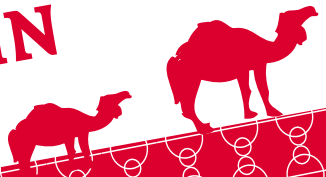
Ausgabe 2/2014 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN



Was ist neu in Perl 5.20?
Neuerungen in Perl

App::Fatpack
Warum Fatpack?

Nr. 30

Subroutinen-Signaturen

Swiss Perl Workshop 2014

Friday, 5. and Saturday, 6. September 2014

Venue: Flörli Olten

Language: English

Meet Perl hackers from Switzerland and other countries

•

Learn from and be inspired by talks

•

Chat and socialise during breaks and attendees dinner

Register today

•

Submit your talk

•

Become a Sponsor

www.perl-workshop.ch

VORWORT

Die Gemeinschaft macht's...

In der letzten Ausgabe hatte ich das Aus für das Perl-Magazin angekündigt. Auf Perl-Community.de gab es daraufhin eine Diskussion, ob es nicht möglich sei, ein Magazin aus der Gemeinschaft heraus zu stemmen: mit einem Forum oder Wiki als Basis um die verschiedenen Artikel zusammenzutragen. Wir wollen das Experiment wagen und suchen daher noch etliche Autoren, die hin und wieder einen Artikel schreiben wollen. Meine Erfahrungen mit $foo haben gezeigt, dass man für jede Ausgabe ca. 10-12 Autoren braucht, die etwas beisteuern. Das hochgerechnet bedeutet, dass man einen Pool aus 25-30 Autoren braucht, um eine komplette Ausgabe inhaltlich zu füllen.

Der entsprechende Thread ist unter http://bit.ly/1eeVfG7 zu finden.

Es wäre klasse, wenn das Experiment glücken würde.

Noch ein Projekt, bei dem die Gemeinschaft wichtig ist: Perl! Im Gegensatz zu vielen anderen Sprachen wird die Perl-Entwicklung nicht von einem Unternehmen stark beeinflusst, sondern hier ist die Gemeinschaft ganz wichtig - auch wenn der Pumpking das letzte Wort hat. An der Version 5.20, die vor kurzen veröffentlicht wurde, haben wieder eine ganze Menge Programmierer aus der großen Perl-Gemeinschaft mitgearbeitet: Von Bugmeldungen über Bugfixes und neuen Features bis hin zum Testen der einzelnen Entwicklerversionen. Viele Features werden von der Community erst ausführlich - bisweilen zu ausführlich - diskutiert.

In zwei Artikeln werden die Neuerungen in Perl 5.20 hier in dieser Ausgabe gezeigt.

Und auf dem CPAN finden sich rund 30.000 Distributionen - allerlei Bibliotheken, die von den Autoren für die Gemeinschaft bereitgestellt werden. Auch die Distributionen, die nicht überall genannt werden, sind extrem wichtig. Vielleicht helfen sie nur eins oder zwei anderen Programmierern/Programmiererinnen, aber auch dann hat es sich schon gelohnt die Distribution zu veröffentlichen.

Ende März fand in Hannover das große "Klassentreffen" der deutschsprachigen Perl-Gemeinschaft statt. Dort wurden interessante Vorträge zu Perl 6 und natürlich Perl 5 gehalten. Solche Treffen sind aber nicht nur wegen der Vorträge wichtig, sondern auch für den Kontakt und persönlichen Austausch mit vielen anderen Perl-Programmierern und Perl-Programmiererinnen.

Ansonsten gibt es auch in dieser Ausgabe wieder viele interessante Artikel, aber wir sind - wie immer - auf der Suche nach Themenvorschlägen und Autoren. Auch die abschließenden beiden Ausgaben wollen wir mit tollen Themen füllen. Aber jetzt erstmal viel Spaß mit der aktuellen Ausgabe.

Renée Bäcker

Die Codebeispiele können mit dem Code

omdbv2

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

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue30

IMPRESSUM

Herausgeber:	Perl-Services.de Renée Bäcker Bergfeldstr. 23 D - 64560 Riedstadt
Redaktion:	Renée Bäcker, Katrin Bäcker
Anzeigen:	Katrin 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

PERL

ANWENDUNGEN

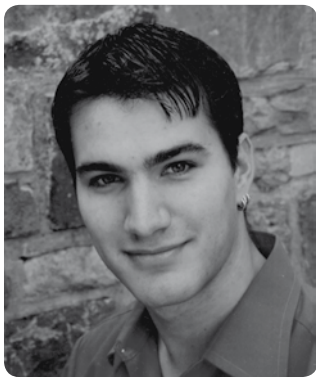
MODULE

NEWS

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

Renée Bäcker

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



Alexander Becker

Alexander Becker schreibt seit dem zweiten Millennium CGI-Skripten mit dem MVC-Framework CGI::Application (neuerdings auch Titanium genannt). Er arbeitet für eine IT-Firma in Luxemburg. Den Rest der Zeit vertreibt er sich als CPAN-Autor, dem Verdreschen von Beachvolleyballbällen oder als Betreiber von http://www.PerlTK.de.

Herbert Breuung

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.

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/

Wolfgang Kinkeldei

Wolfgang Kinkeldei arbeitet als Software-Entwickler bei einem mittelständischen Mediendienstleister in Nürnberg. Zu seinen Hauptaufgaben zählen die Automatisierung von Arbeitsabläufen in der Druckvorstufe sowie die Erstellung von Web-basierten Lösungen. Die meisten seiner Projekte werden mit Perl gelöst.

Renée Bäcker

Merkwürdigkeiten in Perl

Nach etwas längerer Pause hier mal wieder etwas aus dem Fundus "Kuriositäten mit Perl" - heute mit dem Thema "wie kann ich ein valides Perl-Programm erstellen obwohl ich das gar nicht will?". Auf Stackoverflow bin ich auf folgende Frage gestoßen:

Warum ist dieses Programm valide? Ich habe versucht einen Syntaxfehler zu erzeugen.

Der Benutzer wollte seinen git *pre-commit-hook* testen der Commits mit Syntaxfehlern verweigert. Also hat er versucht ein kleines Skript zu schreiben, das den Hook dazu bringt den Commit zu verweigern. Naja, schnell hingesetzt und fertig war das Testprogramm:

```
use strict;
use warnings;

Syntax error!

exit 0;
```

Schnell ein `commit` und dann passierte es - nichts! Der Commit ging problemlos durch. Auf den ersten Blick sieht das nicht nach einem validen Perl-Programm aus. Wenn man sich aber näher damit beschäftigt, sieht man was hier schiefläuft. Kommt jemand sofort drauf? Wer nicht auf die Lösung kommt - macht nichts. Hier kann das Modul `B::Deparse` helfen. Aber wir nähern uns dem Fall ohne gleich auf die Lösung zu schauen, und es werden nicht viele Schritte benötigt bis zur Lösung.

Wenn Perl keinen Fehler ausgibt, muss ja etwas ausgeführt werden. Schmeißen wir den Code am Anfang des Programms raus von dem wir wissen dass es valides Perl ist. Also das `use strict;` und das `use warnings;` weg. Damit bleibt

```
Syntax error!
exit 0;
```

Da Perl hiermit kein Problem hat, kommen zwei Fakten zu Hilfe: 1. Befehle werden mit einem Semikolon abgeschlossen und 2. Leerzeichen sind (meistens) ohne Bedeutung. Was übriggeblieben ist, ist also ein Befehl mit etwas zu viel Leerzeichen und Zeilenumbrüchen. Fassen wir es also zusammen:

```
Syntax error! exit 0;
```

Vielleicht kommt jetzt schon jemand drauf was am Ende die Ursache für diese "Merkwürdigkeit" ist. Wenn nicht, machen wir noch kleinen Schritt. Diesmal fügen wir ein Leerzeichen ein:

```
Syntax error ! exit 0;
```

Wer sich etwas mit Objektorientierung in Perl beschäftigt sieht jetzt vielleicht das Problem. Perl erlaubt verschiedene Schreibweisen für Methodenaufrufe. Die bekannteste Schreibweise dürfte

```
$object->methode( $param );
# bzw.
Klasse->methode( $param );
```

sein. Es gibt aber noch eine weitere Schreibweise, die man hin und wieder antrifft:

```
methode Klasse $param;
```

Das nennt man die indirekte Notation bzw. indirekter Methodenaufruf. Vergleichen wir das mit dem Befehl von oben:

```
methode => Syntax
Klasse    => error
$param    => !exit 0
```

Aber die Klasse *error* und die Methode `Syntax` existieren doch gar nicht. Warum meckert Perl das nicht an? Weil Perl gar nicht bis dahin kommt. Wenn wir den Befehl auf die bekanntere Schreibweise umschreiben, sieht das so aus:

```
error->Syntax( !exit 0 );
```

Die Parameter für die Methode werden zuerst "evaluiert" (also Methoden/Funktionen werden ausgeführt um an die Werte der Parameter zu kommen). Der erste Befehl der in dem oben gezeigten Programm ausgeführt wird ist das `exit 0`. Darum kann Perl keinen Fehler entdecken.

Schauen wir uns noch kurz die Ausgabe von B::Deparse an (Perl 5.14.2)

```
$ perl -MO=Deparse skript.pl
use warnings;
use strict 'refs';
'error'->Syntax(!exit(0));
-e syntax OK
```

Lustigerweise hätte der Test des Hooks geklappt wenn nur Kleinigkeiten anders gewesen wären: Wenn statt des `exit 0` ein `print "done"` gestanden hätten oder wenn statt des `!` ein Semikolon gestanden hätte.

Solche Ungereimtheiten können bei der indirekten Notation vorkommen. Deshalb kann man das Modul `indirect` einsetzen:

```
use strict;
use warnings;
no indirect;

Syntax error!

exit 0;
```

Wirft den Fehler

```
rect call of method "Syntax" on
  object "error" at -e line 4.
```

Renée Bäcker

Was ist neu in Perl 5.20?

Wie in den vergangenen Jahren gibt es auch in diesem Frühjahr eine neue Version von Perl 5. Mittlerweile gibt es schon die Version 20 und auch diese bringt einige Neuerungen, die in diesem Artikel beschrieben werden sollen. Auch wenn es das Perl-Magazin in Zukunft nicht mehr geben wird, wird es diese Art von Artikel jährlich geben: zu finden unter http://perl-academy.de

Im Gegensatz zu den letzten Versionen gibt es in Perl 5.20 eine große entscheidende Veränderung: Es gibt Subroutinen-Signaturen. In älteren Ausgaben von $foo wurden schon Module vorgestellt die Methoden-Signaturen einführen: Artikel "Methoden-Signaturen" in Ausgabe 12 und als Teil des Moose-Tutorials in Ausgabe 16.

Um die weiteren nützlichen Änderungen in Perl 5.20 dennoch ausreichend würdigen zu können, gibt es die Signaturen als eigenständigen Artikel direkt im Anschluss an diesen hier.

Widmen wir uns aber den Neuerungen in der neuen Perl-Version.

Postfix-Dereferenzierung

Das Dereferenzieren von Datenstrukturen ist nicht gerade eine Wohltat für die Augen. Manchmal erkennt man vor lauter Sigils und geschweiften Klammern gar nicht mehr, worum es eigentlich geht. Abhilfe - auch wenn das ebenfalls gewöhnungsbedürftig ist - verspricht die Postfix-Dereferenzierung.

Auch hier gehen die Perl 5 Porters den Weg, viele neue Features erst einmal als *experimentell* zu kennzeichnen, und das gilt auch für dieses Feature. Und es muss über das *feature* Pragma aktiviert werden:

```
use feature 'postderef';
no warnings 'experimental::postderef';
```

Danach kann zusätzlich zu dem altbewährten Dereferenzieren auch die Postfix-Dereferenzierung verwendet werden:

```
my @colors    = qw(blue yellow red);
my $color_ref = \@colors;

# bisheriges Derefenzieren
my @copy = @{$color_ref};

# Postfix-Dereferenzierung
my @copy = $color_ref->@*;
```

Statt dem vorangestellten Sigil und den -- in manchen Situationen überflüssigen - geschweiften Klammern wird der bekannte Pfeil -> verwendet und das passende Sigil gefolgt vom Asterisk. Dieses Schema wird auch bei den weiteren Dereferenzierungen eingesetzt:

```
$sref->$*;  # same as ${ $sref }
$aref->@*;  # same as @{ $aref }
$href->%*;  # same as %{ $href }
$cref->&*;  # same as &{ $cref }
$gref->**;  # same as *{ $gref }
```

Das funktioniert natürlich auch für komplexere Datenstrukturen:

```
my $data = {
    servers => {
        server1 => {
            ip => [
                '127.0.0.1',
                '192.168.2.161',
            ],
        },
    },
};

my @ips = $data->{servers}->{server1}
              ->{ip}->@*;
```

Damit erkennt man schneller als bei der altbewährten Methode, dass am Ende ein Array herauskommt. Der Vorteil bei der neuen Methode ist auch, dass der komplette Ausdruck von links nach rechts gelesen werden kann, ohne dass man das Sigil am Anfang im Hinterkopf behalten muss.

Möchte man auf einen Slot innerhalb einer *Globreferenz* zugreifen, wird der abschließende Asterisk durch den Zugriff auf den Slot ersetzt:

```
# same as *{ $gref }{ $slot }
$gref->*{ $slot };
```

Die Postfix-Dereferenzierung funktioniert auch bei Slices:

```
$aref->@[ ... ];  # same as @$aref[ ... ]
$href->@{ ... };  # same as @$href{ ... }
$aref->%[ ... ];  # same as %$aref[ ... ]
$href->%{ ... };  # same as %$href{ ... }
```

Hier ist deutlich schneller erkennbar, dass ein Slice einer Referenz erwünscht ist und nicht die Dereferenzierung eines Array- bzw. Hashelements.

Einige dieser Ausdrücke können auch interpoliert werden. Dazu muss aber das verwandte Feature *postderef_qq* aktiviert werden:

```
use feature qw/postderef postderef_qq/;

my $name_ref = \'$foo';
print "Sie lesen $name_ref->$*\n";
```

Folgende Ausdrücke können interpoliert werden:

```
$sref->$*
$aref->@*
$aref->@[ ... ]
$href->@{ ... }
```

Um an den letzten Index in einer Arrayreferenz zu kommen, braucht man nicht mehr

```
my $max_index = $#$aref;
```

sondern

```
my $max_index = $aref->$#*;
```

Performanz-Verbesserungen

Ein paar Perl 5 Porters haben sich auch der Verbesserung der Performanz verschrieben. In den nachfolgenden Absätzen ist zu sehen, dass es viele kleine Verbesserungen gibt, die für sich genommen nicht immer die großen Schritte machen, aber in der Summe kommt doch einiges zusammen.

Hash-Lookups bei Slices

Es ist schon länger so, dass zur Compile-Zeit der interne Hashwert für Hash-Lookups mit konstanten Schlüsseln (z.B. `$hash{schluessel}`) vorberechnet wurde. Damit wird der Hash-Lookup stark beschleunigt. Diese Optimierung wurde jetzt auch für Hash-Slices (z.B. `$hash{qw/schluessel1 schluessel2/}`) umgesetzt.

and- und or-Oparatoren im void-Kontext

Kombinierte `and`- und `or`-Operatoren im *void*-Kontext, wie z.B. bei `unless( $a && $b )` oder `if( $a || $b )`, kürzen jetzt ab, so dass bei `unless( $a && $b )` das `$b` erst gar nicht überprüft wenn `$a` schon unwahr ist.

Patternmatching mit Nicht-Unicode

Eine Performanzregression wurde in Perl 5.11.2 für Nicht-Unicde Patternmatching, bei denen die Groß-/Kleinschreibung keine Rolle spielt, eingebaut. Dabei wurde eine Optimierung für Zeichen aus dem ASCII-Bereich deaktiviert. Diese Optimierung ist in Perl 5.20 wieder aktiviert.

Copy-On-Write

Perl hat einen neuen Copy-On-Write-Mechanismus, der das Kopieren des internen Stringpuffers vermeidet wenn ein Skalar einem anderen zugewiesen wird. Das macht das Kopieren von großen Strings viel schneller. Wenn einer der beiden Skalare nach der Zuweisung verändert wird, wird das wirkliche Kopieren angestoßen. Durch diesen neuen Mechanismus ist es unnötig Strings aus Effizienzgründen als Referenz an z.B. Subroutinen zu übergeben.

Dieses Feature gab es schon in Perl 5.18.0, war aber standardmäßig deaktiviert. Das wurde jetzt geändert, so dass man Perl nicht mehr mit der *Configure*-Option

```
-Accflags=PERL_NEW_COPY_ON_WRITE
```

kompilieren muss.

Soll das Copy-On-Write beim Kompilieren deaktiviert werden, muss man

```
-Accflags=PERL_NO_COW
```

verwenden.

Hier ein kleines Beispiel, an dem man das Copy-On-Write (*COW*) erkennen kann:

```
$ perl -MDevel::Peek -e
    '$a="abc"; $b = $a; Dump $a; Dump $b'
SV = PV(0x260cd80) at 0x2620ad8
  REFCNT = 1
  FLAGS = (POK,IsCOW,pPOK)
  PV = 0x2619bc0 "abc"\0
  CUR = 3
  LEN = 16
  COW_REFCNT = 2
SV = PV(0x260ce30) at 0x2620b20
  REFCNT = 1
  FLAGS = (POK,IsCOW,pPOK)
  PV = 0x2619bc0 "abc"\0
  CUR = 3
  LEN = 16
  COW_REFCNT = 2
```

Man sieht, dass beide Skalare den gleichen *PV*-Puffer verwenden und der COW-Referenzzähler auf 1 steht. Bei den *FLAGS* ist auch vermerkt, dass für den Skalar "Copy-On-Write" aktiv ist. Ändert man z.B. `$b` durch ein

```
$b =~ s/a/u/g;
```

dann sehen die Dumps so aus:

```
SV = PV(0x260cd80) at 0x2620ad8
  REFCNT = 1
  FLAGS = (POK,IsCOW,pPOK)
  PV = 0x2619bc0 "abc"\0
  CUR = 3
  LEN = 16
  COW_REFCNT = 2
SV = PV(0x940cf90) at 0x941da60
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x9418ea0 "ubc"\0
  CUR = 3
  LEN = 16
```

Bei dem geänderten Skalar ist das Flag *IsCOW* verschwunden und der String wurde in einen anderen Puffer kopiert.

