und die Dokumentation von *ElasticSearch.pm* liefert Beispiele von *scrolling*. Für das Beispiel hier reichen die 5000 Einträge aber vollkommen aus.

`MetaCPAN::API` liefert für verschiedene Informationsentitäten Methoden. Für die Autoren gibt es die oben gezeigte Methode `author`, die entweder zum Suchen oder zum Extrahieren von Informationen über einen bestimmten Autor benutzt werden kann:

```
perl -MMetaCPAN::API -MData::Dumper \
  -E 'my $mcpan = MetaCPAN::API->new;
      my $autor = $mcpan->author("RENEEB");
      print Dumper $autor'
$VAR1 = {
        'country' => 'DE',
        'website' => [
                    'http://perl-services.de',
                    'http://perlybook.org',
                    'http://renee-baecker.de',
                    'http://perl-magazin.de'
                      ],
        'asciiname' => 'Renee Baecker',
        'name' => 'Renee Baecker',
        'dir' => 'id/R/RE/RENEEB',
        'pauseid' => 'RENEEB'
      };
```

Ähnliche Methoden gibt es auch für Module, Sourcecode, Releases und POD.

Nach der Suche ist es recht einfach, die Gravatare herunterzuladen und abzuspeichern. Ich benutze dazu `Mojo::UserAgent` im Zusammenspiel mit `Mojo::IOLoop`.

```
my $ua    = Mojo::UserAgent->new;
my $delay = Mojo::IOLoop->delay;

my @hits    = @{ $hits->{hits}->{hits} };
my $counter = 1;

my $step = 200;

while (
    ( ( $counter - 1 ) * $step )
     <= scalar @hits
) {
    my $start   = ($counter - 1) * $step;
    my $stop    = ($counter * $step) -1;

    $stop = $#hits if $stop > $#hits;

    my @subhits = @hits[$start .. $stop];
    $counter++;

    for my $hit ( @subhits ) {
        my $id  = $hit->{_id};
        my $end = $delay->begin(0);

        my $url = $hit->{fields}
                      ->{gravatar_url};
        say "Grab img for $id: $url ...";

        next unless $url;

        $ua->get( $url => sub {
            my ($ua, $tx) = @_;

            my $type = $tx->res
                          ->headers
                          ->content_type;
            my $suffix =
                $type =~ m{jpe?g}i ?
                  'jpg' :
                  'png';

            $tx->res
               ->content
               ->asset
               ->move_to(
                File::Spec->catfile(
                    $target_dir,
                    $id . '.' . $suffix
                ),
            );

            $end->( $id );
        });
    }

    Mojo::IOLoop->start
        unless Mojo::IOLoop->is_running;
}
```

Durch `Mojo::IOLoop` werden die Abfragen asynchron abgearbeitet und mittels `Mojo::IOLoop::Delay` schließlich wieder synchronisiert. Mit `$delay->begin` wird der Zähler der aktiven Eventhandler inkrementiert. Die Methode liefert eine Methodenreferenz, über die der Zähler wieder dekrementiert werden kann.

Dem `get`-Aufruf wird ein Callback übergeben, der aufgerufen wird, sobald die Antwort vom angefragten Server vorliegt. Innerhalb des Callbacks wird dann der Gravatar gespeichert. Dazu werden die Methoden von Mojolicious genommen:

```
$tx->res
   ->content
   ->asset
   ->move_to(
     File::Spec->catfile(
       $target_dir,
       $id . '.jpg'
     ),
   );
```

Die Aufteilung in mehrere Schritte (über die while-Schleife und Arrayslices) ist hier notwendig, weil sonst 5000 Dateihandles geöffnet werden und das führt zu einem Fehler:.

```
   Event "error" failed: Can't open file
"gravatare/KOHA.jpg": Zu viele offene Dateien
at ...
   Event "error" failed: Can't open file
"gravatare/RATCLIFFE.jpg": Zu viele offene
 Dateien at ...
```

Jetzt liegen die Bilder auf der Festplatte und der nächste Schritt kann erfolgen. Für das Mosaikbildchen brauchen wir eine Maske, in der die Gravatare am Schluss erscheinen sollen, einen weißen Hintergrund und ein Bild mit den ganzen Gravataren.

Für die Bildbearbeitung gibt es auf CPAN einige Module; zu den bekanntesten und größten zählen `GD` und `Image::Magick`. Diese beiden Module kommen auch hier zum Einsatz.

Für die Erzeugung der Maske wird `GD` verwendet. Hier wird in zwei Schritten vorgegangen:

1. Es wird ein sehr großes Bild erzeugt, in das der Schriftzug eingefügt wird. Es ist jedoch sehr viel freie Fläche rund um den Text. Beim Hinzufügen des Textes kann man die benötigte Breite und Höhe bestimmen. Auf Basis dieser Werte wird ein ...

2. ... zweites Bild erzeugt, das nur einen kleinen Rahmen um den Schriftzug lässt.

Das erste Bild wird mit der Größe 1000 x 1000 erstellt. Als nächstes muss man noch Farben initialisieren; in diesem Fall reicht es aus nur "weiß" zu initialisieren. Danach wird der Schriftzug eingefügt. Da das Bild wieder verworfen wird,

wird für den Text keine neue Farbe herangezogen, sondern in weiß gehalten. Da eine TrueType Schriftart verwendet wird, wird die Methode `stringTTF` aufgerufen. Als Rückgabewert bekommt man die Koordinaten des Schriftzugs. Auf Basis der Koordinaten lassen sich Breite und Höhe bestimmen:

```
my $image = GD::Image->new( 1000, 1000 );

# initialize some colors
my $w = $image->colorAllocate(255,255,255);

my @bounds = $image->stringTTF(
    $w, $params{font}, $params{size},
    0.0, 10, 990, $params{text},
);

my $width  = $bounds[4];
my $height = (1000 - $bounds[5]) + 20;

return ($width, $height);
```

Der nächste Schritt ist dann die Erzeugung der Maske. Im fertigen Bild soll es einen weißen Hintergrund geben und einen Schriftzug gefüllt mit den kleinen Gravataren. Also muss der Schriftzug transparent sein. Auch für dieses zweite Bild wird das Modul `GD` genommen. Jetzt kennen wir die endgültige Größe:

```
# mask for mosaic
my $image = GD::Image->new(
   $params{width} + 20,
   $params{height} + 20,
);
```

Für den Hintergrund muss die Farbe weiß initialisiert werden und für den Schriftzug brauchen wir eine zweite Farbe, die dann auf Transparent gesetzt wird.

```
# initialize some colors
my $w = $image->colorAllocate(255,255,255);
my $black = $image->colorAllocate(0,0,0);

# set black as transparent
$image->transparent($black);
```

Anschließend wird der Schriftzug eingefügt und zwar mit der Farbe, die Transparent ist

```
$image->stringTTF(
    $black, $params{font}, $params{size},
    0.0, 10, $params{height}-10,
    $params{text},
);
```

Da wir das Bild noch in einem weiteren Schritt zum Mosaik benötigen, wird das Bild im PNG-Format gespeichert.

```
my $file = 'overlay.png';
   open my $png, ">", $file or die
       "can't write to output file $file";
   print $png $image->png;
   close $png;

   return $file;
```

Jetzt stehen die Gravatare zur Verfügung und die Maske wurde erzeugt. Bei der Erzeugung des Mosaiks ist es so, dass zwei Bilder übereinandergelegt werden. Auf dem "unteren" sind die ganzen Bilder (Gravatare) zu sehen und das obere ist die Maske. Der weiße Hintergrund überdeckt die Gravatare des unteren Bilds und nur an den transparenten Stellen des oberen Bildes schauen die Gravatare durch.

Der dritte Schritt auf dem Weg zum Mosaik ist als die Erstellung des "unteren" Bildes. Dazu braucht man erstmal die Pfade zu den ganzen Gravataren:

```
my $dir   = './tiles';
my @files = File::Find::Rule->file
                ->name(qr/.*\.(jpg|png)/i)
                ->maxdepth(1)->in( $dir );
@files    = grep{ -s $_ }@files;
```

Auch dieses Bild soll in der gleichen Größe wie die Maske entstehen. Zusätzlich wird dieses Bild als TrueColor-Bild gespeichert.

```
my $image = GD::Image->new(
    $width,
    $height,
    1,        # use TrueColor
);
```

Um möglichst viele Bilder unterzubekommen, werden die eingebundenen Bilder nur max. 25 Pixel groß sein. Die Gravatare müssen also noch entsprechend skaliert werden, da sonst nur ein 25 Pixel breiter Ausschnitt zu sehen ist. Auf Basis dieser 25 Pixel wird berechnet wie viele Bilder in eine Reihe passen.

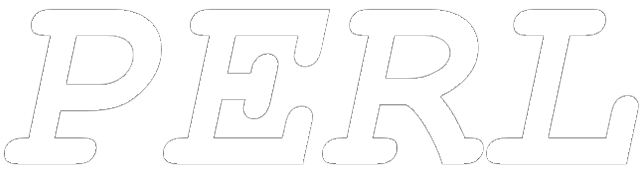


Abbildung 1: Die Maske mit transparentem Schriftzug

```
my $x = 0;
my $y = 0;

my $row_total = $height / 25;
```

Danach werden die Bilder nach und nach hinzugefügt.

```
while( $y <=  $row_total ){
    $x = 0;
    while( $x <= $width ){
        my $currimage = pop @files;
        my $sub = $currimage =~ m{\.png$}
           ? 'newFromPng'
           : 'newFromJpeg';

        my $tileimg =
            GD::Image->$sub( $currimage);
        $image->copy(
            $tileimg,$x,($y*25),
            0,0,
            25,25,
        );

        $x += 25;
    }

    $y++;
}
```

Abschließend wird das Bild gespeichert.

Für den letzten Schritt, das Zusammenführen von Hintergrundbild und Maske, wird `Image::Magick` benutzt. Zuerst werden aus den beiden Bildern jeweils `Image::Magick`-Objekte erzeugt

```
# load mask
my $dest = Image::Magick->new;
$dest->Read( $params{overlay} );
 # load background image (gravatars)
my $magick = Image::Magick->new;
$magick->Read( $tiles_file );
```

Um das Originalbild mit den Gravataren zu behalten, wird ein Klon erzeugt:

```
my $clone = $magick->Clone;
```

Über diesen Klon wird die Maske gelegt

```
$clone->Composite(
    image   => $dest,
    compose => 'over',
);
```

Das Mosaik muss jetzt nur noch gespeichert werden

```
my $mosaic = 'mosaic.png';
$clone->Write( $mosaic );
```

Damit ist das Mosaik fertig (Abb. 3).

Abbildung 2: Das Hintergrundbild mit den Gravataren

Abbildung 3: Das fertige Mosaik

Ullrich Horlacher <framstag@rus.uni-stuttgart.de>

fexsrv: ein schlanker spezieller Webserver in Perl

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.

Auf (UNIX-)Serverseite läuft der F*EX Server, der vollständig in Perl implementiert ist. Er besteht aus:

- dem zentralen `fexsrv`
- dem Modul `dop` (Document OutPut)
- der Konfigurationsdatei `fex.ph`
- diversen Programmen für den Dateitransfer

Das Installationsprogramm legt alles unter `/home/fex/` ab und testet vorab ob *xinetd* (das den *fexsrv* startet) und *sendmail* bzw. ein kompatibler MTA vorhanden sind.

