Ausgabe 4/2013 – www.foo-magazin.de - 6,00 €

$foo PERL MAGAZIN

plenv
Eine Alternative zu perlbrew

ZeroMQ
Ein Baustein für verteilte, asynchrone Systeme

Mensch oder nicht Mensch
... Bot oder nicht Bot

Nr 28

16. Deutscher Perl-Workshop

26. bis 28. März 2014

Hannover
Kulturzentrum FAUST

http://act.yapc.eu/gpw2014/

VORWORT

Hackathons - Jugend, Startups, Perl, ...

Hackathons werden immer beliebter. Mittlerweile gibt es für alles und jeden einen Hackathon. In den letzten Wochen und Monaten habe ich an mehreren Hackathons teilgenommen. Alle hatten mit dem Thema „Programmierung“ zu tun, aber keiner war spezifisch für Perl-Programmierer. Es waren wirklich gute Veranstaltungen - sowohl organisatorisch als auch von der Idee her.

Ein Teil versucht, Menschen schon in jungen Jahren für die Informatik zu begeistern. So gab es das CodeCamp im Harz für Studenten und andere junge Menschen und nur wenige Tage später gab es in Berlin „Jugend hackt“ für die 12- bis 18-jährigen. Eine unterstützenswerte Idee, so dass wir diese Veranstaltungen durch Sponsoring gefördert haben.

Ein Großteil der Hackathons haben das Ziel neue Ideen umzusetzen und ggf. auch Startups hervorzubringen. So gibt es extra StartupBootcamps und StartupWeekends usw. Leider steht man da als Perlaffiner Mensch relativ alleine da...

Aber auch in der Perl-Welt gibt es etliche Hackathons. Im Gegensatz zu den Veranstaltungen mit großen Firmen gibt es dort keine Preise und häufig geht es nicht darum etwas Neues zu schaffen, sondern Bestehendes zu verbessern.

Anfang Dezember gibt es die zweite Ausgabe eines Hackathons, den es in Zukunft regelmäßig geben soll: „patch -p1“. In Paris werden an drei Tagen (06.-08. Dezember) Perl-Projekte bearbeitet.

Allen Hackathons, die ich kenne, ist gemein, dass sie offen sind für alle interessierten Personen. Das bietet die Gelegenheit zu einem Austausch mit Personen und mit diesen Personen ein Projekt zu bearbeiten. Das kann auch zu einem Blick über den Tellerrand führen und auch dazu, mögliche „Trends“ zu erkennen. So war bei „Jugend hackt“ und bei dem DeveloperGarden/Box.com-Hackathon das Thema „Finden von Wohnung bzw. lebenswerte Wohngegend“ in mehreren Projekten zu finden.

Die abschließenden Fragen an die Leser: Sind Hackthons für Sie interessant? Würden Sie gerne mal an einer solchen Veranstaltung teilnehmen? Welche Themen sollten dabei behandelt werden? Sollte Perl besser vertreten sein bei den nicht-spezifischen Hackathons? Wie kann man das erreichen?

Über Meinungen und Antworten würde ich mich freuen. Entweder an feedback@perl-magazin.de oder auf Twitter mit dem Hashtag #foohackathon

Wer auf der Suche nach einem passenden Hackathon ist, sollte mal auf hackerleague.org vorbeischauen.

Viel Spaß beim Lesen der 28. Ausgabe des Perl-Magazins,

Renée Bäcker

Die Codebeispiele können mit dem Code

pmcri5

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/issue28.

IMPRESSUM

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

INHALTSVERZEICHNIS

ALLGEMEINES

PERL

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

Herbert Breunung

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

Boris Däppen

Boris Däppen lernte Perl im Umfeld der Finanzdienstleister in Zürich kennen und vertiefte seine Kenntnisse der Sprache später bei perl-services.de. Er hat kürzlich als erster Absolvent den neuen Master „Technik und Philosophie" an der TU Darmstadt abgeschlossen und arbeitet nun als Freelancer in der Schweiz.

Thomas Fahle

Thomas Fahle, Perl-Programmierer und Sysadmin seit 1996.
Websites:

- http://www.thomas-fahle.de
- http://Perl-Suchmaschine.de
- http://thomas-fahle.blogspot.com
- http://Perl-HowTo.de

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.

Thomas Klausner

Just in case you like to know, I'm currently full-time father of 2 teenagers, half-time Perl hacker, sort-of DJ, bicyclist, no longer dreadlocked and more than 34 years old and not only too lazy to update my profile once a year but also to translate this bio into German.

Wolfgang Kinkeldei

Net::OpenSSH

ssh und Perl

`ssh` ist das übliche Werkzeug, um sichere Verbindungen zu entfernten Rechnern aufzubauen, Dateien auszutauschen oder Tunnel für TCP-Verbindungen zu erstellen. Dazu gehören auch einige ausführbare Programme wie `scp` oder `sftp`.

Da diese Werkzeuge auf *nix Plattformen als binär ausführbare Programme zur Verfügung stehen, genügt prinzipiell ein `system` Aufruf, um in den Genuss der Funktionalität der Programme zu gelangen.

```
# Dateien auf dem entfernten System
# auflisten:
system 'ssh', 'user@hostname.domain',
  'ls', '-l', '/verzeichnis';
say $?; # 0 wenn kein Fehler

# Eine Datei auf das entfernte System
# kopieren:
system 'scp', '/lokal/datei.txt',
  'user@hostname.domain:/verzeichnis/';
say $?; # 0 wenn kein Fehler
```

So weit, so gut. Solange nichts schief geht, eventuelle Ausgaben der Kommandos auf dem Standard-Ausgabe oder -Fehler Kanal keine Rolle spielen, kann man so arbeiten. Doch leider gibt es eine Reihe von Unschönheiten bei diesem Ansatz:

• Jedes einzelne abgesetzte Kommando führt eine erneute Authentifizierung beim entfernten Rechner durch, weswegen zum Teil unangenehme Laufzeiten entstehen können oder im schlimmsten Fall jedes Mal eine Passwort-Eingabe notwendig wird.

• Jede Operation muss als ssh/scp Kommando zusammengestellt werden, schön wäre es, wenn die relevanten Inhalte der Kommandozeilen irgendwie gekapselt werden könnten.

• Das Abfangen von Fehlern und eventuell Fehlermeldungen zur späteren Weiterverarbeitung wäre nett.

• Zumindest bei der Benutzung von `system` können keine Daten auf dem Standard-Eingabe Kanal des entfernt laufenden Prozesses gelegt werden.

ssh auf CPAN

Perl ist nicht nur die Sprache mit den vermutlich meisten Web Frameworks, sondern es gibt auch eine Menge an ssh Abstraktionen. Ein paar davon sind:

• Net::OpenSSH
• Net::SSH
• Net::SSH::Perl
• Net::SSH::Any
• Net::SSH::Expect
• Net::SSH::Tunnel
• Net::SSH::W32Perl
• Net::SSH2
• Net::SSH2::Expect
• IPC::PerlSSH

Jedes einzelne dieser Module hat seine Vor- und Nachteile oder ist für bestimmte Aufgaben besonders gut einsetzbar. Der Autor von `Net::OpenSSH` hat sich die Mühe gemacht, die meisten der oben genannten gegeneinander zu vergleichen. Bei ihm erntet jedes Kritik, was sein persönlicher Grund war, `Net::OpenSSH` zu entwickeln.

Der für mich persönlich ausschlaggebende Grund, `Net::OpenSSH` einzusetzen, war dessen Performance. 100 ssh Verbindungen sind gegenüber dem direkten ssh-Aufruf aus ei-

ner Shell bei diesem Benchmark um Faktor 10 schneller. "v-webserver" soll einen Webserver in der DMZ unserer Firma sein. Der Aufruf erfolgte von meinem Arbeitsplatz-Rechner aus.

```
$ time for ((i=0; i<100; i++));
  do ssh v-webserver true; done

real   0m50.627s
user   0m0.917s
sys    0m0.508s

$ time perl -MNet::OpenSSH \
    -e '$ssh = Net::OpenSSH->
      new("v-webserver");
        $ssh->system("true") for 1..100'

real   0m4.950s
user   0m0.495s
sys    0m0.473s
```

Was passiert hier? Warum ist `Net::OpenSSH` deutlich schneller? `Net::OpenSSH` verwendet für Verbindungen zum entfernten SSH-Server den sogenannten Slave-Modus. Technisch gesehen wird hierbei lediglich eine TCP-Verbindung zum entfernten Rechner geöffnet und die ssh-Verbindung inklusive Authentifizierung darüber hergestellt. Jedes einzelne abgesetzte Kommando startet zwar lokal und entfernt die notwendigen ssh- und Kommando-Prozesse, nutzt aber die bereits bestehende ssh-Verbindung ohne eine weitere Authentifizierung ausführen zu müssen. Je nach Latenz im Netzwerk können die Geschwindigkeitssteigerungen mehr oder weniger groß sein. Erfahrungsgemäß liegt die Steigerung im Bereich von Faktor 5 bis Faktor 20.

Der Slave-Modus steht seit OpenSSH Version 4.1 zur Verfügung, welches auf Mai 2005 zurückzudatieren ist. Auf den meisten heute eingesetzten Systemen muss man sich damit keine Sorgen machen

Beispiel

```
#!/usr/bin/env perl
use strict;
use warnings;
use Net::OpenSSH;

# Verbindung aufbauen
my $remote = Net::OpenSSH->new(
  'me:secret@server.de');

# Standard Ausgabe zeilenweise abholen
my @output = $remote->capture
  ('sudo grep "snmpd" /var/log/syslog');

# Datei auf den entfernten Rechner kopieren
 $remote->scp_put('/local/file',
   '/remote/dir_or_file');
```

Was bietet Net::OpenSSH alles?

Verbindungsaufbau

Zum Aufbau einer Verbindung kann wahlweise ein sprechender String mit Benutzer, Passwort, Host und Port oder einzelne Argumente in Hash-Schreibweise übergeben werden.

Zusätzliche SSH-Optionen sind möglich. Außerdem kann das Passwort für den entfernten Rechner oder den lokal hinterlegten Schlüssel angegeben oder abgefragt werden.

Fehlerbehandlung

Die meisten Methoden von `Net::OpenSSH` liefern einen wahren Wert zurück, wenn die Ausführung erfolgreich war. Im Fehlerfall kann über die Methode `error` der Fehler der letzten Operation erfragt werden.

```
$remote->execute('kommando', @argumente)
    or die "execute Fehler: " .
     $remote->error;
```

Ausführung von entfernten Kommandos

Im einfachsten Fall lassen sich Kommandos auf der entfernten Seite ausführen. Die Methode `system` verhält sich wie sein lokales Äquivalent, sollte lediglich der positive Ausgang (exit Status: 0) eines Kommandos interessant sein, so kann die Methode `test` verwendet werden. Diverse Varianten von `capture` sowie `pipe_in`, `pipe_out` und das Supertalent `open3` stehen zur Verfügung, um Daten von und zu den Standard Ein- oder Ausgabekanälen zu schleusen.

```
my ($in, $out, $err, $pid) =
    $remote->open3('kommando', @argumente)
        or die "open3 Fehler: "
          . $remote->error;

# beliebige viele Dinge an den entfernten
# Prozess senden
 print $in "Irgendwas...\n";
 close($in);

# warten bis der entfernte Prozess
# beendet ist
waitpid $pid, 0;
```

Dateien kopieren

Hier hat man die Wahl zwischen `scp_get` bzw. `scp_put`, um einzelne Dateien mit dem entfernten Rechner zu tauschen. Für ganze Verzeichnisse stehen `rsync_get` und `rsync_put` zur Verfügung. Wer innerhalb einer Sitzung viele Dateien austauschen will, kann auch `sftp` zum Einsatz bringen. Hier ein Beispiel, mit dem gleichzeitig eine Bandbreiten-Limitierung auf 1 MBit/s gesetzt wird.

```
$remote->scp_get({ bwlimit => 1000 },
  '/etc/passwd', '/tmp/passwd')
   or die "scp Fehler: " . $remote->error;
```

Erwähnenswert ist noch, dass einige der erwähnten Kommandos auch asynchron (durch die Option `{async => 1}` verwendet werden können. In solchen Fällen wird immer die Prozess-ID des lokal gestarteten SSH Prozesses zurückgeliefert.

Erweiterungen

MooseX::Role::Net::OpenSSH

eine Moose-Rolle, die die Attribute `ssh`, `ssh_hostname`, `ssh_username` sowie `ssh_options` mitbringt. Die SSH-Verbindung wird beim ersten lesenden Zugriff auf das Attribut `ssh` aufgebaut und bleibt für die Lebensdauer der diese Rolle konsumierenden Klasse erhalten.

Net::OpenSSH::Compat

bietet eine Reihe von Adaptern, mit denen das Verhalten diverser anderer SSH-Module emuliert wird.

Net::OpenSSH::Parallel

erlaubt die parallele Ausführung zahlreicher Kommandos auf einer Reihe verschiedener Rechner.

Einschränkungen

Wer sich für `Net::OpenSSH` entscheidet, kann lediglich Verbindungen mit dem SSH Protokoll Version 2 zum entfernten Rechner aufbauen. Aufgrund zahlreicher Verwundbarkeiten der älteren Version dürfte Version 2 heute der Standard sein.

Das erforderliche OpenSSH in Version 4.1 existiert seit Mai 2005. Wer nicht gerade extrem betagte Server manipulieren muss, wird damit sicher kein Problem haben.

`Net::OpenSSH` unterstützt kein Windows. Für manche dürfte das vermutlich die einzige Einschränkung sein, die schwerwiegend ist.

Boris Däppen

Sicheres Deployment mit Pinto

Je länger und tiefgreifender uns IT-Produkte im Alltag begleiten, desto mehr wächst der Druck Software mit guter Qualität zu liefern. Ein zentraler Punkt von guter Software besteht nun nicht nur in einem ansprechenden Design für den Benutzer, sondern liegt zu einem bedeutenden Teil auch in der Stabilität und Verfügbarkeit des durch die Software offerierten Dienstes. Je mehr wir im Alltag mit Software zu tun haben, desto Ausfallsicherer muss sie sein, da unsere Handlungen und Planungen auf das Funktionieren dieser Software setzen. Bei den heute üblichen vernetzten Client- und Server-Architekturen betrifft die Gewährleistung des Dienstes somit nicht nur den für den Kunden sichtbaren Teil (GUI/Frontend), sondern im Besonderen gerade das Backend, da dieses für die eigentliche Abwicklung verantwortlich ist. Mit Perl und CPAN lassen sich nun hervorragend Backends schreiben oder unterstützen. Dieser Artikel widmet sich daher der Frage, wie man mit Perl den heutigen Anforderungen der Kunden betreffend Softwarequalität gerecht werden kann. Hierbei werfen wir einen Blick auf *Pinto* [1] von *Jeffrey Ryan Thalhammer* und dem damit in Zusammenhang stehenden Webservice *Stratopan* [2].

Grundvoraussetzungen für gutes Testen

In Hinsicht auf die hohen Anforderungen betreffend der Ausfallsicherheit und Qualität integrieren die meisten Software-Entwicklungsprozesse ausgiebige Testphasen. Im September schreibt Heise unter Verweis auf den *World Quality Report*: "Die Ausgaben für Software-Testing und Qualitätssicherung machen mittlerweile 23 Prozent der weltweiten IT-Budgets aus. Der durchschnittliche Anteil der Ausgaben ist damit 5 Prozent höher als noch im Vorjahr" [3]. Die Qualität welche durch Testen sichergestellt wird, ist nicht unbedingt die Qualität des Codes selbst! Eine Funktion kann unterschiedlich implementiert sein oder sogar Fehler enthalten. Solange sie den Anforderungen genügt, erfüllt sie aber den Test, da für ihn die Software als Blackbox - das heißt nur in Betrachtung der Ein- und Ausgaben - bewertet wird. Wenn in diesem Artikel von Qualität gesprochen wird, dann ist damit lediglich die Sicherheit gemeint, zu Wissen dass die Software den Erwartungen entsprechend reagiert. Diese Art von "Qualität" kann auch auf lausig programmierte Software zutreffen.