Allgemeine Verbesserungen und Änderungen

Es gibt etliche allgemeine Verbesserungen und Änderungen, die in den folgenden Abschnitten gezeigt werden.

readdir() und die Fehlervariable $!

Die Funktion `readdir()` setzt die Spezialvariable `$!` nur im Fehlerfall. Diese Variable wird beim abschließenden `undef` nicht mehr auf `EBADF` gesetzt, wenn der Systemaufruf nicht diese Variable setzt.

Zeichen matchen in Regulären Ausdrücken

Eine Regression seit Perl 5.18.0 Reguläre Ausdrücke betreffend wurde gefixt. In dem Fehler wurden Zeichen aus dem Bereich `\x80 - \xFF` nicht gematcht wenn in einer Zeichenklasse neben `[:^ascii:]` noch andere Zeichen waren.

Neues Attribut für Subroutinen

Subroutinen können jetzt mit dem `prototype`-Attribut versehen werden. Wenn die Subroutine definiert oder deklariert wird, kann der Prototyp innerhalb eines `prototype`-Attributs anstelle der runden Klammern nach dem Namen geschrieben werden. Aus

```
sub foo ($$) {
}
```

wird dann

```
sub :prototype($$) {
}
```

Warum das neue Attribut notwendig wurde, wird im Artikel über die Subroutinen-Signaturen erläutert.

Zufallszahlengenerator für rand

Bisher verwendete Perl einen plattformsspezifischen Zufallszahlengenerator, das war entweder `rand()`, `random()` oder `drand48()`. Das bedeutete, dass die Qualität der Zufallszahl in Perl von Plattform zu Plattform variiert. Das geht von den 15 Bits von `rand()` auf Windows bis zu 48 Bit von `drand48()` auf POSIX-Plattformen wie Linux.

Mit Perl 5.20 verwendet Perl auf allen Plattformen seine eigene Implementierung von `drand48()`, was Perls `rand()`-Funktion kryptographisch aber nicht sicher macht.

Bessere 64-bit Unterstützung

Auf 64-Bit-Plattformen benutzen die internen Array-Funktionen jetzt 64-Bit-Offsets, wodurch es möglich ist, dass Perl Arrays mehr als 2**31 Elemente haben können -- solange genug Speicher zur Verfügung steht.

Die Engine für die Regulären Ausdrücke unterstützt jetzt Strings die länger sind als 2**31 Zeichen.

Einige `PerlIO_*`-Funktionen haben jetzt Parameter und Returnwerte von der Größe `SSize_t` anstatt *int*.

Neue slice Syntax

Für Hashes und Arrays gibt es eine neue Form von Slices. Mit `%hash{...}` bzw. `%array[...]` liefert eine Liste von Schlüssel-Wert-Paaren bzw. bei Arrays Index-Wert-Paaren.

```
use Data::Dumper;

my @array = qw(perl magazin foo);
my %map   = %array[0,2];

print Dumper \%map;
__END__
$VAR1 = {
          '2' => 'foo',
          '0' => 'perl',
        };


use Data::Dumper;

my %hash = qw(
    sprache  => 'perl',
    magazin  => 'foo',
    webseite => 'cpan.org',
    modul    => 'Moose',
);
my %map  = %hash{qw/sprache magazin/};

print Dumper \%map;
__END__
$VAR1 = {
          'magazin' => 'foo',
          'sprache' => 'perl',
        };
```

Neue Warnung bei möglichen Vorrangsproblemen

Beim Programmieren fällt es nicht unbedingt sofort auf wenn man in die Fallen der Vorrangsregeln von Operatoren tappt. Für solche Fälle ist es praktisch und wichtig, wenn der Compiler eine Warnung ausgibt. Eine solche Warnung wurde jetzt eingeführt für Fälle in denen ein Operator wie `return` und einem Operator mit niedriger Vorrangsstufe wie `or` gemischt werden.

Schreibt man

```
sub foo {
    return $a or $b;
}
```

bekommt man die Warnung

```
Possible precedence issue with \
    control flow operator at ....pl
```

angezeigt. Perl parst den `return` Befehl als

```
sub foo {
    (return $a) or $b;
}
```

und dadurch wird das `or $b` nie ausgeführt. Die Zeile ist also effektiv ein reines `return $a;`. Durch die Warnung wird man darauf aufmerksam gemacht, dass man entweder runde Klammern benutzen sollte (`return ($a or $b)`) oder Operatoren mit hoher Priorität (`return $a || $b`).

Namen für Dateihandles

Ein lexikalischer Dateihandle (wie er in `open my $fh ...` vorkommt) hat üblicherweise einen Namen basierend auf dem aktuellen Package und dem Namen der Variablen, also z.B. `main::$fh`. Im Falle von Rekursion hat der Dateihandle den *$fh*-Teil verloren. Dieser Fehler wurde beseitigt.

Auslesen von freigegebenem Speicher

In den seltenen Fällen dass ein Perl-Programm mit einem *HEREDOC*-Teil aufhörte und die abschließende Zeile keinen Zeilenumbruch enthielt konnte es passieren, dass der freigegebene Speicher während des Parsens ausgelesen wurde. Dieses Verhalten wurde abgestellt.

Kommentar oder kein Kommentar

Seit Perl 5.001 wurde in Regulären Ausdrücken wie `/[#$a]/x` oder `/[#]$a/x` das `#`-Zeichen als Kommentar betrachtet. Dabei sollte in Zeichenklassen das Zeichen seine besondere Bedeutung verlieren. Durch diesen Fehler wurde die Variable nie interpoliert. Dieser Fehler ist jetzt endlich behoben.

Nützliche Infos und Dokumentation

Das Debugging mit `gdb` ist für unerfahrene Programmierer nicht so einfach. In der *perlhacktips*-Dokumentation wurden einige zusätzliche Beispiele für die Verwendung von `gdb` hinzugefügt.

Kernmodule

Die Perl5-Porters tun sich nicht leicht damit, langjährige Mitglieder der Kernmodule wieder aus dem Kern zu entfernen und neue Mitglieder aufzunehmen. Manchmal ist es aber dennoch notwendig und auch mit Perl 5.20 gibt es da etwas Bewegung. Einige Module wurden entfernt, zu den Wichtigsten gehören `CPANPLUS`, `Log::Message` und `Archive::Extract`.

Zu den Modulen, die in (nicht ganz so naher Zukunft) aus dem Kern entfernt werden, gehört jetzt auch `CGI.pm`. Wer Webanwendungen schreibt und dabei das Modul verwendet, sollte es auf jeden Fall in die Liste der Abhängigkeiten eintragen. Bis so eine Ankündigung dass ein Modul entfernt wird in die Tat umgesetzt wird, vergehen üblicherweise viele Jahre, also kein Grund zur Hektik.

Neu hinzugekommen ist neben ein paar "kleineren" Modulen auch `IO::Socket::IP`.

Mit `IO::Socket::IP` bekommt der Perl-Kern ein Modul mit dem unabhängig von Protokollen mit IPv4 und IPv6 gearbeitet werden kann. Das Modul ist als Ersatz für das weitverbreitete `IO::Socket::INET` gedacht. Ein paar Inkompatibilitäten gibt es, die in der Dokumentation von `IO::Socket::IP` aufgelistet sind. In den meisten Fällen sollte ein `s/IO::Socket::INET/IO::Socket::IP/g` reichen um seine Anwendungen IPv6-fähig zu machen.

Die ganzen Updates werden hier nicht im Detail dargestellt, da alleine die Auflistung der aktualisierten Module mehrere Seiten in Anspruch nehmen würde. Eine Aktualisierung aber doch ins Auge gesprungen: Die Ausgabe von `Data::Dumper` hat sich - je nach Daten und Einstellungen - geändert. Wer also auf die "Exaktheit" der Ausgabe vertraut, sollte sich den Code nochmal anschauen.

Mit der Version 1.35 von `List::Util` kommen einige nützliche Funktionen in den Kern: `any`, `all`, `none`, `notall` und `product` wurden neu hinzugefügt und die Funktionen `reduce` und `first` werden implementiert auch wenn *MULTICALL* nicht gesetzt ist.

Die hier gezeigten Änderungen sind nur ein Teil dessen was sich in der neuen Perl-Version alles getan hat. Es zeigt auch, dass die Perl-Entwicklung nicht stehen bleibt. Gerade für größere Features wäre es gut, wenn möglichst viele Programmiererinnen sich auch mal Entwicklerversionen von Perl installieren und die neuen Features ausprobieren. Dank *perlbrew* bzw. *plenv* ist das kein Problem.

Mehr Informationen zu den ganzen Neuerungen finden sich in der *perldelta*-Dokumentation.

Renée Bäcker

Subroutinen-Signaturen

Das Thema Subroutinen-Signaturen für den Perl-Kern liegt schon länger auf dem Tablett und Peter Martini hat viel Zeit damit verbracht, verschiedene Wege auszuprobieren. Er hat immer wieder bei den Perl 5 Porters Dokumentation und Code vorgebracht, die ausführlich diskutiert wurden. Dabei wurden auch immer wieder Grenzfälle und Schwächen der Implementierung deutlich. Durch die verschiedenen Entwicklungsstadien wurde das Feature immer ausgereifter. Und mit Perl 5.20 gibt es die Subroutinen-Signaturen auch im Perl-Kern. Da es ein großes Thema ist, wurde es aus dem Artikel "Neues in Perl 5.20" herausgezogen und als eigener Artikel geschrieben.

Obwohl lange an dem Feature herumgefeilt und diskutiert wurde, wird es dennoch zunächst als experimentell gekennzeichnet. Damit sollen die Probleme vermieden werden, wie sie beim Smartmatch aufgetaucht sind. Zusätzlich muss das Feature mit `use feature 'signatures';` aktiviert werden.

Da es ein experimentelles Feature ist, wird bei der Verwendung eine Warnung ausgegeben, die mittels `no warnings "experimental::signatures"` ausgeschaltet werden kann.

Um uns dem Feature zu nähern, nehmen wir den in Listing 1 dargestellten Perl-Code als Basis:

Das Programm kann dann so aufgerufen werden, wie in Listing 2 gezeigt.

Es kann also das Jahr übergeben werden für das man die Statistiken der Teilnehmer sehen möchte. Diese Angabe bekommt auch die Subroutine übergeben, genauso wie den "UserAgent".

```
use strict;
use warnings;

use feature qw/say/;

use Time::Piece;
use Mojo::UserAgent;
use HTML::TableExtract qw(tree);

my $time = localtime;
my $ua   = Mojo::UserAgent->new;

get_stats( $ua, @ARGV );

sub get_stats {
    my ($ua, $year) = @_;

    my $tx = $ua->get(
        'http://act.yapc.eu/gpw' .
        $year .
        '/stats'
    );
    my $body = $tx->res->body;

    my $xtract = HTML::TableExtract->new(
        depth     => 1,
        subtables => 1,
    );

    $xtract->parse( $body );
    my @tables = $xtract->tables;
    my $table  = $tables[0]->tree;
    say $table->as_text;
}
```

Listing 1

```
$ perl workshop.pl 2013
Deutschland 128 (115)
Niederlande 6 (6)
Österreich 4 (3)
# noch mehr länder
$ perl workshop.pl 2014
Deutschland 85 (65)
Österreich 5 (2)
Niederlande 4 (4)
# noch mehr länder
```

Listing 2

Das Skript von oben wird jetzt nach und nach an die Möglichkeiten des neuen Features angepasst. Betrachten wir uns die erste Subroutine:

```
sub get_stats {
    my ($ua, $year) = @_;

    my $tx = $ua->get(
        'http://act.yapc.eu/gpw' .
        $year .
        '/stats'
    );
    my $body = $tx->res->body;

    # code zum extrahieren der tabellen
}
```

Der erste Schritt ist, das Feature über das `feature`-Pragma zu aktivieren:

```
use feature qw/signatures/;
```

Um anderen Entwicklern schon beim Betrachten des Perl-Codes klarzumachen, dass ein sehr neues Perl gebraucht wird, kann man noch ein `use 5.019009;` einsetzen. Ohne diesen Zusatz bekommt man bei der Ausführung des Programms gleich eine Fehlermeldung `Feature "signatures" is not supported by Perl 5.18.1`.

Dann kann es aber losgehen. Die wichtigste Änderung ist erstmal von

```
sub get_stats {
    my ($ua, $year) = @_;
}
```

nach

```
sub get_stats ($ua, $year) {
}
```

Schön, nicht? Der Rest kann bleiben und das Programm funktioniert bei den Aufrufen wie oben auch noch genauso. Nur eine Warnung bekommt man jetzt zu sehen - weil die Signaturen noch als experimentell gekennzeichnet sind. Schon in Ausgabe 27 von $foo wurde genauer auf solche Warnungen eingegangen. Deshalb hier nur die eine Zeile, die diese Warnung deaktiviert:

```
no warnings 'experimental::signatures';
```

Jetzt soll das Programm aber noch etwas angepasst werden. Wenn man die Jahreszahl weglässt, sie also nicht an die Subroutine übergeben wird, soll immer die Statistik zum aktuellen Jahr ausgegeben werden. Dann kann ein Standardwert verwendet werden:

```
use Time::Piece;

my $time = localtime;
sub get_stats ($ua, $year = $time->year) {
}
```

Wunderbar, jetzt funktioniert

```
$ perl workshop.pl
```

genauso wie

```
$ perl workshop.pl 2014
```

Achtung: Die Sache mit dem Standardwert hat nur einen Haken: Dieser wird nur dann gesetzt, wenn der Parameter nicht übergeben wird. Sobald ein *undef* übergeben wird, wird der Standardwert nicht gesetzt.

Ein

```
my $ua   = Mojo::UserAgent->new;
my $year = shift @ARGV;

get_stats( $ua, $year );
```

wird also bei dem Aufruf

```
$ perl workshop.pl
```

nicht das gewünschte Ergebnis liefern. Das ist einer der Punkte, bei denen ich noch Nachbesserungsbedarf sehe.

Gibt der Benutzer neben der Jahreszahl noch etwas anderes ein, so trifft man auf ein weiteres Umsetzungsdetail der Signaturen:

```
$ perl workshop.pl 2014 nochwas
Too many arguments for subroutine at \
    workshop.pl line 18.
```

Und ohne den Standardwert liefert auch ein

```
$ perl workshop.pl
```

einen Fehler:

```
Too few arguments for subroutine at \
    workshop.pl line 18.
```

Die Subroutine erwartet also genauso viele Parameter wie in der Signatur angegeben sind. Möchte man flexibel bleiben, muss man *slurpy* Parameter verwenden, also solche, die den ganzen Rest noch aufnehmen. Das kann entweder ein Array oder ein Hash sein:

```
sub get_stats ($ua, $year, @rest) {
}
```

Jetzt kann man auch

```
get_stats($ua, 2014, 'noch', 'weiteres');
```

aufrufen und man bekommt das richtige Ergebnis. Da hier dieser *slurpy* Parameter aber nur dazu verwendet wird, dass es keine Fehlermeldung gibt wenn man zu viele Parameter an die Subroutine übergibt und die Werte daraus nicht gebraucht werden, braucht man auch keinen Namen anzugeben:

```
sub get_stats ($ua, $year, @) {
}
```

Das Weglassen des Namens funktioniert auch mit allen anderen Parametern. Wenn z.B. immer drei Parameter übergeben werden, der zweite aber nicht gebraucht wird, dann kann man

```
sub my_function ( $first, $, $third ) {
}
```

schreiben.

Innerhalb der Signatur kann man für die Standardwerte auch einen Bezug auf vorher genannte Parameter herstellen.

```
sub user ($first, $last, $nick = $first) {
}
```

Wenn hier kein Nickname übergeben wird, wird automatisch der Vorname als Nickname verwendet.

Noch ein Wort zu den *slurpy* Parametern: Für diese Parameter kann man keine Standardwerte festlegen. Ein

```
sub get_stats ( @params = qw(1 2 3) ) {
}
```

funktioniert also nicht.

Der geänderte Code ist in Listing 3 zu sehen.

Die in diesem Artikel gezeigten Möglichkeiten der Signaturen funktionieren auch bei anonymen Subroutinen:

```
my $get_stats =
    sub ($ua, $year = $time->year ){
        # code
    };
```

Soweit zu den Möglichkeiten der Signaturen. Ein Problem entsteht aber durch die Signaturen: Es gab schon eine Verwendung für runde Klammern nach dem Subroutinennamen. Es war etwas, was gerade Perl-Einsteiger häufig mit Signaturen verwechselten – die Prototypen. In Ausgabe 24 von $foo wurden die Subroutinenprototypen ausführlich behandelt, deswegen an dieser Stelle nur ein kleines Beispiel:

```
sub hello ($) {
  my $name = shift;
  print "hello $name\n";
}
```

Hier ist das `($)` keine Signatur, sondern ein Prototyp der besagt, dass der Parameter an *hello* als Skalar behandelt wird. Sobald man `use feature 'signatures';` verwendet, wird das aber als Signatur verstanden und es ist in diesem speziellen Fall auch eine gültige Signatur: Es muss ein Parameter übergeben werden, der aber nicht gebraucht wird. Der Code funktioniert auch weiterhin. `@_` wird durch die Signaturen nicht angefasst. Aber dennoch ist es verwirrend.

Wer weiterhin mit Prototypen arbeiten möchte, kann das auch tun. Für Entwickler/innen, die sich damit auskennen,

```
use strict;
use warnings;

use feature qw/signatures say/;
no warnings 'experimental::signatures';

use Time::Piece;
use Mojo::UserAgent;
use HTML::TableExtract qw(tree);

my $time = localtime;
my $ua   = Mojo::UserAgent->new;

get_stats( $ua, @ARGV );

sub get_stats ($ua, $year = $time->year ){
    my $tx = $ua->get(
        'http://act.yapc.eu/gpw' .
        $year .
        '/stats'
    );
    my $body = $tx->res->body;

    my $xtract = HTML::TableExtract->new(
        depth     => 1,
        subtables => 1,
    );

    $xtract->parse( $body );
    my @tables = $xtract->tables;
    my $table  = $tables[0]->tree;
    say $table->as_text;
}
```

Listing 3

können Prototypen ganz nützlich sein. In so einem Fall muss mit dem *prototype* Attribut gearbeitet werden:

```
sub hello :prototype($) ($name) {
    print "hello $name\n";
}
```

Das *prototype* Attribut muss – genauso auch andere Attribute – vor der Signatur stehen.