(HTTP-)Webserver gibt es nun wahrlich genug, warum also *noch* einer? Erstmal gilt für uns Perlianer: TIMTOWTDI :-) Aber es gibt auch noch andere Gründe: Zur Entstehungszeit von F*EX gab es keinen einzigen Webserver ohne 32-bit-Bug: es war nicht möglich HTTP POST mit mehr als 4 GB zu verarbeiten. Zudem waren alle getesteten Webserver erschreckend ineffizient und damit langsam bei der Verarbeitung von *großen* Dateien. Auf derselben Hardware schafft *fexsrv* zehn mal mehr Durchsatz als ein Apache.

Aber Dateitransfer interessiert hier nicht weiter, da dies bereits im Vorgängerartikel ausführlich behandelt worden ist. Was kann also *fexsrv* noch anders/besser als z.B. Apache?

- HTML mit eingebettetem Perl-Code
- HTML mit URL-Parametern
- HTML mit bedingten if..then..elseif..end-Blöcken
- on-the-fly zip, tar und tgz Streaming Output
- Verzeichnisauflistung

HTML mit eingebettetem Perl-Code

Um Perl- und HTML-Code beliebig zu mischen ohne dafür extra ein CGI zu installieren, bietet *fexsrv* die Möglichkeit von Inline-Perl in HTML-Dateien. Dazu muss der betreffende Perl-Code in <<Begrenzer>> stehen. Ein Beispiel: Auf http://fex.rus.uni-stuttgart.de/index.html steht:

```
we live in the year 44 after
  invention of the internet
```

Dieser Text wird aber nicht statisch ausgegeben sondern erzeugt mittels:

```
we live in the year
<<my @x=gmtime(time); print $x[5]-69>>
after invention of the internet
```

Der <<Perl-Code>> darf sich auch über mehrere Zeilen erstrecken.

Auch die Ausführung von Shell-Code ist damit möglich:

```
<<`sh code`>>
```

Beispiel:

```
Umgebungsvariablen:<p><pre><<`env`>></pre>
```

Noch einfacher ist die Ausgabe von einzelnen Umgebungsvariablen. Hierzu reicht die Angabe von `$VARIABLE$`, Beispiel:

```
Für diese Seite ist $PATH=$PATH$ gesetzt.
```

HTML mit URL-Parametern

fexsrv setzt eine Reihe von Umgebungsvariablen, darunter auch `QUERY_STRING`. Damit kann auf mit `?` angehängte URL-Parameter zugegriffen werden. Beispiel:

http://fex.rus.uni-stuttgart.de/dynamic.html?xxxx

Enthält den HTML-Code

```
URL parameter=$QUERY_STRING$
```

und es erscheint dann im Webbrowser:

```
URL parameter=xxxx
```

Alternativ wäre folgendes möglich:

```
<<print "URL parameter=
  $ENV{QUERY_STRING}\n">>
```

Eine Besonderheit ist der URL-Parameter "!": damit kann man sich den Sourcecode der betreffenden HTML-Seite anzeigen lassen, also z.B.:

```
http://fex.rus.uni-stuttgart.de/dynamic.html!
```

HTML mit bedingten if..then..elseif..end-Blöcken

Für eine bedingte HTML-Ausgabe stehen die Kontrollanweisungen `#if #elseif #else #endif` zur Verfügung, die am Zeilenanfang stehen müssen und direkt auf Perl-Variablen zugreifen können. Beispiel:

```
<<our $y = 1>>
#if $y > 2
  <h1>y=$y$ Highscore!</h1>
#elseif 0
  <h1>NULL</h1>
#elseif 4 > 3
  <h1>################</h1>
#else
  <h1>zzz</h1>
#endif
```

on-the-fly zip, tar und tgz Streaming Output

Neben statischen Dokumenten und dynamischen HTML-Seiten kann *fexsrv* auch tar-, tgz- oder zip-Archive *on-the-fly* ausgeben. Dazu muss nur eine Steuerungsdatei mit Endung .stream erstellt werden, in der die Dateien aufgelistet sind. Bei entsprechender Anfrage wird dann ein passendes Archiv generiert und ausgegeben. Beispiel:

http://fex.rus.uni-stuttgart.de/foo.stream enthält:

```
foo_1.jpg
foo_2.jpg
foo_3.jpg
foo_4.jpg
foo_5.jpg
foo_6.jpg
```

Das ist eine ganz normale Textdatei. Wenn nun aber auf

- http://fex.rus.uni-stuttgart.de/foo.tar
- http://fex.rus.uni-stuttgart.de/foo.tgz
- http://fex.rus.uni-stuttgart.de/foo.zip

zugegriffen wird, so generiert *fexsrv* in Echtzeit das entsprechende Archiv und gibt dies aus.

Verzeichnisauflistung

fexsrv gibt Dokumente (nur) unterhalb von `/home/fex/htdocs/` aus. Wird eine URL verwendet, die mit `/` endet, so wird zuerst versucht ein darin enthaltenes `index.html` auszugeben. Existiert das nicht, wird die Datei `.htindex` ausgewertet. Dort drin darf ein in Perl gültiger regulärer Ausdruck stehen, der angibt, welche Dateien in einer Verzeichnisauflistung angezeigt werden. Ein leeres .htindex steht dabei für "liste alles". Unterverzeichnisse und das übergeordnete Verzeichnis werden nur angezeigt, wenn sie wiederum selbst ein `.htindex` enthalten. Beispiel siehe Listing 1.

Implementation

fexsrv ist komplett in Perl geschrieben und nur 30 kB groß - inklusive Ausgabemodul *dop* und Kommentare! Es werden

```
http://fex.rus.uni-stuttgart.de/sw/share/fstools-0.0/doc/

/sw/share/fstools-0.0/doc/
/sw/share/fstools-0.0/

2011-08-10 16:31:58       3,676 Bundesbehörde_für_Lebensverwaltung
2011-04-27 09:16:58       4,054 Computer-Problem-Melde-Formular
2001-01-02 10:00:39       9,236 Hohlkopf-Antwort-Formular.txt
2006-05-07 19:34:35       4,900 Merkbefreiung
2002-10-08 08:32:06         301 TOFU
2000-12-11 16:27:19         614 Witzigkeit
2002-11-22 21:36:32       1,191 goldenes_posting
```

Listing 1

nur Module verwendet, die zum Perl CORE gehören.*fexsrv* ist leicht verständlich und modifizierbar.

Der on-the-fly-Archivcode umfasst gerade einmal 1400 Bytes. Dabei wird allerdings etwas "gemogelt", indem auf externe Programme zugegriffen wird: *tar*, *gzip* und *zip* müssen installiert sein, was aber sowieso für fast alle UNIX-Systeme zutrifft. Die CPAN-Module für die Verarbeitung von *tar*- und *zip*-Dateien erwiesen sich als unbrauchbar, da diese die zu generierende Datei zuerst komplett im RAM aufbauen. Das funktioniert deshalb nur für kleine Dateien und ein streaming ist auch nicht möglich.

Der zentrale Code-Teil fürs Streaming sieht so aus:

```
while (<$streamfile>) {
  chomp;
  push @files,$_;
}

if ($request =~ /\.tar$/) {
  open $file,'-|',qw(tar cf -),
     @files or http_error(503);
} elsif ($file =~ /\.tgz$/) {
  open $file,'-|',qw(tar czf -),
     @files or http_error(503);
} elsif ($file =~ /\.zip$/) {
  open $file,'-|',qw(zip -q -),
     @files or http_error(503);
} else {
  http_error(400);
}
```

Danach wird das Filehandle `$file` an die Routine zur Dokumentenausgabe übergeben, die auch normale Dateien ausgibt.

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* nach Einlesen des HTTP-Headers die Kontrolle vollständig an das CGI und läuft nicht mehr als vermittelnder Prozess mit. Das CGI muss also selber "HTTP antworten".

Diese andere Architektur wurde gewählt, weil damit ein deutlich größerer Datendurchsatz (mehr als Faktor 10!) zu erreichen war. Der Nachteil ist, dass *fexsrv* nicht CGI-kompatibel gemäß der original Spezifikation ist und deshalb fremde CGI-Skripte nicht ohne Modifikationen laufen. Eigene *fexsrv*-CGI-Skripte können aber leicht selber geschrieben werden. Dem Installationspaket liegen diverse Beispiele bei. In vielen Fällen ist aber gar kein CGI notwendig, da *fexsrv/dop* von sich aus durch die Inline-Perl-Funktionalität viele Aufgaben erfüllen kann.

Zur Sicherheit: *fexsrv* läuft unter einer eigenen UID (*fex*) und hat keine *root*-Rechte und ist deshalb auch nicht mehrbenutzerfähig.

Auf einem Linux-x86_64-System benötigt *fexsrv* pro Prozess 50 MB RAM, egal ob eine HTML-Seite oder ein anderes Dokument ausgegeben wird, oder ob ein 50 GB-Archiv hochgeladen wird. Ein *fexsrv*-Prozess terminiert, wenn der Client ein HTTP CLOSE schickt. Innerhalb einer HTTP-Session können mehrere Dokumente angefordert werden, was auch die allermeisten Webbrowser machen.

fexsrv ist kein eigenständiger Daemon, der ständig läuft und auf seinem Port lauscht ob Anfragen kommen. Stattdessen wird er bei Bedarf vom *xinetd* gestartet, wo er auch registriert ist. Wenn HTTPS gewünscht wird, kann dies einfach durch Vorschalten von *stunnel* realisiert werden. Es muss nur ein passendes SSL-Zertifikat abgelegt werden, so wie bei anderen Webservern auch.

Download: http://fex.rus.uni-stuttgart.de/fex.html

Renée Bäcker

Was ist neu in Perl 5.18?

Noch in diesem Monat soll eine neue Version von Perl5 veröffentlicht werden: Perl 5.18. Auch mit dieser Version wird es einige Neuerungen geben, von denen die wichtigsten in diesem Artikel vorgestellt werden.

Wie schon in den Vorgänger-Versionen sind es eher kleinere Änderungen und nicht der ganz große Wurf an neuen Features. Schon in der vergangenen Ausgabe von *$foo* wurde der neue Hashing-Algorithmus vorgestellt. Deshalb an dieser Stelle nur der Hinweis, dass man sich jetzt auch auf ein und demselben Rechner nicht mehr auf die Reihenfolge von Schlüsseln in einem Hash verlassen kann.

```
test.pl

use Data::Dumper;

my %h = (1..6);
print Dumper \%h;
```

Unter bisherigen Perl-Versionen war die Reihenfolge eines Hashs bei jedem Lauf gleich - wobei die Reihenfolge bei jeder Maschine anders war.

```
$ perlbrew switch perl-5.16.2
$ perl test.pl
$VAR1 = {
          '1' => 2,
          '3' => 4,
          '5' => 6
        };
$ perl test.pl
$VAR1 = {
          '1' => 2,
          '3' => 4,
          '5' => 6
        };
```

Unter Perl 5.18 (und den letzten Developer-Releases) ist die Reihenfolge bei jedem Programmlauf unterschiedlich.

```
$ perlbrew switch perl-5.17.10
$ perl test.pl
$VAR1 = {
          '3' => 4,
          '5' => 6,
          '1' => 2
        };
$ perl test.pl
$VAR1 = {
          '5' => 6,
          '1' => 2,
          '3' => 4
        };
```

Da es zu einiger Unsicherheit bei Programmierern kam, noch ein kurzer Hinweis zu `keys` und `values`: Wenn man `keys` aufruft kann man auch weiterhin sicher sein, dass `values` die dazu passenden Werte in der richtigen Reihenfolge liefert.

```
$ perl test.pl
$VAR1 = {
          '5' => 6,
          '1' => 2,
          '3' => 4
        };
keys: 5, 1, 3
values: 6, 2, 4
```

Änderungen bei Regulären Ausdrücken

Spezialvariablen `$&`, `$'` und `` $` `` sind nicht länger langsam. Bisher hat die Verwendung dieser Spezialvariablen das komplette Programm ausgebremst (siehe auch $foo Ausgabe Nr. 4), und man hat die verwendeten Module nicht unter Kontrolle. Mit Perl 5.18 sind diese Variablen keine Performancefresser mehr.