Erhöhte Test-Bemühungen zahlen sich nur aus, wenn sie in dem Umfeld stattfinden, in welchem dann später auch die Produktion läuft. Nur dann können die Tests überhaupt verlässliche Ergebnisse liefern. Denn es ist eben nicht mit Sicherheit voraussagbar welche Komponente welchen Test wie beeinflusst. Hier kann es immer Überraschungen geben. Dem Kunden ist später nicht geholfen wenn man ihm erklärt, dass die Software *theoretisch* hätte funktionieren müssen. Der Test muss eben gerade die *eigentliche* Situation erfassen, und nicht eine *theoretische*.

Wer ein größeres Backend in Perl schreibt, kann seine Produktivität erheblich steigern wenn er hierfür auf ausgewählte Module von CPAN zugreift. Software-Module anderer Programmierer oder ganzer Communities können viele bekannte Probleme einfach lösen und sind meist besser als selbst Geschriebenes. Benutzt man allerdings solche Module, verliert man schnell die Übersicht über den Code welchen man in seinem Produkt einsetzt. Es entsteht das Problem, dass man die Module managen muss, was aber allzuoft unterlassen wird.

Es soll Unternehmen geben welche Perl einsetzen, aber - aus Angst vor "fremdem" Code - den Einsatz von CPAN untersagen. Wir werden im Verlauf des Artikels aber sehen, dass

sich durch gute Testvoraussetzungen dieser "fremde" Code durchaus als vertrauenswürdig betrachten lässt. So können CPAN-Module und die damit verbundenen Vorteile auch in sicherheitsbewussten Unternehmen Anwendung finden.

Software auf CPAN kann nicht als "stable" betrachtet werden. Wie ein Modul jeweils gepflegt wird, ist von Autor zu Autor verschieden. Blindes Vertrauen auf CPAN-Module ist nicht zu empfehlen. Selbst dann nicht wenn nur CPAN-Module verwendet werden, welche sehr zuverlässige Autoren haben. Denn auch wenn Modul-Updates immer mit aller Sorgfalt durchgeführt werden, so weiß doch keiner wie ein Update genau mit *deiner* Software interagieren wird.

Besonders drängend wird das Problem wenn das Produkt auf mehreren verschiedenen Umgebungen läuft. Die üblichen Perl-Installationen bieten hier keinen Mechanismus welcher sicherstellt, dass auf allen Maschinen die gleichen Module installiert sind. Wenn ich heute eine Maschine aufsetze und hierbei die Module von CPAN installiere, kann die Maschine, welche ich morgen aufsetze, schon anders aussehen, da inzwischen ein Modul auf CPAN ein Update erhalten haben könnte. Mit der Zeit können sich so immer mehr Unterschiede einschleichen, bis schlussendlich Entwickler-, Test-, Produktions- und Kundenmaschinen ein ziemlich zufälliges Setup an Softwaremodulen aufweisen.

Lösungen hierzu gibt es schon länger. Abhilfe schafft ein explizites Verwalten aller Versionsnummern, verbunden mit deren jeweiliger Installation. Manuell durchgeführt bringt dies einige praktische Problemen mit sich. Die Installation von Abhängigkeiten wird schnell unübersichtlich. Da die Module auf CPAN meistens auch von anderen abhängen, sieht man sich sehr schnell einer ganzen Kette von Modulen gegenüber, deren Zahl sehr schnell die 100 überschreiten kann. Ein Lösungsansatz bietet das Betreiben eines eigenen Repositories für die Module. Auf diese Weise hat man eine gewisse Kontrolle darüber was man installiert. Hier würde ein Repository helfen, welches extra auf den Zweck zugeschnitten ist einen stabilen Zustand in die verwendeten Module zu bringen. Und genau dieser Anforderung versucht *Thalhammer* mit Pinto gerecht zu werden.

Es gibt bereits verschiedene Lösungen in Perl ein eigenes Repository aufzusetzen. Pinto wirbt damit mehr zu bieten als die üblichen bekannten Lösungen, was auch der Grund ist, weshalb wir es hier exklusiv anschauen. In der Dokumentation sind die wichtigsten Merkmale aufgelistet [4]:

- Unterstützung für verschiedene sogenannte "Stacks". Dies kann z.B. zur Unterscheidung zwischen Produktion und Entwicklung verwendet werden.

- Möglichkeit zur Blockierung einer spezifischen Modulversion mittels "Pin". So erfährt dieses Modul auch dann kein Upgrade wenn eine andere Abhängigkeit dies ausgelöst hätte.

- Versionskontrolle bezüglich der Änderungen an den verwalteten Modulen.

- Die Möglichkeit von beliebigen Quellen Module einzuspielen.

- Unterstützung von "Team Development".

- Robuste Kommandozielen-Schnittstelle.

- Erweiterbarkeit.

Hinzuzufügen ist auch die hervorragende Dokumentation und die Möglichkeit zur Nutzung von Pinto als Webservice unter https://stratopan.com/. Hierzu später mehr.

Installation von Pinto

Für die Installation gibt es einiges zu Beachten. Dies liegt aber nicht an Pinto sondern ist viel mehr der Natur der Sache (und Pintos voraussichtigem Umgang damit) geschuldet. Der Perlprogrammierer ist dazu geneigt sich die Software gleich wie üblich mit einem CPAN-Client zu installieren. Doch hier ist Vorsicht geboten! Pinto selbst hat viele Abhängigkeiten, ist aber gerade dazu da, Probleme mit dem Management von Abhängigkeiten zu lösen. Wer Pinto in seine gewohnte Umgebung installiert, riskiert, dass Pinto genau in die Probleme rennt, die es eigentlich lösen soll. Auch Pinto ist kein "Münchhausen" und kann sich daher nicht selbst am Schopf packen und aus dem "Sumpf" ziehen. Für die Installation steht unter `Pinto::Manual::Installing` eine ausführliche Anleitung zur Verfügung. Demnach ist es das beste Pinto mitsamt all seinen Abhängigkeiten in ein eigenes Verzeichnis zu installieren. Hierfür steht im Web ein Skript zur Verfügung:

```
wget -O -
  http://getpinto.stratopan.com | bash
```

Das Skript installiert (per Default) Pinto an folgenden Ort:

```
$HOME/opt/local/pinto
```

Um die mit Pinto kommenden Kommandos in der Shell nutzen zu können, müssen sie geladen werden:

```
source $HOME/opt/local/pinto/etc/bashrc
```

Dieses Konzept ist dem "Perler" bereits von *Perlbrew* her bekannt. Am besten schreibt man den Befehl in die Datei `~/.bashrc` (oder ein Äquivalent dazu), damit die Kommandos automatisch zur Verfügung stehen.

Auf diese Weise ist Pinto nun lokal installiert, was für unser Szenario hier vorerst reicht. Pinto lässt sich aber auch als Serverdienst betreiben. Hierfür ist ebenfalls eine Installationsanleitung vorhanden.

Ein Projekt mit Pinto aufsetzen

Wer mit Pinto starten möchte kann dies mit einem einfachen Kommando tun:

```
pinto -r MeinProjekt init
```

Dies erstellt einen Ordner "MeinProjekt" und legt dort eine Dateistruktur an, welche von Pinto benötigt wird. Man kann diesen Ordnern jederzeit löschen und das Kommando so rückgängig machen.

Später lässt sich z.B. mit

```
pinto -r MeinProjekt list
```

diesen Ordner gefiltert anschauen.

Allerdings erzeugt der Befehl im Moment noch keine Ausgabe, da an dem Projekt noch nichts gemacht wurde. Ersichtlich wird aber bereits die generelle Struktur der Kommandozeilen-Argumente. Die Option `-r` heißt ausgeschrieben `--root` und muss `immer` angegeben werden. Pinto muss eben wissen auf welchen Daten es operieren soll. Die Option lässt sich aber als Umgebungsvariable auslagern. Wer `PINTO_REPOSITORY_ROOT` setzt, kann sich die Option auf der Kommandozeile sparen:

```
export PINTO_REPOSITORY_ROOT=$HOME/
  MeinProjekt
```

Nun reicht es aus als Befehl z.B. `pinto list` zu schreiben, was doch um einiges angenehmer und weniger anfällig für Tippfehler ist.

Wenn wir schon bei den Umgebungsvariablen sind: Zum aktuellen Zeitpunkt (Ende September) sollte dafür gesorgt sein, dass eine Umgebungsvariable `EDITOR` gesetzt ist, da Pinto sonst bei den kommenden Kommandos mit Fehler abbricht. Da dies bei vielen Linux-Distributionen nicht per Default der Fall ist, muss dies manuell eingerichtet werden. Pinto braucht einen Editor um die Commit-Messages (analog zu Git) entgegen zu nehmen. Hierzu ist aber bereits ein Bugreport eröffnet. In Zukunft kann Pinto den Editor evtl. selber finden, oder der Schritt ist in der Installationsanleitung vermerkt. Für den Zeitpunkt dieses Artikels muss also noch folgendes gemacht werden:

```
export EDITOR=vi
```

Nun sind alle Schritte getan: Ich kann die ersten Module in das Repository einspielen. Dies ist denkbar einfach. Mit diesem Kommando installieren wir z.B. das Modul `JSON`:

```
pinto pull JSON
```

Nun erzeugt auch der Befehl `list` endlich eine Ausgabe. Wie zu sehen ist, wurde das Modul installiert.

```
pinto list
[rf-] JSON                          2.59
   MAKAMAKA/JSON-2.59.tar.gz
[rf-] JSON::Backend::PP               0
   MAKAMAKA/JSON-2.59.tar.gz
[rf-] JSON::Boolean                   0
   MAKAMAKA/JSON-2.59.tar.gz
[rf-] JSON::PP5005                 1.10
   MAKAMAKA/JSON-2.59.tar.gz
[rf-] JSON::PP56                   1.08
   MAKAMAKA/JSON-2.59.tar.gz
[rf-] JSON::backportPP::Boolean    1.01
   MAKAMAKA/JSON-2.59.tar.gz
```

Der Befehl `pull` installiert per Default alle Abhängigkeiten mit! Auf diese Art und Weise ist ein Projekt schnell aufgesetzt, selbst wenn es insgesamt auf hunderte von Modulen angewiesen ist. Einfach die direkt benutzten Module angeben, der Rest installiert sich von selbst in das Repository von Pinto:

```
pinto pull Dancer JSON YAML
```

Es lassen sich auch problemlos lokale Archive einbinden. Bedingung ist lediglich, dass dieses als Perl-Modul gepackt vorliegen. Hierzu wird dann der Befehl `add` verwendet:

```
pinto add ./Mein-Modul-0.1.tar.gz
```

Auch hier werden die Abhängigkeiten per Grundeinstellung gleich mit installiert.

Wer also seine Applikation *komplett* mit Pinto managen möchte, muss die Applikation selbst als Paket zur Verfügung stellen. Oft ist aber eine Applikation zu komplex um sie als einfaches Perl-Modul zu packen. So spielen meist noch andere Komponenten eine Rolle: Datenbanken, Webserver, Betriebssysteme, Skripte, Cronjobs, Webdienste, etc. In diesem Fall lassen sich alternativ nur die Perl-Abhängigkeiten mit Pinto managen um damit dann verschiedene Instanzen der Software aufzusetzen.

Eine einfache Applikation als Beispiel

Im Folgenden soll der komplette Prozess zum Aufsetzen eines Pinto-Repositories für eine Applikation durchgespielt werden. Hierzu verwenden wir eine kleine Applikation, welche exemplarisch für den Rest des Artikels verwendet werden soll. Es handelt sich um einen Webservice, welcher `JSON` entgegennimmt und als `YAML` formatiert zurückgibt: Nennen wir das Ganze `json2yaml.app`. Folgender Code soll hierfür als Beispiel ausreichen:

```
use Dancer qw( get dance params);
use JSON   qw( decode_json );
use YAML   qw( Dump );

get '/:json' => sub {
    Dump(decode_json(
        params->{json}
    ));
}; dance;
```

Der Code liegt in der Datei `json2yaml.app.pl` [5] und lässt sich nun wie folgt starten:

```
perl json2yaml.app.pl
>> Dancer 1.3118 server 7793
  listening on http://0.0.0.0:3000
== Entering the development dance floor ...
```

Als Funktionstest können wir z.B. mit `curl` auf der Kommandozeile einen Aufruf tätigen. Wir wollen z.B. den JSON-String `{"foo":"bar"}` übergeben, was als URL codiert als `%7B%22foo%22%3A%22bar%22%7D` dargestellt wird. Und tatsächlich, das Programm gibt wie erwartet `YAML` aus:

```
curl http://0.0.0.0:3000/
  %7B%22foo%22%3A%22bar%22%7D
---
foo: bar
```

Nehmen wir nun also an, dies sei unsere "komplexe" Applikation, dessen Perl-Abhängigkeiten wir in den Griff bekommen möchten, damit die Testergebnisse auch für die Produktion Geltung finden können.

Aufsetzen der Beispiels-Applikation

Von der Einführung her wissen wir bereits was zu tun ist. Wir müssen ein Pinto-Repository aufsetzten und dort die Abhängigkeiten installieren:

```
export PINTO_REPOSITORY_ROOT=$HOME/webapp
pinto init
pinto pull Dancer JSON YAML
```

Dies ist bereits alles was es braucht. Zur Kontrolle lässt sich mit `list` die Ausgabe anzeigen. Hier die Ausgabe mit Hilfe von `cut` und `sort` auf das Wichtige gefiltert:

```
pinto list | cut -d/ -f2 | sort -u
Dancer-1.3118.tar.gz
Encode-Locale-1.03.tar.gz
File-Listing-6.04.tar.gz
HTML-Parser-3.71.tar.gz
HTML-Tagset-3.20.tar.gz
HTTP-Body-1.17.tar.gz
HTTP-Cookies-6.01.tar.gz
HTTP-Daemon-6.01.tar.gz
HTTP-Date-6.02.tar.gz
HTTP-Message-6.06.tar.gz
HTTP-Negotiate-6.01.tar.gz
HTTP-Server-Simple-0.44.tar.gz
HTTP-Server-Simple-PSGI-0.16.tar.gz
IO-HTML-1.00.tar.gz
JSON-2.59.tar.gz
libwww-perl-6.05.tar.gz
LWP-MediaTypes-6.02.tar.gz
MIME-Types-2.04.tar.gz
Module-Runtime-0.013.tar.gz
Net-HTTP-6.06.tar.gz
Test-Deep-0.110.tar.gz
Test-NoWarnings-1.04.tar.gz
Test-Tester-0.109.tar.gz
Try-Tiny-0.18.tar.gz
URI-1.60.tar.gz
WWW-RobotRules-6.02.tar.gz
YAML-0.84.tar.gz
```

Wir sehen: Alle direkten und indirekten Abhängigkeiten der Applikation stehen nun unter der Verwaltung von Pinto. An dieser Stelle ergibt sich gleich eine Gelegenheit nochmals auf die gute Dokumentation von Pinto hinzuweisen. Ein Blick in `App::Pinto::Command::list` zeigt, dass eine eigene Option für die Formatierung von `list` zur Verfügung steht. Damit kann obige Ausgabe auch ohne die Pipe in `cut` erledigt werden:

```
pinto list --format %D | sort -u
```

Deployment der Applikation

Die Modul-Abhängigkeiten für `json2yaml.app` sind nun komplett in Pinto verwaltet. Hierbei ist es vorerst nebensächlich welche Version die jeweiligen Module haben (im Moment ist dies ja die aktuell auf CPAN veröffentlichte Version). Wichtig ist lediglich, dass die Entwicklung, das Testing und auch das Deployment im selben Zyklus mit dieser Version arbeiten.