Dieser Artikel hat gezeigt, dass die Subroutinen-Signaturen für positionale Parameter sehr nützlich und praktisch sind. In der weiteren Entwicklung sollte es noch um benamte Parameter und ein paar Verbesserungen gehen, dann wäre das Feature komplett.

Alexander Becker

Komodo mit PerlTidy

Komodo IDE, eine Entwicklungsumgebung von ActiveState, bringt nützliche Werkzeuge zur Quellcodeverschönerung mit. Für Perl ist leider von Haus aus noch kein Formatierer dabei. Glücklicherweise erlaubt es Komodo neue, eigene Formatierer per Kommandozeile zu verwenden.

Hier bietet sich perltidy an: ein Perl-Skript zum Einrücken und Formatieren von anderen Perl-Skripten. Das erhöht die Lesbarkeit: fremde Skripte können beispielsweise in das eigene, präferierte Format übertragen werden, oder der eigene Programmierstil weicht den Anforderungen an ein gemeinschaftliches Code-Repository. Wie der Quellcode genau formatiert wird, kann konfiguriert werden.

Zur Verwendung in Komodo bietet es sich an, einfach das Modul Perl::Tidy zu installieren. Das Modul liefert praktischerweise gleich das Skript perltidy mit. Unter Verwendung von ActiveState-Perl auf Windows wird das Programm standardmäßig hier abgelegt: *C:\Perl\site\bin\perltidy.bat*

Um den Perl-Quellcodeverschönerer in Komodo zu verwenden sind folgende Schritte erforderlich:

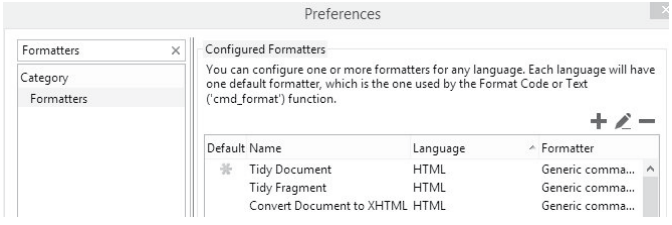


Bild 1: Kategorie Formatters auswählen

1. In den Komodo-Einstellungen wird die Kategorie Formatters aus der Liste der Einstellungen gewählt (siehe Bild 1).

2. Dort ermöglicht ein Klick auf das Plus-Symbol die Anlage eines neuen Formatierers (siehe Bild 2).

3 .Über die sogenannte Formatter Configuration wird Komodo zuerst mitgeteilt, dass ein generischer Formatierer für die Programmiersprache Perl konfiguriert werden soll. Der Name für diesen Formatierer kann frei gewählt werden (z.B. PerlTidy) - siehe Bild 3.

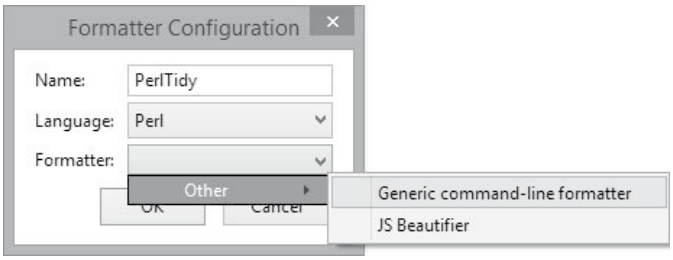


Bild 2: Neuen Formatierer anlegen

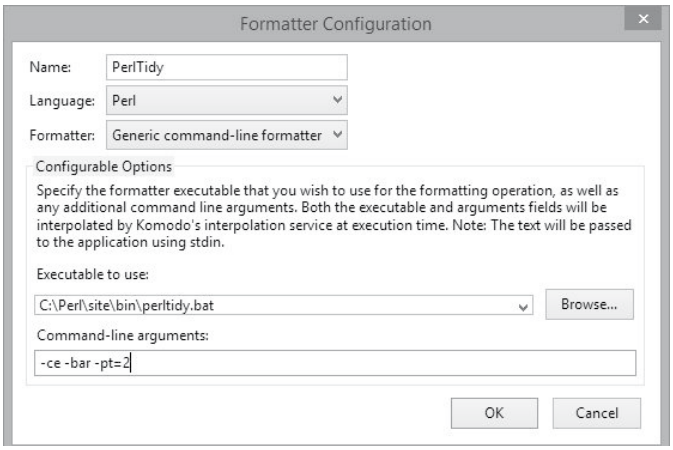


Bild 3: Konfiguration des Formatierers

Ulli Horlacher

vv : visual versioning

Emacs und kompatible Editoren erstellen beim Abspeichern automatisch ein Backup, wo der Dateiname am Ende ein ~ angefügt bekommt.

Beispiel: file.txt --> file.txt~

Oft reicht aber nur eine (letzte) Backup-Version nicht aus und außerdem wird so das aktuelle Verzeichnis mit den ganzen ~-Backups vollgekleistert, was sich bei Kommandos wie ls oder grep unangenehm auswirkt.

Beide Nachteile umgeht das Perl-Programm vv (visual versioning), das bis zu 10 Backup-Versionen in einem eigenen .versions/ subdirectory ablegt und dazu weitere Operationen anbietet:

```
usage: vv [-l] [file]
       vv -r version-number file [new-file]
       vv -d version-number file
       vv -v version-number file
       vv -s file
       vv -e file
       vv -p
       vv -m
options: -l   list available versions
         -r   recover file
         -d   show diff
         -v   view version
         -s   save file to new version
         -e   edit file with
                 $EDITOR (with versioning)
         -p   purge orphaned versions
                 (without current file)
         -m   migrate backup files
                 to version files
```

Klassische Versionskontrollsysteme wie RCS, SCCS, subversion oder Git muss man explizit aufrufen, bei vv passiert das implizit durch den Editor.

Deshalb ist vv ideal für den typischen Perl-Hacker, der sich nicht mit einem umständlichen Versionskontrollsystem herumschlagen will, aber trotzdem gerne auf die vor-vor-vorletzte Version seines Sourcecodes zurückgreifen möchte, wenn er sich mal wieder verprogrammiert hat.

vv kann in jeden Editor integriert werden, der via Kommandozeile gestartet wird. Backup-Versionen werden beim Abspeichern automatisch und unsichtbar angelegt. Nur im Notfall greift man dann durch expliziten Aufruf von vv drauf zu. Beispiel in Listing 1.

Hmmm... war doch nicht so gut, ich hätte doch gerne wieder die Version, die ich grad überschrieben habe... kein Problem: vv legt nämlich vor dem recovery ein Null-Backup an, das man sich wieder holen kann!

```
framstag@fex:/sw/share/fstools-0.0/bin:
                                  vv -r 0 fexsend
./.versions/fexsend~0~ -> fexsend
```

Für vim sieht die vv-Integration z.B. so aus (im .vimrc):

```
autocmd BufWritePre
  * execute '! vv -s ' . shellescape(@%)
autocmd BufWritePost
  * execute '! vv -b ' . shellescape(@%)
```

Falls dieses Backup-Schema jemandem bekannt vorkommen sollte: VMS RMS macht das so, allerdings für ALLE Dateien und nicht bloß für die editierten.

vv sollte auf jedem UNIX System laufen, getestet habe ich es allerdings nur auf Linux, BSD und Solaris. vv benötigt nur Perl Core Module und die externen Programme rsync und diff.

```
framstag@fex:/sw/share/fstools-0.0/bin: vv fexsend
version  bytes        date time
.        68500  2014-04-19 20:24:06
1        68500  2014-04-15 15:44:09
2        68500  2014-04-10 09:37:52
3        68464  2014-04-10 09:28:22
4        68308  2014-04-10 09:17:37
5        68226  2014-04-10 09:11:12
6        68086  2014-03-28 09:33:08
7        68068  2014-03-28 09:31:32
8        68081  2014-03-28 09:30:31
9        68068  2014-03-28 09:29:42

framstag@fex:/sw/share/fstools-0.0/bin: vv -d 2 fexsend
--- ./.versions/fexsend~1~      2014-04-15 15:44:09.000000000 +0200
+++ fexsend     2014-04-19 20:24:06.344993000 +0200
@@ -2415,7 +2415,7 @@
   $xx =~ s:.*/::;
   $url = "$proxy_prefix/fop/$from/$from/$xx?ID=$id";

-  sendheader("$server:$port","GET $url HTTP/1.1","User-Agent: $useragent");
+  sendheader("$server:$port","GET $url HTTP/1.0","User-Agent: $useragent");
   http_response();
   while (<$SH>) {
     s/\r//;

framstag@fex:/sw/share/fstools-0.0/bin: vv -r 2 fexsend
fexsend -> ./.versions/fexsend~0~
./.versions/fexsend~2~ -> fexsend
```

Listing 1

vv umfasst (ohne Doku) 400 Zeilen Code und ist "klassisch" programmiert: d.h. ohne Zusatzmodule und ohne Objektorientierung. Deshalb ist es sofort auf jedem UNIX lauffähig und kann auch ohne große Programmiererfahrung verändert werden.

Berücksichtigt wurde "Perl Best Practices" von Damian Conway.

http://fex.rus.uni-stuttgart.de/fstools/vv.html

Wolfgang Kindeldei

App::Fatpack

Warum Fatpack?

Gelegentlich ergeben sich Notwendigkeiten, in Perl entwickelte Programme auf unterschiedlichen Systemen zur Verfügung haben zu müssen. Nicht immer ist ein Zugriff auf CPAN oder firmeneigene Äquivalente möglich, Benutzerrechte sind unzureichend, vorhandene Perl-Module entsprechen nicht den Wünschen oder es fehlen C-Compiler und Co. Kurzum, es muss ein Weg gefunden werden, unsere Programme einfach und schnell zu installieren.

Die einfachste zu installierende Variante wäre es, nur auf CORE Module zurück zu greifen und sämtlichen ausführbaren Code in eine einzige Datei zu packen. Doch damit erkauft man sich schnell Spagetti-Code oder Wartungs-Höllen, es muss also nach Alternativen gesucht werden.

Ein möglicher Weg ist der Einsatz von `App::Fatpacker`. Alternativ kann man auch `PAR::Packer` einsetzen.

Technischer Hintergrund

Das Endresultat eines Pack-Vorganges ist eine ausführbare Datei, in der neben dem eigentlichen Script auch sämtliche (rein in Perl geschriebenen) Module mit enthalten sind. Grund genug zunächst einmal zu untersuchen, welche Mechanismen im Hintergrund dafür notwendig sind, damit so etwas überhaupt funktioniert.

Es muss sichergestellt sein, dass das Laden von Modulen via `use` oder `require` funktionieren. Dazu verrät die Manpage `perldoc -f require` den Trick: das `@INC` Array darf auch Code-Referenzen oder Objekte enthalten, um einen gezielten Eingriff in den Ladevorgang von Modulen vornehmen zu können.

Eine einfache Lade-Routine könnte demnach so aussehen:

```
BEGIN {
    # Definition aller gepackten Module
    my %fatpacked = (
        'My/X.pm' =>
              'package My::X; our $x = 42;'
    );

    # Lade-Routine
    unshift @INC, sub {
        my ($this_subref, $filename) = @_;

        my $source_code =
               $fatpacked{$filename}
            or return;
        open my $fh, '<', \$source_code;
        return $fh;
    };
}
```

Neben diesem Trick findet auch das Idiom der im Speicher gehaltenen Dateien Verwendung, das in `perldoc -f open` beschrieben ist. Verwendet man anstelle eines Dateinamens eine Skalare Referenz, so wird der Inhalt der skalaren Variable wie eine Datei behandelt und kann über die üblichen Datei-Operationen behandelt werden. Damit fällt die eigentliche Lade-Routine relativ kurz aus und das Haupt-Augenmerk liegt darauf, sämtliche notwendigen Module zusammenzutragen und in den Hash zu packen.

Einschränkungen

Solange Module über `require` oder `use` geladen werden und es sich dabei um reine Perl Module handelt, ist alles in Ordnung. Sogar ein `do 'My/X.pm'` wäre in obigem Falle ausführbar.

Wehe aber, wenn versucht werden sollte, Plugin-Mechanismen einzusetzen, die Verzeichnisse traversieren, ergeben sich Probleme. Damit sind Module wie *Module::Pluggable* leider nicht einsetzbar. Der Grund liegt darin, dass ja eben

keine Dateien für die ladbaren Kandidaten der Module vorhanden sind.

Ebenfalls draußen bleiben müssen in C geschriebene Module, da zum einen die erzeugten Bibliotheken plattformabhängig sind und zum Anderen der Linker bei der Ausführung nichts mit Daten aus einem Perl Hash anfangen kann.

Ebenfalls draußen ist, wer mit *Module::ShareDir* oder vergleichbaren arbeiten mag. Auch hier ist der Grund derselbe: es gibt keine Dateien.

Beispiel Anwendung

Um zu sehen, was alles beim Packen passiert, wollen wir eine einfache Anwendung stricken. Wir nutzen *Moo* als Objektsystem und *MooX::Options* zur Handhabung von Kommandozeilen-Argumenten und packen die eigentliche Ablauf-Logik in eine zusätzliche Klasse.

Unser Script ist dann relativ einfach:

```
#!/usr/bin/env perl
use MyApp;
MyApp->new_with_options->run;
```

In der Applikation `MyApp` werden lediglich Kommandozeilen Argumente entgegen genommen und dann weiter delegiert.

```
package MyApp;
use Moo;
use MooX::Options;
use MyApp::Worker;

option units => (
    is       => 'ro',
    required => 1,
    format   => 'i',
    short    => 'u',
    doc      => 'Select no of units to '
        . 'work [required]',
);

has worker => (
    is => 'lazy',
);

sub _build_worker { MyApp::Worker->new }

sub run {
    my $self = shift;

    $self->worker->work($self->units);
}

1;
```

Und ganz primitiv fällt der Umfang unserer Arbeits-Klasse aus:

```
package MyApp::Worker;
use 5.010;
use Moo;

sub work {
    my ($self, $units) = @_;

    say "You chose $units units to work";
}

1;
```

Packen mit App::FatPacker

Leider sind vier Schritte notwendig, um aus dem gegebenen Script eine gepackte Variante zu erstellen, aber diese Schritte sind exakt in dieser Reihenfolge in der Dokumentation von *App::FatPacker* beschrieben.

Der erste Aufruf startet das Script und fängt dabei sämtliche Ladevorgänge ab. Jedes hierbei geladene Modul wird gemerkt und als jeweils eine Zeile in der Datei `fatpacker.trace` im aktuellen Verzeichnis mitgeschrieben.

```
$ fatpack trace bin/my_app.pl
```

Anhand der geladenen Module wird nun versucht, die durch die Installation mit einem CPAN Client entstandenen `.packlist` Dateien zu finden. Das ist notwendig, damit anschließend sämtliche in der zu Grunde liegenden Distribution enthaltenen Dateien gefunden werden können. Das Ergebnis ist eine zweite Arbeits-Datei, in der die Pfade sämtlicher `.packlist` Dateien enthalten sind.

```
$ fatpack packlists-for
  `cat fatpacker.trace` >packlists
```

Nun werden sämtliche `.packlist` Dateien durchforstet und alle darin enthaltenen Dateien zusammengetragen. Dazu wird das Verzeichnis `fatlib` angelegt und in der notwendigen Hierarchie befüllt.

```
$ fatpack tree `cat packlists`
```

Zum Abschluss werden die Shebang-Zeile des Scripts, die Inhalte sämtlicher `.pm` Dateien aus dem Verzeichnis `fatlib` sowie der Rest des Scripts zu einer Datei zusammengefasst. Warnungen erscheinen für alle in den Pack-Listen enthaltenen Dateien, die nicht die Datei-Endung `.pm` tragen und können gerne ignoriert werden.

```
$ fatpack file myscript.pl >packed.pl
```

Voilá! Wir haben eine (leider ca. 1MiB große) Datei, in der alles enthalten ist, das wir brauchen. Und das beste: sie funktioniert sogar.

```
$ ./packed.pl -u 42
You chose 42 units to work
```

Vereinfachung

Wer so etwas häufiger benötigt, wird die vier Schritte vermutlich in einem `Makefile` unterbringen oder für die ganz faulen wie mich durch das *Dist::Zilla* Plugin *Dist::Zilla::Plugin::FatPacker*. Dadurch wird die Benutzung besonders einfach, lediglich eine geeignete `dist.ini` Datei ist notwendig. In diesem Fall werden sämtliche im `lib` Verzeichnis gespeicherten Module nicht mit in die entstehende Distribution gepackt, da uns nur das ausführbare Binärprogramm interessiert.

```
name             = MyApp
version          = 0.01
author           = Wolfgang Kinkeldei
                   <wolfgang@kinkeldei.de>
license          = Perl_5
copyright_holder = Wolfgang Kinkeldei
copyright_year   = 2014

[@Basic]

[PruneFiles]
match = ^lib/

[FatPacker]
script = bin/my_app.pl
```

Gepackt wird dann ganz einfach:

```
$ dzil build
```

Nun kann bequem die entstandene `.tar.gz` Datei verteilt und am Zielort entpackt und das Script ohne weitere Aktivitäten genutzt werden.