Diese Spezialvariablen können verwendet werden, um den gematchten Teilstring, den Teil vor dem Match bzw. nach dem Match herauszubekommen:

```
$ perl -E 'q~$foo - DAS Magazin zur
   Softwareentwicklung mit Perl~ =~
    m{Magazin}; say $&; say $`'
Magazin
$foo - DAS
```

Einfache ungeschützte geschweifte Klammern sind jetzt als *deprecated* markiert. Geschweifte Klammern müssen jetzt entweder mit einem Backslash geschützt werden oder Teil eines Ausdrucks sein, in dem die Klammer ein Metazeichen ist.

Erlaubt:

```
/test{2,4}/
/test\{hallo\}/
```

Nicht mehr erlaubt:

```
/test{,4}/
```

Whitespaces matcht man mit `\s`. Dazu zählen jetzt auch die vertikalen Tabs.

In Regulären Ausdrücken kann man Perl-Code einbetten. Dazu gibt es die Konstrukte `(?{})` und `(??{})`. Zwar bleibt dieses Feature als experimentell gekennzeichnet, aber Dave Mitchell hat in den letzten Monaten das komplette Feature überarbeitet und dabei etliche Fehler behoben.

Die Code-Blöcke werden zur gleichen Zeit geparst wie der umgebende Code.

Ein Beispiel für eingebetteten Perl-Code in Regulären Ausdrücken:

```
$_ = 'a' x 8;
m<
   (?{ $cnt = 0 })        # Initialize $cnt.
   (
     a
     (?{
       local $cnt = $cnt + 1; # Update $cnt,
                         # backtracking-safe.
     })
   )*
   aaaa
   (?{ $res = $cnt })    # On success copy to
                     # non-localized location
>x;
```

Ein neues Feature ist die Einführung von Set-Operationen in Regulären Ausdrücken. Damit kann man Zeichenklassen zusammen führen, einzelne Zeichen aus einer großen Zeichenklasse herausnehmen oder Alternativen einführen. Sollen z.B. alle Großbuchstaben von A bis Z gematcht werden mit den Ausnahmen C, K bis N und Y, dann kann man das bisher so schreiben:

```
/[ABD-JO-XZ]/
```

Mit den neuen Sets ist folgendes möglich

```
/(?[ [A-Z] - [CK-NY] ])/
```

Das wirft unter Perl 5.18 noch eine Warnung *The regex_sets feature is experimental in regex;...*, die man mit `no warnings "experimental::regex_sets"` ausschalten kann.

Lexikalische Subroutinen

Ein lang gehegter Wunsch vieler Programmierer wird jetzt Wirklichkeit: Lexikalische Subroutinen. Es ist zwar noch als experimentell gekennzeichnet, aber es ist schon ganz brauchbar. Um damit arbeiten zu können, muss man folgende drei Zeilen schreiben:

```
use 5.018;
no warnings "experimental:lexical_subs";
use feature "lexical_subs";
```

Jetzt kann man folgendes schreiben:

```
package TestLexicalSubs;
{
  use 5.018;
  no warnings "experimental::lexical_subs";
  use feature "lexical_subs";

  my sub test;
  sub test {
    say "test";
  }

  sub hallo {
    say "hallo";
    test();
  }
}

package main;

use feature 'say';

TestLexicalSubs::hallo();
eval{
  TestLexicalSubs::test()
} or say "Fehler: $@";
```

Das erzeugt folgende Ausgabe:

```
hallo
test
Fehler: Undefined subroutine
  &TestLexicalSubs::test called ...
```

Wie man sieht, kann man auf die Subroutine `test` nur innerhalb des Blocks in dem sie deklariert wurde aufrufen.

Neben `my sub ...` kann man auch `our sub ...` und `state sub ...` verwenden. In der Dokumentation `perlsub` findet man einen größeren Abschnitt zu lexikalischen Subroutinen.

Sonstiges

`qw()` kann nicht mehr als Klammernpaar bei `for`-Schleifen verwendet werden. An vielen Codestellen konnte man `qw()` als Ersatz für runde Klammern finden. Z.B.

```
for my $name qw(hannah martin paula) {
    say $name;
}
```

Schon mit Perl 5.14 wurde das als *deprecated* markiert. Mit Perl 5.18 bricht das Programm ab:

```
syntax error at -e line 1, near
    "$name qw(hannah martin paula)"
Execution of -e aborted due to
    compilation errors.
```

In nachgestellten Konstrukten funktioniert das aber weiterhin:

```
$ perl -E 'say for qw(hannah martin paula)'
hannah
martin
paula
```

Neuer Mechanismus für experimentelle Features: Wie schon in einigen Beispielen in diesem Artikel gezeigt, gibt es einen neuen Mechanismus für experimentelle Features. Der `use feature`-Aufruf bleibt wie bisher, allerdings geben die (meisten) experimentellen Features eine Warnung aus, die mittels `no warnings "experimental::<feature_name>"` deaktiviert werden können.

Perl 5.18 wird mit Unicode 6.2 ausgeliefert.

Wenn ein Zeichen, das mittels `\N{...}` benannt wird, nicht existiert, wird jetzt ein Fehler geworfen. Eigene Aliase (siehe *CUSTOM ALIASES* in `charnames`) können jetzt auch Nicht-Latin1-Zeichen enthalten. Somit wäre auch ein `\N{þ€ł}` möglich.

Kein neues Feature, aber eine Änderung an einem bestehenden Feature, ist das Markieren der *switch*-Familie als experimentell. Ein anderes Feature, das mit Perl 5.10 eingeführt wurde, ist jetzt als *deprecated* markiert: Das lexikalische `$_`. Es gab damit einige Probleme: Einige CPAN-Module erwarten, dass sie mit dem globalen `$_` arbeiten (z.B. `List::Util`). Prototypen haben damit Probleme gehabt und selbst XS-Code konnte nicht darauf zugreifen.

Seit Perl 5.10 wurde `CPANPLUS` als alternativer CPAN-Client mitgeliefert. Nachdem einige Probleme mit `CPAN.pm` unter VMS beseitigt wurden, wird `CPANPLUS` nicht mehr mit ausgeliefert.

In bisherigen Perl-Versionen wurden alle Kategorien an Warnungen ausgeschaltet, wenn nur eine bestimmte Kategorie aktiviert wurde:

```
use warnings;

print $*; # "deprecated" Warnung

use warnings "void";
print $*; # keine Warnung
```

Ab Perl 5.18 bleiben einige Kategorien aktiv, bis explizit `no warnings` aufgerufen wurde. Dazu zählen auch *deprecated*-Warnungen:

```
use warnings;

print $*; # "deprecated" Warnung

use warnings "void";
print $*; # keine Warnung
```

Dieses Beispiel gibt die Warnung

```
$* is no longer supported, and will
   become a syntax error at - line 3.
$* is no longer supported, and will become
   a syntax error at - line 7.
```

aus.

Wenn beim Einlesen von Dateien der Variablen `$/` eine Re-

ferenz auf eine Zahl zugewiesen wurde, wurde mit <> diese Anzahl von Bytes gelesen:

```
$/ = \500;
open my $fh, '<', $file or die $!;
my $five_hundred_bytes = <$fh>;
```

Das kann Probleme machen wenn die Datei UTF-8-Daten enthält und diese auch mit dem `:encoding`-Perl I/O-Layer geöffnet wurde. Dann konnte es nämlich passieren, dass bei einem Zwei-Byte-Zeichen nur die Hälfte des Zeichens eingelesen wurde. Ab Perl 5.18 werden bei der Verwendung des `:encoding`-Perl I/O-Layer nicht *n* Bytes, sondern *n* Zeichen gelesen:

```
$/ = \500;
open my $fh, '<:encoding(utf-8)', $file
    or die $!;
my $five_hundred_chars = <$fh>;
```

Geschwindigkeitsverbesserungen

Auch an der Geschwindigkeit wurde gearbeitet. So gibt es einige signifikante Verbesserungen:

Wenn eine Kopie eines Strings, auf den ein Match ausgeführt wird, notwendig ist, wird nur der Teilstring kopiert, der für die Capture-Variable notwendig ist. Bisher wurde immer der komplette String kopiert, was bei sehr großen Strings viel Speicher und Zeit benötigt hat.

```
$&;
$_ = 'x' x 1_000_000;
1 while /(.)/;
```

profitiert sehr stark von dieser Optimierung.

Bei Regulären Ausdrücken ist das Matchen von Unicodezeichen schneller geworden. Der größte Sprung wurde bei der Verwendung von `\X` erreicht, bei dem ein Geschwindigkeitsvorteil von bis zu 40% erreicht wird. Auch Zeichenklassen mit Zeichen, deren Codepoint über 255 liegt, sind jetzt schneller.

Bei der Verwendung von *NO_TAINT_SUPPORT* bei der Kompilierung von Perl wird der Taint-Modus von Perl komplett abgeschaltet. Das sollte man allerdings nur benutzen, wenn man genau weiß, wofür das Perl verwendet wird. Man sollte dabei auch bedenken, dass dann die Tests einiger CPAN-Module und auch des Perl-Kerns selbst fehlschlagen. Allerdings bekommt man einen kleinen Geschwindigkeitsvorteil.

Sicherheit

Neben dem Rehashing, das oben beschrieben wurde, gibt es noch weitere Sicherheitsfixes und die Dokumentation wurde teilweise angepasst:

Wenn es Benutzern erlaubt ist, beliebige Faktoren für `x` zu bestimmen (z.B. `'string' x $userinput`), kann das für einem DoS-Angriff genutzt werden. In Perl < 5.15 konnte es sogar für einen Pufferüberlauf genutzt werden. Dieses Problem ist jetzt behoben.

Thomas Fahle

How To? String::Dump
Was genau ist in einem String enthalten?

Manchmal steckt in einem String nicht das drin, was man erwartet, z.B. nicht druckbare oder UTF-Zeichen.

String::Dump - `Dump strings of characters (or bytes) for printing and debugging` von `Nick Patch` vereinfacht das Debuggen solcher Probleme erheblich, indem es jedes Byte und jeden Codepoint eines Strings ausgibt.

Funktionen

String::Dump stellt **sechs** unterschiedliche Ausgabeformate über folgende Funktionen zur Verfügung.