Nun gilt es, eine Instanz der Applikation aufzusetzen. Pinto bringt hierfür ein eigenes `install` Kommando mit sich. Da ich aber gerne die Tools verwende die ich in dem Zusammenhang immer benutze nehme ich für das Beispiel `cpanm`. Ich verweise meinen CPAN-Client einfach auf das lokale Pinto-Archiv:

```
cpanm           \
  -L src_prod   \
  --mirror file://$HOME/webapp \
  --mirror-only \
    Dancer JSON YAML
```

Mit `-L src_prod` sage ich, dass die Module in das Verzeichnis `src_prod` installiert werden sollen. Die Option `--mirror-only` garantiert mir, dass wirklich nichts direkt von CPAN geholt wird.

Auf diese Weise lässt sich nun zu jeder Zeit bei der Installation sicherstellen, dass die richtigen (weil getesteten) Module zur Installation kommen.

Management des Repositories

Nun ist es so, dass Pinto erst hier anfängt seine Stärke langsam ins Spiel zu bringen. Pinto belässt es eben gerade nicht dabei, lediglich ein Repository anzubieten, sondern liefert die Tools, dieses dann für den Betrieb zu verwalten. Die wichtigsten Punkte sollen hier kurz Erwähnung finden, eine komplette Hinführung mit Beispielen würde aber den Artikel sprengen. Hier ist dem interessierten Leser mit der detaillierten Dokumentation und den Tutorials aber gut geholfen.

Stacks

In dem Beispiel wurde die Applikation aufgesetzt und kann nun in den Betrieb gehen. Was aber, wenn nun auf CPAN ein Modul in einer neuen Version erscheint und dies integriert werden soll? Wie können verschiedene Zyklen der Entwicklung abgebildet werden? Hierfür bieten sich Stacks an. Der aktuell verwendete Stack heißt `master` (analog zu Git). Von diesem Stack lässt sich nun eine Kopie erstellen (die Option `-r` haben wir nach wie vor über die Umgebungsvariable definiert):

```
pinto copy master develop
```

Nun existiert der Modul-Stack zwei Mal: Unter dem Namen `develop` wurde eine Kopie angelegt. Der `master` kann weiterhin für die Produktion verwendet werden. Auf dem Entwicklungs-Stack werden die neuen Module installiert:

```
pinto pull --stack develop Dancer~1.3118
```

Der so veränderte Stack kann mit demjenigen der Produktion verglichen werden:

```
pinto diff master develop
```

Auch sind alle Änderungen jederzeit nachvollziehbar:

```
pinto log --stack develop
```

Soll eine Umgebung mit dem neuen Repository installiert werden, kann der Stackname einfach an die URL angehängt werden:

```
cpanm           \
  -L src_prod   \
  --mirror file://$HOME/webapp/stacks/
    develop \
  --mirror-only \
    Dancer JSON YAML
```

So kann sich eine Installation aus dem Entwicklungs-Stack bedienen um die Konfiguration zu testen.

Pins

Mit dem Befehl `pin` können einzelne Module "festgepinnt" werden, was ihr Versionsnummer angeht. Dieses Modul kann dann kein Update mehr erfahren. Der Gegenbefehl hierzu ist `unpin`. Mit `pinto pin YAML` wird z.B. verhindert, dass das Modul `YAML` versehentlich ein Update erfährt. Dies macht dann Sinn, wenn bekannt ist, dass eine neuere Version Probleme verursachen würde. Wenn die Probleme mit dem Modul behoben sind lässt sich eine Kopie des Stacks erstellen - wobei die Pins mitgenommen werden. Auf der Kopie kann der Pin dann gelöst werden (mit `unpin`) und das Verhalten kann getestet werden. Natürlich kann mit `add` jederzeit auch ein lokal gepatchtes Modul nachgereicht werden um aktuelle Probleme zu beheben.

Stratopan: Pinto "al Dente"

Pinto hilft einem seine Abhängigkeiten zu verwalten. Allerdings hat die Sache einen Preis. Zusätzlich zur eigenen Software muss nun auch noch ein zusätzliches Produkt gewartet werden: Pinto. Dieser Faktor ist nicht zu unterschätzen. Es gibt aber die Möglichkeit den Service von Pinto als Dienstleistung zu beziehen. Hierzu steht unter *stratopan.com* ein Service zur Verfügung (ist u.U. noch in der Beta-Phase).

Stratopan bietet die komplette Funktionalität von Pinto mit einem bequemen und gewarteten GUI. Repositories können angelegt und Ressourcen verwaltet werden. Stacks, Pins, kopieren: alle Funktionen sind vorhanden. Links zu den Repositories werden automatisch generiert. Repositories können hierbei privat oder öffentlich sein.

Für das hier erstelle Programm `json2yaml.app` habe ich unter meinem Namen ein öffentliches Repository angelegt [6]. Mit Hilfe dieses Repositories kann die Software nun aufgesetzt werden. Ein Verzeichnis erstellen (`mkdir src_prod`) und die Module darin installieren:

```
cpanm           \
  -L src_prod   \
  --mirror https://stratopan.com/
    borisdaeppen/webapp/prod \
  --mirror-only \
    Dancer JSON YAML
```

Die Applikation herunterladen:

```
wget https://raw.github.com/borisdaeppen/
  foo-PerlMagazin-Examples/master/28/
  json2yaml.app.pl
```

Die Applikation starten:

```
perl -I src_prod/lib/perl5 json2yaml.app.pl
```

Und gegebenenfalls den `curl`-Aufruf von vorhin ausführen um die Applikation zu testen.

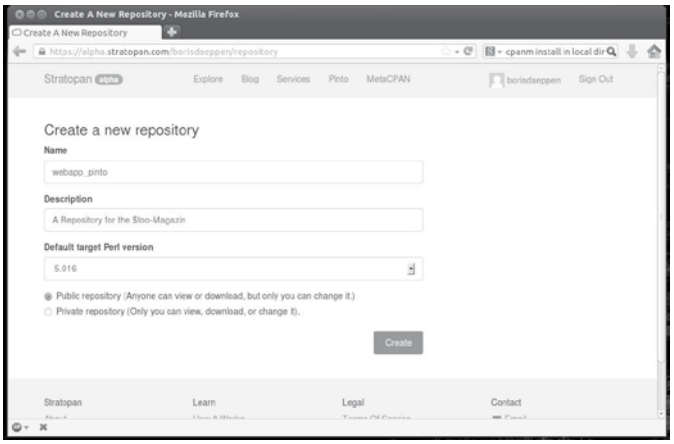


Abbildung 1: Dialog bei Stratopan für das Anlegen eines euen Projekts.

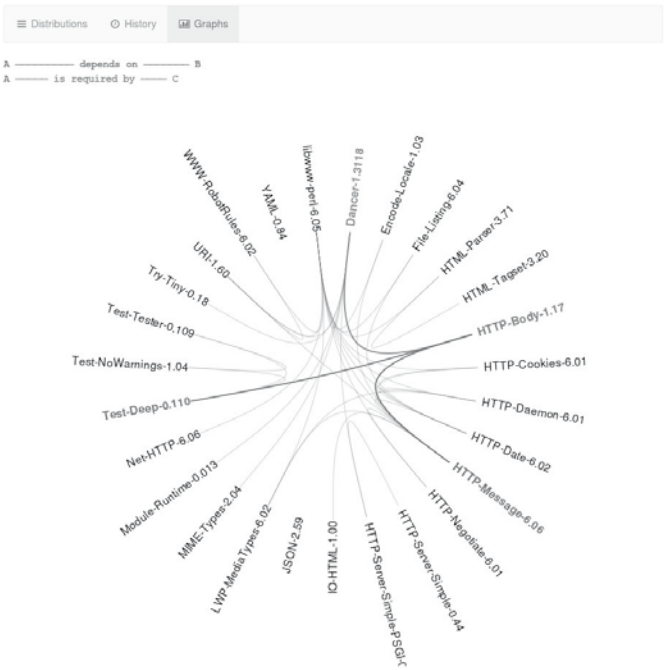


Abbildung 2: Grafik von Stratopan zur Visualisierung der Abhängigkeiten

Hilfe

Ein großer Vorteil von Pinto besteht in der - im Vergleich zu den meisten anderen CPAN-Distributionen - wirklich großartigen Dokumentation. Die meisten Fragen werden mit einem Blick in `Pinto::Manual::Introduction`, `Pinto::Manual::QuickStart`, `Pinto::Manual::Tutorial` oder `Pinto::Manual::Installing` beantwortet. Zusätzlich gibt es zu jedem Kommando eine eigene Seite wo die Funktionalität detailliert beschrieben ist. Für das Kommando `list` ist diese z.B. unter `App::Pinto::Command::list` zu finden. Eine Übersicht der Dokumentation im Web erhält man am besten über die Release-Übersicht von *metacpan* unter *https://metacpan.org/release/Pinto*.

Man muss sich aber nicht zwingend ins Internet begeben um mit der Dokumentation zu arbeiten. Pinto bringt alles auf der Kommandozeile mit. Spätestens hier zeigt sich mit wie viel Sorgfalt und Liebe zum Detail *Thalhammer* vorgegangen ist. Ein einfaches `pinto help` listest alle möglichen Kommandos mit Kurzbeschreibung auf und lädt dazu ein diese auszuprobieren. Die komplette und wirklich detaillierte Dokumentation für jedes einzelne Kommando lässt sich dann (z.B. für `list`) mit `pinto manual list` lesen. Die Information ist hierbei jeweils dieselbe wie auch auf CPAN.

Fazit

Pinto verspricht seinem Vorhaben gerecht zu werden. Es bietet ein reichhaltiges Set an Kommandos (von dem hier nicht alles vorgestellt wurde) um die Grundvoraussetzung für zuverlässiges Testen aufrecht zu erhalten: die stabile bzw. gleichbleibende Modullandschaft. Allerdings bietet Pinto alleine keine Garantie für erfolgreiches Deployment. Nur in Kombination mit anderen Massnahmen (betreffend allem was nicht Perl ist) soweit einer voraussichtigen Nutzung von Pinto selbst, kann ein Nutzen aus dem Produkt gezogen werden. Stratopan bietet hier zusätzliche Möglichkeiten, da es von Pintos Kommandozeile abstrahiert und so auch Personal mit weniger Know-How die Pflege der Repositories erlaubt. Auch die Ausgelagerte Wartung des Produktes kann einen Vorteil darstellen. Die 4000 Doller welche von *Brian d Foy* im Mai diesen Jahres für die Weiterentwicklung von Pinto mittels Crowdfunding gesammelt wurden [7] scheinen mir gut investiert zu sein. Die Liste der Spender kann übrigens mit `pinto thanks` angezeigt werden. Zum Schluss möchte ich mich auch noch beim Unternehmen *plusW* für die Unterstützung für den Artikel bedanken.

Links

[1] https://metacpan.org/module/Pinto
[2] https://stratopan.com/
[3] http://www.heise.de/newsticker/meldung/World-Quality-Report-IT-Budgets-fuer-Qualitaetssicherung-steigen-1955351.html
[4] https://metacpan.org/module/Pinto#FEATURES
[5] https://raw.github.com/borisdaeppen/foo-PerlMagazin-Examples/master/28/json2yaml.app.pl
[6] https://stratopan.com/borisdaeppen/webapp/prod
[7] https://www.crowdtilt.com/campaigns/specify-module-version-ranges-in-pint

Thomas Klausner

ZeroMQ & Perl

Was ist ZeroMQ?

ZeroMQ (ØMQ, 0MQ, zmq) ist ein Baustein für verteilte, asynchrone Systeme. ZeroMQ ist quasi eine Message Queue, nur ohne die Queue, bzw. ohne eine zentrale Verwaltung der Queue ("Broker"). ZeroMQW hat Bindings für mehr als 40 Sprachen. "ØMQ sockets are the world-saving superheroes of the networking world". ZeroMQ implementiert "sockets on steroids". Oder, wie Nicholas Perez mal so schön getweetet hat: "ZeroMQ is a horribly leaky abstraction with propaganda as documentation" [0].

OK, genug Bullshit-Bingo.

Mit ZeroMQ kann man relativ leicht Programme bauen, die mit anderen Programmen kommunizieren. Die Kommunikation kann über verschiedene sogenannte `transports` erfolgen, wie z.B. `inproc` (innerhalb eine Programms), `IPC` (auf einem Rechner) und `TCP` (über das Netzwerk). Weiteres erfolgt die Kommunikation nach gewissen `patterns` wie `PUB-SUB` (Broadcast), `PUSH-PULL` (Pipeline) oder `REQ-REP` ("klassischer" Client-Server).

Anders als fertige Message Queues wie z.B. `AMQP` oder `Gearman` ist ZeroMQ kein fertiges Produkt, sondern eben eine library, die man in den eigenen Code integrieren muss. Je nach den eigenen Anforderungen kann das sehr leicht gehen, oder aber sehr komplex werden. In letzterem Fall ist man vielleicht besser beraten, eine der unzähligen fertigen Lösungen zu verwenden. ZeroMQ kümmert sich z.B. weder um Authentifizierung [1] noch um Persistenz der Messages. Dadurch ist es sehr einfach, ein System zu bauen, das keine geheime Daten transportiert und bei dem ein Verlust von einzelnen Messages keinen (oder nur geringen) Schaden verursacht. Will man aber ein sicheres System bauen, muss man einiges an Zeit und Hirnschmalz in die Architektur stecken.

Zu ZeroMQ gibt es eine ausführliche Dokumentation [2] und die ausgesprochen lesenswerte ØMQ Guide, die mit sehr einfachen (aber kaum praxistauglichen) Beispielen beginnt, in späteren Kapiteln dann aber zeigt, wie man stabile und performante Systeme mit ZeroMQ baut.

ZMQ::LibZMQ3

`ZMQ::LibZMQ3` von Daisuke Maki ist das low-level Perl Interface zu ZeroMQ. Die API ist sehr nahe an der Original-API und deswegen nicht sehr perlig. Hier zeige ich als kurzes Beispiel einen kleinen Echo-Server, der eine einfache Message annimmt, "Hello" davorstellt und wieder an den Client zurückschickt.

Der Server - Listing 1

In Zeile 3, 4 und 5 lade ich `ZMQ::LibZMQ3`, `ZMQ::Constants` und `AnyEvent`. `ZMQ::LibZMQ3` arbeitet leider sehr intensiv mit Konstanten, deswegen müssen dieses über `ZMQ::Constants` importiert werden. `AnyEvent` brauchen wir für den Event Loop.

Zeile 7 initiiert einen `ØMQ context`. Dieser `context` ist ein Container für alle weiteren Sockets und verwaltet diese.

In Zeile 8 wird ein neues Socket angelegt. `ZMQ_REP` gibt den Typ dieses Sockets an, in diesem Fall eben `REP`.

In Zeile 9 verknüpfe ich nun das Socket mit einem `transport`, hier `tcp://*:10001`, also Port 10001 am `localhost`.

Zeilen 12 bis 21 implementieren den Event Loop dieses Servers über `AnyEvent`. ZeroMQ kommt zwar mit einem eigenen Event Loop (`zmq_poll`), aber in Perl fahren wir mit

AnyEvent besser. Damit AnyEvent aber was zum Arbeiten hat, brauchen wir den in Zeile 10 via zmq_getsockopt ausgelesenen FileDescriptor. Sobald nämlich ZeroMQ eine Message empfängt, wird diese Message als Read-Event an diesem File Descriptor gemeldet. Dann wird der in Zeile 15 bis 19 definierte callback aufgerufen.

Via zmq_recvmsg und zmq_msg_data komme ich an die Payload der Message. ZeroMQ kümmert sich nicht im Geringsten um Inhalt, Art oder Encoding der Message, kann allerdings keine Referenzen übermitteln. Sollten also komplexe Datenstrukturen übertragen werden, müssen diese vorher serialisiert werden (z.B. als JSON). In diesem Fall ist $msg aber ein einfacher String.

In Zeile 18 schicke ich die Antwort des Servers ("Hallo $msg") an den Client.

Der Client - Listing 2.
Wieder lade ich ZMQ::LibZMQ3 und ZMQ::Constants. In Zeile 6 lese ich das erste über die Kommandozeile übergebene Argument aus, oder verwende "World" als default.