Renée Bäcker

Schneller arbeiten mit MCE

OPAR ist eine Art CPAN für OTRS-Erweiterungen. Dort kann jeder seine eigenen Erweiterungen der Allgemeinheit zur Verfügung stellen. Bevor die Erweiterung jedoch von jedem gefunden werden kann, müssen noch ein paar Aufgaben erledigt werden. Einer dieser Aufgaben ist es, das Paket zu analysieren. Zu dieser Analyse gehören folgende Punkte

- Validität und Wohlgeformtheit des XML
- Überprüfung der Programmierrichtlinien (Perl::Critic)
- Code-Layout (Perl::Tidy)
- Vorhandensein von Dokumentation und Tests
- Abhängigkeiten
- Lizenz

Der Aufwand dafür ist relativ groß. Dazu muss man sich anschauen was so alles dahinter steckt, beginnen wir also mit der Erweiterung an sich. OTRS-Erweiterungen (OPM-Dateien) sind einfache XML-Dateien mit einem bestimmten Aufbau. Der Aufbau und die Metadaten, die im XML stecken, interessieren hier an dieser Stelle nicht. Aber in diesem XML stecken auch alle Dateien, die im OTRS installiert werden.

```
<?xml version="1.0" encoding="utf-8"?>
 <otrs_package version="1.0">
 <Name>TicketAttachments</Name>
 <Version>1.0.5</Version>
 <Framework>3.0.x</Framework>
 <Framework>3.3.x</Framework>
 <Vendor>Perl-Services.de</Vendor>
 <URL>http://www.perl-services.de/</URL>
 <Filelist>
  <File Permission="644"
   Location="doc/en/TicketAttachments.pod"
   Encode="Base64">PWhlYWQxIE5BTUUKClRpY2t
   ldEF0dGFjaG1lbnRzIC0gaGFuZGxlIGF0dGFjaG
   1lbnRzIG9mIGEgdGlja2V0Cgo9aGVhZDEgREVTQ
   [...]1JJUFRJT04KClRoaXMgbW9kdWxlIHNob3d
  </File>
  <File Permission="644"
   Location="Kernel/Config/Files/Att.xml"
   Encode="Base64">PD94bWwgdmVyc2lvbj0iMS4
   IiBlbmNvZGluZz0iaXNvLTg4NTktMSI/Pgo8b3R
   [...]yc19jb25maWcg
  </File>
 </Filelist>
</otrs_package>
```

Listing 1

Ein kleiner (gekürzter) Ausschnitt aus einer solchen OPM-Datei finden Sie in Listing 1.

Für die Tests von Code-Layout und Einhaltung der Programmierrichtlinien müssen also diese Dateien erst Base64-dekodiert werden, danach müssen die ganzen Regeln von `Perl::Critic::OTRS` darüber laufen. Zum Abschluss noch Perltidy auf diese Dateien angewendet werden. Da demnächst noch viele Code-Überprüfungen dazukommen sollen, wird die Laufzeit immer wichtiger - auch weil es für die Analyse der Pakete nur eine eingeschränkte Anzahl von Jobs gibt. Der Autor des Pakets soll aber nicht "ewig" auf die Ergebnisse warten.

In Listing 2 ist die aktuelle Umsetzung zu sehen. Dort wird die OPM-Datei geparst und die Liste der Dateien herausgezogen. Jede der gefundenen Dateien wird dekodiert und im Objekt gespeichert. Auf jede dieser Dateien werden dann sämtliche Analyse-Rollen losgelassen. Und genau dieser Schritt soll parallelisiert werden.

Es muss also etwas an der Performanz verbessert werden. Der erste Gedanke waren Threads, aber wer schon mal mit Threads in Perl gearbeitet hat, weiß, dass es weitaus angenehmere Themen bei Perl gibt. Außerdem ist nicht überall ein Threaded-Perl installiert. Bei der Recherche nach Möglichkeiten, bin ich dann auf `MCE` gestoßen. MCE steht für Multi-Core Engine und die Dokumentation des Moduls verspricht die optimale Ausnutzung der Cores einer CPU. Und damit soll die parallele Abarbeitung der Dateien ermöglicht werden.

In diesem Artikel soll MCE vorgestellt werden und die Analyse der Dateien beschleunigt werden.

```
my $parser = XML::LibXML->new;
my $tree   = $parser->parse_file( $self->opm_file );
my $root   = $tree->getDocumentElement;

# retrieve file information
my @files = $root->findnodes( 'Filelist/File' );

FILE:
for my $file ( @files ) {
    my $name = $file->findvalue( '@Location' );

    #next FILE if $name !~ m{ \. (?:pl|pm|pod|t) \z }xms;
    my $encode         = $file->findvalue( '@Encode' );
    next FILE if $encode ne 'Base64';

    my $content_base64 = $file->textContent;
    my $content        = MIME::Base64::decode( $content_base64 );

    # push file info to attribute
    $self->add_file({
        filename => $name,
        content  => $content,
    });
}

[ ...]

# do all the checks that are based on the content of files
my %roles   = $self->roles;

for my $file ( $self->files ) {

    ROLE:
    for my $role ( @{ $roles{file} || [] } ) {
        my ($sub) = $self->can( 'analyze_' . lc $role );
        next ROLE if !$sub;

        my $result   = $self->$sub( $file );
        my $filename = $file->{filename};

        $analysis_data{$role}->{$filename} = $result;
    }
}
```

Listing 2

Einführung in MCE

MCE verspricht ja, die Cores der CPU(s) voll auszunutzen. Also mal schnell einen Blick darauf werfen, wie viele Cores denn hier als Beispiel genutzt werden können. Dafür kann man mal schnell `Sys::Statistics::Linux` verwenden - siehe Listing 3.

Es stehen also vier Cores zur Verfügung, die voll genutzt werden wollen. Schauen wir uns also mal an, was MCE genau ist. Laut Dokumentation verwendet MCE einen "Banking queue model". Es gibt also viele Bankschalter und eine lange Schlange von Personen die in der Bank etwas erledigen wollen. Immer wenn ein Schalter frei wird, kann eine Person aus der Schlange bedient werden.

Bei MCE sind die Bankschalter die Worker, und die Personen die Eingabedaten. Die Worker werden zu Beginn erstellt und bleiben bis zum Schluss am Leben. Für die Erstellung der Worker kennt MCE mehrere Mechanismen. Steht Threading zur Verfügung, wird dieses verwendet, ansonsten wird der Fallback `forks` bzw. `fork` verwendet.

Die einzelne Person aus der Schlange in der Bank ist ein "Chunk", also ein Stück der Eingabedaten. MCE kann auch so konfiguriert werden, dass die Stückgröße ungleich 1 ist. Auch die Anzahl der Worker ist konfigurierbar.

MCE kennt drei Arten von Modulen. Das sind zum einen die Kernmodule, die Addons und die Models. Die Kernmodule stellen die Kernfunktionalität von MCE bereit. `MCE::Core` stelle die API für die Multi-core Engine zur Verfügung, die von

```
perl -MData::Dumper
     -MSys::Statistics::Linux
     -e 'my $s = Sys::Statistics::Linux->new(sysinfo => 1);
         my $info = $s->get;
         print Dumper $info'
$VAR1 = bless( {
  'sysinfo' => {
    'swaptotal' => '8081404 kB',
    'arch' => 'x86_64',
    'version' => '...',
    'release' => '3.11.0-19-generic',
    'hostname' => 'perl-services',
    'countcpus' => 4,
    'pcpucount' => 1,
    'tcpucount' => 4,
    'idletime' => '5d 8h 58m 17s',
    'domain' => '(none)',
    'interfaces' => 'eth0, lo, wlan0',
    'kernel' => 'Linux',
    'memtotal' => '7873488 kB',
    'uptime' => '2d 2h 3m 15s'
  }
}, 'Sys::Statistics::Linux::Compilation' );
```

Listing 3

den anderen Modulen verwendet wird. Mit `MCE::Signal` kann man das Verhalten von MCE beeinflussen. So kann man z.B. das Löschen von temporären Dateien verhindern. `MCE::Util` stellt momentan nur eine einzige Methode zur Verfügung. Mit dieser kann die Anzahl der verfügbaren Cores bestimmt werden.

Die Addons erweitern die Funktionalität von MCE. So stellt `MCE::Subs` einige Funktionen zur Verfügung, die nur ein Wrapper um die Methoden aus `MCE::Core` sind. So gibt es dann die Funktionen `mce_print` um etwas auszugeben oder `mce_forseq` um auf einer Sequenz zu arbeiten.

Die Models konfigurieren schon für einige Anwendungsfälle die MCE-Funktionalitäten. Mit diesen lässt sich in einigen Fällen schon mit wenigen Zeilen statt des "normalen" Perl-Codes die MCE-Funktionalität verwenden. Als fertige Models werden `MCE::Loop`, `MCE::Grep`, `MCE::Map`, `MCE::Flow`, `MCE::Step` und `MCE::Stream` mitgeliefert. Einige davon werden in den folgenden Abschnitten verwendet.

Erste kleine Beispiele

Auch ohne Eingabedaten funktioniert MCE. So kann man mit dem Kernmodul einfach Worker starten, die eine Aufgabe erledigen:

```
use MCE;

my $mce = MCE->new(
  max_workers => 4,
  user_func => sub {
    my ($mce) = @_;
    print "Hello from ", $mce->wid, "\n";
  }
);

$mce->run;
__END__
Hello from 3
Hello from 1
Hello from 2
Hello from 4
```

Hier werden vier Worker gestartet und jeder dieser Worker gibt seine eigene ID aus. Mit `max_workers` wird die maximale Anzahl der Workers eingestellt. Hier kann man entweder eine Zahl fest eintragen oder - wenn man die Anzahl in Abhängigkeit der verfügbaren Cores setzen will - ein String wie z.B. *'auto-1'* (Verfügbare Cores minus 1) oder *'AUTO-3'* (Verfügbare Cores minus 3). Hier sind alle mathematischen Operationen erlaubt.

`user_func` ist die Funktion, die in jedem Worker ausgeführt wird. Es gibt auch noch `user_begin` und `user_end`, die beim Start des Workers bzw. bei dessem Ende ausgeführt werden. Das kann nützlich sein, wenn der Worker für seine Arbeit eine bestimmt Umgebung benötigt (z.B. eine Datenbankverbindung). Dann kann diese in `user_begin` hergestellt werden.

Möchte man existierenden Code einfach Umschreiben, könnten die Models helfen. Ein bestehendes `grep` kann mit Hilfe von `MCE::Grep` mit kleinsten Änderungen parallelisieren. Man muss hier nur beachten, dass MCE hier einen kleinen Overhead mit sich rumschleppt, so dass bei einfachen Blöcken das originale `grep` schneller ist als das `mce_grep`.

```
use MCE::Grep;

## Array or array_ref
my @a = mce_grep { $_ % 5 == 0 } 1..10000;
my @b = mce_grep { $_ % 5 == 0 }
                              [ 1..10000 ];
```

Soll ein `grep` auf den Zeilen einer Datei ausgeführt werden, könnte `mce_grep_f` ganz hilfreich sein:

```
## File_path, glob_ref, or scalar_ref
my @c = mce_grep_f { /phrase/ }
                           "/path/to/file";
my @d = mce_grep_f { /phrase/ }
                              $file_handle;
my @e = mce_grep_f { /phrase/ } \$scalar;
```

Auch für Sequenzen gibt es ein `grep`-Ersatz:

```
## Sequence of numbers
##           (begin, end [, step, format])
my @f = mce_grep_s { $_ * 3 == 0 }
                           1, 10000, 5;
my @g = mce_grep_s { $_ * 3 == 0 }
                           [ 1, 10000, 5 ];

my @h = mce_grep_s { $_ * 3 == 0 } {
                begin => 1, end => 10000,
                step => 5, format => undef
};
```

Beispiele für den Einsatz von MCE

Ein paar Einsatzbeispiele für MCE finden sich auch im Netz, z.B. auf Stackoverflow. Die Beispiele der Dokumentation und auch das Monte-Carlo-Beispiel in der Distribution halte ich für zu wenig bzw. nicht passend für die meisten Programmierer. Nachfolgend ein Beispiel, wie es auch in meinem Arbeitsalltag auftaucht - das Einlesen großer Dateien.

Gerade beim Einlesen und Verarbeiten von sehr großen Dateien (ggf. mehrere Gigabyte) kann das Parallelisieren helfen. Die folgenden Beispiele sind zum Teil von StackOverflow übernommen bzw. angepasst. In dem Beispiel geht es darum, eine größere Apache-Logdatei zu analysieren und auszuwerten. Dabei sollen kleinere Statistiken zu Land und Browser erstellt werden.

Einlesen mit mce_loop_f

Um schnell etwas mit den Zeilen einer Datei zu machen, kann man die Funktion `mce_loop_f` aus `MCE::Loop` nehmen. Da reicht es auch, wenn man als Parameter den Pfad zur Datei angibt. Alternativ kann man auch direkt ein Filehandle oder eine Skalarreferenz übergeben.

```
use ApacheLog::Parser
              qw(parse_line_to_hash);
use MCE;
use MCE::Loop;

use Data::Dumper;

our %hash;

mce_loop_f  {
 my ($mce, $chunk_ref, $chunk_id) = @_;

 foreach my $line ( @$chunk_ref ) {
    chomp $line;
    next if !$line;
    my %ref = parse_line_to_hash( $line );
    my ($client) = (
      join '.',
        (split /\./, $ref{client})[0..2]
    ) . '.x';
    MCE->do( 'set', $client );
 }
} 'access.log';

sub set {
  my ($client) = @_;
  $hash{$client}++;
}

warn Dumper \%hash;
```

Zuerst werden - was MCE spezifischen Code angeht - die Module `MCE` und `MCE::Loop` geladen. Danach wird das `mce_loop_f` definiert. Aus dem Block wird eine Subroutine, die drei Parameter bekommt: das MCE-Objekt, eine Referenz auf die Teile (hier die Zeilen) und die ID des Teils. Danach wird die Zeile geparst und die wichtigen Informationen werden herausgefiltert.

Um die Informationen im Hauptprozess sichtbar machen zu können, muss man `MCE->do( 'name', ... )` machen. Da die Worker ja geforkte Prozesse sind, haben die jeweils ihr eigenes `%hash`. Mit dem `do` kann eine Funktion im Hauptprozess (dem Manager) ausgeführt werden. Einfach den Namen der Subroutine (hier: `set`) angeben und die Parameter.

mce_loop_f - Variante 2

Eine zweite Variante mit `mce_loop_f` ist es, mit einer `chunk_size` von 1 zu arbeiten. Dann kann man innerhalb des Blocks auch mit `$_` arbeiten:

```
use ApacheLog::Parser
              qw(parse_line_to_hash);
use MCE;
use MCE::Loop;

MCE::Loop::init( {chunk_size => 1} );

use Data::Dumper;

our %hash;

mce_loop_f  {
  my $line = $_;
  chomp $line;
  next if !$line;
  my %ref = parse_line_to_hash( $line );
  my ($client) = (
    join '.',
      (split /\./, $ref{client})[0..2]
  ) . '.x';
  MCE->do( 'set', $client );
} 'access.log';

sub set {
  my ($client) = @_;
  $hash{$client}++;
}

warn Dumper \%hash;
```

Die `foreach`-Schleife innerhalb des Blocks ist weggefallen. Damit das funktioniert, muss man vorher aber `MCE::Loop::init` aufrufen und die `chunk_size` auf 1 setzen.

MCE - Kernfunktionalitäten

Das ganze kann man auch nur mit dem Core-Modul lösen. Hier sagt man

```
use ApacheLog::Parser
              qw(parse_line_to_hash);
use MCE;
use Data::Dumper;

our %hash;

sub parse {
  my ($mce, $chunk_ref, $chunk_id) = @_;

  foreach my $line ( @$chunk_ref ) {
    # [ ... wie im Beispiel oben ... ]
  }
} 'access.log';

sub set {
  my ($client) = @_;
  $hash{$client}++;
}

my $mce = MCE->new(
  input_data => './access.log',
  user_func  => \&parse,
);

$mce->run;

warn Dumper \%hash;
```

Blockweises Einlesen

Hat man Dateien, die nicht zeilenweise eingelesen werden sollen, sondern blockweise, kann man in der `mce_loop_f`-Variante einfach ein `local $/ = "..."` schreiben. Am besten macht man daraus einen eigenen Scope, damit man sonstige Nebeneffekte vermeidet (wenn man z.B. noch eine Datei einlesen will und vergisst `$/` wieder zurückzusetzen).

```
use MCE;
use MCE::Loop;

MCE::Loop::init({chunk_size => 1});

local $/ = "\n---\n";
mce_loop_f  {
  chomp;
  warn ">>$_<<";
} 'info.txt';
```

Getrennte Worker für Einlesen und Verarbeiten

In dieser Abwandlung des Beispiels (Listing 4) wird das Einlesen und die Verarbeitung voneinander getrennt. Die von der einen Worker-Gruppe eingelesenen Zeilen werden in eine Queue geschoben, die dann die Worker aus der anderen Gruppe bearbeiten.

Was wird hier alles Neues genutzt? `Threads::Queue` ist jetzt hier kein Thema, man könnte hier auch andere Queueing-Methoden nehmen. Neu ist z.B. `use_slurpio`, mit dem `$_` im Task immer eine Referenz ist (egal ob `chunk_size` 1 ist oder nicht) und immer eine Referenz auf den Puffer ist.

Mit `user_tasks` kann ein Array von Tasks an MCE übergeben werden. Für jeden der Tasks werden Worker gestartet, wobei da beliebig forks und threads gemischt werden können. Jedem Task können mehrere Parameter übergeben werden: `max_workers`, `user_func`, `use_threads`, `task_end` und weitere.

Soll etwas am Ende eines Tasks ausgeführt werden, kann man `task_end` eine Subroutinenreferenz übergeben, die der Worker ausführt wenn er mit seiner Aufgabe fertig ist.

Ausgangsszenario umschreiben

Nachdem es jetzt die Einführung in MCE gab, wird es spannend. Was bringt der Einsatz von MCE im eingangs beschriebenen Fall? Hier sind die Änderungen nur minimal, weil es

```
use threads;
use threads::shared;
use Thread::Queue;
use MCE;

my $R_QUEUE = Thread::Queue->new;
my $queue_workers = 8;
my $process_workers = 8;
my $chunk_size = 1;

my $input_file = '/path/to/file';

sub buildQueue {
  my ($self, $chunk_ref, $chunk_id) = @_;
  if ($R_QUEUE->pending() < 100) {
    $R_QUEUE->enqueue($chunk_ref);
    $self->sendto('stdout',
        "Queue Size: " . $R_QUEUE->pending
        ."\n");
  }
}

sub processQueue {
  my $self = shift;
  my $wid = $self->wid;
  while (my $buff = $R_QUEUE->dequeue) {
    $self->sendto('stdout',
        "Thread " . $wid . " got $$buff");
  }
}

my $mce = MCE->new(
  input_data => $input_file,
  chunk_size => $chunk_size,
  use_slurpio => 1,

  user_tasks => [
    { # queueing task
      max_workers => $queue_workers,
      user_func => \&buildQueue,
       # we'll use threads to have access
       # to the parent's variables in
       # shared memory.
      use_threads => 1,
       # signal stop to our process
       # workers when they hit the
       # end of the queue.
      task_end => sub {
          $R_QUEUE->enqueue(
            (undef) x $process_workers
          )
      }
    },
    { # process task
      max_workers => $process_workers,
      user_func => \&processQueue,
      use_threads => 1,
      task_end => sub {
          print "Finished processing!\n";
      }
    }
  ]
);

$mce->run();

exit;
```

Listing4

nur zwei Stellen sind, die angepasst werden können. Wir fangen mit dem spannenderen Teil an.