- dump_hex($string): Hexadecimal (base 16)
- dump_dec($string): Decimal (base 10)
- dump_oct($string): Octal (base 8)
- dump_bin($string): Binary (base 2)
- dump_names($string): Unicode character name
- dump_codes($string): Unicode code point

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

use utf8;

use String::Dump qw( :all );

my $string = 'Føø Bār';

print dump_hex($string), "\n";

print dump_dec($string), "\n";

print dump_oct($string), "\n";

print dump_bin($string), "\n";

print dump_names($string), "\n";

print dump_codes($string), "\n";
```

Listing 1

Beispiel

Statt vieler Worte ein einfaches Beispiel in Listing 1.

Das Programm erzeugt folgende Ausgabe:

```
66 F8 F8 20 62 101 72

102 248 248 32 98 257 114

146 370 370 40 142 401 162

1100110 11111000 11111000 100000 1100010
100000001 1110010

LATIN CAPITAL LETTER F, LATIN SMALL LETTER
O WITH STROKE, LATIN SMALL LETTER O WITH
STROKE, SPACE, LATIN SMALL LETTER B,
LATIN SMALL LETTER A WITH MACRON, LATIN
SMALL LETTER R

U+0066 U+00F8 U+00F8 U+0020 U+0062
U+0101 U+0072
```

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

use String::Dump qw( :all );

use utf8;

my $string = 'Føø Bār';
print dump_hex($string),    "\n";
print dump_names($string), "\n";

{
  # Der gleiche String ohne
  # Pragma utf8
  no utf8;
  my $string = 'Føø Bār';
  print dump_hex($string),    "\n";
  print dump_names($string), "\n";
}
```

Listing 2

Debugging-Tipps

Algemein gilt: Wenn man UTF-8 debuggen möchte, sollten Strings auch UTF8-kodiert sein.

Literale Strings im Quelltext

Bei der Verwendung von Unicode-Strings im Quelltext sollte das Pragma `utf8` eingeschaltet sein (siehe Listing 2).

Das Programm erzeugt folgende Ausgabe:

```
46 F8 F8 20 42 101 72

LATIN CAPITAL LETTER F, LATIN SMALL LETTER
O WITH STROKE, LATIN SMALL LETTER O WITH
STROKE, SPACE, LATIN CAPITAL LETTER B,
LATIN SMALL LETTER A WITH MACRON, LATIN
SMALL LETTER R

46 C3 B8 C3 B8 20 42 C4 81 72

LATIN CAPITAL LETTER F, LATIN CAPITAL
LETTER A WITH TILDE, CEDILLA, LATIN
CAPITAL LETTER A WITH TILDE, CEDILLA,
SPACE, LATIN CAPITAL LETTER B, LATIN CAPITAL
LETTER A WITH DIAERESIS, HIGH OCTET PRESET,
LATIN SMALL LETTER R
```

Kommandozeilenargumente und Dateihandles

Auch hier sollte der Entwickler (m/w) sicherstellen, dass **alle** Eingaben UTF-8-kodiert sind oder eben nicht.

Für UTF-8 bietet sich die Verwendung des CPAN-Moduls `utf8::all` an, das alle Dateihandles und `@ARGV` automatisch UTF-8-kodiert.

Sonstige Eingabequellen

Strings aus Quellen wie Netzwerksockets, Formulardaten usw. sollten ggf. vorher mit `Encode::decode` von UTF8 dekodiert werden.

```
use Encode qw( decode );

print dump_hex(
           decode('UTF-8', $string)
      ) , "\n";
```

Kommandozeilentool dumpstr

Zum Debuggen auf der Kommandozeile eignet sich das mitgelieferte Tool `dumpstr`.

Der Schalter `-m` bzw. `--mode` legt das Ausgabeformat (hex, dec, oct, bin, names, codes) fest.

```
$ dumpstr -m names 'Føø Bār'
```

Herbert Breunung

Rezension - Gemeinschaft und Perl

Jono Bacon
The Art of Community
O'Reilly Juni 2012
572 Seiten, Softcover
ISBN 978-1-4493-1206-0
broschiert: €32,00 (englisch)
PDF, EPUB, MOBI: €25,49

Michael Schilli
Perl-Bundle
129 S., 31 Artikel
linux-magazin.de
PDF: €10,00

Auch in dieser Folge gibt es wieder Spiele, Spaß und Abwechslung. Es gibt einen Blick in die bunten und anregenden Perlartikel des Linux Magazins vom "Perlmeister" Michael Schilli.

Zuvor soll jedoch die Trilogie für werdende Organisationstalente abgeschlossen werden. Im Winter 2012 wurde eine kleines Büchlein über Projektmanagement vorgestellt, im Frühjahr 2013 ging es um die Planung und Aufbau einer erfüllenden Karriere und dieses Mal um die Leitung eines erfolgreichen, freien Softwareprojektes.

The Art of Community

Hierfür ist die Kunst der Kommunikation wesentlich. Denn die Kernidee freier Software besteht nicht nur darin, Programme aus innerer Befriedigung zu erstellen, sondern auch jeden (in verschiedenen Formen) mitmachen zu lassen. Deshalb hätte man das Buch ebenfalls "The Art of Communication" nennen können. Doch keine Sorge: Es ist kein gummiweiches Gerede eines selbsternannten "Social-Media-Experten". Jono Bacon *http://www.jonobacon.org/*, welcher als Komoderator des Linux-Podcasts "Lug Radio" bekannt wurde und unter anderem Community-Manager bei Ubuntu ist, kennt die Bewegung der freien Software aus verschiedenen Perspektiven. Besonders beim Thema *facebook* stellt der Autor klar, welchen Wert er welcher Art Gruppendynamik beimisst.

Selbst wenn vieles was er schreibt, im Groben absehbar war, so spricht er doch aus Erfahrung und sah, was funktioniert. Der Leser kann nach dieser Lektüre für die verschiedenen Phasen seines Projektes planen und ist auch für verschiedenste Situationen gut vorbereitet. Von der Ankündigung des Projektes per Blogpost, über die Eröffnungsrede einer Konferenz bis zu den mitfühlenden und aufmunternden Worten beim Ableben eines Kernentwicklers gibt der Inhalt das Wirken eines Oberhäuptlings gut wieder.

Bekannte Leithammel wie Linus Torvalds geben am Ende in Gesprächen noch ihre Sicht hinzu und auch zwischendurch gibt es Gastkommentare bekannter Köpfe. Die Abschnitte, nach denen man das Buch wieder ablegen könnte, sind meist nur etwas größer als eine Seite, aber der Inhalt baut sich in Bögen von ca. 40 Seiten auf. Nicht nur deswegen, sondern weil auch eine Art emotionale Intelligenz vermittelt wird, ist es keine leichte Lektüre für Zwischendurch. Von der Organisation der Benutzertests bis zur Wartung der Webinhalte wird ein weites Feld abgedeckt. Die 572 Seiten wurden sicher nicht durch Weitschweifigkeit gefüllt, trotz der manchmal etwas längeren Sätze. In der Erinnerung reduzieren die sich sowieso, da es weniger um die relativ dünn gesäten Einzelinformationen geht. Die dichteren Checklisten sind ja jederzeit nachsehbar.

Die zweite Auflage wurde deutlich erweitert und wirkt angenehm abgerundet. Der Text selber liest sich auch mit mäßigen Englischkenntnissen leicht, von gelegentlichen Idiomen abgesehen. Eine deutsche Ausgabe gibt es leider nicht. Wer auch nur daran denkt, ein freien Projekt zu gründen, kann hier mindestens ermahnt werden, an was alles zu denken ist.

Perl-Bundle

An die hart arbeitende Bevölkerung: Wann habt ihr das letzte Mal in Perl etwas zum Spaß programmiert? Irgend etwas, das in der Vorstellung reizvoll war, oder für das eigene Leben praktisch wertvoll wäre. Michael Schilli tut das ständig und schreibt darüber in der monatlichen Kolumne "Perl Snapshot" für das Linux Magazin. Seine inhaltlich und erzählerisch wirklich guten Bücher *Goto Perl 5* (deutsch) und *Perl Power* (englisch) wurden hier übergangen, da sie aus dem letzten Jahrtausend stammen. Als ihre größte Schwachstelle empfand ich die kaum vorhandene didaktische Struktur. In Artikellänge fährt Mike, der sich zu Recht "Perlmeister" nennt, allerdings zu voller Stärke auf. Denn diese Texte haben alles: gute Geschichten, knappe Erzählweise, ein Ziel, funktionierende Beispiele und relevante Themen. Selbst Leser, die vorher keine Erfahrung mit dem speziellen Thema hatten, können die Beispiele voll verstehen, da sie alles Wesentliche auf den zwei bis fünf Seiten (abzüglich der Quelltexte) erfahren. Für circa 33 Eurocent das Stück sind die Artikel unter *http://www.linux-magazin.de/E-Bibliothek/Themenpakete/Das-grosse-Perl-Bundle-fuer-Perlensammler* auch noch günstig zu haben, selbst wenn oder gerade weil sie nicht mehr taufrisch sind (Jahrgang 2007 bis 2010). Themen wie automatisierte Bildbearbeitung, rückwärts abgespielte mp3 oder ein sich akustisch bemerkbar machender Webserver werden jedoch nicht so schnell alt, weswegen sich der Kauf selbst im Jahr 2013 lohnt. Einzig der amüsante Beitrag von Prof. Dr. Jürgen Schröter über Denkfehler von Perl-Lernenden fällt etwas aus dem Rahmen. Sein "Grundwissen Perl" sollte hier auch noch besprochen werden.

Für Leser, die auf den Geschmack gekommen sind und den leiseren aber treffenden Humor vermissen, gibt es immer wieder aus gleicher Quelle das Sonderheft "Powerhouse Perl" für 10 Euro. Dessen beigelegte Linux-CD und der sehr gute Kurzeinstieg in Perl vom Zauberer Randal L. Schwartz interessieren $foo-Leser wahrscheinlich weniger, die 16 Schilli-Artikel, welche sich fast nicht mit denen dieser Sammlung überschneiden, sind aber immer noch deutlich günstiger als die Einzelnen als PDF für 99 Cent. Aber auch die Seite des Autors *http://perlmeister.com/* enthält viel Nützliches, inklusive Links zu seinen Modulen wie etwa dem bereits 4/2008 vorgestellten `Log::Log4perl`.

Ausblick

Die folgende Ausgabe wird hauptsächlich für Anfänger bereichernd sein. Für Einsteiger in die *Web-Entwicklung mit Perl und Mojolicious* gibt es diesen Titel von Michael Mangelsdorf. Michael Fitzgerald schrieb ein neues O'Reilly-Tierbuch namens *Einstieg in Reguläre Ausdrücke*. Und der vor allem durch seine Rundmail *Perl Weekly* bekannte Gabor Szabo, ist derzeit auch mit seiner *Perl Maven*-Seite aktiv, welche auch ein allgemeines Anfängerbuch umfasst.