In Zeile 8 bis 10 initiiere ich wieder den context, generiere ein Socket (diesmal vom Typ REQ) und verbinde mich zum Server.

ZeroMQ bietet 2 Möglichkeiten ein Socket mit einem Transport zu verküpfen. zmq_bind wird für die "stabilere" bzw. länger laufende Seite verwendet (man könnte diese Seite auch "Server" nennen, aber oft macht diese alte Nomenklatur wenig Sinn). Die andere Seite (der "Client") verwendet zmq_connect. Noch ein wenig verwirrender wird das Ganze dadurch, dass ZeroMQ erlaubt, dass sich Clients an nicht vorhandene Server verbinden. Sobald dann ein Server verfügbar ist, springt der Client an. Das kann allerdings zu verlorenen Messages etc. führen, weshalb bei komplexeren Architekturen die Implementierung eines Handshake-Systems Sinn machen kann.

Wieder zurück zum Beispiel: in Zeile 11 wird via zmq_send die Message an den Server geschickt.

Da ich hier nur einen sehr einfachen Client zeige, der nur einen Request abhandelt und der auch lange auf die Antwort warten kann (und währenddessen inaktiv bleibt, also blocked), braucht der Client keinen Event Loop. Stattdessen wird in Zeile 15 einfach auf die Antwort des Servers gewartet.

Sobald diese eingetroffen ist, wird sie ausgegeben, und der Client stoppt.

Dieses Beispiel enthält weder Error Handling noch kann es mit Netzwerkproblemen umgehen [4]. Dadurch zeigt es zwar die Basiskonzepte von ZeroMQ sehr schön, und auch, mit wie wenig Aufwand man verteilte System bauen kann. Allerdings werden in den meisten echten Systemen Fehler und Netzwerkprobleme auftreten. Um mit diesen umzugehen, muss der Code um einiges komplizierter gemacht werden. Meistens braucht es dann auch zusätzlich zu den eigentlichen Kommunikationskanälen weitere Sockets, über die Metainformation ausgetauscht werden kann.

```
#!/usr/bin/env perl
use 5.016;
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(ZMQ_REP ZMQ_FD);
use AnyEvent;

my $context = zmq_ctx_new();
my $server = zmq_socket(
  $context, ZMQ_REP );
zmq_bind( $server, 'tcp://*:10001');
my $fd = zmq_getsockopt( $server, ZMQ_FD );

my $w = AnyEvent->io(
    fh   => $fd,
    poll => "r",
    cb   => sub {
        my $msg = zmq_msg_data(
                    zmq_recvmsg($server) );
        say "server got >$msg<";
        zmq_send( $server, "Hello $msg!" );
        say "replied with >Hello $msg!<";
    },
);
AnyEvent->condvar->recv;
```

Listing 1

```
#!/usr/bin/env perl
use 5.016;
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(ZMQ_REQ);

my $text = $ARGV[0] || 'World';

my $context = zmq_ctx_new();
my $client = zmq_socket(
                $context, ZMQ_REQ );
zmq_connect( $client,
                'tcp://localhost:10001' );

zmq_send( $client, $text );
say "client sent >$text<";

my $reply = zmq_msg_data(
                 zmq_recvmsg($client) );
say "client got >$reply<";
```

Listing 2

ZeroMQ bietet z.B. die Möglichkeit, über einen `zmq_socket_monitor` Events, die auf einem Socket passieren (z.B. connects) via ZeroMQ an ein anderes Socket (eben den Monitor) zu schicken. Aber einfach zu verwenden ist das dann nicht mehr.

ZMQx::Class

Weil uns [5] `ZMQ::LibZMQ3` zu low-levelig war, haben wir einen Wrapper gebaut: `ZMQx::Class`. Unsere Designziele waren:

- einfacheres Setup von Sockets & `sockopts`
- einfaches Senden und Empfangen von einfachen und multipart Messages
- automatisches Handling des `contexts`, vor allem beim `forken`
- `AnyEvent` Hilfsmethoden

Wir verwenden `ZMQx::Class` seit ein paar Monaten in einem Projekt, das z.Z. noch in Entwicklung ist. Es könnten sich also noch einige Bugs in `ZMQx::Class` verstecken, trotzdem würden wir uns über Benutzer/-innen und Feedback freuen!

Seit Anfang Oktober gibt es ein neues Perl-binding: `ZMQ::FFI` von Dylan Cali. Nicht nur verwendet es FFI (was anscheinend der schlauere / faulere Weg ist, libraries einzubinden), sondern es bietet auch eine etwas angenehmere, Perl-artigere API. `ZMQx::Class` ist nach wie vor noch bequemer und bietet auch noch weitere Features als `ZMQ::FFI`. Aber wir werden uns `ZMQ::FFI` auf jeden Fall noch genau anschauen und auf die eine oder andere Art mit `ZMQx::Class` verbinden.

Ein Beispiel: ZeroBlog

Um mal herzuzeigen, was ZeroMQ so kann, und wie `ZMQx::Class` in Action ausschaut, habe ich als kleines Beispiel eine alte Spielerei von mir nach ZeroMQ portiert.

Auf meiner Website habe ich einen kleinen Microblog [6], quasi mein privates Twitter. Bespielt wird dieser Bereich direkt aus dem `IRC` über ein `irssi` Plugin, über das ich mittels eines eigenen Kommandos IRC-Messages via HTTP an einen kleinen Dancer-Server schicke, der die Message dann in eine SQLite Datenbank einträgt. Damit die Nachricht auf meiner statischen Website aufscheint, muss ich diese neu bauen, was derzeit nur durch ein manuell einzugebendes Kommando am Server möglich ist.

Dieses System möchte ich nun auf ZeroMQ umstellen.

Überblick über die Architektur

Weiterhin soll aus dem `IRC` mittels eines `irssi`-Plugins eine Message geschickt werden. Diese soll nun aber anstatt über HTTP mit ZeroMQ an einen Broker übertragen werden. Der Broker authentifiziert die Messages mittels eines shared keys und schickt sie im Erfolgsfall über ein `PUB` Socket weiter. Beliebig viele andere Prozesse, die auch auf beliebigen Rechnern rennen können, subskribieren sich mittels eines `SUB` Sockets. Einer dieser Prozesse wird am Webserver laufen und dort die Nachricht speichern und meine Webseite neu bauen. Ein anderer Prozess könnte auf deinem Rechner rennen und eine Desktop Notifikation auslösen. Ein weiterer könnte meine Nachricht an Twitter weiterleiten... (siehe Abbildung 1.)

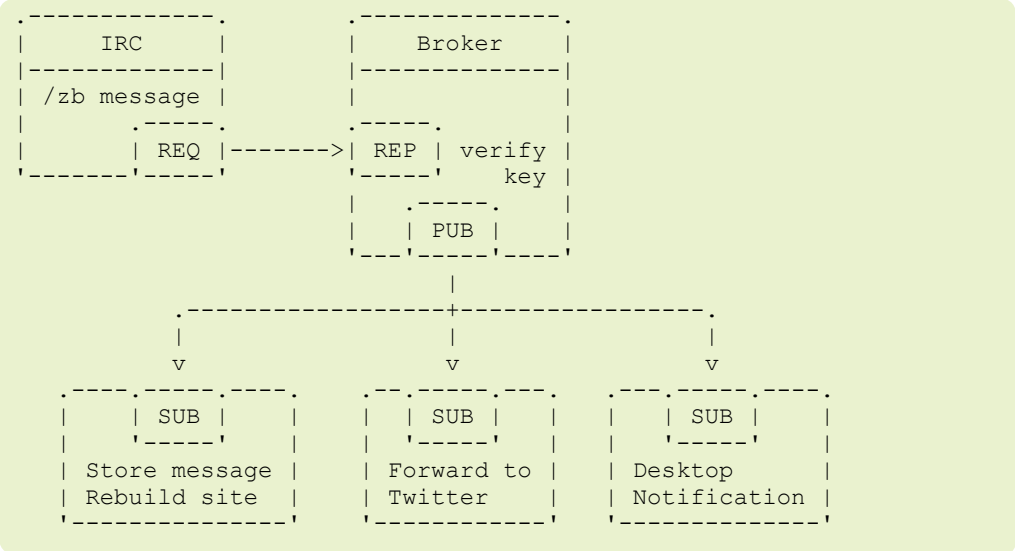


Abbildung 1

Der gesamte Code ist auf github verfügbar: https://github.com/domm/ZeroBlog

```
package ZeroBlog::Broker;
use 5.014;
use Moose;
use ZMQx::Class;
use AnyEvent;
use Digest::SHA1 qw(sha1_hex);

has 'base_port' => (is=>'ro',isa=>'Int',default=>'3333');
has 'secret' => (is=>'ro',isa=>'Str',required=>1);

has 'receiver' => (is=>'ro',isa=>'ZMQx::Class::Socket',lazy_build=>1,required=>1);
sub _build_receiver {
    my $self = shift;
    return ZMQx::Class->socket( 'REP', bind => 'tcp://*:'.$self->base_port );
}
has 'publisher' => (is=>'ro',isa=>'ZMQx::Class::Socket',lazy_build=>1,required=>1);
sub _build_publisher {
    my $self = shift;
    return ZMQx::Class->socket( 'PUB', bind => 'tcp://*:'.($self->base_port+1) );
}

sub run {
    my $self = shift;

    say "Receiver listening on ".$self->receiver->get_last_endpoint;
    say "Publisher sending on ".$self->publisher->get_last_endpoint;

    $self->loop;
}

sub loop {
    my $self = shift;

    my $receiver = $self->receiver;
    my $publisher = $self->publisher;

    my $watcher = $receiver->anyevent_watcher( sub {
        while ( my $msg = $receiver->receive ) {
            my ($token, $message) = @$msg;
            say "got ".join(' - ',@$msg);
            my $check_token = sha1_hex($message,$self->secret);
            if ($check_token eq $token) {
                $receiver->send('ok');
                $publisher->send($message);
            }
            else {
                $receiver->send('bad token, message rejected');
            }
        }
    });
    AnyEvent->condvar->recv;
}

__PACKAGE__->meta->make_immutable;
1;
```

Listing 3

Der Broker - Listing 3

Fangen wir mal mit dem kompliziertem Stück an, dem Broker:

Ich verwende hier `Moose`, `ZMQx::Class`, `AnyEvent` und `Digest::SHA1`. Die beiden Attribute `base_port` und `secret` können von der Benutzerin gesetzt werden, wobei `secret` über keinen default-Wert verfügt und gesetzt werden muss.

In `_build_receiver` und `_build_publisher` werden die beiden Sockets initiiert, nämlich ein `REP`-Socket, das die Messages annimmt; und ein `PUB`-Socket, dass die Messages weiterleitet. Hier sieht man schön, wie viel einfacher das Socket-Setup in `ZMQx::Class` ist:

```
ZMQx::Class->socket(
  'REP', bind => 'tcp://*:3333' );
```

versus

```
my $context = zmq_ctx_new();
my $client = zmq_socket(
                $context, ZMQ_REP );
zmq_bind( $client, 'tcp://*:3333' );
```

Die `run`-Methode ist eher langweilig, weswegen wir gleich zur `loop`-Methode weitergehen.

Mit `anyevent_watcher` kann eine Callback-Funktion definiert werden, die aufgerufen wird, wenn das Socket eine Message empfängt. In diesem Callback kann mit `$socket->receive` einfach eine Multipart-Message empfangen werden.

Hier besteht diese aus zwei Teilen, einem `$token` und der tatsächlichen `$message`. In Zeile 41 überprüfe ich, ob das übermittelte `$token` und ein neu berechnetes gleich sind. Wenn ja, hat der Absender wohl dasselbe shared secret verwendet und ich akzeptiere die `$message`.

Dazu schicke ich an den Absender ein kurzes `ok` zurück. Danach schicke ich die `$message` an das `PUB` socket. Wenn die Authentifizierung nicht geklappt hat, schicke ich eine kurze Fehlermeldung zurück.

Der Broker ist in dieser Form (als reine Klasse) natürlich noch nicht lauffähig. Deswegen braucht es noch das kleine Script `bin/broker.pl`, das die Klasse verwendet:

```
package RunBroker;
use strict;
use warnings;
use FindBin;
use lib "$FindBin::Bin/../lib";
use local::lib "$FindBin::Bin/../local";
use 5.014;

use Moose;
extends 'ZeroBlog::Broker';
with 'MooseX::Getopt';

__PACKAGE__->new_with_options->run;
1;
```

Nach diversem `@INC`-Gefrickel wird in Zeile 10 die aktuelle Klasse `RunBroker` als Subklasse von `ZeroBlog::Broker` definiert. In der nächsten Zeile wird die Rolle `MooseX::Getopt` dazugeladen, wodurch die Attribute von `ZeroBlog::Broker` nun von der Kommandozeile aus gesetzt werden können. Das ist im Übrigen eine sehr praktische Methode, die ich gerne verwende.

So kann ich also den Broker starten:

```
~/perl/ZeroBlog$ bin/broker.pl
   --secret Sup3rGehe1m
Receiver listening on tcp://0.0.0.0:3333
Publisher sending on tcp://0.0.0.0:3334
```

Der Publisher - Listing 4
Anstatt nun den "Client" direkt in `irrsi` einzubauen, macht es Sinn, zuerst mal eine generische Klasse zu bauen, die dann von `irssi` (oder auch von woanders aus) verwendet wird.

Wieder definiere ich einige Attribute, u.a. das `secret` und den `endpoint`. Der `endpoint` ist die Adresse, unter der das `REP`-socket des Brokers rennt. Der `requestor` ist das `REQ`-socket, das hier wieder mit `ZMQx::Class` gebaut wird. In Zeile 13 und 14 werden sogenannten `sockopts` gesetzt. Damit kann das Verhalten des Sockets gesteuert werden.

In der `send`-Methode wird die Message ein wenig aufgeräumt und dann via `sha1_hex` und dem shared secret das `$token` erzeugt. In Zeile 28 wird nun die Multipart-Message losgeschickt. `ZMQx::Class` macht das sehr einfach, indem ein `ArrayRef` übergeben wird. In Zeile 29 wird `receive(1)` aufgerufen. Da `1` übergeben wird, ist dieser Aufruf `blocking`, d.h. das Programm pausiert hier so lange, bis eine Antwort zurückkommt.

Nun folgt noch ein wenig Error Handling bzw. die Rückgabe der Antwort des Brokers.

Hier ist ein kleiner Commandline Client, `bin/commandline.pl`, der wieder den schon im Broker beschriebenen Subclass-Trick verwendet (siehe Listing 5).

Wenn der Broker gerade rennt, kann ich nun so eine erste Botschaft verschicken:

```
~/perl/ZeroBlog$ bin/commandline.pl
  --secret Sup3rGehe1m --endpoint
    tcp://localhost:3333
ok
```

Der Broker sollte in etwa sowas ausgeben:

```
got 5eb4a80c96f65f6b56abcb5e045309b61b815d08
  - die erste botschaft
```

Die Einbindung des Publishers in `irrsi` ist relativ mühsam, weil `irssi` ein recht ... altes Interface anbietet. Wer mag, kann sich die fast 100 Zeilen glue-code in der Klasse `ZeroBlog::Publisher::IrssiPlugin` im github-repo durchlesen [7].