Analyse parallelisieren

Der Teil der Analyse ist spannender, weil besonders die Überprüfung der Programmierrichtlinien relativ lange dauert. Die Überprüfung der einzelnen Datei lässt sich im Moment nicht besonders einfach schneller machen. Hier gibt es vielleicht in nicht allzu ferner Zukunft mit `Perl::Lint` - das gerade im Rahmen eines TPF-Grants entwickelt wird - einen guten Ersatz für `PPI`. Aber wenn mehrere Dateien parallel überprüft werden können, würde das schon einiges an Zeit sparen - so jedenfalls meine Hoffnung.

Original:

```
for my $file ( $self->files ) {

  ROLE:
  for my $role ( @{ $roles{file} || [] } ) {
      my ($sub)
      = $self->can( 'analyze_' . lc $role );
      next ROLE if !$sub;

      my $result   = $self->$sub( $file );
      my $filename = $file->{filename};

     $analysis_data{$role}->{$filename}
                                  = $result;
   }
}
```

Mit `MCE`:

```
use MCE::Loop;
MCE::Loop::init( { chunk_size => 1 } );

mce_loop {
    ROLE:
   for my $role ( @{ $roles{file} || [] } ) {
       my ($sub) =
       $analyzer->can( 'check_' . lc $role );
        next ROLE if !$sub;

        my $result   = $analyzer->$sub( $_ );
        my $filename = $_->{filename};
   }
} $self->files;
```

Dekodieren parallelisieren

In den meisten Paketen sind nicht so arg viele Dateien enthalten. Bei den meisten werden es maximal 20 bis 25 Dateien sein. Große Pakete wie z.B. KIXCore von c.a.p.e IT sind da eher die Ausnahme. Von daher muss man in diesem Fall schauen, ob es tatsächlich einen Vorteil bringt wenn man MCE einsetzt.

Original:

```
FILE:
for my $file ( @files ) {
  my $name = $file->findvalue( '@Location' );

  #next FILE if
    $name !~ m{ \. (?:pl|pm|pod|t) \z }xms;
  my $encode          =
      $file->findvalue( '@Encode' );
  next FILE if $encode ne 'Base64';

  my $content_base64 = $file->textContent;
  my $content        = MIME::Base64::decode(
                           $content_base64 );

  # push file info to attribute
  $self->add_file({
      filename => $name,
           content  => $content,
  });
}
```

Mit MCE:

```
use MCE;
use MCE::Loop;
MCE::Loop::init( { chunk_size => 1 } );

mce_loop {
   my $name    = $_->findvalue( '@Location' );
   my $encode = $_->findvalue( '@Encode' );
   MCE->next if $encode ne 'Base64';

   my $content_base64 = $_->textContent;
   my $content        = MIME::Base64::decode(
                            $content_base64 );

   # push file info to attribute
   MCE->do( 'add_file',
       $self, {
       filename => $name,
       content  => $content,
   });
} @files;
```

Hier sieht man die Verwendung von `MCE->next`, womit man dem Worker sagen kann, dass er mit dem nächsten Stück der Eingabedaten weitermachen soll.

Benchmarking

Zum Benchmarken habe ich eine lokale Kopie von OPAR genommen, so dass ich alle möglichen Größen an OPM-Dateien habe. Dann habe ich ganz naiv die Zeiten gemessen und mir die Werte für Konstellationen gemerkt:

- Zeiten für alle Pakete
- Zeiten für kleinstes Paket

Original:

```
$VAR1 = {
  'time' => '497.312350749969',
  'min_file' => {
    'length' => 1,
    'time' => '0.803714990615845',
    'name' => '/NewTickets-0.0.1.opm'
  }
};
```

Mit MCE:

```
$VAR1 = {
  'time' => '406.970098972321',
  'min_file' => {
    'length' => 1,
    'time' => '1.04557919502258',
    'name' => '/NewTickets-0.0.1.opm'
  }
};
```

Man sieht, dass bei kleinen Paketen der Overhead zu groß ist, aber insgesamt ist der Zeitgewinn schon enorm.

Fazit

Man sollte nicht jedes `map`, `grep` oder `foreach` mit den MCE-Äquivalenten ersetzen, aber bei zeitaufwändigen Aufgaben lohnt sich ein Blick auf Module. MCE ist ein gutes Modul um seinen Code in solchen Fällen schneller zu machen. Unter Umständen muss man auch etwas herumspielen um das Optimale herauszufinden - wann lohnt sich schon MCE und wann fährt man mit den Originalen besser? Je größer die Ausgangsdatenmenge ist (je mehr Arrayelemente, größer die Datei oder zeitaufwändiger der Code) umso mehr kann sich der Einsatz von MCE rechnen. Man darf nicht vergessen, dass das Erzeugen der Worker einen gewissen Overhead erzeugt, der bei kleinen Datenmengen die Vorteile überwiegen kann.

In diesem Beispiel ist die Zeitersparnis schon sehr groß und es ist davon auszugehen, dass die Vorteile von MCE noch größer werden wenn weitere Code-Überprüfungen hinzukommen.

Renée Bäcker

Web-Single-Sign-On mit LemonLDAP::NG

Das Projekt LemonLDAP::NG bezeichnet sich selbst als "Web-SSO der nächsten Generation". Leider setzt es vollkommen auf Apache, wodurch es nicht ganz zur "nächsten Generation" gehört, da andere Webserver immer mehr zum Einsatz kommen. Der Vorteil von Apache liegt hier in der sehr guten Integration von Perl dank mod_perl.

Doch bevor wir zu den Grundlagen von LemonLDAP::NG (*LLNG*) kommen, erst noch ein paar Vorbemerkungen zu der Umgebung, für die der Einsatz der SSO-Lösung getestet wurde. Bei dem betrachteten Kunden gibt es mehrere webbasierte Anwendungen, die zum Einsatz kommen. Das sind unter anderem OTRS als HelpDesk-System und kivitendo als Basis-ERP-System. Damit man sich in diese beiden Anwendungen nicht getrennt einloggen muss und weil bei beiden Anwendungen sowieso die Daten aus einem LDAP stammen, soll ein Single-Sign-On eingerichtet werden. Ziel ist es auch, möglichst einfach zusätzliche Anwendungen bei der SSO-Lösung integrieren zu können.

Da trifft es sich gut, dass mit LemonLDAP::NG nicht nur Personen authentifiziert werden können, sondern dass es einen vollen "triple-A"-Schutz bietet, also Authentifikation, Autorisierung und Accounting.

Womit wir auch schon beim grundsätzlichen Ablauf bei LLNG kommen - und zur Erklärung warum im Moment alles so auf Apache ausgerichtet ist. LLNG besteht aus hauptsächlich drei Komponenten: Den Manager, den Handler und das Portal. Der Manager ist nur für das Konfigurieren des SSO auf LLNG-Seite zuständig. Das Portal bietet die Möglichkeit, sich zu authentifizieren und auch wieder auszuloggen. Es ist die zentrale Komponente, gegen die eine Webanwendung prüft ob ein Benutzer angemeldet ist oder nicht.

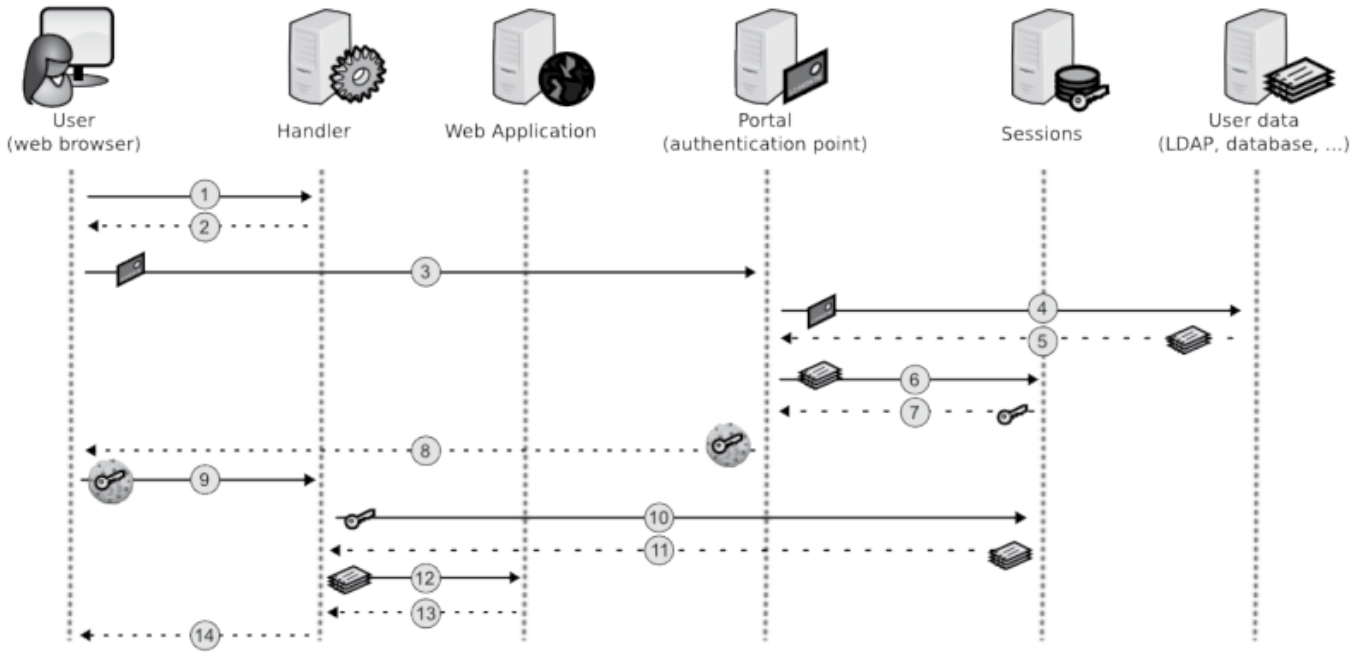


Abbildung 1: Ablauf bei LemonLDAP::NG

Diese Prüfung wird durch den Handler erledigt. Der Handler ist ein mod_perl Modul, das in die Abarbeitung einer HTTP-Anfrage eingreift. Ist der Benutzer noch nicht angemeldet, wird er auf die Login-Seite des LLNG-Portals weitergeleitet. Dort muss sich der Benutzer anmelden, woraufhin eine Session in der Datenbank abgelegt und ein Session-Cookie erzeugt wird. Danach wird der Benutzer zur ursprünglichen Anwendung umgeleitet. Der Handler erkennt jetzt, dass ein Session-Cookie vorliegt und der Nutzer kann die Anwendung verwenden.

Dieser Ablauf ist auch in Abbildung 1 dargestellt.

Installation

Für Debian/Ubuntu gibt es fertige Pakete, die man installieren kann. Auch auf CPAN sind die Module zu finden. Die Installation über CPAN würde ich hier jedoch nicht empfehlen, da dann zu viel Nacharbeiten per Hand zu machen sind. Das ist ein Punkt, der noch stark verbesserungsfähig ist.

Hier wird also die Installation über das Archiv gemacht. Bevor es dazu kommt muss man eine ganze Reihe an Abhängigkeiten installieren, die in einer langen Liste auf der Webseite aufgeführt sind. Für *apt-get* bzw. *yum* gibt es unter *http://lemonldap-ng.org/documentation/latest/prereq* auch den Befehl zum einfachen kopieren.

Nachdem diese Vorarbeiten abgeschlossen sind, muss man erstmal das Archiv herunterladen. Der Link dazu ist unter *http://lemonldap-ng.org/download* zu finden. Nachdem das Archiv heruntergeladen wurde, einfach entpacken und dann kann es losgehen:

```
tar zxvf lemonldap-ng-1.3.3.tar.gz
cd lemonldap-ng-1.3.3
make
make test
```

Bevor man jetzt weiter macht, sollte man sich ein paar Gedanken machen: Unter welchem User/Gruppe läuft der Apache? Unter welcher Domain ist die Seite erreichbar? Wie soll der VHost des Apache konfiguriert werden.

Soll der Standard verwendet werden, kann man jetzt einfach

```
make install
```

starten. Läuft der Apache nicht unter dem Standard-User (was z.B. bei OTRS-Installationen empfohlen ist), muss man das angeben. Standardmäßig werden die Konfiguration für die Domain *example.com* angelegt. Hier soll aber alles unter der Domain `fooapps.de` erreichbar sein:

```
make install APACHEUSER=otrs \
    APACHEGROUP=otrs \
    DNSDOMAIN=fooapps.de
```

Anschließend müssen die Konfigurationsdateien im Apache eingebunden werden (siehe Listing 1).

Die *test-apache2.conf* ist für die zwei Testanwendungen, die mit LLNG mitgeliefert werden. Diese wird in einer Produktivumgebung nicht benötigt, kann aber gut als Vorlage dienen.

Achtung: Die Apache-Konfigurationen sind für Apache <= 2.2. In Apache 2.4 hat sich ein wenig in den Befehlen geändert. Hieß es bisher

```
Order deny, allow
Deny from all
```

muss es jetzt

```
Require all denied
```

bzw. statt

```
Order allow, deny
Allow from all
```

jetzt

```
Require all granted
```

```
$ ln -s /usr/local/lemonldap-ng/etc/test-apache2.conf /etc/apache2/conf-enabled/
$ ln -s /usr/local/lemonldap-ng/etc/handler-apache2.conf /etc/apache2/conf-enabled/
$ ln -s /usr/local/lemonldap-ng/etc/portal-apache2.conf /etc/apache2/conf-enabled/
$ ln -s /usr/local/lemonldap-ng/etc/manager-apache2.conf /etc/apache2/conf-enabled/
$ service apache2 restart
$
```

Listing 1

Ohne diese Änderungen wird man immer auf *http://auth.fooapps.de* umgeleitet und bekommt eine Fehlerseite angezeigt.

Läuft der Apache ohne Probleme, kann man einfach mal *http://test1.fooapps.de* aufrufen. Die Anwendung wird dann feststellen, dass kein User eingeloggt ist und man wird daher auf das Portal unter *http://auth.fooapps.de* umgeleitet (Abbildung 2). LemonLDAP::NG hat für die Testumgebung ein paar feste Benutzer hinterlegt.

Loggt man sich jetzt mit der Nutzer-Kombination `msmith` / `msmith` ein, wird man wieder auf die Testanwendung umgeleitet in der man alle möglichen Angaben zu Umgebungsvariablen etc. findet.

Damit läuft das Single-Sign-On für Webanwendungen prinzipiell schonmal. Nur kommt im echten Einsatz eine fest hinterlegte Nutzerliste nicht in Frage und die gesamte "Architektur" der Anwendungen kann eine ganz andere sein. Im nächsten Schritt soll ein LDAP als Nutzerquelle verwendet werden. Das dürfte gerade in größeren Unternehmen die häufigste Form der Datenhaltung von Benutzerdaten sein.

LemonLDAP::NG unterstützt aber auch eine ganze Reihe anderer Backends: Eine Datenbank, die über DBI angebunden wird, Facebook, Google, BrowserID, Apache (Kerberos, NTLM, OTP, ...), OpenID, Radius, Twitter und viele mehr. Man hat also die freie Wahl. Bei einigen Backends kann allerdings kein Passwort geändert werden wie z.B. bei Google oder Facebook.

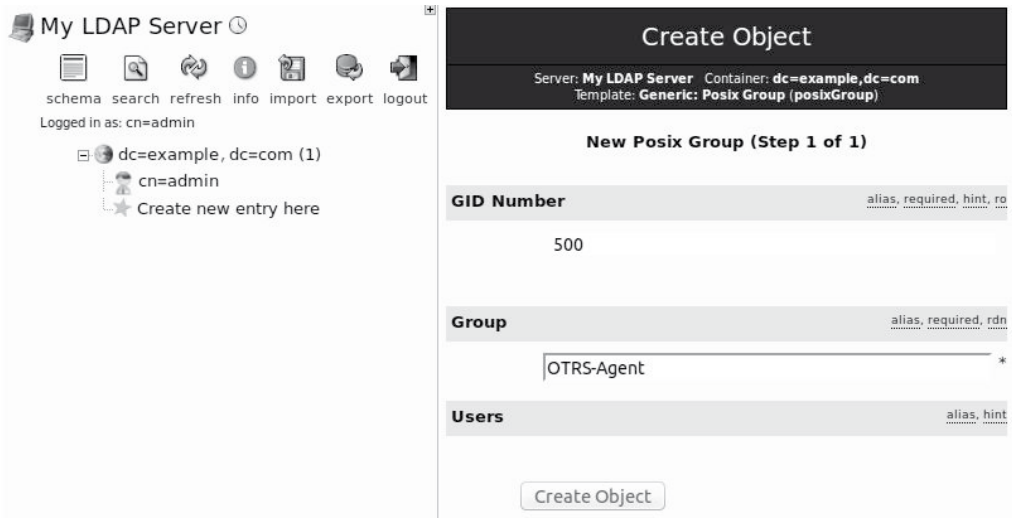


Abbildung 2: Formular zum Anlegen einer POSIX Gruppe

Wie bereits gesagt, wollen wir aber ein LDAP-Backend verwenden. Denn dort können neben den Stammdaten einer Person auch Gruppenzugehörigkeiten und andere Informationen gespeichert werden. Die Gruppenzugehörigkeiten spielen dann eine wichtige Rolle, wenn nicht alle Benutzer auch wirklich alle Anwendungen benutzen können sollen.

OpenLDAP einrichten

Um ein LDAP als Backend verwenden zu können, muss erstmal ein LDAP eingerichtet werden. Dazu gibt es schon viele Tutorials im Netz und in der 3. Ausgabe von *$foo* gab es ein `Net::LDAP`-Tutorial von Martin Fabiani. Deshalb hier nur kurz die Installation von OpenLDAP.

```
$ apt-get install slapd
```

Damit ist das LDAP installiert. Nun müssen noch ein paar Leute eingetragen werden. Damit ich nicht alles mit einem LDIF machen muss, habe ich in der Testumgebung *phpLdapAdmin* installiert (Wer hätte Lust mit mir ein perlLdapAdmin zu schreiben?). Damit erstelle ich alle notwendigen Benutzer, die auf die Webanwendungen zugreifen können sollen, und die Gruppen um schließlich noch Berechtigungen umsetzen zu können.