Renée Bäcker

Mit Git, Jenkins und App::Cmd automatisiert OTRS-Pakete erstellen

In diesem Artikel stelle ich meine Vorgehensweise bei der Erstellung und Auslieferung von OTRS-Paketen vor. Kurz zu der Gesamtsituation:

Ich habe mehrere Kunden für die ich OTRS-Erweiterungen programmiere. Diese Erweiterungen werden gemäß der "Spezifikation" für Erweiterungen erstellt. Teilweise haben mehrere Kunden die gleichen Erweiterungen. Wenn eine Erweiterung ausgeliefert werden soll, sollte man ein paar Tests laufen lassen. Wenn alles in Ordnung ist, kann man das Paket bauen. Abschließend muss das Paket zum Kunden gelangen.

Diese Schritte kann man alle manuell gehen. Der Schritt "Programmieren" dürfte klar sein. Da kann man nicht allzu viel automatisieren. Man sollte sich hier an die Strukturen von OTRS halten und dann sind die nächsten Schritte recht einfach.

Für die Erstellung der Erweiterungspakete liefert OTRS ein Skript mit. Die Auslieferung an den Kunden kann man per Mail machen.

Je mehr man macht, desto schneller tauchen gewisse Probleme auf, aber bevor ich diese Probleme angehe, noch ein paar Worte zu den OTRS-Erweiterungen.

Über den sogenannten Paketmanager kann man in OTRS AddOns installieren. Dazu müssen die Pakete einer gewissen Spezifikation genügen. Das Paket an sich ist eine XML-Datei, in der alle Informationen zu finden sind:

- Metadaten wie Paketname, Host auf dem das Paket gebaut wurde, Datum
- für welche OTRS-Version(en) das Paket geeignet ist
- welche Dateien das Paket liefert
- Die Dateien selbst sind Base64-kodiert ebenfalls in der XML-Datei enthalten
- Änderungen, die an der Datenbank vorgenommen werden sollen (z.B. bei Installation oder Deinstallation)
- Code, der bei Installation bzw. Deinstallation ausgeführt werden soll

Der Paketmanager nimmt diese XML-Datei auseinander, installiert die Dateien, macht die Datenbankänderungen und führt ggf. Code aus.

Jetzt aber zu den Problemen, die bei der manuellen Erstellung und Auslieferung der Pakete auftreten können.

Zum Bau des Pakets wird eine Spezifikations-XML-Datei herangezogen, die dem endgültigen Paket sehr ähnlich ist (aber natürlich die Dateiinhalte nicht enthält). Diese Datei ist eventuell zwar gültiges XML, definiert aber nicht die richtigen Inhalte.

Das Programm von OTRS für den Bau der Pakete prüft das XML nicht auf Korrektheit.

Wenn es auf der Platte mehr Dateien in der Ordnerstruktur gibt als in der Spezifikations-XML-Datei aufgeführt sind, werden diese beim Bau einfach nicht berücksichtigt. Was, wenn der Programmierer einfach vergessen hat, diese Dateien aufzulisten? Es gibt zwar im
Repository der OTRS AG ein Skript, das das überprüft, aber es ist nicht im eigentlichen Programm enthalten.

Läuft das Paket auch wirklich mit allen möglichen OTRS-Versionen? Wie sieht es mit unterschiedlichen Perl-Versionen und unterschiedlichen Datenbanksystemen aus? Beim Testen vergisst man doch leicht mal einiges.

Ein weiteres Problem ist, dass man sich merken muss, welcher Kunde welches Paket bekommen muss und auch noch, welche OTRS-Version dieser Kunde einsetzt.

Liebt man es nicht gerade als Perl-Programmierer, Dinge zu automatisieren? Jedenfalls versuche ich das immer wieder. Also musste auch hier etwas her, das möglichst viel des Prozesses automatisieren soll und die oben genannten Probleme möglichst vermeidet.

In meinem Unternehmen wird eine größere Anzahl an Tools verwendet. Diese sollten, soweit es geht, auch hier verwendet werden um das Know-How einsetzen zu können.

Eines der Tools ist Jenkins. Jenkins ist ein Continuous Integration Server. Eine kurze Einleitung folgt gleich noch. Dieses Tool soll also die Tests ausführen um einige der Probleme zu vermeiden.

Problem: Es gibt noch nichts Gescheites, das die Dateien des OTRS-Pakets prüfen kann. Hier muss also etwas Eigenes her. Was das ist, werde ich später noch genauer vorstellen.

Die Tests sollen möglichst automatisch angestoßen werden. In Jenkins einloggen und bei dem zu bauenden Modul "jetzt bauen" klicken ist keine Lösung!

Da wir - wie vermutlich jeder mit "gesundem Menschenverstand" - für den Code eine Versionsverwaltung einsetzen, kann man sogenannte Hooks verwenden. In unserem Fall ist es ein "post-receive" hook in *git*.

Also sieht der Ablauf jetzt wie folgt aus (siehe Abbildung 1)

1. Schritt: Code ist klar, keine Anpassungen notwendig.

2. Schritt: Code in *git*. Ist auch klar. Code wird regelmäßig committed. Auf die Einzelheiten gehe ich hier nicht ein, da der Vortrag keine Einführung in *git* sein soll. Der geneigte Leser kann sich gerne online kundig machen. Da der Prozess aber auch mit anderen Systemen wie Mercurial, SVN oder CVS funktioniert, ist *git* hier nur als Platzhalter zu verstehen.

3. Schritt: git-Hook. Hier ist der erste Schritt zur Automatisierung feststellbar. Das Bauen des Pakets muss automatisch angestoßen werden. Der Schritt soll ausschließlich vom "zentralen" Repository (was dem dezentralen Verständnis von git widerspricht) aus getriggert werden. Auch soll nicht jeder Commit das Bauen des Pakets anstoßen.

4. Schritt: Jenkins. Durch den git-Hook werden Basistests und der Paketbau gestartet.

5. Schritt: Auslieferung an den Kunden.

Git-Hooks

Im Gegensatz zu Unittests soll das Bauen der Pakete nicht bei jedem Commit ausgeführt werden, weil auch "unfertiger" Code im Respository landet. Erst wenn die Änderungen im Hauptbranch landen und die Version des Pakets angepasst wurde, soll das Paket gebaut werden.

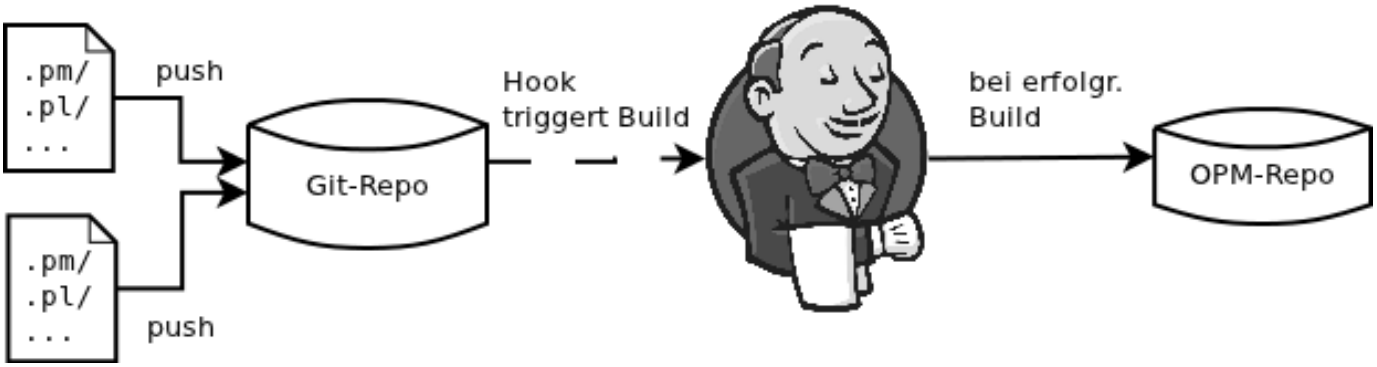


Abbildung 1

Es gibt verschiedene Hooks bei git. Einige Hooks können auf der lokalen Kopie ausgeführt werden, andere Hooks greifen auf dem Server. Hier kommt ein *post-receive*-Hook zum Einsatz, weil der Hook nur auf dem Server ausgeführt werden soll. Das hat mehrere Gründe:

Erstens sollen die Hooks nicht auf jeder neuen Entwicklungsumgebung installiert werden, sondern nach dem Klonen des Repositories soll man direkt loslegen können. Zweitens muss Jenkins auf das Repository mit den Änderungen zugreifen können um die Änderungen laden zu können.

Der Hook ist als kleines Perl-Skript realisiert worden. Man kann diese Hooks aber in jeder beliebigen Programmiersprache programmieren. Es muss nur darauf geachtet werden, dass der Hook ausführbar ist.

Hier der aktuelle git-Hook in Listing 1.

Informationen zu dem "alten" Stand, dem "neuen" Stand und zum Branch auf dem die Änderungen gemacht wurden, werden auf *STDIN* an den Hook übergeben.

Als nächstes wird hier ein *diff* erstellt, um feststellen zu können, ob das Paket überhaupt gebaut werden soll. Wie schon oben geschrieben, ist die Spezifikationsdatei für OTRS-Pakete eine Datei im XML-Format. Erst wenn das Paket auf die nächste Version gehoben wird - und das bedeutet, dass in der XML-Datei z.B. aus

```
<Version>0.9.99</Version>
```

ein

```
<Version>1.0.0</Version>
```

wird.

```
#!/usr/bin/perl

use strict;
use warnings;

use LWP::UserAgent;
use HTTP::Request;

my @STDIN = <STDIN>;
my ($old,$new,$ref) = split /\s+/, $STDIN[0];

exit 1 if $ref !~ m{/master\z};

my $diff = `git diff-tree -t -r -U $new`;

my ($sopm_diff) = $diff =~ m{
    diff \s+ --git \s+
      a/.*?\.sopm (.*?)
      (?: (?:diff \s+ --git) | \z )
}xms;

if ( $sopm_diff && $sopm_diff =~ m{
    ^ \+ \s*
    <Version> .*? </Version>
}xms ) {

    my $ua  = LWP::UserAgent->new;
    my $req = HTTP::Request->new(
        GET => 'http://hostname.de/job/GPW/build?token=GPWToken'
    );

    $req->authorization_basic(
        'otrs_builder',
        'fb235abce98fd13532',
    );

    my $response = $ua->request( $req );
}
```

Listing 1

Wenn diese Version angepasst wird, wird die API von Jenkins über einen einfachen HTTP-Request angesprochen. Der Benutzer der API muss sich mittels HTTP-Auth authentifizieren.

Jenkins

Jenkins ist - wie bereits erwähnt - ein sogenannter Continuous Integration Server. Mit Integration ist das Zusammenbauen der einzelnen Programmkomponenten zur Gesamtanwendung gemeint. In den meisten Projekten wird das Bauen der Gesamtanwendung erst kurz vorm Release gemacht, so dass Fehler im Prozess erst kurz vor wichtigen Terminen auffallen. Werden Programmkomponenten regelmäßig, also kontinuierlich, zusammengeführt, fallen die Fehler früher auf.

Ursprünglich ist Jenkins auf die Java-Welt zugeschnitten, aber durch Plugins und eigene Software kann man es auch für beliebige andere Projekte nutzen. Wie man eigene Perl-Projekte damit testen und bauen kann, wurde in der Ausgabe 21 des Perl-Magazins gezeigt.

Ein Plugin, das hier benötigt wird, ist git. Es wird benötigt, um den Quellcode der Pakete aus dem git-Repository zu holen.