Die Subscriber - Listing 6

Für die Subscriber (oder auch Worker) baue ich wieder eine Klasse, die die Funktionalität implementiert:

```
package ZeroBlog::Publisher;
use 5.014;
use Moose;
use ZMQx::Class;
use Digest::SHA1 qw(sha1_hex);

has 'secret' => (is=>'ro',isa=>'Str',required=>1);
has 'endpoint' => (is=>'ro',isa=>'Str',required=>1);
has 'requestor' => (is=>'ro',isa=>'ZMQx::Class::Socket',lazy_build=>1,required=>1);
sub _build_requestor {
    my $self = shift;
    return ZMQx::Class->socket( 'REQ', connect => $self->endpoint, {
        rcvtimeo=>500,
        linger=>0
    } );
}

sub send {
    my ($self, $message) = @_;

    $message =~ s/\s+$//;
    return 'error','no message' unless $message;
    my $token = sha1_hex($message,$self->secret);

    my $socket = $self->requestor;
    my $rv;
    eval {
        $socket->send([$token, $message]);
        $rv = $socket->receive(1);
    };
    if ($@ || !$rv) {
        return ('error', $@ || 'connection timeout' );
    }
    else {
        return $rv->[0], '';
    }
}

__PACKAGE__->meta->make_immutable;
1;
```

Listing 4

```
package RunCommandline;
use strict;
use warnings;
use FindBin;
use local::lib "$FindBin::Bin/../local";
use lib "$FindBin::Bin/../lib";
use 5.014;

use Moose;
extends 'ZeroBlog::Publisher';
with 'MooseX::Getopt';

my $client = __PACKAGE__->new_with_options;
my ($status, $error) = $client->send(join(' ',@{$client->extra_argv}));
say "$status $error";

1;
```

Listing 5

```
package ZeroBlog::Subscriber;
use 5.014;
use Moose;
use ZMQx::Class;
use AnyEvent;

has 'endpoint'  => ( is => 'ro', isa => 'Str', required => 1 );
has 'subscribe' => ( is => 'ro', isa => 'Str', default  => '' );
has 'subscriber' => (
    is         => 'ro',
    isa        => 'ZMQx::Class::Socket',
    lazy_build => 1,
    required   => 1
);

sub _build_subscriber {
    my $self = shift;
    return ZMQx::Class->socket( 'SUB', connect => $self->endpoint );
}

sub run {
    my ( $self, $callback ) = @_;

    my $subscriber = $self->subscriber;
    $subscriber->subscribe( $self->subscribe );
    my $watcher = $subscriber->anyevent_watcher(
        sub {
            while ( my $msg = $subscriber->receive ) {
                &$callback( $self, $msg );
            }
        } );
    AnyEvent->condvar->recv;
}
1;
```

Listing 6

Der Code sollte jetzt schon recht bekannt aussehen. Das Attribut `endpoint` bestimmt die Adresse des `PUB`-Servers. Der `subscriber` ist ein Socket, das über `ZMQx::Class` gebaut wird. Zwei interessante Punkte gilt es hier aber noch anzusprechen.

Erstens, `subscribe`: ZeroMQ `PUB-SUB` Sockets verfügen über die Möglichkeit, Interesse an nur bestimmten (oder allen) Messages zu bekunden. Das passiert eben über einen Aufruf der `subscribe`-Methode (bzw. `zmq_setsockopt ZMQ_SUBSCRIBE` in der low-level Variante). Wenn ein `SUB` Socket sich an `PUB` Socket verbindet, muss es diesem mitteilen, welche Messages geschickt werden sollen. Dazu kann ein beliebiger String gesetzt werden (oder ein leerer String, um alle Messages zu erhalten). Das `PUB` Socket schickt dann nur Messages an das `SUB` Socket, deren erstes Frame (also der erste Teil der Message) diesem String entspricht. Regex sind hier leider nicht möglich.

Hier wollen wir alle Messages erhalten, deswegen subskribieren wir einen leeren String (Zeile 25 bzw. 8).

Zweitens erwartet die `run` Methode ein Coderef als Argument. Dadurch kann ich in der `run` Methode den generischen Event Loop initialisieren und einkommende Messages entgegennehmen. Die Messages werden dann an den Callback übergeben, der eben außerhalb des Modules definiert wird.

Schauen wir uns mal diesen außen liegenden Code an, hier z.B. das Script `bin/store_blio_recreate.pl`, das die eingehenden Messages in eine SQLite-Datenbank speichert und dann meine Website neu baut (siehe Listing 7).

Die ersten 15 Zeilen kenn wir schon. In Zeile 17 und 18 wird der Speicherort meiner Website im Filesystem und ein DBI-Handle definiert (ja, das könnte auch über Moose Attribute gelöst werden).

In den Zeilen 20 bis 31 übergebe ich nun den Callback-Code als Subref an die `run` Methode von `ZeroBlog::Subscriber`. In dem Callback wird die Message in die SQLite-DB gespeichert und dann ein Shell-Script aufgerufen, das meine Website neu baut.

```
package StoreBlioRecreate;
use strict;
use warnings;
use FindBin;
use local::lib "$FindBin::Bin/../local";
use lib "$FindBin::Bin/../lib";
use 5.014;

use Moose;
extends 'ZeroBlog::Subscriber';
with 'MooseX::Getopt';

use Path::Class;
use DBI;
use DateTime;

my $blio_base = dir('/home/domm/privat/domm.plix.at');
my $DBH = DBI->connect("dbi:SQLite:dbname=".$blio_base->file('sqlite','microblog.db'),'','');

__PACKAGE__->new_with_options->run(sub {
    my ($self, $msg) = @_;
    my ($message) = @$msg;

    say "storing message in DB";
    $DBH->do(
        'INSERT INTO microblog (date,message) values (?,?)',undef,
        DateTime->now(time_zone=>'local')->iso8601,
        $message
    );
    say "rebuilding website";
    system($blio_base->file('build')->stringify);
});

1;
```

Listing 7

Fertig!

Im `ZeroBlog` github repository findet sich noch ein weiterer Subscriber, der eingehende Messages einfach im Terminal ausgibt (`libnotify` für richtige Desktop-Notifikationen hab ich noch nicht zum Laufen gebracht). Der Twitter-Weiterleiter bleibt eine Aufgabe für die Leser.

Falls sich ein werter Leser oder eine werte Leserin direkt via ZeroMQ an meinen Microblog-Feed hängen mag, gebe ich gerne die `endpoint`-Adresse meines Publishers via Email (oder IRC) bekannt.

Zusammenfassung

Mit ZeroMQ lassen sich tatsächlich relativ leicht verteilte System bauen. Nicht nur kann so die Last auf mehrere Rechner verteilt werden (horizontales Scaling), es können auch komplett verschiedene Sprachen zur Implementierung der unterschiedlichen Komponenten verwendet werden. Mit `ZMQx::Class` können (hoffentlich!) solche Komponenten in Perl auf native und elegante Weise geschrieben werden.

Man darf aber nicht übersehen, dass ein stabiles und verlässliches (reliable) System, in dem Messages niemals verloren gehen, in ZeroMQ zwar möglich ist, aber dann einiges mehr an Aufwand braucht, als ich hier gezeigt habe.

Links

• *http://zeromq.org*: ZeroMQ Homepage mit Links zu API-Dokumentation, Guide, Downloads, etc.

• *http://zguide.zeromq.org*: "ØMQ - The Guide" - lesenswert

• *https://metacpan.org/release/ZMQ-LibZMQ3*: Perl Bindings, low level

• *https://metacpan.org/release/ZMQx-Class*: OO Interface zu ZeroMQ, entwickelt vom werten Autor und Klaus Ita. Danke an Validad GmbH für die Veröffentlichung auf CPAN.

• *http://nanomsg.org/documentation-zeromq.html*: Einige Probleme von ZeroMQ beschrieben aus der Sicht von nanomsg, einem ZeroMQ-Rewrite von Martin Sustrik, einem der Autoren von ZeroMQ

• *http://domm.plix.at/talks/zeromq_perl.html*: Slides zu meinem Talk über ZeroMQ und Perl (u.a. mit Details zu Multipart Messages). Ein Video von der YAPC::Europe 2013 in Kiev gibt's auch.

Fußnoten

0: https://twitter.com/perlyarg/statuses/327067223824953344

1: In der neuen Version 4.0, die soeben releaset wird, wurde ein security framework integriert.

2: bzw Propaganda :-)

3: Multipart-Messages sind komplizierter.

4: die auf jeden Fall auftreten werden :-)

5: http://validad.com

6: http://domm.plix.at/microblog.html

7: https://github.com/domm/ZeroBlog/blob/master/lib/ZeroBlog/Publisher/IrssiPlugin.pm

Herbert Breunung

Sereal - sichert alle Daten - schnell und kompakt

Wenn der Benutzer das Licht ausknipst, fallen alle Siliziumchips in den tiefen Schlaf des Vergessens. Damit die ganze Rechnerei aber nicht vergebens war, speichern Programme ihre Ergebnisse auf Festplatte oder Flash-Speicher. Dafür gibt es eine sehr große Auswahl an Formaten. Voriges Jahr fügten die Entwickler der Hotelreservierungsplattform *Booking.com* sogar noch eines dazu. Denn das Programm einfach nur am nächsten Tag (oder auf einem anderen Rechner) fortzufahren, war bisher (zumindest im Land der Dromedare) nicht so einfach.

Nichts gegen YAML, JSON, Storable oder Data::Dumper - aber was diese Module interessiert, sind lediglich Daten im Sinne von Zahlen, Text, Arrays und Hashes. YAML kann etwas mehr, da der bekannte Perl-Haudegen Ingy daran beteiligt war. Es kann auch Perl-Objekte, *typeglob* und Referenzen serialisieren (in eine Form bringen, die man in einem Schwung in eine Datei legen kann und aus der sich später alles wieder herstellen lässt). Doch wer Perl näher kennt, weiß, dass es zwischen Himmel und Hölle noch weit mehr Dinge gibt, wie etwa schwache Referenzen. Eines der Ziele von Sereal war es, wirklich alles realitätsgetreu speichern und wiedergeben zu können, selbst bereits kompilierte reguläre Ausdrücke.

```
use v5.14;
use Sereal qw/encode_sereal decode_sereal/;

my $rxout = encode_sereal(qr/d(.+)r/);
say decode_sereal($rxout);
say ref decode_sereal($rxout);
```

ergibt:

```
(?^:d(.+)r)
Regexp
```

Sereals grundsätzliche API ist denkbar einfach und funktioniert nach dem bekannten freeze/thaw - Prinzip: je eine Funktion pro Richtung der Umwandlung. Die zweite Ausgabe zeigt: es ist eine Regex, welche im beim print/say der ersten Ausgabe ihren Inhalt präsentiert. Jede Datenstruktur, die Perl im Speicher halten kann, lässt sich nun festhalten - keine Bedenken mehr, ob ein Attribut etwas Problematisches enthielt. Für so eine Arbeit muss man Ahnung von Perl's internen Speicherstrukturen haben. Die ist zweifellos bei Hauptautor Steffen Müller (tsee) zu finden, der bereits seit Jahren sehr aktiv in der p5p ist. Auch Helfer wie Yves Orton (demerhq), der den Großteil des 5.10er Regex-Umbaus gestemmt hat, oder der ehemalige Pumpking Rafaël Garcia-Suarez besitzen Kaliber, was am klaren und gut strukturierten Quellcode deutlich sichtbar ist.

Nicht um die Begrenzungen des Serialisierers herumprogrammieren zu müssen war Ziel von Sereal. Ein anders war Geschwindigkeit und eine kompakte Ausgabe. Und obwohl ersteres Priorität hatte, scheint es in beiden Disziplinen sämtliche andere CPAN-Module zu überflügeln, auch den bisherigen Rekordhalter `Data::MessagePack`. Dies war natürlich nur mit einem durchdachten, binären Format möglich, welches mit C-Funktionen erzeugt und gelesen wird. Das Perlmodul enthält nur die XS-Schnittstelle und Dokumentation. Wer mehr Kompression zu Lasten der Rechenzeit bevorzugt, kann Google's Snappy-Algorithmus aktivieren.

```
encode_sereal($data, {snappy => 1});
```

Eine Reihe weiterer Optionen zeigen, dass Sereal ein robustes Werkzeug für schwere Einsätze sein möchte.

Version 2 - weiter gehts!

Diesen Herbst, noch vor seinem ersten Geburtstag, erfuhr das Format eine kleinere Neugestaltung - mit voller Vorwärts- und Rückwärtskompatibilität. Es ging hauptsächlich darum, zwischen die Datenblöcke Metainformationen

abzulegen. Dadurch kann man auch schnell auf wesentliche Informationen zugreifen, ohne den gesamten Datensatz entpacken zu müssen.

Die Praxistauglichkeit scheint sich herumzusprechen, denn im Github-Repository *http://github.com/Sereal/Sereal* finden sich bereits Implementationen für sieben weitere Sprachen. Dies gibt Anlass zur Hoffnung, dass Sereal breit akzeptierter Standard wird, wie bereits YAML oder TAP.

Links

- http://blog.booking.com/sereal-a-binary-data-serialization-format.html
- http://blog.booking.com/the-next-sereal-is-coming.html

Wolfgang Kinkeldei, Renée Bäcker

plenv -- eine Alternative zu perlbrew

Ursprung

Sich die Lieblingsprogrammiersprache lokal zu installieren ist inzwischen eine gewisse Mode geworden. Gründe dafür können nicht vorhandene Administrator-Berechtigungen, hohe Flexibilität oder die Einfachheit eines solchen Unterfangens sein.

Wir kennen alle *perlbrew* als eine Möglichkeit, Perl lokal zu installieren. Aus der Ruby Welt hat das dort verbreitete *rbenv* (*http://rbenv.org*) inzwischen einen Nachahmer bei Perl gefunden. Macht es Sinn, über diese Alternative nachzudenken? Wir gehen dem auf den Grund.

Funktionsweise

Genau wie beim Ruby-Vorbild wird der Suchpfad für ausführbare Kommandos für eine Shell lediglich einmal gesetzt, nachdem `plenv` aktiviert wurde. Innerhalb des Verzeichnisses, in dem `plenv` installiert wurde, existieren zwei Unterverzeichnisse für Binär-Dateien: *bin* für `plenv` und `shims` für alle zu einem bestimmten Perl gehörenden oder dazu installierten Kommandos.

Die Dateien im *shims* Verzeichnis sind alle via Hard-Link mit demselben Shell-Script verbunden. Dieses löst prinzipiell das nachfolgende Kommando aus:

```
plenv exec "gewünschtes Programm"
 argumente...
```

Ein dadurch aktiviertes Shell Script findet die gewünschte Perl Version, setzt temporär den Suchpfad entsprechend und startet das passende Kommando.

Jedes Mal wenn ein neues Perl oder neue Distributionen mit ausführbaren Kommandos installiert wird, muss die Liste der Kommandos im *shim* Verzeichnis aktualisiert werden. Das erfolgt normalerweise automatisch, kann aber im Bedarfsfall manuell mit dem `plenv rehash` Befehl angestoßen werden.

Auswahl der gewünschten Version

Ebenfalls gemäß dem Vorbild von Ruby stehen mehrere Mechanismen mit aufsteigender Priorität zur Verfügung, mittels derer die auszuführende Perl-Version gewählt wird: ein globaler Schalter, innerhalb von Verzeichnissen liegende magische Dateien sowie die Umgebungsvariable `PLENV_VERSION`.