Als erstes werden POSIX-Gruppen angelegt, weil für die Personen eine Gruppe angegeben werden muss. Dazu muss man im LDAP-Manager die Domäne auswählen, in der die Personen auch angelegt werden sollen. Danach auf "Create new entry here" klicken und die Checkbox von "Generic: Posix Group" aktivieren. In dem Formular einen Gruppennamen angeben (Abbildung 2) und speichern. Achtung: Bei phpLdapAdmin muss jede Änderung auch nochmal committed werden.

Ist die Gruppe angelegt, können jetzt die Nutzer der Anwendungen angelegt werden. Dazu wieder einen neuen Eintrag für die Domäne

erstellen - diesmal ein "Generic: User Account". Alle notwendigen Felder ausfüllen, Objekt erstellen und committen.

Für das Beispiel hier wird keine Schemaänderung benötigt. Im Unternehmens-LDAP kann das Schema natürlich angepasst sein, aber die meisten Anwendungen - so auch LLNG und OTRS - erlauben die Definition der Attribute und Filter um die richtigen Infos aus dem LDAP zu bekommen.

LDAP als Backend verwenden

Wenn das LDAP soweit steht, kann es als Backend verwendet werden. Alle Konfigurationseinstellungen werden im Manager gemacht. Dazu einfach auf *http://manager.fooapps.de*; gehen. Ist man nicht als Admin unterwegs, muss man erst auf dem Portal vorbeischauen und sich ausloggen und wieder als Admin einloggen.

Übrigens: Sollte beim Aufruf der Manager-URL nur die *index.pl* heruntergeladen werden, dann muss noch das CGI-Modul im Apache aktiviert werden.

```
$ ln -s /etc/apache2/mods-available/cgi.
            load /etc/apache2/mods-enabled/
$ service apache2 restart
```

Die Einstellungen zum LDAP sind unter den *General Parameters* zu finden (Abbildung 3). Zuerst muss für *Authentication Module*, *Users Module* und *Password Module* jeweils *LDAP* eingestellt werden. Bei jeder Änderung im Manager sollte *Apply* geklickt werden, damit die Änderungen auch wirklich gespeichert werden.

Anschließend müssen noch weitere Dinge eingestellt werden. Sobald bei den eben genannten Parametern *LDAP* gewählt wurde, erscheint im Menü der Konfiguration der Punkt *LDAP Parameters*. In diesen Parametern müssen die Daten für eine Verbindung zum LDAP und evtl. einzusetzende Filter eingegeben werden. Die Filter werden benötigt um die Einträge für Personen und für Gruppen von denen zu trennen, die nicht für die Authentifizierung herangezogen werden sollen.

In der Testumgebung waren die Standardwerte vollkommen ausreichend - außer dem Account für den LDAP-Zugriff, ansonsten kann mit Sicherheit der LDAP-Administrator weiterhelfen.

Man sollte nicht vergessen, in der Konfiguration der Virtual Hosts für *manager.fooapps.de* den Zugriff auf die Konfiguration zu sichern. Dazu muss die Regel *default* (unter dem Punkt *Rules* zu finden) noch die *uid* desjenigen eingetragen werden, der auch in Zukunft den SSO-Manager bedienen soll.

Sind alle notwendigen Änderungen an der Konfiguration vorgenommen worden, muss die Konfiguration noch mit dem Button *Save* gesichert werden.

Wer nicht über den Manager gehen möchte um LLNG zu konfigurieren, kann das auch direkt auf Dateiebene arbeiten. Unter */usr/local/lemonldap-ng/etc/* liegt die Datei *lemonldap-ng.ini*, in der alle möglichen Einstellungen vorgenommen werden können. Eine Liste mit den ganzen Parametern ist im Wiki von LLNG einsehbar.

In der Regel ist es aber bequemer über den Manager zu arbeiten. Änderungen kann man über verschiedene Dateien in */usr/local/lemonldap-ng/data/conf* verfolgen. Bei jedem *Save* wird eine neue Datei nach dem Schema `lmConf-#` angelegt.

Jetzt kann man sich unter *http://auth.fooapps.de* als "dwho" ausloggen und mit seinem LDAP-Account anmelden. Als Benutzername dient hierbei die *uid* aus dem LDAP.

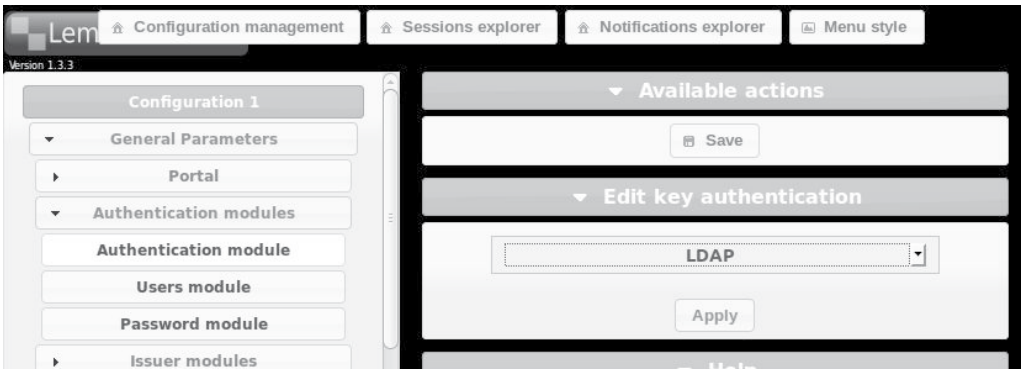


Abbildung 3: Menü im Manager

Konfigurationsbackend ändern

Sollen die Konfigurationen nicht in normalen .ini-Dateien verwaltet werden, kann das Backend angepasst werden. Hier kann man unter anderem eine Datenbank oder das LDAP verwenden. Der Vorteil bei LDAP als Konfigurationsbackend ist, dass es sich leichter zwischen Servern verteilen lässt wenn diese Remote auf das LDAP zugreifen können und es ist leichter SSL/TLS-Unterstützung einzurichten.

Die Konfiguration im LDAP wird dann in einem eigenen Zweig verwaltet: `ou=conf,ou=applications,dc=fooapps,dc=de`. Auch hier werden immer neue Versionen abgelegt:

```
cn=lmConf-1,ou=conf,ou=applications,
  dc=fooapps,dc=de
```

Die Konfiguration wird über die Objektklasse *applicationProcess* abgebildet. Sollte das im LDAP-Server noch nicht abbildbar sein, muss das Schema entsprechend angepasst werden.

Im nächsten Schritt muss der Zweig für die Konfigurationen erstellt werden. Die DN sollte man sich merken, weil die *.ini* von LLNG auch noch angepasst - wie im folgenden Listing zu sehen - werden muss.

```
[configuration]
type = LDAP
ldapServer = ldap://localhost
ldapConfBase = ou=conf,ou=applications,
   dc=fooapps,dc=de
ldapBindDN = cn=manager,dc=fooapps,dc=de
ldapBindPassword = secret
```

Anwendungen anbinden

Jetzt ist LemonLDAP::NG soweit konfiguriert, dass die Benutzer der Anwendungen im LDAP stehen. Nun geht es darum, dass die Anwendungen daran angebunden werden. Voraussetzung für eine Anbindung ist, dass die Anwendung Benutzer als authentifiziert erkennt, wenn der LLNG-Handler entsprechende HTTP-Header setzt.

Anwendung im Portal anzeigen

Das Portal ist nicht nur für den Login da. Man kann jederzeit auf das Portal gehen und als angemeldeter Nutzer bekommt man unter Umständen alle möglichen Informationen angezeigt. Unter anderem auf welche Anwendungen man zugreifen kann. Diese Anzeige muss im Manager konfiguriert werden.

Unter dem Konfigurationspunkt "Portal" gibt es den Unterpunkt "Menu". Unter "Modules activation" muss die Anwendungsliste aktiviert sein. Die Anwendung selbst muss dann unter "Categories and applications" eingetragen werden. Entweder in eine der bestehenden Kategorien oder man legt selbst eine neue Kategorie an.

Anwendungen unter Apache

Als Beispiel für Anwendungen unter Apache dient hier OTRS. OTRS an LLNG anzubinden ist sehr einfach, da es auch ein HTTP-Auth-Modul zur Authentifizierung anbietet. Man muss jetzt nur noch einen Handler definieren, der für das OTRS prüft ob es eine gültige Session und somit einen eingeloggten Benutzer gibt oder nicht.

Als erstes sollte OTRS in einem VirtualHost laufen, z.B. mit dem `ServerName` *support.fooapps.de*. Dieser Name muss dann ins DNS bzw. in der Testumgebung in die */etc/hosts* eingetragen werden.

Arbeiten im Manager

Im Manager von LLNG muss dieser neue VirtualHost ebenfalls eingetragen werden. Im Hauptmenü gibt es den Punkt *Virtual Hosts*, bei dem man den neuen Virtuellen Host hinzufügen kann. Sobald der Virtuelle Host im Manager angelegt ist, kann man auch hier Regeln hinterlegen um den Zugriff feiner zu steuern. Sollen noch mehr eigene HTTP-Header gesetzt werden, ist das auch hier möglich, ist aber für OTRS nicht notwendig.

OTRS den Benutzernamen mitteilen

Wir müssen OTRS jetzt noch irgendwie den Benutzernamen mitteilen. Dazu muss die Apache-Konfig für das OTRS angepasst werden. In der *Location /otrs* Direktive muss noch ein

```
PerlRequire /pfad/zu/MyBasicAuth.pm
PerlHeaderParserHandler My::BasicAuth
```

eingefügt werden. Sollte die *MyBasicAuth* nicht vorliegen (z.B. weil man alles per CPAN installiert hat), kann man die einfach irgendwo erstellen und den folgenden Code eintragen.

```
package My::AuthBasic;

# Load Auth Basic Handler
use Lemonldap::NG::Handler::
       SpecificHandlers::AuthBasic;
@ISA = qw(Lemonldap::NG::Handler::
       SpecificHandlers::AuthBasic);

__PACKAGE__->init(
    {

        # See Lemonldap::NG::Handler
    }
);

1;
```

Dann muss natürlich der Pfad in der Apache-Konfiguration angepasst werden.

OTRS konfigurieren

In der OTRS-Konfiguration (*$OTRS_HOME/Kernel/Config.pm*) ist nur eine einzige Zeile hinzuzufügen:

```
$Self->{AuthModule} =
  'Kernel::System::Auth::HTTPBasicAuth';
```

Man muss nur noch darauf achten, dass die Agenten auch in der OTRS-Datenbank eingetragen sind.

Mit diesen wenigen Änderungen ist OTRS in LLNG eingebunden.

Für einige Anwendungen wie Sympa oder Zimbra gibt es in der `LemonLDAP::NG::Handler`-Distribution schon fertige Handler.

Anwendungen unter Nicht-Apache

Dieser Fall ist nicht so einfach. Wie schon weiter oben erläutert, ist LLNG auf die Verwendung von Apache ausgerichtet auch wenn es im Jira des Projekts verschiedene Tickets für die `nginx`-Unterstützung gibt. In solchen Fällen würde ich nginx als Reverse Proxy verwenden und einen Plack-Server (z.B. `Starman`) für die Ausführung der Anwendung.

nginx-Konfiguration

In Listing 2 ist eine Beispielkonfiguration für OTRS zu sehen

Als erstes wird der Upstream definiert, wo die Anwendung läuft. In diesem Fall ist die Anwendung über 127.0.0.1:3067 erreichbar. Dann wird der Server für *support.fooapps.de* definiert. Wenn man größere Anhänge an den Tickets akzeptieren möchte, muss man noch `client_max_body_size` setzen. Der Standard lässt nur kleinere Anfragen zu. Mit dem `rewrite` wird ein Aufruf von *http://support.fooapps.de* nach *http://support.fooapps.de/otrs/index.pl* umgeleitet.

Um nicht die ganzen statischen Dateien wie Bilder, CSS- oder JavaScript-Dateien über die Anwendung ausliefern zu lassen, wird hierfür eine `Location` definiert. Unter `/otrs` sind dann die ganzen Skripte zu finden. Deswegen werden diese Anfragen an die Anwendung weitergeleitet.

Nicht-Plack-Anwendungen mit Plack

Hat man das Problem, dass die Anwendung nicht schon nativ eine Plack-Anwendung ist (OTRS ist z.B. grundsätzlich immer noch eine CGI- bzw. mod_perl-Anwendung), kann man das leicht ändern.

Dank `Plack::App::CGIBin` und `Plack::Builder` kann man jede CGI-Anwendung schnell Plackfähig machen. In Listing 3 wird gezeigt, wie man OTRS Plackfähig machen kann.

Das handling der IP-Adresse ist notwendig damit die richtige IP-Adresse im OTRS für das Sessionhandling benutzt wird. Da die Anfrage von nginx weitergeleitet wird, würden sonst alle Agenten als IP-Adresse die 127.0.0.1 haben.

```
upstream otrs_app {
  server 127.0.0.1:3067;
}

server {
  listen 80;
  server_name support.fooapps.de;

  client_max_body_size 20M;

  rewrite ^/?$ /otrs/index.pl permanent;

  location /otrs-web {
    alias /opt/otrs/var/httpd/htdocs;
  }

  location /otrs {
    proxy_read_timeout 300;
    proxy_pass http://otrs_app;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For \
      $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-HTTPS 0;
  }
}
```

Listing 2

```
#!/usr/bin/perl

# file: otrs.psgi

use strict;
use warnings;

use Plack::App::CGIBin;
use Plack::Builder;
use Module::Refresh;

my $reverse_proxy = $ENV{OTRS_REVERSE_PROXY};
my $real_ip       = $ENV{OTRS_REAL_IP} || 'X-Forwarded-For';

my $app_main = Plack::App::WrapCGI->new(
  script => "/opt/otrs/bin/cgi-bin/index.pl"
)->to_app;

builder {
    enable sub {
        my $app = shift;
        sub {
            my $env = shift;

            if ( $reverse_proxy ) {
                $real_ip =~ s/-/_/g;
                $real_ip = uc $real_ip;

                $env->{REMOTE_ADDR} = $env->{"HTTP_" . $real_ip};
            }

            $app->($env);
        };
    };

    mount "/otrs/index.pl" => sub {
        Module::Refresh->refresh;
        $app_main->(@_);
    };
};
```

Listing 3

Jetzt kann man nach der Installation von `Starman` einfach

```
OTRS_REVERSE_PROXY=1 \
nohup starman --listen 127.0.0.1:3067 \
bin/otrs.psgi &
```

aufrufen und startet so das OTRS.

Plack::Middleware

Jetzt läuft das OTRS unter Starman hinter einem nginx ReverseProxy und man kann nicht mehr die existierenden Handler von nehmen. Mit `Plack::Middleware`-Modulen kann man in die Requests eingreifen ohne dass man etwas am Code der Anwendung ändern muss. Ich habe `Plack::Middleware::LemonLDAP::BasicAuth` geschrieben, mit dem man auch OTRS unter einem Plack-Server wie Starman an LemonLDAP anbinden kann. Nach der Installation des Moduls muss man nur den Aufruf von Starman etwas anpassen:

```
OTRS_REVERSE_PROXY=1 \
nohup starman \
-e 'enable "LemonLDAP::BasicAuth", \
    portal => "http://auth.fooapps.de"' \
--listen 127.0.0.1:3067 \
bin/otrs.psgi &
```

Eigene Anwendungen

Schreibt man eine Anwendung für sich selbst und möchte sie von Anfang an für den Einsatz unter `LemonLDAP::NG` vorsehen, sollte man gleich so planen, dass die Anwendung Benutzer erkennt wenn der Benutzername in `$ENV{REMOTE_USER}` oder im HTTP-Header (dort können noch mehr Informationen wie Gruppen oder E-Mailadresse übertragen werden) übergeben wird.

Man kann man sich das alles auch viel einfacher machen. In vielen Fällen kommt (im Hintergrund) immer noch `CGI.pm`

zum Einsatz. In so einem Fall kann man einfach CGI durch `LemonLDAP::NG::Handler::CGI` ersetzen. Also anstatt

```
use CGI;
my $cgi = CGI->new;
```

einfach

```
use LemonLDAP::NG::Handler::CGI;
my $cgi =
  LemonLDAP::NG::Handler::CGI->new;
```

Dann kann man später im Code mit

```
$cgi->authenticate;
```

prüfen ob der Benutzer authentifiziert ist. Auch an die Benutzerdaten kommt man sehr einfach:

```
# Get attributes (or macros)
my $cn = $cgi->user->{cn}

# Test if user is member of Lemonldap::NG
# group (or LDAP mapped group)
if( $cgi->group('admin') ) {
  # special html code for admins
}
else {
  # another HTML code
}
```

Für moderne Frameworks wie Dancer, Mojolicious oder auch Catalyst die man mit Plack betreiben kann, kann man auch die Lösung mit der Plack::Middleware einsetzen.

Multidomain-SSO

Nach der Authentifizierung am Portal wird ein Cookie gesetzt in dem die Session-Informationen gespeichert werden. Werden jetzt Webanwendungen nicht nur unter *fooapps.de* sondern auch unter *perlhelp.de* betrieben, hat man das Problem dass das Cookie nur für die Domain gilt, für die das Portal konfiguriert wurde. Aus Sicherheitsgründen werden die Cookies einer Domain nicht an eine andere Domain geschickt, was in 99,9% der Fälle auch richtig ist. Nur in diesem Fall ist es eher hinderlich.

Der Benutzer meldet sich also am Portal an, bekommt das Cookie für die Domain des Portals gesetzt und besucht dann eine Webanwendung, die auf der anderen Domain läuft. Die Anwendung sieht jetzt dieses Cookie nicht und leitet den Benutzer wieder auf das Portal um. Jetzt muss man in der Konfiguration noch die *Cross-Domain-Authentication* (CDA) aktivieren. Dann erkennt das Portal, dass der Benutzer das SSO-Cookie hat, aber von einer anderen Domain kommt. Das Portal leitet den Benutzer dann wieder auf die Anwendung zurück und übergibt die Session-ID als URL-Parameter.

Der Handler vor der Anwendung erkennt diesen Parameter und erstellt für die Domain der Anwendung ein SSO-Cookie.