Konfiguration von Jenkins

In den folgenden Absätzen zeige ich, wie Jenkins konfiguriert werden muss. Es geht dabei nur darum, wie das Bauen der OTRS-Pakete konfiguriert wird. Es geht nicht um die Konfiguration von Jenkins an sich.

Für jedes Paket muss ein eigener *Job* erstellt werden. Wie bereits erwähnt, ist Jenkins ursprünglich nicht für Perl- oder auch OTRS-Projekte gedacht. Aus diesem Grund muss ein "Free Style"-Projekt erstellt werden.

Als Name für das Beispielprojekt nehmen wir "GPW". In der Beschreibung sollte man kurz darlegen, was der Job macht. Alte Builds sollten verworfen werden, da alte Builds nicht wirklich benötigt werden. Die Historie der Codeänderungen ist im git zu finden und die Ausgaben des Paketbauens werden nicht unbedingt benötigt.

Als Source-Code-Management Software wird dann *git* ausgewählt. Ein Problem besteht mit dem git-Plugin aber noch: Man kann kein Passwort zum Auschecken angeben. Deshalb wurde ein SSH-Schlüssel ohne Passwort erstellt und ein User "Jenkins" wurde im git-Repository eingerichtet. Dieser hat aber nur Leserechte auf dem Repository.

Als nächstes muss man noch den Branch einstellen, der ausgecheckt werden soll. Ich habe hier festgestellt, dass es besser ist, direkt "master" einzutragen. Es kam sonst vor, dass der falsche Branch ausgecheckt wurde und damit ein falsches Paket erstellt wurde.

Der Build-Auslöser ist skriptgesteuert (nämlich durch den git-Hook). Ich mache es auch so, dass für jedes Paket ein eigenes Token für die Authentifizierung generiert wird.

Bisher wurde also nur konfiguriert, wie Jenkins an den Quellcode des Pakets kommt. Als nächstes folgen die Schritte, die zum endgültigen Paket führen.

Es sind mehrere Schritte erforderlich, die alle eine Ausführung auf der Shell bewirken.

```
opmbuild sopmtest ${WORKSPACE}/GPW.sopm
```

Als erstes wird die Spezifikationsdatei an sich getestet, unter anderem auf Korrektheit des XML. `${WORKSPACE}` ist eine Variable von Jenkins und ist der Pfad zu dem Verzeichnis, in das der Code geklont wird.

Der nächste Test prüft, ob alle Dateien in der Spezifikationsdatei auch auf der Festplatte zu finden sind und umgekehrt.

```
opmbuild filetest ${WORKSPACE}/GPW.sopm
```

Als drittes wird das Paket gebaut.

```
opmbuild build
--output /opt/packages/ ${WORKSPACE}/GPW.sopm
```

Abschließend wird noch eine Mailadresse eingetragen, die informiert wird, wenn ein Buildvorgang mal fehl schlägt (und dann wieder erfolgreich ist).

Schlägt der Bau des Pakets fehl, bekomme ich eine Mail (siehe Listing 2).

OTRS::OPM::Maker

Die Hauptarbeit bei der gesamten Angelegenheit wird von dem Modul `OTRS::OPM::Maker` bzw. von dessen Programm *opmbuild* erledigt. Die Befehle wurden schon in dem Abschnitt über die Konfiguration gezeigt.

In diesem Abschnitt zeige ich, wie das Modul bzw. das Programm umgesetzt wurde. Bei der Entwicklung ging es darum, möglichst wenig Arbeit selbst machen und das Programm leicht erweiterbar zu halten.

Zum Glück gibt es für sehr viele Aufgaben schon fertige Module auf dem CPAN. Ich habe mich hier für `App::Cmd` entschieden.

Die Stärken des Moduls werden deutlich wenn man sich die Umsetzung von `OTRS::OPM::Maker` anschaut.

Das Programm *opmbuild* besteht nur aus wenigen Zeilen:

```
use strict;
use warnings;

use Getopt::Long;

use OTRS::OPM::Maker;
OTRS::OPM::Maker->run;
```

Das ist schon alles. Damit ist klar, dass das Wichtige über die Module läuft. Die Methode `run` wird von `App::Cmd` bereitgestellt.

Damit hat auch das Modul nur wenige Zeilen:

```
package OTRS::OPM::Maker;

use App::Cmd::Setup -app;

# ABSTRACT: Module/App to build and test
# OTRS packages

our $VERSION = 0.05;

1;
```

Das `use App::Cmd::Setup -app` entspricht `use base qw(App::Cmd);`. Mit der ersten Variante ist es aber möglich, direkt Plugins einzubinden:

```
Started by user Renee Baecker
Building in workspace <http://hostname.de/job/GPW/ws/>
Checkout:GPW / <http://hostname.de/job/GPW/ws/> - hudson.remoting.LocalChannel@6f8f3f1
Using strategy: Default
Cloning the remote Git repository
Cloning repository git@hostname.de:GPW.git
git --version
git version 1.7.0.4
Fetching upstream changes from git@hostname.de:GPW.git
Seen branch in repository origin/HEAD
Seen branch in repository origin/master
Commencing build of Revision 0aaae88da265bd064c878458c0a660f48eda5265
 (origin/HEAD, origin/master)
Checking out Revision 0aaae88da265bd064c878458c0a660f48eda5265 (origin/HEAD, origin/master)
No change to record in branch origin/HEAD
No change to record in branch origin/master
[GPW] $ /bin/sh -xe /tmp/hudson1799527762675301040.sh
+ opmbuild sopmtest <http://hostname.de/job/GPW/ws/GPW.sopm>
[GPW] $ /bin/sh -xe /tmp/hudson5829059171426614887.sh
+ opmbuild filetest <http://hostname.de/job/GPW/ws/GPW.sopm>
Files listed in .sopm but not found on disk:
    - Kernel/System/PostMaster/Filter/GPWMails.pm
Files found on disk but not listed in .sopm:
    - Kernel/System/PostMaster/Filter/Mails.pm
[GPW] $ /bin/sh -xe /tmp/hudson8820500498704184856.sh
+ opmbuild build --output /home/jenkins/opms <http://hostname.de/job/GPW/ws/GPW.sopm>
Can't read <http://hostname.de/job/GPW/ws/Kernel/System/PostMaster/Filter/GPWMails.pm>:
  No such file or directory at /usr/share/perl5/Path/Class/File.pm line 60.
Build step 'Execute shell' marked build as failure
```

Listing 2

```
use App::Cmd::Setup -app => {
    plugins => [ 'Prompt' ],
};
```

Mit dem Skript und dem Hauptmodul ist die Basis für die Anwendung gelegt. Jetzt muss diese noch mit Leben gefüllt werden.

Das Leben kommt über die Kommandos der Anwendung, z.B.

- build
  opmbuild build ...

- filetest
  opmbuild filetest ...

- sopmtest
  opmbuild sopmtest ...

Jedes Kommando wird als eigenständiges Modul entwickelt. `App::Cmd` erkennt diese Module dann als Kommandos und kann diese aufrufen.

Ein Modul wird zu einem Kommando durch die Verwendung von `use OTRS::OPM::Maker -command;`.

Das Modul muss/kann dann noch einige Subroutinen definieren. Zum Beispiel eine Methode, die eine allgemeine Beschreibung zurückgibt, die bei einem einfachen Aufruf von `opmbuild` angezeigt wird:

```
$ opmbuild
Available commands:

  commands: list the application's commands
  help: display a command's help screen

  build: build package files for OTRS
  dbtest: Test db definitions in .sopm files
  dependencies: list dependencies for OTRS
    packages
```

Die Methode sieht dann wie folgt aus:

```
sub abstract {
  return "build package files for OTRS";
}
```

Mehr Informationen zur Verwendung des Kommandos werden ausgegeben, wenn man einfach `opmbuild build` aufruft:

```
$ opmbuild build
Error: need path to .sopm
Usage: opmbuild <command>

opmbuild build [--output <output_path>]
  <path_to_sopm>
      --output      Output path for OPM file
```

Dafür ist die Methode `usage_desc` zuständig:

```
sub usage_desc {
    return "opmbuild build
  [--output <output_path>] <path_to_sopm>";
}
```

Man sieht an der Beschreibung des Aufrufs, dass auch Parameter für die Aufrufe einzelnen erlaubt sind. Dazu muss man eine Methode implementieren, die diese Parameter spezifizieren:

```
sub opt_spec {
    return (
        [ "output=s",
             "Output path for OPM file" ],
    );
}
```

Für jeden Parameter muss eine Arrayreferenz definiert werden, in der zum einen der Parameter mit erwartetem Datentyp (z.B. `=s` für Strings), zum anderen eine Beschreibung des Parameters zu finden sind.

Neben diesen Parametern können weitere Argumente angegeben werden. Diese können und sollten validiert werden, damit man sicher sein kann, keine falschen Daten zu verwenden. Die Methode dafür heißt `validate_args`.

```
sub validate_args {
  my ($self, $opt, $args) = @_;

  $self->usage_error(
   'need path to .sopm' ) if
      !$args ||
      !$args->[0] ||
      !$args->[0] =~ /\.sopm\z/ ||
      !-f $args->[0];
}
```

Die Methode bekommt neben dem Objekt selbst auch die Parameter und die weiteren Argumente übergeben.

Die Fehlermeldung, die hier angegeben wurde, ist auch in der Ausgabe des `opmbuild build` wiederzufinden.

In der Methode `execute` ist dann der Code zu finden, der bei dem Kommando ausgeführt wird. Diese Methode bekommt die gleichen Parameter übergeben wie `validate_args`.

```
sub execute {
  my ($self, $opt, $args) = @_;

  # ... Code to build opm
}
```

Die Anwendung ist durch die Verwendung von `App::Cmd` und dessen Architektur sehr leicht erweiterbar. Neue Kommandos sind durch neue Module einfach umsetzbar.

Auslieferung an den Kunden

Über die SysConfig in OTRS ist es möglich, Online-Repositories in den Paketmanager einzubinden. Das machen wir uns bei der Auslieferung von Paketen an Kunden zu Nutze.

Da es zu umständlich ist, für jeden Kunden ein eigenes Repository zu erstellen, wurde eine Mojolicious-Anwendung geschrieben, die die Repositories "virtualisiert". In der Konfiguration werden nur noch IDs für die Repositories eingetragen und welche Pakete den einzelnen Repositories zugeordnet sind.
Folgende Routen sind eingerichtet:

http://otrsrepos.perl-services.de/<id>/
http://otrsrepos.perl-services.de/<id>/otrs.xml
http://otrsrepos.perl-services.de/<id>/paket-0.0.1.opm

Die ersten beiden URLs bedeuten das Gleiche. In der *otrs.xml* sind alle Pakete aufgeführt, die in dem Repository zur Verfügung stehen. Der Name der Datei ist dabei von OTRS vorgegeben, da der Paketmanager den Namen automatisch an die URL des Repositories anhängt (siehe Abbildung 2).

Die letzte URL liefert dann das Paket aus (siehe Abbildung 3).