Installation

Leider ist `plenv` noch nicht als CPAN Distribution verfügbar, insofern muss man sich aus dem GitHub-Repository des Autors bedienen. Aber die Installation ist relativ einfach. Nachfolgend die auszuführenden Befehle, die bis zum Verlassen der Shell gelten. Optimaler Weise stehen solche Anweisungen in der Start-Datei der Shell (siehe Listing 1).

```
# nach ~/.plenv auschecken (Verzeichnis wird
# dabei mit angelegt)
$ git clone https://github.com/tokuhirom/
     plenv.git ~/.plenv

# Suchpfad erweitern
$ export PATH="$HOME/.plenv/bin:$PATH"

# Auto-Vervollständigung und Shims
# aktivieren
  $ eval "$(plenv init -)"
```

Nun steht plenv prinzipiell bereit, allerdings erleben wir gleich die erste Überraschung.

```
$ plenv install 5.18.1
plenv: Please install perl-build.
See https://github.com/tokuhirom/plenv/
  blob/master/README.md#installation
```

Na gut, dann folgen wir den Anweisungen:

```
$ git clone git://github.com/tokuhirom/
  Perl-Build.git \
    ~/.plenv/plugins/perl-build/

$ plenv install 5.18.1
Fetching 5.18.1 as /home/wolfgang/.plenv/
  cache/perl-5.18.1.tar.gz
```

Na also -- geht doch! Dann sorgen wir für die richtige Spielwiese:

Perl Versionen installieren

Verschiedene Perl Versionen zu installieren ist relativ einfach

```
# Auflisten aller verfügbaren Perl Versionen
   dank Vervollständigung:
$ plenv install (tab)(tab)

# installieren einer ausgewählten Version
$ plenv install 5.16.3
$ plenv install 5.14.4

# Auflisten aller installierten:
$ plenv versions
  * system
    5.16.3
    5.18.1

# globales Auswählen einer Version
$ plenv global 5.18.1

$ perl -v
This is perl 5, version 18, subversion 1 ...
```

Wer gerne mit `cpanm` Distributionen installiert, wird sich über diesen einfachen Weg, es zu installieren freuen. Leider muss man diesen Schritt für jedes installierte Perl einmal wiederholen.

```
$ plenv install-cpanm

$ plenv global 5.16.3
$ cpanm --help

plenv: cpanm: command not found

The 'cpanm' command exists in these Perl
 versions:
    5.18.1
```

Kennt man den Namen eines ausführbaren Kommandos, kann man schnell herausfinden, bei welchen Perl Versionen es zur Verfügung steht:

```
$ plenv whence cpanm
5.14.4
5.18.1
```

Umgang mit verschiedenen Perl Versionen

Wie oben schon erwähnt, stehen drei unterschiedlich prior behandelte Möglichkeiten zum Umschalten auf eine andere Perl-Version zur Verfügung. Die jeweils zu verwendende Version wir hierbei in einer Datei *version* innerhalb des Plenv-Verzeichnisses festgehalten.

```
$ plenv global 5.18.1
$ perl -v
This is perl 5, version 18, subversion 1
$ plenv version
5.18.1 (set by /home/wolfgang/.plenv/
  version)
$ cat ~/.plenv/version
5.18.1
```

Sehr praktisch ist die vom Ruby-Vorbild übernommene Funktionalität, die gestattet je Verzeichnis festzulegen, welche Perl-Version innerhalb dieses Verzeichnisses zu wählen ist. Zu beachten ist dabei, dass dieses Verzeichnis (oder ein darunter liegendes) das aktuelle Verzeichnis sein muss! Es genügt nicht, ein ausführbares Kommando innerhalb eines solchen Verzeichnisses auszuführen!

```
$ mkdir -p projekte
$ cd projekte
$ plenv version
5.18.1 (set by /home/wolfgang/.plenv/
  version)
$ plenv local 5.14.4
$ plenv version
5.14.4 (set by /home/wolfgang/projekte/
  .perl-version)
$ cd ..
$ plenv version
5.18.1 (set by /home/wolfgang/.plenv/
  version)
```

Mit dem Aufruf des Skripts direkt mit `perl` oder mit `#!/usr/bin/env perl` in der Shebang kann ein Perl-Skript aus jedem beliebigen Verzeichnis heraus aufgerufen werden und es wird mit dem "richtigen" Perl ausgeführt:

```
$ mkdir 5181
$ cd 5181/
$ cat version.pl
#! /usr/bin/env perl
print $],"\n";
$ perl version.pl
5.014002
$ cat .perl-version
5.18.1
$ perl version.pl
5.018001
$ cd ..
$ perl 5181/version.pl
5.018001
$ perl version.pl
5.014002
$ ./5181/version.pl
5.018001
```

Umschaltung in Server Umgebungen

Will man die Vorzüge der Umschaltung einer Perl-Version in Cron-Umgebungen oder *init.d* Scripten kommen, braucht man lediglich das passende ausführbare Kommando im *shims* Verzeichnis von plenv anzugeben. Da es sich bei allen shim-Kommandos um Shell-Scripte handelt, die über die Versions-Umschaltmechanismen das passende Kommando in der gewünschten Perl-Version finden, bekommt man die passende Perl-Version automatisch serviert.

Damit reduziert sich der Aufwand bei z.B. Cron-Scripten auf eine dieser Möglichkeiten:

```
# nichts tun, die globale Perl Version ist
# korrekt gesetzt (hoffentlich)
~/pfad/zu/plenv/shims/kommando

# ODER: zu einem passenden Verzeichnis
# wechseln
cd /web/data/www.website.de/app
~/pfad/zu/plenv/shims/kommando

# ODER: per Environment-Variable das
# passende Perl wählen
export PLENV_VERSION=5.16.3
~/pfad/zu/plenv/shims/kommando
```

Unterschiede zu Perlbrew

Während `plenv` lediglich zwischen Perl Versionen unterscheidet, bietet Perlbrew die Möglichkeit mittels `perlbrew lib` je installiertem Perl mehrere Verzeichnisse zur Aufnahme unterschiedlicher Distributionen zu verwalten. Eine solche *lib* verhält sich von der Umschaltung her wie eine eigene Perl Version. So könnte man Projekt- oder Kundenspezifische Installationen perlbrew-intern sammeln. Plenv bietet solch einen Komfort nicht, allerdings ist die Installation von Distributionen in eigene Verzeichnisse und das korrekte Setzen der `PERL5LIB` Umgebungsvariablen nicht dramatisch.

Die Umschaltung zwischen Perl-Versionen kann bei perlbrew zwar temporär mit `perlbrew use` für die aktuelle Shell vorgenommen werden, eine Automatik wie bei plenv gibt es hier leider nicht.

Perlbrew bietet mit dem `exec` Befehl die Möglichkeit, ein und dasselbe Kommando mit jeder installierten Perl Version auszuführen. Für Tests oder Benchmarks können solche Kleinigkeiten sehr angenehm sein.

Die Installation von `cpanm` bieten beide Kandidaten, plenv installiert im aktuellen Perl, perlbrew installiert an zentraler Stelle.

Perlbrew bietet einen *self-upgrade*, der bei plenv fehlt.

Plenv erlaubt die Suche nach ausführbaren Kommandos und liefert mit `plenv whence` die Liste aller Perls, die dieses Kommando kennen, perlbrew kennt solch eine Option nicht.

Plugins

Diese Kommandos, die perlbrew kennt und in plenv fehlen, können aber über das Plugin-System von plenv nachträglich installiert werden. Ein erstes Plugin wurde auch schon installiert: perl-build.

Tatsuhiko Miyagawa hat die nützlichen perlbrew-Kommandos `exec`, `lib` und `use` in jeweils einem weiteren Plugin umgesetzt. Mittels

```
git clone git://github.com/miyagawa/
  plenv-contrib.git ~/.plenv/plugins/
  plenv-contrib/
```

können die Plugins installiert werden. Das perlbrew-Kommando heißt unter plenv dann aber nicht `exec` sondern `exec-all`. Wie bereits erwähnt, eignet sich das hervorragend für Tests. Besonders für Programme und Module, bei denen nicht feststeht unter welcher Perl-Version sie eingesetzt werden, ist der Test hilfreich. Auch die bevorstehende

Migration von "internen" Projekten auf eine neuere Version ist ein mögliches Einsatzszenario.

```
$ plenv exec-all perl version.pl
5.18.1
==========
5.018001

5.19.3
==========
5.019003
```

Gleiche Perl-Versionen aber unterschiedliche Projekte bedeutet entweder unter mehreren Benutzern mit plenv arbeiten oder ein Benutzer mit dem halben CPAN für die geforderte Perl-Installation. Ein dritter Weg ist die Verwendung des `lib`-Kommandos. Zu beachten ist dabei, dass das Modul `local::lib` installiert sein muss (siehe Listing 1).

Allerdings ist es nicht möglich, in der *.perl-version*-Datei eine Versionsangabe á la `5.18.1@projekt1` festzulegen. Da wird hoffentlich noch nachgebessert.

Eigene Plugins

Sollten die bisher gezeigten Kommandos nicht ausreichen, ist es möglich, eigene Plugins zu schreiben. Hier wird zum einen das Kommando `cpan-reporter` als Bash-Skript und das Kommando `list-inc` als Perl-Skript umgesetzt.

Das `cpan-reporter`-Kommando installiert in jeder mit plenv installierten Perl-Version *cpanm* und `App::cpanminus::reporter` und startet den setup-Vorgang. Damit kann man bei der Verwendung von plenv auch schnell noch etwas Gutes für die Perl-Community machen.

Das Skript für das Kommando ist sehr kurz:

```
#!/usr/bin/env bash
set -e
[ -n "$PLENV_DEBUG" ] && set -x

if [ -z "$PLENV_ROOT" ]; then
PLENV_ROOT="${HOME}/.plenv"
fi

for path in "${PLENV_ROOT}/versions/"*; do
if [ -d "$path" ]; then
curl -L http://cpanmin.us |
   PLENV_VERSION="${path##*/}" plenv exec
   perl - App::cpanminus
      PLENV_VERSION="${path##*/}" plenv
   exec cpanm App::cpanminus::reporter
      plenv rehash
fi
done

cpanm-reporter --setup
```

Das `list-inc`-Kommando gibt einfach die Verzeichnisse aus, die in `@INC` enthalten sind. Das kann für das Debugging hilfreich sein.

```
#!/usr/bin/env perl

use strict;
use warnings;

print $_,"\n" for @INC;
```

Die hier gezeigten Plugins sind auch unter *https://github.com/reneeb/plenv-plugins-foo* zu finden.

```
$ plenv lib create 5.18.1@projekt1
Creating lib 'projekt1' for 5.18.1
Attempting to create directory ....
$ plenv use 5.18.1@projekt1
A sub-shell is launched with PLENV_VERSION=5.18.1 and local::lib activated for @projekt1.
  Run 'exit' to finish it
$ cpanm Mojolicious
... # install Mojolicious
$ perl -MMojolicious -e 1
$ exit
$ plenv lib create 5.18.1@projekt2
Creating lib 'projekt1' for 5.18.1
Attempting to create directory ....
$ plenv use 5.18.1@projekt2
A sub-shell is launched with PLENV_VERSION=5.18.1 and local::lib activated for @projekt2.
  Run 'exit' to finish it
$ perl -MMojolicious -e 1
Can't locate Mojolicious.pm in @INC (...)
```

Listing 1

Renée Bäcker

Mensch oder nicht Mensch - das ist hier die Frage?

Oder zumindest eine der vielen Fragen, die man sich als (Web-)Entwickler stellt um z.B. Bots von der Registrierung auszuschließen oder den Login-Vorgang sicherer zu machen. Bei der Registrierung eines JiffyBox-Accounts für einen Test der JiffyBox-API (siehe auch Artikel von Boris Däppen in der letzten Ausgabe) ist mir eine interessante Möglichkeit aufgefallen, wie man einen Account aktivieren kann. Die Standardvorgehensweise in so einem Fall ist es, eine Mail an die angegebene Adresse zu schicken und dort einen Aktivierungslink anzugeben. Bei einem Klick auf diesen Link wird der Account freigeschaltet.

Bei JiffyBox muss man eine Telefonnummer angeben und man bekommt einen Aktivierungscode per Telefon mitgeteilt. Bei der Suche, wie man so etwas selbst implementieren kann, bin ich auf developergarden.com von der Telekom gestoßen. Dort werden APIs zu verschiedensten Diensten wie Immobilienscout24 und ClickAndBuy bereitgestellt und auch die Telekom Tropo API. Diese kann auch Telefonanrufe machen.

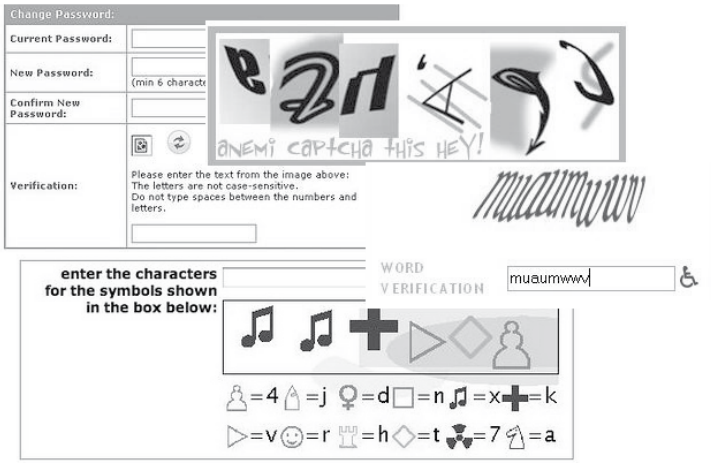


Abbildung 1: Ein paar Captcha-Beispiele

Bots bei der Registrierung ausschließen

Der erste Anwendungsfall, den wir betrachten wollen, ist die Registrierung eines Accounts auf einer Webseite. Bots möchte man nach Möglichkeit ausschließen und nur echte Menschen als Benutzer zulassen. Die erste Hürde, die viele in ihren Formularen einbauen, sind sogenannte *Captchas*. Es gibt auf CPAN etliche Module, die Captchas erzeugen oder bestehende APIs nutzen können. Es gibt ein paar Probleme mit Captchas, aber dieser Schutz ist relativ schnell eingebaut. Im nächsten Abschnitt werden verschiedene Module vorgestellt.

Ein anderer Weg ist - wie oben beschrieben -, dass man den Webseitenbesucher anruft und diesem automatisiert einen Code übermittelt. Hierzu gibt es noch nicht allzu viele APIs. Zwei davon werden in diesem Artikel vorgestellt.

Captchas

Wer kennt sie nicht, die Bilder auf denen man etwas erkennen muss und dann das Erkannte in ein Textfeld geben muss. Ein paar Beispiele sind ein Abbildung 1 zu sehen.

Idealerweise sind Captchas so gestaltet, dass sie von Menschen relativ leicht und von Bots relativ schwer zu lösen sind. Dass das nicht so einfach ist, ist schon an den Beispielen zu sehen. Auch wenn durch die Umwandlung in Graustufen das Erkennen der Schrift nochmal etwas schwieriger ist, ist es auch in der farbigen Version nicht so einfach. Je leichter es für den Menschen ist, das Captcha zu lösen, umso leichter ist es aber auch für Bots per Mustererkennung die Captchas zu lösen.

Deshalb haben sich auch Alternativen zu den bildbasierten Captchas entwickelt: die Audio- und die Video-Captchas. Googles reCAPTCHA bietet sowohl die bildbasierte Version als auch eine Audioversion. Aber auch da kann es zu Problemen kommen. Im Mai 2012 hatten einige Hacker herausgefunden, dass Google die nur 58 Wörter für die Audioversion verwendeten. Das eingeblendete Hintergrundgeräusch stammte aus nur einer kleinen Anzahl von Radiosendungen. So konnten die Hacker ein Skript schreiben, das mittels maschinellem Lernens dies Version knackte.

Google hat aber nachgebessert und reCAPTCHA gilt als eines der sichersten Verfahren.