Damit das alles funktioniert muss im Manager unter *General Parameters -> Cookies* das *Multiple domains* auf On gesetzt werden.

Corporate Identity für das Portal

Im Standard ist das Portal zwar ganz nett, aber in der Regel ist es eher gewünscht, dass sich Login-Masken etc. an eine bestimmte *Corporate Identity* halten. Aus diesem Grund kann es notwendig sein, dass man das Aussehen des Portals anpassen muss. In den neueren Versionen von LLNG ist das kein Problem mehr.

Schon bei der Installation des Portals gibt es drei *Skins*:
- pastel
- impact
- dark

Diese treffen wahrscheinlich eher nicht die Wünsche der Marketing-Abteilung. Dann ist es notwendig, dass ein eigener Skin erstellt wird. Einfach einen existierenden Skin zu überschreiben ist nicht geschickt, da diese Änderungen mit dem nächsten Upgrade von LemonLDAP::NG wieder überschrieben werden.

Ein Skin besteht dabei aus mehreren Komponenten:
- Templates: `HTML::Template`-Dateien mit der Endung *.tpl* Hier soll der ganze HTML-Inhalt rein.
- CSS-Dateien: Für das hübsche Aussehen des HTMLs.
- JavaScript-Dateien: Wenn JavaScript benötigt wird
- Bilder und weitere Medien

Als erstes muss die entsprechende Ordnerstruktur angelegt werden:

```
cd /usr/local/lemonldap-ng/htdocs/portal/
                                        skins
mkdir foo
mkdir foo/css
mkdir foo/js
mkdir foo/images
```

Danach einfach alle *.tpl*-Dateien aus einem bestehenden Skin rüber kopieren (man möchte vielleicht nicht alle HTML-Dateien bearbeiten):

```
cp pastel/*.tpl foo/
```

Und dann kann der Spaß losgehen. In der *customhead.tpl* werden alle CSS-Dateien eingebunden sowie alle weiteren Infos aus dem `<head>`-Bereich einer HTML-Seite gepflegt. Der HTML-Code für den *header*-div ist in der Datei *customheader.tpl* zu finden, für den *footer*-div in der *customfooter.tpl*.

Sind diese Arbeiten abgeschlossen, muss der Skin im Manager noch aktiviert werden.

Ein Herz, tausend Gesichter

Wird LLNG nicht nur für Anwendungen einer Domain genutzt, sondern für die Anwendungen mehrerer Domains - also z.B. neben fooapps.de noch perlapps.de. In so einem Fall muss das Portal nicht an ein einzelnes Design angepasst werden sondern an mehrere - man möchte dem Nutzer ja das Gefühl geben ständig in einer einheitlichen Umgebung unterwegs zu sein. Denn Sprünge im Aussehen oder im Verhalten führen eher zu einem Abbruch der Arbeiten.

Deshalb bietet LLNG die Möglichkeit, den verwendeten Skin abhängig von der aufgerufenen URL (sprich Anwendung) oder von der aufrufenden IP zu machen. Dazu müssen erstmal verschiedene Skins nach dem oben gezeigten Muster erstellt werden. Danach muss im Manager unter "General Parameters" > Portal > Customization > Skin display rules eine neue Regel erstellt werden.

Dort müssen zwei Felder ausgefüllt werden. Zum Einen die Regel an sich und zum Anderen der Skin, der verwendet werden soll, wenn die Regel zutrifft. Die Regel ist ein Perl-Ausdruck, in dem die Umgebungsvariablen über `%ENV` und die aufgerufene URL über `$_url` verwendet werden kann. Soll der Skin *foo* verwendet werden wenn die Aufrufer-IP die `127.0.0.1` ist oder die aufgerufene URL zur Domain *fooapps.de* gehört, dann muss die Regel

```
$ENV{REMOTE_ADDR} eq '127.0.0.1' ||
$_url =~ m{https?://(.*?\.)?fooapps\.de}
```

heißen.

Das Portal umgehen...

Auch wenn man das Portal jetzt so schön an das Corporate Design angepasst ist, ist das hin und her Weiterleiten auch nicht so prickelnd für den Benutzer. Da ist es doch besser, ein Login-Formular direkt bei der Anwendung zu haben. In so einem Fall muss man folgende Dinge bei der Verarbeitung der Benutzereingaben tun:

- Per SOAP beim Portal eine Session erzeugen
- Beim Portal anfragen, ob der Benutzer die Anwendung sehen darf
- Ein SSO-Cookie erstellen

Der Code um beim Portal eine Session zu erzeugen ist in Listing 4 zu sehen.

Fazit

Die hier gezeigten Möglichkeiten sind nur ein Bruchteil von dem was LemonLDAP::NG kann. Und über Single-Sign-On kann man die Akzeptanz von Anwendungen erhöhen, weil der Nutzer sich nicht mehr an jeder Anwendung einzeln anmelden muss, sondern direkt nach dem Aufruf der Anwendung mit der Arbeit loslegen kann.

Nach einer gewissen Zeit der Einarbeitung - auch um sich einen Überblick über die Features zu verschaffen - ist es einfach, seine Anwendungen an ein SSO anzubinden. Damit ist es dann auch möglich eine zentrale Benutzerverwaltung zu haben und so auch konsistente Daten in allen Anwendungen zu haben.

Hier wurde jetzt nicht betrachtet, wie die einzelnen Komponenten noch besser gegen unbefugte Zugriffe gesichert werden können. Das ist im Zweifelsfall eine Sache die mit den Systemadministratoren abgestimmt werden muss.

```
my $xheader = $env->{'X_FORWARDED_FOR'};
$xheader .= ", " if ($xheader);
$xheader .= $env->{REMOTE_ADDR};

my $soap_headers = HTTP::Headers->new( "X-Forwarded-For" => $xheader );

my $soap = SOAP::Lite->proxy(
    $self->portal || '',
    default_headers => $soap_headers,
)->uri('urn:Lemonldap::NG::Common::CGI::SOAPService');

my $response = $soap->getCookies( $user, $password );
my $cv;

# Catch SOAP errors
if ( $response->fault ) {
    return;
}
else {
    my $res = $response->result();

    # If authentication failed, display error
    if ( $res->{errorCode} ) {
        return;
    }

    $cv = $res->{cookies}->{ $self->cookiename };
}

return 1;
```

Listing 4

Herbert Brenung

Rezension

Mark C. Chu-Carroll
Good Math (englisch)
pragprog.com, 262 Seiten
Vertrieb: O'Reilly Media
1. Auflage, August 2013
ISBN: 193-778-533-5
Paperback: 44$
epub, mobi, PDF: 24$

Jan Goyvaerts,
Steven Levithan
Reguläre Ausdrücke Kochbuch
O'Reilly, 544 Seiten
1. Auflage, Dezember 2009
ISBN: 978-3-89721-957-1
Gebunden: 49€
PDF: 40€

Gabor Szabo
Advanced Perl Maven
perl5maven.com/products
ständig aktualisiert
PDF: $32

Damit es weniger auffällt, dass die Themen der letzten Folge lediglich fortgesetzt werden (vorletzte Ausgabe - Gabors Perl Kurs und Friedl's Reguläre Ausdrücke), haben die heutigen Rezensionen einen etwas unerwarteten Titel. Und was wäre das Programmieren ohne die Mathematik?

Good Math

Nicht nur weil Perl keine hohe Einsteigerhürde hat, tummeln sich in dem Bereich eine deutliche Anzahl Programmierer, die nie von einer Informatikfakultät belästigt wurden. Randal L. Schwartz und cromatic zum Beispiel treten trotzdem (und mit Erfolg) als Experten auf. Aber gerade im Bereich Mathematik geht es um grundsätzliche, logische Sachverhalte, denen man ohne etwas netten Druck von außen, wohl niemals auf den Grund gegangen wäre. Die daraus gewonnenen Erkenntnisse braucht man meist nicht direkt, aber ein solch geschulter Geist ist ungemein nützlich, wie auch Mark Jason Dominus in *Higher Order Perl* (3/2008) zeigte.

Good Math beginnt erfreulicherweise sehr einfach mit ganzen Zahlen (*N*) und steigert sich langsam. Von Anfang hat es jedoch universitäres Niveau, selbst wenn Sprache und die Vergleiche sehr human sind. Vor allem merkt man die Begeisterung des Autors für das Thema, die den Leser über manch einen Hügel mitziehen kann. Man bekommt auch eine gute Portion Mathematikgeschichte, was den Stoff ebenfalls auflockert und hilft Zusammenhänge zu verstehen. Anregend sind auch die kleinen, eingestreuten philosophischen Betrachtungen (Was bedeutet eigentlich null?). Das einzig Unentspannte an dem Werk tritt zu Tage, wenn sich der Chu-Caroll über bestimmte Populärliteratur äußert. Er bezieht sich da berechtigterweise auf Schund, von dem es einen Berg gibt. Daneben gibt es jedoch auch Brilliantes, das fundiert die teilweisen Komplexen Zahlenverhältnisse in ägyptischer Kunst und Architektur analysiert, worüber ja auch akademische Kunststudenten Literatur haben.

Natürlich geht es hauptsächlich um Mengenlehre, Symmetrien, Lambdakalkül, Turing, Gödel bis zum Halteproblem. Doch bereits das klare Verständnis, was genau der Unterschied zwischen irrationalen und transzendenten Zahlen ist, kann beim täglichen 1mal1 hilfreich sein. Es ist ein Mathebuch explizit für Programmierer, das sogar einige Beispielprogramme (in Scala), Beispieldaten und Grafiken enthält. Die Mehrzahl der Seiten ist allerdings mit Text gefüllt, was

jedoch so gut wie nie langweilig wird, denn es gibt keine Längen. Sonst wäre es Mark nicht gelungen mit 250 Seiten auszukommen. Auch sind die Kapitel klein genug, um das Buch nach 15 bis 20 Minuten wieder beiseite zu legen. Es funktioniert als unterhaltsames Weiterbildungsprogramm für den schwer arbeitenden Tastaturklopfer. Das einzige, was vielleicht wirklich stört: Das Buch gibt es nur auf Englisch und die mathematischen Fachausdrücke dürften selbst jenen nicht geläufig sein, die sich ansonsten gut in Englisch verständigen können.

Aber da gibt es noch einen weiteren Grund, warum dieser Titel hier vorgestellt wird. Der Verlag ist interessant. Gut - es sind Ruby-Fans deren weitere Hauptthemen Scala, Apple und funktionale Sprachen sind. Aber die *http://www.pragmaticprogrammer.com/* haben es vorgemacht, wie man im zügigen Takt gute Bücher zu gefragten Themen veröffentlicht, sodass sogar O'Reilly eine Vertriebspartnerschaft einging. Selbst das hauseigene (mit Zeitabstand kostenlose) Magazin *http://pragprog.com/magazines* wäre für manchen hier eine Bereicherung. Mein Wunsch wäre es, dass wir cromatic's Werkzeuge und Ideen nutzen und selber aktuelle, relevante, kompakte und handwerklich gute Bücher herausbringen. (Der Autor versucht sich auch an einem solchen derzeit.) Anregungen und Interesse bitte in Mailform an die Redaktion.

Advanced Perl Maven

Gabor hat seine Vorstellungen, wie er Bücher schreibt, veröffentlicht und vertreibt sie. Zudem verfügt er sogar über eine breite Zuhörerschaft, die seine Rundmail liest, ihm Themen zuträgt, seine freien Artikel verbessert und sie übersetzt. Das war aber bereits Thema - heute wird der Scheinwerfer lediglich auf sein "high-end"-Produkt gerichtet. Wie bereits das "Beginner"-Buch baut er es inkrementell auf, was sich an der vorliegenden Versionsnummer 1.05 ablesen lässt.

Die eins vor dem Punkt kann man gelten lassen, da die großen Themen recht umfassend behandelt werden und sogar etwas weniger wortkarg als im *Beginner Perl Maven*. Diese Themen sind Module, Datenstrukturen und OOP, nebenbei gibt es noch etwas Fehlerbehandlung, Logging, Tests und Debugging.

Zum Beispiel bei den Datenstrukturen werden alle Arten von Referenzen beschrieben, sogar Perls Referenzzähler und seine Tücken, sowie wie sie sich umgehen lassen. Wie erkenne ich Speicherlecks?, Wie kopiere ich komplette Datenstrukturen?, die Module zur Serialisation und sogar etwas CSV und Excel enthält diese Rundschau.

Im Abschnitt Module tauchen `Exporter`, die verschiedenen Schnittstellen die ein Modul bieten kann, Probleme mit `@INC`, `local::lib` sogar `CPAN::Reporter` auf. Auf die Herausforderung Module unter etlichen Betriebssystemen zu installieren wird eingegangen. Was der Abschnitt über die Moose, Moo und die Kern-OOP wohl beinhaltet - dazu haben $foo-Leser (Ausgabe 8,11,12,15-19) nach dem umfangreichen Tutorial eine Vorstellung. Das Bild sollte sich langsam vervollständigen und den Lobgesang auf Aktualität und Korrektheit der letzten Rezension könnte man hier erneut anstimmen.

Am ehesten ließe sich der Inhalt mit O'Reilly's "Intermediate Perl" vergleichen, dass im August 2012 erneuert wurde und der beinah deckungsgleich, wenngleich anders aufgebaut ist. Es wurde als Lehrmaterial konzipiert, mithilfe dessen die Teilnehmer des zweiten Stonehenge-Kurses den in einer kontrollierten Umgebung behandelten Stoff wiederholen können. Die elektronische Ausgabe ist mit 25,49 € günstiger und hat ein wenig mehr Umfang. Der fortgeschrittene Kenner (Maven) enthält dafür etwas Praxiscode und einige Themen, die man erst im *Mastering Perl* (ISBN 978-1-4493-9311-3) von brian d foy (1/2008) findet. Das wurde übrigens erst Januar 2014 aktualisiert und darf sogar im Netz frei gelesen werden (*http://chimera.labs.oreilly.com/books/1234000001527*).

Doch auch Gabor bleibt nicht stehen und hat mit einem Kochbuch und einem Titel über Test-Automation weiteres in in Vorbereitung.

Reguläre Ausdrücke Kochbuch

Kochbücher enthalten Rezepte zum Nachmachen. Und das mit der Spitzmaus auf dem Titel erklärt mit fertigen Programmstücken, wie Aufgaben aus der Praxis mit regulären Ausdrücken gelöst werden. Diese Ein- bist Mehrzeiler stehen hier parallel in den Sprachen *.Net*, *JVM*, *PCRE*, *Perl*, *Python*, *Ruby*, *Javascript*. Mit den ersten drei aufgezählten, sind natürlich Bibliotheken gemeint, die von etlichen weiteren Sprachen genutzt werden (etwa *.Net* von *C#*, *JVM* von *Scala* oder *PCRE* von *PHP*). Da die Anbindung der Bibliotheken oft unterschiedlich ist, finden sich auch manchmal Exempel in zusätzlichen Sprachen.

Doch uns beschäftigt erst einmal nur Perl. Mit einem solch verengten Blick hat das Buch immer noch ca. 450 statt der physischen über 500 Seiten. Das kommt daher, dass der meiste Platz für die Beantwortung der Fragen verwendet wird, wie und warum der Code funktioniert und vor allem wie ich mir selber solche Lösungen herleite. Das auf dem Umschlag beworbene Einstiegs-Tutorial versteckt sich in den Beispielen der vorderen Abschnitte, welche sehr einfach beginnen, aber dennoch nützliche Probleme lösen.

Denn bereits nach kurzen 25 Seiten Einleitung, die vor allem die Vorstellung der verwendbaren Sprachen, Editoren und anderer Werkzeuge umfasst, beginnen die Rezepte, welche sich bis zum letzten Blatt erstrecken. Danach kommt nur der sorgfältig ausgearbeitete Index, der auch nicht zu lang ist, wodurch sich das Gesuchte flott finden lässt. Die (übersetzte, deutsche) Sprache ist flüssig und angenehm, fällt aber nicht mit eigenem Witz oder besonderen Metaphern auf, was mehr Aufmerksamkeit für das Verstehen des Inhalts übrig lässt. Der ist eigentlich nicht übermäßig schwer. Die letztens besprochenen *Reguläre Ausdrücke* vom Friedl enthalten spürbar komplizierteres und exotischeres. Hier liegt der Augenmerk mehr auf den Details in der Anwendung und auf das Zusammenspiel von Funktionen, wohingegen Jeffrey E. F. Friedl versucht, jede Funktion für sich bis in die Tiefe zu erklären. Somit ergänzen sich beide Titel wirklich gut. Die beiden Köche holen auch Friedl's Versäumnis nach, aktuellere Perl-Features zu erwähnen, denn 2009 war zum Glück Perl 5.10 bekannt.

Was den kurzfristigen Nutzen der Küchenarbeit etwas einschränkt, ist der Umstand, dass für viele der beschriebenen Tätigkeiten heute Module zum Einsatz kommen. Neulinge, die versuchen *URL* oder *XML* mit einer *Regex* zu parsen, bekommen von unsereins in der Regel auf die Finger, um sich selbige nicht an der Herdplatte zu verbrennen, da solche Aufgaben unvermutete Falltüren enthalten und es selbst für Fortgeschrittene kaum machbar ist hier eine immer korrekt arbeitende Lösung abzuliefern. Dennoch ist der Ansatz die logische und kurzweilige Fortsetzung zum *Einstieg in Reguläre Ausdrücke* (vorletzten Folge) was hiermit die Trilogie zu den *Regex* beendet.

Ausblick

In der kommenden Ausgabe wird untersucht, ob man mit dem etwas angestaubten Werk von Horst Eidenberger und Elke Michlmayr *Mit Perl programmieren Lernen* kann. Auf der mitp-Anleitung *Anti Patterns* steht: *Entwurfsfehler erkennen und vermeiden*. Das wird ebenfalls überprüft. Und es werden die *mojocast* auseinander genommen. Sind sie die wirklich zeitgemäße Form der Wissensvermittlung? Und wie unterscheiden sie sich von Gabors Lehrvideos?

CPAN News

Wie immer an dieser Stelle werden sechs neue Module vom CPAN vorgestellt.

Test::Cucumber::Tiny

In dieser Ausgabe gab es mal wieder mehrere Seiten über das Thema "Testen". Das Thema ist in der Perl-Community sehr stark verankert und wie bei vielen Themen gilt auch hier das alte Perl-Motto "There is more than one way to do it". Dementsprechend viele Module und Lösungsansätze gibt es auch auf CPAN.