Lieferdienst "Mojolicious"

Da es nur wenige Zeilen sind, die für die Anwendung benötigt werden, wird nur `Mojolicious::Lite` verwendet. Der

Abbildung 2: Online-Repository in der SysConfig eintragen

Abbildung 3: Pakete, die über den Paketmanager installiert werden können

Quellcode sieht aus, wie im folgenden dargestellt:

```
use Mojolicious::Lite;
use OTRS::Repo;

plugin 'RenderFile';

get '/:repo' => sub {
    my $self = shift;

    my $repo     = $self->param( 'repo' );
    my $packages = $self->config->{$repo};

    my $index    =
          OTRS::Repo->create_index( $repo );
    $self->render_text( $index );
};

get '/:repo/*package' => sub {
    my $self = shift;

    my $package  =
        $self->param( 'package' );
    my $repo     = $self->param( 'repo' );
    my $packages = $self->config->{$repo};

    if (
     !$package || $package eq 'otrs.xml' ) {
        my $index =
          OTRS::Repo->create_index( $repo );
        $self->render_text( $index );
        return;
    }

    die if !first{
       $_ eq $package }@{$packages};

    $self->render_file(
        filepath =>
         '/opt/packages/' . $package,
    );
};
```

Fazit und zukünftige Entwicklung

Mit Perl, git und Jenkins lässt sich vieles im Release-Prozess vereinfachen und automatisieren. Diese Methode vermeidet viele Fehler und man kann dem Kunden eine bessere Qualität liefern.

In diesem Szenario gibt es gerade mit den Möglichkeiten von Jenkins noch ein großes Verbesserungspotential und einiges ist auch schon in Planung. So ist geplant, aus Jenkins heraus virtuelle Maschinen mit unterschiedlichen Szenarien (verschiedene OTRS-Versionen, unterschiedliche Datenbanksysteme, etc.) zu erstellen, zu starten und dann die Tests darauf auszuführen.

Leserbriefe

Hallo,

manche öffentliche Debatte um vermeintlichen Sexismus und "unangemessenen" Sprachgebrauch finde ich aufgebauscht. Aber die Wortwahl im Artikel und bei den Programmnamen von Ulli Horlacher und insbesondere Zwischenüberschriften wie "Public SEX" fand ich dann doch störend. Ich glaube auch, dass die Benennung seiner Programme nicht eben akzeptanzfördernd bei den potentiellen Anwendern ist. Je nach Rezipient werden (Perl-)Programmierer so als Nerds, als infantil und gestrig wahrgenommen. Schade.

Jetzt aber noch zu ein paar anderen Punkten.

Die Mojolicious-Einführung gefällt mir und für mich ergibt sich daraus ein Vorschlag für weitere Artikel nach dem Ende dieser Serie: Quasi die gleichen Inhalte nochmal - nur dann auf Basis von Dancer 2, und mit ein paar Erläuterungen zu den erkennbaren Unterschieden und Gemeinsamkeiten der beiden Frameworks.

Wo wir beim Thema Web-Anwendungen sind: Der Adventskalender von Jesse Luehrs et. al. zu OX (http://iinteractive.github.com/OX/advent/index.html) ist spannend. Vielleicht ist er oder einer der anderen Autoren bereit, OX näher vorzustellen?

Thomas Klausner wiederum hat unter http://domm.plix.at/perl/2012_12_getting_started_with_zeromq_anyevent.html einen Blogbeitrag zum Thema ZeroMQ und AnyEvent veröffentlicht, der zum AnyEvent-Artikel in der aktuellen Ausgabe passt. Vielleicht könnte er (oder jemand anderes) auf dem Blog-Beitrag aufsetzen und ZeroMQ in dem Kontext mal näher vorstellen?

Beides sind Themen, die mich interessieren, über die ich aber bisher kaum etwas weiß. Bevor also die Frage kommt: Nein, ich könnte das leider nicht übernehmen.

Und zu guter Letzt: Die Sprache bei dem offenbar übersetzten Artikel von Sawyer X klingt etwas holperig. Das ist nicht weiter schlimm, aber ich für meinen Teil hätte kein Problem damit, wenn $foo auch Artikel in englischer Sprache veröffentlichte.

- Heiko Jansen

Katrin Bäcker

Rezension - Mal ein etwas anderes Buch

Nicole Dornseif & Maximilian Dornseif
Eltern sein - kurz und geek
136 Seiten
ISBN 978-3-86899-827
€ 9,90

„Eltern sein - kurz & geek“ von Nicole und Maximilian Dornseif, 2012 erschienen im O’Reilly Verlag, ist doch das passende Buch für mich, dachte ich, als ich davon hörte. Schließlich habe ich ja selbst einmal viel mit Informatik zu tun gehabt – jedenfalls bevor ich Mutter zweier Kinder (heute 5 Monate und knapp 2 Jahre) geworden bin.

Dass es aber am Ende für mich eine Herausforderung darstellen würde, dieses Buch dann auch wirklich zu lesen und diese Rezension zu schreiben, habe ich anfangs nicht gedacht. Zwei Kinder wollen nun mal unterhalten werden, sie werden immer dann krank, wenn man es nicht gebrauchen kann, und bescheren uns die eine oder andere schlaflose Nacht. Genauso sehen es auch die Autoren des Buches: Schon die Einleitung des Buches trifft es meiner Meinung nach auf den Punkt – Kinder sind nun einmal unberechenbar! Im Leben mit Kindern gibt es immer wieder Überraschungen und ein Handbuch wird bei der Geburt des Kindes auch nicht mitgeliefert ;-)

Das Buch ist in sieben Kapitel unterteilt, die sich unterschiedlichen Schwerpunktthemen widmen:

1. Dein Kind und Du
2. Dein Kind und sein Körper
3. Dein Kind und seine Aktivitäten
4. Mit Kindern unterwegs
5. Dein Kind und die Anderen
6. Dein Kind und die Technik
7. Dein Kind, du und viel Spaß

Besonders hat es mir das zweite Kapitel „Dein Kind und sein Körper“ angetan. Wahrscheinlich liegt es daran, dass es mit zwei Kindern, die noch nicht einmal das Kindergartenalter erreicht haben, einfach das passendste Kapitel ist. Wickeln ist dann notwendig, wenn man gerade gar keine Zeit hat – Ja, kann ich bestätigen. Durchschlafen, ein Gerücht? – Ja, das ist bei uns daheim genauso. Kinder schaffen es immer sich gegenseitig anzustecken, wenn sie krank sind – Stimmt (mussten wir gerade leidvoll erfahren)!

Die Autoren, selbst Eltern, beschreiben in diesem kurzen, sehr witzig geschriebenen Buch, ihre Erfahrungen mit Kindern und geben Ratschläge. Es handelt sich glücklicherweise aber nicht um einen klassischen Elternratgeber (von denen gibt es mehr als genug). Vielmehr zeigen die Autoren viele Situationen auf, die Lesern, die selbst Kinder haben, sehr bekannt vorkommen werden. Dabei geben sie kleine, humorvolle Tipps, wie man mit der ein oder anderen schwierigen Situation umgehen kann. Passende lustige Bilder ergänzen den Text und durch den regelmäßigen Bezug in die Computerwelt ist es ein super Buch für Informatiker. Beispielsweise wird die Neugier von Kindern als „Mehr Input, mehr Input“ dargestellt oder das Fifo-/Lifo-Prinzip verwendet, um die Aufgaben zu beschreiben, die man als Elternteil bewältigen muss.

Das Buch beschreibt sehr schön die Realität mit Kindern und bringt den Leser oft zum Schmunzeln. An vielen Stellen findet man sich selbst wieder. Ein Leben mit Kindern ändert nun mal vieles: Lange Tage und Nächte vor dem PC werden zur Seltenheit und, wie ich anfangs schon erwähnte, kann das Lesen eines ca. 130-seitigen kleinen Buches schon zu einer großen Herausforderung werden!

Kurz zusammengefasst: Ein MUSS für Computerfreaks, die ein Kind erwarten oder schon Kinder haben, aber auch eine empfehlenswerte Lektüre für Menschen, die nicht unmittelbar zur Zielgruppe gehören.

CPAN NEWS

Eigentlich war an dieser Stelle vorgesehen, `jQuery::File::Upload` genauer vorzustellen. Aber das Modul benötigt `Image::Magick` und das lässt sich in der aktuellen Version (und auch in den letzten Versionen zuvor) nicht installieren. Aus diesem Grund wird stattdessen `Data::Faker` näher vorgestellt.

Data::Faker

Bei Tests oder bei Prototypen hat man nicht unbedingt Echtdaten zur Verfügung, aber die verwendeten Daten sollten nicht komplett an den Haaren herbeigezogen sein. Also muss man die Daten fingieren. In diesem Fall ist `Data::Faker` eine gute Wahl.

Benötigt man z.B. Daten für Benutzer einer Anwendung, so kann man das folgendermaßen lösen:

```
use Data::Faker;

my $faker = Data::Faker->new();

print "Name:    ".$faker->name."\n";
print "Company: ".$faker->company."\n";
print "Address: ".$faker->street_address,
                  "\n";
print "         ".$faker->city.", ".
                   $faker->us_state_abbr." ".
                   $faker->us_zip_code."\n";
```

Jetzt ist aber das Problem, dass Daten von Benutzern aus Deutschland benötigt werden. Für diesen Fall muss man Plugins für das Modul registrieren. Als erstes sollte man sich überlegen, welche Daten benötigt werden und wie diese aufgebaut sind. Name und Firma sind in Ordnung, aber die Straßenangabe ist in Deutschland etwas anders aufgebaut. Hier kommt erst der Straßenname und dann die Hausnummer.

Für die deutschen Angaben schreiben wir ein Plugin. Standardmäßig werden alle Module unter `Data::Faker::*` geladen.

```
package Data::Faker::AddressDE;
```

Das Plugin muss von `Data::Faker` erben und die Funktionen registrieren.

```
use base 'Data::Faker';

__PACKAGE__->register_plugin(
    strasse => sub {
        return
          shift->city . 'er Str. '
          . int( rand 200 );
    },
);
```

Der erste Schlüssel bei der Registrierung des Plugins ist der Name der Methode, die dem Objekt dann zur Verfügung steht. Der Wert beschreibt, wie die Daten "gefunden" werden. Hier ist es eine Subroutine. Damit hat man den vollen Einfluss darauf, was zurückgegeben wird.

Das Plugin kann dann direkt im Programm verwendet werden: `print $faker->strasse`. Eine Beispielausgabe:

```
Monterey Parker Str. 150
```

Natürlich sind Straßennamen in Deutschland etwas komplexer, aber das würde hier zu weit führen.

Gibt es nur einen eingeschränkten Wertebereich, kann man auf die Subroutinenreferenz verzichten und stattdessen eine Arrayreferenz nehmen.

```
__PACKAGE__->register_plugin(
    kindergarten_alter => [3, 4, 5],
);
```

`Data::Faker` wählt dann zufällig einen der Werte aus.