Für reCAPTCHA gibt es schon Module auf CPAN: `Captcha::reCAPTCHA`. Das in Webanwendungen zu verwenden ist sehr einfach. Man benötigt einen Account und die hierfür generierten Tokens und Schlüssel.

```
use CGI;
use Captcha::reCAPTCHA;

my $c = Captcha::reCAPTCHA->new;

my $cgi = CGI->new;
print $cgi->header;

my $public_key  = '........';
my $private_key = '........';

my %params = $cgi->Vars();

if ( !%params ) {

    # Output form
    print $cgi->start_form,
          $c->get_html(
             $public_key,
          ),
          $cgi->submit,
          $cgi->end_form;
}
else {

    # Verify submission
    my $result = $c->check_answer(
        $private_key,
        $ENV{'REMOTE_ADDR'},
        $params{recaptcha_challenge_field},
        $params{recaptcha_response_field},
    );

    if ( $result->{is_valid} ) {
        print „Yes!";
    }
    else {
        # Error
        print $result->{error};
    }
}
```

Mojolicious::Plugin::Captcha::reCAPTCHA

```
#!/usr/bin/perl

use strict;
use warnings;

use Mojolicious::Lite;

my $public_key  = 'xxxx';
my $private_key = 'xxxx';

  plugin 'Captcha::reCAPTCHA' => {
    private_key => $private_key,
    public_key  => $public_key,
    options     => { theme => 'white' },
};

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

    $self->stash(
        recaptcha_html =>
            $self->recaptcha
                 ->get_html($public_key),
    );

    $self->render( 'form' );
};

post '/' => sub {
    my $self = shift;
    my $r    = $self->req;
    my $ip   = $r->tx->remote_address;
    my $p    = $r->params->to_hash;

    my $challenge =
        $p->{recaptcha_challenge_field};
    my $response =
        $p->{recaptcha_response_field};

    my $captcha = $self->recaptcha;
    my $result  = $captcha->check_answer(
        $private_key,
        $ip,
        $challenge,
        $response,
    );

    if ( $result->{is_valid} ) {
        $self->render( text => 'Yes' );
    }
    else {
        $self->render(
            text => $result->{error},
        );
    }
};

__DATA__
@@ form.html.ep
<form method="post">
      <%= recaptcha_html %>
      <button type="submit" value="check">
          check
      </button>
  </form>
```

Listing 1

Catalyst::TraitFor::Controller::reCAPTCHA

```
package MyApp::Controller::Comment;
use Moose;
use namespace::autoclean;

BEGIN { extends 'Catalyst::Controller' }
with 'Catalyst::TraitFor::Controller::
        reCAPTCHA';

sub example : Local {
    my ( $self, $c ) = @;

    # validate received form
    if ( $c->forward('captcha_check') ) {
        $c->detach('my_form_is_ok');
    }

    # Set reCAPTCHA html code
    $c->forward('captcha_get');
}

1;
```

Listing 2

Für die meisten modernen Webframeworks gibt es schon fertige Plugins, mit denen die Einbindung in die Webanwendung sehr einfach wird. Hier ein paar Beispiele (siehe Listing 1 und Listing 2).

Man kann aber auch eigene Captchas mittels GD generieren. Allerdings muss man sich hierbei selbst darum kümmern, dass man das richtige Bild zum Request zuordnet.

```
use GD::SecurityImage;

# Create a normal image
my $image = GD::SecurityImage->new(
                width   => 80,
                height  => 30,
                lines   => 10,
                gd_font => 'giant',
            );
   $image->random( $your_random_str );
   $image->create( normal => 'rect' );
my ($data, $mime_type, $number) =
   $image->out;
```

Telefon

Nachdem jetzt gezeigt wurde, wie man mit Captchas arbeiten kann, kommen wir zu dem Telefon-Teil. Hier werden jetzt zwei APIs vorgestellt, mit dem man eine solche Lösung umsetzen kann: Telekom Tropo-API und Twilio. Beide APIs funktionieren fast identisch. Der grundsätzliche Ablauf ist in Abbildung 2 zu sehen.

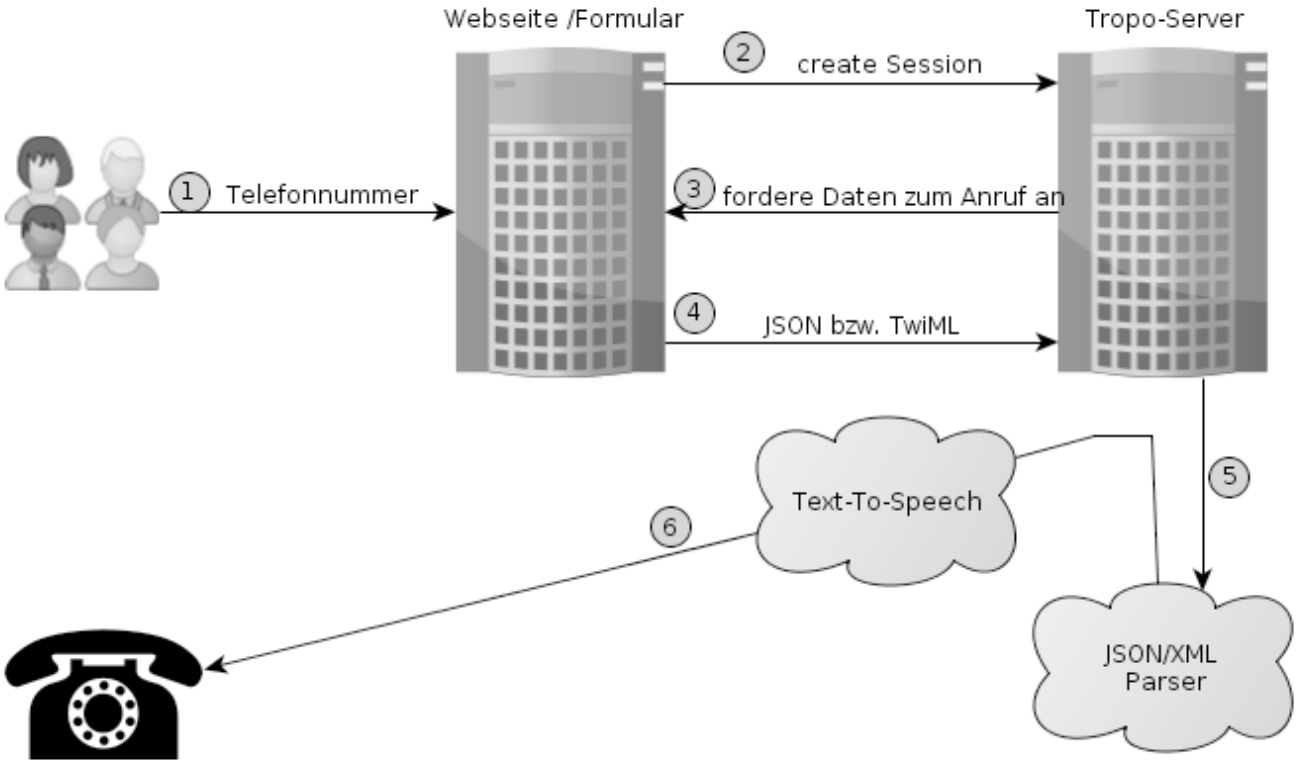


Abbildung 2: Anruf über Webseite auslösen

```
post '/' => sub {
    my $self = shift;

    $self->stash( WILL_CALL => 0 );
    my $phone = $self->param( 'phone' );

    if ( !$phone || !_check_phone( $phone ) ) {
        $self->stash( PHONE_ERROR => 1 );
    }
    else {
        my $id = $self->random_string;

        # save id and phone number in a database

        my $s = Tropo::RestAPI::Session->new(
            url => 'https://tropo.developergarden.com/api/',
        );

        my $data    = $session->create(
            token        => $token,
            call_session => $id,
        ) or $self->app->log->error( $session->err );

         $self->stash( WILL_CALL => 1 );
    }

    $self->render( 'form' );
};
```

Listing 3

```
$VAR1 = {
  'session' => {
                'userType' => 'NONE',
                'parameters' => {
                                  'token' => 'your_api_token',
                                  'action' => 'create',
                                  'call_session' => 'zRlbp7UET5ecDcneDCnoB4'
                                },
                'callId' => undef,
                'initialText' => undef,
                'timestamp' => '2013-09-06T18:53:20.168Z',
                'accountId' => '9183',
                'id' => '9884f64erb41e97948083c25980d63683'
              }
};
```

Listing 4

Der Webseitenbenutzer gibt im Formular der Webseite seine Telefonnummer ein und schickt das Formular ab. Die Webanwendung erzeugt eine Session auf dem API-Server und stößt mit einem Request an den API-Server alles Weitere an. Der API-Server bekommt den Request und ruft eine, bei der Registrierung hinterlegte, URL auf. Das ist eine Seite in der eigenen Webanwendung, die dem API-Server alle Informationen zur Verfügung stellen muss: Wer soll angerufen werden und welcher Text soll gesagt werden. Je nach API gibt es dann noch weitere Einstellungen. Hier ist ein Punkt, in dem sich die Tropo-API und die Twilio-API unterscheiden: der Tropo-Server verlangt die Daten im JSON-Format, Twilio in TwilioML, einem XML-Dialekt.

Tropo API

Für den Einsatz der Tropo API gibt es ein Modul auf CPAN: `Tropo`. Das unterstützt zwar noch nicht alle Tropo-Befehle, aber die Wichtigsten sind bereits implementiert. Die hier gezeigten Code-Stücke zeigen eine `Mojolicious::Lite`-Anwendung.

Um den Anruf zu initialisieren, wird zuerst ein Formular benötigt, in das der Benutzer seine Telefonnummer eingeben muss:

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

    $self->app->log->debug(
        'form requested'
    );

    $self->stash( WILL_CALL => 0 );
    $self->render( 'form' );
};

__DATA__
@@ form.html.ep
<% if ( $WILL_CALL ) { %>
<span style="background-color: green">
    We will call you in a moment</span>
<% } %>
<form action="" method="post">
    Your phone number (international
    format, e.g. +4912345678):
    <input type="text" name="phone" />
    <br />
    <button type="submit" value="Call">
        Call me!
    </button>
</form>
```

Wichtig hierbei ist, dass der Benutzer seine Telefonnummer auch wirklich in dem internationalen Format eingibt, da sonst der Tropo-Server keinen Anruf tätigen wird.

Nachdem der Benutzer das Formular mit seiner Telefonnummer abgesendet hat, muss die Session auf dem Tropo-Server initialisiert werden (Listing 3).

In dem Beispiel wird ein Zufallsstring erzeugt, der hier für die Zuordnung der Session zur Telefonnummer des Benutzers dient. Es ist auch möglich, die Telefonnummer bei der Initialisierung der Tropo-Session zu übergeben, aber dann wird die Nummer unnötig häufig hin und her geschickt.

Bei der Erzeugung des Session-Objekts wird die URL der Tropo-API bei DeveloperGarden angegeben. Die Tropo-API wird bei mehreren Anbietern eingesetzt und über die Angabe der URL kann das Modul auch für andere Anbieter eingesetzt werden.

Mittels `create` wird dann die Session erzeugt. Der Token ist der API-Token, der im „Application-Management" auf der DeveloperGarden-Webseite angegeben ist. Neben diesem Token können beliebige Parameter übergeben werden, die in der Session gespeichert werden sollen. In der Beispielanwendung werden alle relevanten Daten in einer eigenen Datenbank gespeichert, so dass nur die zur Tropo-Session gehörigen Daten identifiziert werden müssen. Das passiert hier über die ID, die in *call_session* übergeben wird.

Dieses *create* stößt den POST-Request an den Tropo-Server an. Danach wird dem Benutzer nur noch angezeigt, dass er in Kürze einen Anruf erhält.

Der Tropo-Server startet dann einen POST-Request an die Webanwendung und übergibt die Session-Daten als JSON im Body des Requests. Das übermittelte JSON ist in Listing 4 dargestellt.

In diesen Daten befindet sich auch die ID, die beim `create` für den Parameter *call_session* übergeben wurde. Damit wird dann die Telefonnummer des Benutzers aus der Datenbank gelesen. Schließlich muss noch der vierstellige Code erstellt werden.

```
post '/tropo/' => sub {
    my $self = shift;

    my $tropo_data = $self->req->json;
    my $session    = $tropo_data
       ->{session}->{parameters}
       ->{call_session};

    # get phone number by call_session
    # id and save it to $phone

    my $data = {};

    if ( $phone ) {
        my $tropo = Tropo->new;

        $tropo->call( $phone );
        $tropo->say(
            'activation code is ' .
            sprintf „%04d",
                int rand (9999)
        );
        $data = $tropo->perl;
    }

    $self->render( json => $data );
};
```

Wie bereits oben beschrieben benötigt der Tropo-Server als Antwort ein JSON-Objekt mit allen notwendigen Angaben. Als erstes wird ein Tropo-Objekt erzeugt. Dieses Objekt soll einen Anruf starten - das geschieht mit der Methode `call`, der die anzurufende Nummer übergeben wird. Über die Methode `say` wird Tropo dann mitgeteilt, was in dem Telefonat gesagt werden soll. Das war es auch schon. Über die Methode `perl` bekommt man die Perl-Datenstruktur, die über das `render` noch in JSON umgewandelt wird. `Tropo` bietet auch die Möglichkeit, direkt JSON zu bekommen, wird hier aber nicht gemacht, um beim Logging einfach ein `Dumper` verwenden zu können.

Das erzeugte JSON sieht dann wie folgt aus:

```
{„tropo":[{„call":{„to":"+4912345656778"}},
  {„say":{„value":"hello world"}}]}
```

Twilio

Twilio funktioniert ähnlich wie Tropo. Für das oben gezeigte Einsatzszenario werden die Module `WWW::Twilio::API` und `WWW::Twilio::TwiML` benötigt. Das Formular ist das Gleiche wie im Tropo-Beispiel. Um den Anruf zu initiieren wird ein Objekt von `WWW::Twilio::API` erzeugt. Für dieses Objekt wird dann die Methode `POST` aufgerufen, der man die Aktion, die Telefonnummern und eine URL übergeben muss:

```
use WWW::Twilio::API;

my $twilio = WWW::Twilio::API->new(
    AccountSid => 'AC12345...',
    AuthToken  => '1234567...',
);

## make a phone call
$response = $twilio->POST(
    'Calls',
    From => '1234567890',
    To   => '8905671234',
    Url  => 'http://domain.tld/call',
);
```

Die Aktion ist der API-Endpunkt bei Twilio und entspricht in diesem Fall *https://api.twilio.com/2010-04-01/Accounts/{YourAccountSid}/Calls*. Die Parameter *From* und *To* dürften selbsterklärend sein. Dem Parameter *Url* muss man die URL zur eigenen Webanwendung übergeben. Unter dieser URL holt sich Twilio die notwendigen Informationen zu dem Anruf ab: Was soll gesagt werden.

Als Antwort erwartet Twilio ein TwiML-Dokument. Im Endeffekt sieht das folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Response>
  <Say>Activation token is ...</Say>
</Response>
```

Soll das Dokument etwas komplexer werden, empfiehlt es sich, das Modul `WWW::Twilio::TwiML` für die Generierung des XML zu verwenden.

```
my $tw = WWW::Twilio::TwiML->new;
 $tw->Response->Say('Activation token is');
 my $twiml = $tw->to_string;
```

Von der Funktionalität und dem Funktionsumfang geben sich Tropo und Twilio nicht viel. Im Test haben beide APIs sehr gut funktioniert.

In der nächsten Ausgabe gibt es den zweiten Teil von „Bot or Not". Darin geht es um die Zwei-Faktor-Authentifizierung.

Herbert Breunung

Rezension - Regex

Barbara Hohensee
Perl Programmierung - Grundkurs
Eigenverlag, ca. 159 Seiten
15. Juli 2013
ASIN: B00DYZAS82
mobi(Kindle): 7,81€

Gabor Szabo
Beginner Perl Maven
perl5maven.com/products 257 Seiten
ständig aktualisiert
PDF: $32

Jeffrey E.F. Friedl
Reguläre Ausdrücke
O'Reilly, 560 Seiten
3.Auflage, Oktober 2007
ISBN 978-3-89721-720-1
Gebunden: €44.90
PDF, EPUB: €36.00

Der Standard in Buchformat zu Regulären Ausdrücken ist im Haus und so soll diese Folge auch nach ihm benannt sein. Doch genauso könnte es auch „Einsteigerliteratur II“ heißen, denn das versprochene Anfängerbuch vom Gabor und ein aktuelles, zumindest vom Erscheinungsdatum her, stehen auf der Menükarte. Der angekündigte Titel über Webprogrammierung wird dafür verschoben.

Grundkurs

Erscheint ein taufrisches Perlbuch in der Landschaft, löst es erst einmal einen leisen aber steilen Begeisterungsimpuls im EEG aus. Und gleich mit Übungen und Lösungen und einem etwas anderen, freundlichen Logo - Interessant. Doch die Freude verflog schnell, sobald die Blicke über die ausschließlich virtuellen Seiten wanderten. Mit „aktuelle Version 5“ war nämlich nicht 5.16 oder gar 5.18 gemeint, sondern 5.0. An der Stelle tat eine Überprüfung des Ausgabedatums not - wahrhaftig: 16 Juli 2013.