Wenn man nicht nur als Entwickler für Tests zuständig ist, sondern auch entwicklungsferne Personen, muss man sich überlegen wie die Tests definiert werden und welchen Ansatz man fährt. Denn eine Person, die mit Perl nichts zu tun hat, kann man schlecht mit `Test::More` oder `Test::Exception` behelligen.

In solchen Fällen fährt man häufig den Ansatz des *Behaviour Driven Development* und entsprechendes Testen. Man möchte beschreiben was passieren soll bzw. was das Ergebnis sein soll und nicht wie der Weg dorthin ist. Unter Ruby-Leuten ist das Testing-Tool *Cucumber* bekannt und auch auf dem CPAN findet man entsprechende Module. Eines davon ist `Test::Cucumber::Tiny`.

Zuerst muss man sich mit allen, die Tests schreiben sollen, auf eine gemeinsame Sprache einigen. Wie soll eine Testbeschreibung aussehen? Was soll alles abgedeckt werden? Wenn diese Dinge festgelegt sind, muss darauf geachtet werden dass sich jeder daran hält. Und die Entwickler können die Basis für die Tests mit der Testsprache legen.

Die Tests können dann entweder im Testskript selbst geschrieben werden oder man erstellt eine YAML-Datei, die die Testszenarios enthält. Eine solche Datei ist in Listing 1 zu sehen.

Im Testskript sieht muss nur noch die Datei eingebunden werden:

```
my $cuc =
    Test::Cucumber::Tiny
        ->ScenariosFromYML( "test.yml" )
        ->Given(...)
        ->When(...)
        ->Then(...)
        ->Test;
```

```
- Scenario: Add 2 numbers
  Given:
    - first, I entered 50 into the calculator
    - second, I entered 70 into the calculator
  When: I press add
  Then: The result should be 120 on the screen

- Scenario: Add 3 numbersqr/^(.+),.+entered (\d+)/
  Given:
    - first, I entered 50 into the calculator
    - second, I entered 70 into the calculator
    - third, I entered 10 into the calculator
  When: I press add
  Then: The result should be 130 on the screen
```

Listing 1

Die Basis, von der weiter oben gesprochen wurde, besteht darin, das `Given(...)`, `When(...)` und `Then(...)` mit Leben zu füllen. Aus den jeweiligen Parts müssen die wichtigen Informationen gezogen werden. Schauen wir uns also an, wie das vonstatten geht.

Betrachten wir als erstes was bei `Given()` passieren muss. Um die Informationen herauszuziehen eignen sich Reguläre Ausdrücke hervorragend:

```
qr/^(.+),.+entered (\d+)/
```

Damit kann man die "Position" (first, second, ...) und die eingegebene Zahl speichern. Als nächstes muss man die Treffer noch verarbeiten. Dazu muss man eine Subroutinenreferenz übergeben, wobei diese Referenz als erstes ein `Test::Cucumber::Tiny`-Objekt und als zweites den zu verarbeitenden Satz – also z.B. *first, I entered 50 into the calculator* – übergeben bekommt.

In dem Beispiel werden wir die "Position" und die Zahl in dem Objekt speichern damit wir später noch mit den Zahlen arbeiten können. Damit sieht der Code für `Given()` so aus:

```
->Given(
    qr/^(.+),.+entered (\d+)/ => sub {
        my $c       = shift;
        my $subject = shift;
        my $key     = $1;
        my $num     = $2;

        $c->{numbers}->{$key} = $num;
        $c->Log($subject);
    }
)
```

Das `Given()` wird automatisch für jeden Eintrag bei *Given* im Szenario ausgeführt.

Als nächstes gibt es `When()`, was den Teil des Tests beschreibt, wenn die Vorbedingungen bzw. -aktionen ausgeführt werden und jetzt der Test gestartet wird. In dem Beispiel ist es, wenn "Ich 'add' drücke". Aber auch hier muss man sicherstellen, dass der Test nicht bei einem falschen Trigger ausgeführt wird.

Auch hier nimmt man dann wieder einen Regulären Ausdruck:

```
qr/press add/
```

Genau wie bei`Given()` muss noch eine Subroutinenreferenz übergeben werden in der die abschließende Aktion durchgeführt wird. In dem hier gezeigten Beispiel werden die eingegebenen Zahlen addiert. Damit ergibt sich folgender Code für `When()`:

```
->When(
    qr/press add/ => sub {
        my $c       = shift;
        my $subject = shift;

        $c->{answer} += $c->{numbers}->{$_}
            for keys %{$c->{numbers}};
    }
)
```

Bleibt noch der Abschluss, bei dem geprüft werden muss, ob die abschließende Aktion – also das Addieren der Zahlen – das richtige Ergebnis geliefert hat. Hierfür ist dann `Then()` zuständig. In der YAML-Datei ist die Beschreibung zu finden was als Ergebnis rauskommen soll, nämlich *The result should be 130 on the screen*. Mit einem Regulären Ausdruck wird das erwartete Endergebnis geholt:

```
qr/result.+should be (\d+)/
```

Genau wie bei den beiden vorherigen Methoden auch wird anschließend noch eine Subroutinenreferenz übergeben, in der der eigentliche Test stattfindet:

```
->Then(
    qr/result.+should be (\d+)/ => sub {
        my $c        = shift;
        my $subject  = shift;
        my $expected = $1;

        is $c->{answer},
            $expected,
            $subject;
    }
)
```

Wenn man das Skript mit der oben gezeigten YAML-Datei ausführt, ergibt sich dann die Ausgabe, die hier zu sehen ist.

```
$ perl test.pl
ok 1 - Then The result should
          be 120 on the screen
ok 2 - Then The result should
          be 130 on the screen
1..2
```

Pye

Logging ist ein Thema, das bei fast jeder Anwendung auftaucht. Es hilft ja auch bei der Fehlersuche und zum Nachverfolgen der Aktionen die während des Ablaufs ausgeführt werden. Es gibt jede Menge Ansätze für das Logging, auch an CPAN-Modulen zu dem Thema mangelt es nicht. Jetzt hat sich noch ein weiteres Modul zu der Sammlung hinzugesellt - `Pye`.

Die meisten existierenden Logging-Lösungen arbeiten mit Dateien. `Pye` arbeitet mit *MongoDB*. Das ist der eine große Unterschied. Ein weiterer Unterschied ist, dass es bei `Pye` keine Log-Level gibt. Alle Aufrufe der `log`-Methode führen zu einem Eintrag in der Datenbank. Das Gute bei Pye ist es, dass alles nach einer Session-ID gruppiert wird.

Mit dem Kommandozeilentool `pye` kann man sehr einfach auf die MongoDB-Einträge zugreifen. So bekommt man auch ganz schnell alle Informationen zu einer Session-ID. Der Vorteil hierbei ist, dass die Logeinträge nicht über viele Zeilen in der Logdatei verteilt sind wobei zwischendrin auch noch Logeinträge anderer Sessions vorkommen. Auch von einer Rotation von Logdateien ist das Logging mit `Pye` nicht betroffen.

In Listng 2 ein Beispiel für eine einfache Anwendung in der das Logging mit `Pye` umgesetzt ist.

```
use strict;
use warnings;

use Parallel::ForkManager;
use Mojo::UserAgent;
use Pye;

my $pm = Parallel::ForkManager->new( 4 );

my @urls = qw(
   http://perl.org
   http://perl-services.de
   http://perl-magazin.de
   http://perl-academy.de
);

my $logger = Pye->new;

for my $url ( @urls ) {
    $pm->start and next;
    _find_links( $url, $logger );
    $pm->finish;
}

sub _find_links {
    my ($url, $logger) = @_;

    my $ua    = Mojo::UserAgent->new;
    my $res   = $ua->get( $url );
    my @links =
        grep{$_}map{ $_->attr('href' ) }
        $res->res->dom->find( 'a' )->each;

    $logger->log(
        $$,
        "Found links" => {
            links => \@links,
        },
    );
}
```

Listing 2

Hier werden von mehreren URLs die Links auf der Startseite gesammelt. Damit man unterschiedliche Session-IDs bekommt, wird das Einholen der Startseite und Extrahieren der Links pro URL geforkt. Ein nettes Modul dafür ist `Parallel::ForkManager`. Dann wird der Logger mit

```
my $logger = Pye->new;
```

initialisiert. Dem Konstruktor kann man noch einige Optionen mitgeben. Grundsätzlich kann man Optionen mitgeben, die auch der Konstruktor des Moduls `MongoDB::MongoClient` erlaubt. Setzt man auf dem eigenen Rechner MongoDB mit den Default-Einstellungen ein, muss man auch `Pye` nichts mitgeben. Neben den Parametern für den MongoDB-Client können auch noch folgende weitere Optionen verwendet werden:

```
my $logger = Pye->new(
    log_db       => 'db_fuer_logs',
    log_coll     => 'log_collection',
    session_coll => 'session_collection',
    be_safe      => 0,
);
```

Mit diesen Optionen werden alle Logs in der Datenbank mit dem Namen *db_fuer_logs* gespeichert. Die Information, welche Session es gibt, wird in der Collection *session_collection* gespeichert und die Logging-Nachrichten werden in der Collection *log_collection* abgelegt. Wenn *be_safe* auf einen wahren Wert gesetzt wird, stirbt das Programm wenn beim Einfügen von Lognachrichten in die MongoDB etwas schiefläuft. Standardmäßig ist der Wert aber auf 0 gesetzt.

Das Logging an sich wird dann mit

```
$logger->log(
    $$,
    "Found links" => {
        links => \@links,
    },
);
```

erledigt. Der erste Parameter ist die Session-ID zu der die Logmeldung gespeichert wird. In diesem Beispiel wird einfach die Prozess-ID als Session-ID verwendet. Als nächstes kommt eine einfache Meldung. Und zum Schluss kann man noch eine Hashreferenz mit zusätzlichen Informationen zu der Logmeldung übergeben. Diese Extradaten sind ganz sinnvoll wenn man z.B. zu der Meldung "User kann nicht angelegt werden" noch die Informationen hat, mit welchen eingegebenen Daten es versucht wurde. Sonst kommt man bei der Fehlersuche nicht wirklich weiter.

Über das Logging-Objekt kann man sich auch alle Logmeldungen zu einer Session-ID holen:

```
$logger->session_log( $session_id );
```

Damit bekommt man eine Liste alle Meldungen sortiert nach dem Datum.

Alternativ kann man jetzt auch mit dem Tool `pye` arbeiten:

```
$ pye
Latest sessions logged:
.------------+----------+-------------.
| Date       | Time     | Session ID  |
+============+==========+=============+
| 2014-02-22 | 13:30:23 | 16655       |
| 2014-02-22 | 13:30:23 | 16656       |
| 2014-02-22 | 13:30:22 | 16658       |
| 2014-02-22 | 13:30:22 | 16657       |
| 2014-02-22 | 13:27:02 |             |
'------------+----------+-------------'
```

Und die Meldungen zu einer Session:

```
$ pye -S 16657
2014-02-22, 13:30:22: Found links
   {
      "links" : []
   }

$ pye -S 16658
2014-02-22, 13:30:22: Found links
   {
      "links" : [
         "/kontakt",
         "/impressum",
         "/ueber_uns",
         "/",
         "#",
         "/moose",
         "/perl_critic",
         "/mojolicious",
         "/regex",
         "/firmenschulung",
         "perl_critic",
         "http://perl-magazin.de",
         "/firmenschulung",
         "/kontakt",
         "http://mojolicio.us",
         "http://perl.org"
      ]
   }
```

Capture::Tiny

Hin und wieder steht man vor der Aufgabe, ein externes Programm laufen zu lassen und die Ausgaben auszuwerten. Ein nettes leichtgewichtiges Modul hierfür ist `Capture::Tiny`. Hiermit lassen sich *STDOUT* und *STDERR* in verschiedene Variablen speichern.

So könnte man die Ausgaben der Modulinstallation abfangen:

```
use Capture::Tiny ':all';
my ($out, $err, $exit) = capture {
    system 'cpanm', 'Memory::Stats';
};
```

Der `system`-Befehl sollte bekannt sein. An dessen Stelle kann aber auch beliebiger Perl-Code stehen:

```
use Capture::Tiny ':all';
my ($out, $err, $exit) = capture {
    print "test";
};
```

Hierbei landet das *test* in `$out`. Zu beachten ist allerdings, dass "test" jetzt nicht mehr ausgegeben wird. Möchte man die Ausgabe duplizieren, also einmal wirklich ausgeben und einmal capturen, so kann man `tee_stdout` verwenden:

```
perl -MCapture::Tiny=:all -wE '
> my $out = tee_stdout{ say "test" };
> say "Captured: <<$out>>";
> '
test
Captured: <<test
>>
```

Das gleiche funktioniert auch mit *STDERR*.

Memory::Stats

Welcher Teil der Anwendung verbraucht so viel Speicher? Mit `Memory::Stats` lässt sich das herausfinden. Dabei geht es den tatsächlich gebrauchten physikalischen Speicher (*resident set size*). Das Modul funktioniert zumindest unter Linux und Mac OS X. Uns so sieht es in der Anwendung aus:

```
use Memory::Stats;

my $stats = Memory::Stats->new;

$stats->start;

test1();

$stats->stop;
$stats->report;

sub test1 {
    my $var = '1' x 10_000_000;
}
```

Bei diesem Skript wird folgende Statistik ausgegeben:

```
$ perl community/memory_stats.pl
--- Memory Usage ---
start: 15380480
stop: 25378816 - delta: 9998336 -
                 total: 9998336
--- Memory Usage ---
```

Aber selten will man bei einem solch einfachen Skript den Speicherverbrauch messen. Deshalb kann man auch *checkpoints* setzen. Dann wird der Speicherverbrauch an jedem dieser Zeitpunkte gemessen und am Schluss in der Statistik ausgegeben.

```
use Memory::Stats;

my $stats = Memory::Stats->new;

$stats->start;
$stats->checkpoint( "before 1st test1" );
my $t = test1();
$stats->checkpoint( "before 2nd test1" );
my $u = test1();
$stats->checkpoint( "after 2nd test1" );
$stats->stop;
$stats->report;

sub test1 {
    my $var = '1' x 10_000_000;
}


$ perl community/memory_stats.pl
--- Memory Usage ---
start: 15384576
before 1st test1: 15384576
        - delta: 0 - total: 0
before 2nd test1: 25505792
        - delta: 10121216 - total: 10121216
after 2nd test1: 35508224 - delta: 10002432
        - total: 20123648
stop: 35508224 - delta: 0 - total: 20123648
--- Memory Usage ---
```

Test::Synopsis

Ein ganz nützliches Modul für Modulautoren ist `Test::Synopsis`. Für Programmierer auf der Suche nach tollen Modulen ist die Dokumentation ganz wichtig und um einen ersten Eindruck zu bekommen - oder um die CPAN-News für *$foo* zu schreiben - wird dann der Code aus der *SYNOPSIS* kopiert und laufen gelassen. So hat man z.B. so eine Dokumentation:

```
=head1 NAME

Ein super geniales neues Modul

=head1 SYNOPSIS

  use Geniales::Modul;

  my $object = Geniales::Modul->new;
  $object->tue_etwas( $ein_scalar );
```

Da ist es natürlich hinderlich wenn dieser Code nicht lauffähig ist - wie hier weil die Variable `$ein_scalar` nicht definiert ist. Um das zu vermeiden kann man als Modulautor einfach einen Test einbauen:

```
use Test::Synopsis;
all_synopsis_ok();
```

Damit wird der SYNOPSIS-Code aller Module getestet. Manchmal ist das aber gar nicht gewünscht und es soll nur der Code des Hauptmoduls getestet werden, so ist das auch einfach möglich:

```
use Test::Synopsis;
synopsis_ok('lib/Geniales/Modul.pm');
```

Um dann so ein Problem wie oben gezeigt zu beseitigen kann man einfach in der Dokumentation

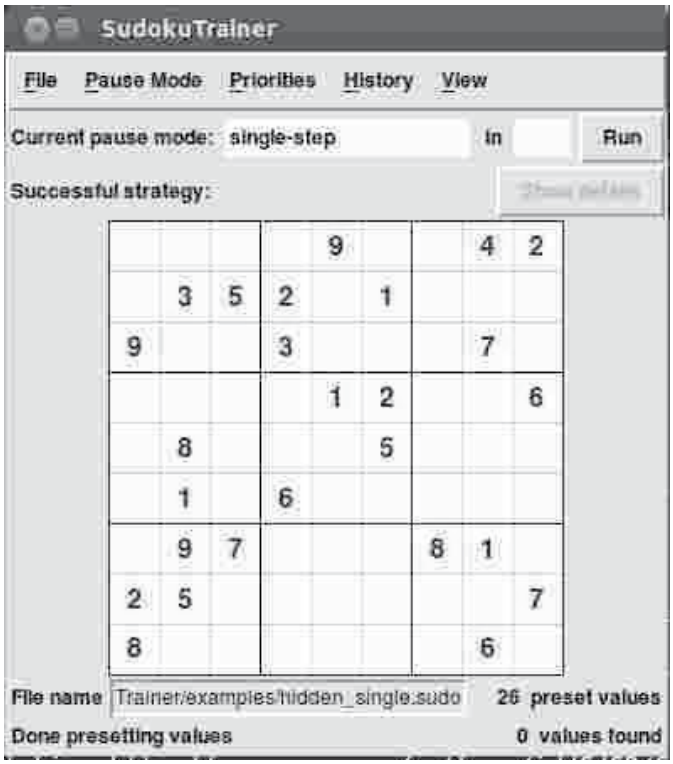
```
=for test_synopsis
my $ein_scalar = 'testwert';
```

eintragen.

SudokuTrainer

Auch wenn der große Hype um Sudokus etwas abgeflaut zu sein scheint, ist es immer noch ein sehr netter Zeitvertreib. Und Perl6-Kenner Moritz Lenz betreibt mit Sudokugarden. de eine Seite auf der man täglich spielen kann. So ganz einfach sind die Rätsel nicht, deshalb kann man seine Spieltaktiken trainieren. Dafür gibt es `SudokuTrainer` auf CPAN.

Damit ist man auch für die nächste Sudoku-Meisterschaft gewappnet...



http://perl-academy.de

Moderne Objektorientierung
Reguläre Ausdrücke für Könner
Webentwicklung mit Mojolicious
Perl::Critic und Programmierrichtlinien