Ein nettes Feature von `Data::Faker` ist, dass man bei numerischen Werten nicht nur direkt mit Zahlen arbeiten kann, sondern auch mit Platzhaltern.

```
__PACKAGE__->register_plugin(
    preis => [
        '#.###,##',
        '###,##',
        '##,##',
        '#,##'
    ],
);
```

Das Modul ersetzt alle `#` durch zufällige Ziffern. Mit dem gerade gezeigten Plugin sind dann folgende Ausgaben möglich:

```
92,64
7,51
2.743,51
851,74
7,85
```

Sollten mehrere Plugins die gleiche Methode zur Verfügung stellen, besteht leider keine Möglichkeit, einzelne Methoden von Plugins auszuschließen. Vielmehr wird eine der Methoden zufällig gewählt. Soll eine bestimmte Methode herangezogen werden, kann man entweder den vollen Namen angeben

```
$faker->Data::Faker::PluginName::methode();
```

oder nur bestimmte Plugins laden:

```
use Data::Faker qw(Plugin1 Plugin2);
```

HTML::Zoom

Als gäbe es nicht schon genug Template-Engines, gibt es jetzt eine weitere: `HTML::Zoom`. Doch diese Template-Engine funktioniert etwas anders als die üblichen. Man findet im Template keine Template-Variablen, Schleifen oder ähnliches. Die Daten kommen über einen anderen Weg in die Ausgabe - über Selektoren. Am besten sieht man es an einem Beispiel - siehe Listing 1.

Hier ist nichts zu erkennen, das auf ein typisches Template schließen lässt. Das ist auch schon einer der Vorteile von `HTML::Zoom`. Designer können "ganz normal" eine Webseite erzeugen oder später auch bearbeiten. Sie müssen sich nicht darum kümmern, ob das Element noch zur Schleife gehört oder nicht. Auch im Browser lässt sich das Template anschauen ohne dass es Darstellungsprobleme gibt und durch die "Echtwerte" lässt sich das Aussehen einer ausgelieferten Seite nicht nur erahnen.

```
my $tmpl = q~
<html>
  <head><title>1</title></head>
  <body>
    <div id="list">
      <h1 id="header">Test</h1>
    </div>
    <ul id="list">
      <li>Punkt</li>
    </ul>
  </body>
</html>
~;
```

Listing 1

Im Perl-Programm wird als erstes ein Objekt benötigt:

```
my $zoom = HTML::Zoom->from_html( $tmpl );
```

Dann kann es losgehen. Die Teile, die ersetzt werden sollen, werden mit einem CSS-Selektor herausgesucht:

```
my $output =
  $zoom->select( '#header' )
       ->replace_content( 'Perl-Magazin' );
```

Und danach kann man dann das Ergebnis ausgeben:

```
print $output->to_html;
```

Die Ausgabe sieht folgendermaßen aus:

```
<html>
  <head><title>1</title></head>
  <body>
    <div>
      <h1 id="header">Perl-Magazin</h1>
    </div>
    <ul id="list">
      <li">Punkt</li>
    </ul>
  </body>
</html>
```

Auch Wiederholungen lassen sich leicht umsetzen:

```
my $loop_output =
  $zoom->select( '#list' )
       ->repeat_content(
         [
           map{ my $nr = $_;
             sub {
               $_->select('li')
                 ->replace_content("$nr");
             }
           }(1..3),
         ]
       );

print $loop_output->to_html;
```

Und die dazugehörige Ausgabe:

```
<html>
  <head><title>1</title></head>
  <body>
    <div>
      <h1 id="header">Test</h1>
    </div>
    <ul id="list">
      <li>1</li>

      <li>2</li>

      <li>3</li>
    </ul>
  </body>
</html>
```

Man sieht, dass der Perl-Code mit `HTML::Zoom` nicht unbedingt kürzer wird. Noch umständlicher sieht es aus, wenn komplexere Sachen wie Attribute für ein Tag festlegt bzw. gefüllt werden sollen. Bleiben wir bei obigem Beispiel. Am Ende soll jedes `li` eine eindeutige `id` haben. Es bringt also nichts, einfach im Template `<li id="item">` zu schreiben, denn dann würden bei der Ausgabe drei Elemente die gleiche `id` haben.

Das `map` in dem `repeat_content` muss folgendermaßen aussehen:

```
map{ my $nr = $_;
  sub {
    $_->select('li')
      ->add_to_attribute(
        id => "item$nr",
    )
    ->then
    ->replace_content("$nr");
  }
}(1..3),
```

Das `then` führt das vorherige `select` erneut aus, damit mehrere Aktionen auf dem gleichen Element ausgeführt werden können.

Sollen neben der `id` noch weitere Attribute zu dem Element hinzugefügt werden, müssen weitere `add_to_attribute`/`then`-Aufrufe erfolgen. Das macht deutlich, dass solche Konstrukte nicht so schön sind. Aber der Vorteil mit dem Standard-HTML als Template sowie die Stärken beim Streaming machen `HTML::Zoom` zu einer echten Alternative. Auch ist das Modul sehr nützlich wenn man es in einem Proxy einsetzt, bei man auf die HTML-Ausgabe einer Anwendung keinen Einfluss hat, am Endergebnis aber noch etwas ändern möchte.

PerlIO::http

Schluss mit Datei herunterladen und dann einlesen. Mit `PerlIO::http` kann man das direkt machen, ohne ein zusätzliches Modul zu laden. Es wird automatisch geladen wenn der "http"-Layer verwendet wird. Die HTTP-Statusmeldungen werden in "normale" Fehlermeldungen übersetzt.

Zwei, drei Beispiele werden den Einsatz des Moduls am besten verdeutlichen können.

```
  perl -E 'open my $fh, "<:http","http://www.
ascii-art.de/ascii/t/truck.txt";
  while ( <$fh> ){ print; last if $. == 16 }'
```

Heraus kommt dabei die nette Frontansicht eines LKW.

```
          .--------------.
          |~              ~|
          |H______________H|
          |.------------.|
          ||::..      __ ||
          |'--------'--''||
          | '. _______ .' |
          | _  |======| _  |
          |(_)|======|(_)|
          |___|======|___|
          [________________]
          |##|        |##|
    jrei  '""'        '""'
```

Versucht man eine nicht-existierende Datei zu öffnen, bekommt man auch die entsprechende Fehlermeldung.

```
  perl -E 'open my $fh, "<:http","http://www.
perl-magazin.de/404.html" or die $!'
  Datei oder Verzeichnis nicht gefunden at -e
line 1.
```

App::GitHubPullRequest

Mittlerweile haben viele Entwickler ihre CPAN-Module und andere Open Source Software auf Github liegen. Ein erfreulicher Moment ist es, wenn man einen "Pull Request" von einer Programmiererin erhält. Jetzt aber die gewohnte Programmierumgebung (Terminal) verlassen und sich auf der Webseite von Github den Pull Request anschauen? Unpraktisch, da man aus seinem normalen Arbeitsablauf gerissen wird. Praktisch ist da das Modul `App::GitHubPullRequest`, mit dem man die Pull Requests von der Kommandozeile aus managen kann.

`git pr` listet einfach die offenen Pull Requests auf. Mit `git pr show #` bekommt man die Kommunikation zu dem Pull Request angezeigt. Über `git pr checkout 1` wird ein neuer Branch *pr/1* erzeugt, in dem der Code zu finden ist. Danach noch ein `git merge` und `git push` und schon kann man ein neues Release erstellen.

Wer sich vor dem Checkout erstmal das Diff anschauen möchte, kann das über `git pr patch 1` machen.

```
cd ~/dev/git/CPAN/Perl-Critic-OTRS
git pr
Open pull requests for
   'reneeb/Perl-Critic-OTRS':
1 2013-03-31T22:29:52Z
    Some small improvements.
git pr show 1
Date:    2013-03-31T22:29:52Z
From:    mgruner
Subject: Some small improvements.
Number:  1

This PR updates the git repository [...]
```

again

Die Anwendung läuft in einer persistenten Umgebung und hin und wieder ändert sich etwas an einem Modul. Also Server/Daemon stoppen und neu starten. Gerade während der Entwicklung kann das nerven und große Anwendungen brauchen eventuell auch einige Zeit bis alles hochgefahren wurde. Für den Apache gibt es z.B. `Apache::Reload`, das sich dieses Problems annimmt und das Modul neu lädt, wenn sich am Code etwas geändert hat.

Für andere Anwendungen wurde `again` geschaffen. Einfach das Modul, das sich in der Entwicklung befindet mit `use again 'Modul'` laden.

```
# default import
use again 'LWP::Simple';
# no import
use again 'LWP::Simple', [];
# import only get
use again 'LWP::Simple', [qw(get)];
# default import (!!)
use again 'LWP::Simple', ();
# import only get
use again 'LWP::Simple', qw(get);

use again;
require_again 'Foo::Bar';
```

Carp::Always:: SyntaxHighlightSource

Ein wirklich nützliches Modul ist `Carp::Always::SyntaxHighlightSource`. Damit bekommt man für Warnungen und Fehler immer einen Code-Ausschnitt auf der Kommandozeile angezeigt und die wichtige Zeile wird markiert. Das bezieht sich aber nicht nur auf die Code-Zeile, die die Warnung oder den Fehler ausgibt, sondern auf den gesamten Stacktrace. Das Modul muss nicht gleich im Quellcode geladen werden, sondern kann, sobald man einem Fehler analysieren möchte, nachträglich geladen werden.

In der Abbildung 1 ist zu sehen, wie die Konsolenausgabe dann aussieht.

```
use Carp;
use Carp::Always::SyntaxHighlightSource;

warn "Warnung";
```

```
bash-4.2$ perl carp_always_syntaxhighlightsource.pl
context for carp_always_syntaxhighlightsource.pl line 9:

   6: use Carp;
   7: use Carp::Always::SyntaxHighlightSource;
   8:
   9: warn "Warnung";
  10: carp "Warnung (carp)";
  11:
  12: croak "Fehler (croak)";
==========================================================
Warnung at carp_always_syntaxhighlightsource.pl line 9
context for /home/reneeb/perl5/perlbrew/perls/perl-5.17.10/li

  99:
 100: sub croak   { die shortmess @_ }
 101: sub confess { die longmess @_ }
 102: sub carp    { warn shortmess @_ }
 103: sub cluck   { warn longmess @_ }
 104:
 105: BEGIN {
==========================================================
```

Abbildung 1: Konsolenausgabe

NEWS

Mai 2013

02. Treffen Dresden.pm
07. Treffen Frankfurt.pm
08. Treffen Niederrhein.pm
13. Treffen Erlangen.pm
Treffen Ruhr.pm
14. Treffen Stuttgart.pm
15. Treffen Darmstadt.pm
25./26. Polnischer Perl-Workshop
28. Treffen Bielefeld.pm
29. Treffen Berlin.pm

Juni 2013

03.-05. YAPC:NA
04. Treffen Frankfurt.pm
06. Treffen Dresden.pm
Treffen Suttgart.pm
10. Treffen Ruhr.pm
17. Treffen Erlangen.pm
12. Treffen Niederrhein.pm
19. Treffen Darmstadt.pm
25. Treffen Bielefeld.pm
26. Treffen Berlin.pm

Juli 2013

02. Treffen Frankfurt.pm
Treffen Stuttgart.pm
04. Treffen Dresden.pm
08. Treffen Ruhr.pm
10. Treffen Niederrhein.pm
15. Treffen Erlangen.pm
17. Treffen Darmstadt.pm
30. Treffen Bielefeld.pm
31. 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.

Hier könnte Ihre Werbung stehen!

Interesse?

Email: werbung@foo-magazin.de
Internet: http://www.foo-magazin.de (hier finden Sie die aktuellen Mediadaten)