Unfassbar - die gute Frau kommt aus einer Zeit, als OS/2 und Perl 4 sich noch in freier Wildbahn bewegten und Menschen tatsächlich den Windows-Standardeditor benutzen um Perl zu schreiben - nicht *notepad*, sondern seinen ASCII Vorgänger *edit*, den ich ebenfalls bis circa 1993 benutzte. Die einzigen Bildschirmabbildungen zeigen wirklich dieses Programm. Gilbert Steger hat bei seinem doppelt so umfangreichen und nur halb so teuren eBuch *Zukunft.pl* (3/2012) wenigstens EPIC verwendet.

Sie erwähnt, dass Padre einsteigerfreundlich sei, verwendet ihn jedoch nicht. Aber richtig einsteigerfreundlich ist das ganze Buch nicht. Während eine Sprachfunktion nach der nächsten aufgezählt wird, werden Fachbegriffe wie *Strings*, *Fließkommazahlen*, *Fluchtsymbole* (Hohensee meint *escape-sequenzen*) ohne genaue Erläuterung eingeführt. Eine didaktische Struktur erschloss sich für mich, trotz teilweise cleverer Einzelschritte, nicht. Während sie eine `while`-Schleife erklärt, geht es fast nur um `last` ohne die anderen Sprungbefehle anzusprechen. `STDIN` wird zu Anfang verwendet, `STDOUT` erst in der Mitte und `strict` erst nach einem Drittel. Natürlich ist für sie Web gleich `CGI`, *HTML*-Tags werden mit Regex geparst und *typeglobs* sind „veraltet“. Derart „aus-

gebildete“ Neulinge werden es in Internetforen nicht leicht haben. Es wird nicht einmal der Umgang mit der Dokumentation gelehrt und (bis auf `CGI`) kein Modul angefasst. Die Beispiele, welche keine besonders hilfreichen Programme darstellen, enthalten Tippfehler und in Sachen Rechtschreibung wäre ein guter Lektor ebenfalls fündig geworden.

Wahrscheinlich waren die Perlkurse der Autorin in den 90ern gut für die Zuhörer und die positive Stimmung der Teilnehmer bewog sie es zu veröffentlichen. Doch ohne den Aufwand in Abrede zu stellen, die eine Verschriftlichung abverlangt, und die unübersehbare Liebe zu manchem Detail nicht ignorierend, empfinde ich das Werk als aktive Rufschädigung an der verwendeten Sprache.

Beginner Perl Maven

Nächster Kandidat - gleiche Disziplin, aber was für ein Unterschied.

Gleich zu Beginn: 3 anfängerfreundliche Distributionen, 11 Editoren (grummel grummel ohne Kephra), Perl in der Shell, erstes Skript mit strict und warnings, Erklärung warum das gut ist, Kommentare und Dokumentation (schreiben und benutzen). Ein derart ermächtigter Neuling wird bereits beinah auf zwei Beinen stehen können und seinen Weg finden.

Fast systematisch bekommt er nun die Schreibweisen von Literalen, Variablen und Kontrollstrukturen zu sehen, teilweise mit weniger Text als im Buch zuvor. Es werden auch kaum Begriffe geklärt, aber inhaltlich ist der Kenner (englisch Maven) wesentlich breiter gefächert und der Inhalt ist vor allem auf dem aktuellen Stand und (meist) unbedenklich. Wem die die knappen Sätze nicht reichen, der kann sich zusätzlich die passenden Filme (I und II für je 59$) dazu kaufen (eine Rezension ist ebenfalls geplant). Zugegeben, das ist nicht billig, aber von echtem Wert. Den 10-Stunden-Video-Kurs mit Legende Randal L. Schwartz *http://shop.oreilly.com/product/0636920014430.do* lässt sich O'Reilly 150 Dollar kosten. Leider gibt es beides, wie auch das Buch selber nur auf Englisch.

Mit den Filmen kommen jeweils auch die Übungen und Lösungen. Das Buch besitzt lediglich zwei sehr detailliert ausgearbeitete Indizes. Einen inhaltlichen und einen nach Schlüsselworten, Variablen, Operatoren, Regex und Weiterem. Ab Seite 168 kommt ein wirklich langer Appendix in dem echte Probleme besprochen werden wie etwa das berühmte Jahr *19100*, „konnte Modul nicht finden“ oder die listigen Nebenwirkungen des `open` mit zwei Argumenten. Sachlich und in kleineren Absätzen kommen seine Erläuterungen hierbei auf den Punkt. Die kleinen Beispielprogramme sind nützlich und sehen aus wie aus der täglichen Praxis. Auch die vielen Links zu Internetseiten aber auch selbst verfassten Videos und Artikeln werten das Buch spürbar auf. Da seine Netzpräsenz vor circa einem Jahr umzog, müssen jedoch noch einige Verknüpfungen aktualisiert werden. Das könnte sogar schnell geschehen, da dieses Werk wie seine letztens besprochene PerlMaven-Seite ständig und inkrementell weiterentwickelt wird. Die vorliegende Version 1.24 ist keine willkürlich gewählte Zahl und die Hoffnung ist berechtigt, dass noch einige sehr dünne, vordere Kapitel etwas mehr Text gewinnen. Mit dem Kauf erhält man das Recht, über *PerlMaven* auch zukünftige Versionen beziehen zu dürfen.

Reguläre Ausdrücke

Nur alle 5 Jahre (die gerade wieder um sind) wurde der Schmöker vom Friedl aktualisiert - das oft zitierte „Eulenbuch“. (Wobei kleinere Korrekturen unter *http://regex.info/book.html* nachgereicht werden.) Es bietet auf rund 500 Seiten Breite und Tiefgang und ist für all jene, die nicht nur die ganzen Funktionalitäten und ihre Schreibweisen kennen lernen wollen, um irgendetwas in einem Text zu suchen oder zu ersetzen. (Das bietet auch die `perldoc`.) Es ist für die Sucher nach dem großen Warum und Wozu. Dorthin führen diese Seiten mit einem angenehmen Ton, der nicht zu trocken oder langatmig ist, in Gedanken- und Handlungsbögen von Teilgebiet zu Teilgebiet.

Dabei hat man trotz der regelmäßigen Beispiele und Tabellen das Gefühl eine Geschichte zu lesen, die sich fast beliebig an jeder Überschrift beiseite legen und später wieder aufnehmen lässt, ohne den Zusammenhang zu verlieren. Freimütig gibt der Autor zu, erst durch das Schreiben dieser Kapitel und mit Hilfe einer Reihe von Koryphäen selbst zum Experten geworden zu sein. Doch gerade der dabei betriebene Aufwand und die Rücksicht gegenüber Lernenden zeichnen diesen Titel aus. Ohne überfordert zu werden, kann ein gerne denkender Mensch, hier die verschiedenen Regex-Dialekte kennen

lernen, ihre Fähigkeiten, Schreibweisen und Laufzeiten abwägen und er beginnt hier auch zu verstehen, was mit Fachsimpeleien wie NFA oder Backtracking gemeint ist.

Das 84 Seiten lange Kapitel 7 befasst sich ausschließlich mit den Besonderheiten von Perl's Regex-Maschine, danach auch mit *Java*, *.Net* und *PHP*, was allerdings nur noch ein Viertel des Gesamtumfangs ausmacht. Es ist daher das derzeit tiefschürfendste Buch zu Perls regulären Ausdrücken. Nur schmerzt es ein wenig, dass es nicht aktuell ist. (Das 2012er-Datum bezieht sich nur auf die Konvertierung in ein elektronisches Buch.)

Denn just in den eingangs erwähnten, letzten fünf Jahren seit 5.10 hat sich wesentliches geändert. Rekursive Ausdrücke, benannte Teilausdrücke und die neuen Unicode-Fähigkeiten sind alle nicht erwähnt. Natürlich war das Erscheinungsdatum wenige Monate vor 5.10, aber mit etwas Beratung eines Perl-Insiders oder einem Blick in die p5p hätte man das vermeiden können. Andererseits spricht es vollständig für die Perl-Porter, dass mit den Mengenoperatoren für Zeichenklassen, die 5.18 brachte, sämtliche Verbesserungsvorschläge vom Friedl für Perl bereits umgesetzt sind. Mit Mengenoperatoren auf Zeichenklassen lässt sich zum Beispiel recht lesbar ausdrücken: „An dieser Stelle sollte ein Buchstabe des deutschen oder norwegischen Zeichensatzes mit Umlauten stehen, aber kein ü“. Ebenfalls eine Sache, die ich zu gerne in diesem Buch gesehen hätte: Damian Conways unglaublicher `Regexp::Debugger` (*http://metacpan.org/module/Regexp::Debugger*) mit dem man in einem selbst bestimmten Tempo der Regex beim Backtracking zusehen kann. Doch zum Glück gibt es gute Erläuterungen im Netz zu „neueren“ Regex-Funktionen wie etwa *http://perltraining.com.au/tips/*, welche aber leider nicht die Tiefe haben und so gut ausformuliert sind wie das Pfund von Jeffrey E. F. Friedl (übersetzt von Andreas Karrer).

Ausblick

Nach dem Einstieg in die Regulären Ausdrücke im letzten Heft, welches seine Leser lediglich so fix wie möglich produktiv werden läss und der heutigen Erklärung die „Alles“ enthielt, wird im Frühjahr 2014 diese Trilogie mit dem *Kochbuch Reguläre Ausdrücke* von Jan Goyvaerts und Steven Levithan, also reiner Praxisarbeit abgeschlossen. Der ständig emsige Gabor hat noch sein eBuch für Fortgeschrittene in petto. Und für Abwechslung sorgt Mark C. Chu-Carroll mit „Good Math“ - Mathematik für Programmierer. Das wird zusammen mit seinen ebenfalls sehr tüchtigen Herausgebern, den *Pragmatic Publishers* vorgestellt.

Thomas Fahle

HowTo - Crypt::SaltedHash

Crypt::SaltedHash - eine einfach zu bedienende Bibliothek zum Erzeugen und Validieren gesalzener Hashes

Crypt::SaltedHash von Sascha Kiefer bzw. Gerda Shank ist eine einfach zu bedienende Bibliothek zum Erzeugen und Validieren *gesalzener Hashes*.

Salted Hashes werden oft zur sicheren Speicherung von Passwörtern verwendet. Über die Sicherheit gesalzener Hashes mag man trefflich streiten. Diese kleine Anleitung wendet sich an Anwender dieses in der Praxis oft anzutreffenden Verfahrens und behandelt die beiden praxisrelevanten Fälle Erzeugen und Prüfen von *Salted Hashes*.

Beispiel: Salted Hash erzeugen

Die Methode `generate()` erzeugt einen gesalzenen Hash aus einem Klartextpasswort und gibt diesen als RFC2307-(userPassword)-codierte Zeichenkette zurück.

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

use Crypt::SaltedHash;

my $csh = Crypt::SaltedHash->new(
          algorithm => 'SHA-1' );

my $cleartext = 'secret';

$csh->add( $cleartext );

my $salted = $csh->generate;

print „Salted: $salted\n";
```

Das Programm erzeugt z.B. folgende Ausgabe:

```
Salted:
  {SSHA}9GnzDgL3ChgeupyOkQtSrN/0/v8sGBf3
```

In den geschweiften Klammern steht eine Kennung für den verwendeten Algorithmus, gefolgt vom einem MIME Base 64 codiertem String, der wiederum die Verkettung des Hashes und des zufälligen Salts enthält.

Das zufällige `Salt`, welches zusammen mit dem Hash ausgegeben wird, kann mit der Methode `salt_hex()` einfach ausgelesen werden.

Stärkere Algorithmen, bitte!

Der gewünschte Hashingalgorithmus kann über den Parameter `algorithm` eingestellt werden - hier sind alle Algorithmen zulässig, die von *Digest* unterstützt werden, z.B. `SHA-256`, `SHA-384` oder `SHA-512`. Per Vorgabe wird `SHA-1 (FIPS 160-1)` verwendet.

Beispiel:

```
my $csh = Crypt::SaltedHash->new(
          algorithm => 'SHA-256' );
```

Das Beispielprogramm von oben erzeugt nun z.B. folgende Ausgabe:

```
Salted: {SSHA256}fqlu4iT+
  JI2MsDWAH7OtlFARRrKL8OibOawogcsezQs/v9Qg
```

Beispiel: Salted Hash validieren

Die Methode `validate()` kann einen vorgegebenen gesalzenen Hash validieren:

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

use Crypt::SaltedHash;

my $csh = Crypt::SaltedHash->new(
           algorithm => 'SHA-1' );

my $cleartext = 'secret';

my $salted =
  '{SSHA}9GnzDgL3ChgeupyOkQtSrN/0/v8sGBf3';

my $valid = Crypt::SaltedHash->validate(
             $salted, $cleartext );

if ($valid) {
   print „OK\n";
} else {
   print „Not OK\n";
}
```

Crypt::SaltedHash kann natürlich auch die gesalzenen Hashes, die mit anderen Programmen, z.B. `slappasswd`, dem OpenLDAP-Passwort-Werkzeug, erzeugt wurden, validieren.

In Listing 1 habe ich das Klartextpasswort `123456` ein paar Mal durch `slappasswd` laufen lassen, um die unterschiedlichen gesalzenen Hashes für dasselbe Passwort zu erzeugen.

Das Programm erzeugt folgende Ausgabe:

```
OK ({SSHA}jppWV0DCxDJeOtaSS434nv5WewNYZSCS)
OK ({SSHA}ilJ95JAFKS4TgDbjRkcYrBMBRp+4mmCE)
OK ({SSHA}vrPil747oBHVgriDBbE+04XAs6BhXis0)
OK ({SSHA}ew3Xf1C9vK+H0kNeJ12Gc1M4fpbT41+x)
OK ({SSHA}9bRacd1WzrrUTiav7QBkMrxHtKTbjhpi)
OK ({SSHA}4m91OtOLhIfL+xDlfO/L5YdYaABKF60U)
OK ({SSHA}b+8sUhnuy6gnmlpf5BD58pww8NiQDZ3S)
OK ({SSHA}SeN6Yubt+pFOfTYDPPQ8JVp7MSfBkr1l)
OK ({SSHA}oSX0fmh8wNPGaJgEGQHY28RMiawsmraB)
OK ({SSHA}K4+dl7hft3VY7gfjPOdz8OOyXflbxA1f)
OK ({SSHA}7X/z6qeaDP6NpKQM3PYUQrERTTTj+VPD)
```

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

use Crypt::SaltedHash;

my $csh = Crypt::SaltedHash->new(
           algorithm => 'SHA-1' );

#$ slappasswd -s 123456

my @salted = qw!
  {SSHA}jppWV0DCxDJeOtaSS434nv5WewNYZSCS
  {SSHA}ilJ95JAFKS4TgDbjRkcYrBMBRp+4mmCE
  {SSHA}vrPil747oBHVgriDBbE+04XAs6BhXis0
  {SSHA}ew3Xf1C9vK+H0kNeJ12Gc1M4fpbT41+x
  {SSHA}9bRacd1WzrrUTiav7QBkMrxHtKTbjhpi
  {SSHA}4m91OtOLhIfL+xDlfO/L5YdYaABKF60U
  {SSHA}b+8sUhnuy6gnmlpf5BD58pww8NiQDZ3S
  {SSHA}SeN6Yubt+pFOfTYDPPQ8JVp7MSfBkr1l
  {SSHA}oSX0fmh8wNPGaJgEGQHY28RMiawsmraB
  {SSHA}K4+dl7hft3VY7gfjPOdz8OOyXflbxA1f
  {SSHA}7X/z6qeaDP6NpKQM3PYUQrERTTTj+VPD
  !;

my $cleartext = '123456';

foreach my $salted (@salted) {

    my $valid = Crypt::SaltedHash->validate(
                 $salted, $cleartext );
    if ($valid) {
        print „OK ($salted)\n";
    } else {
        print „Not OK ($salted)\n";
    }
}
```

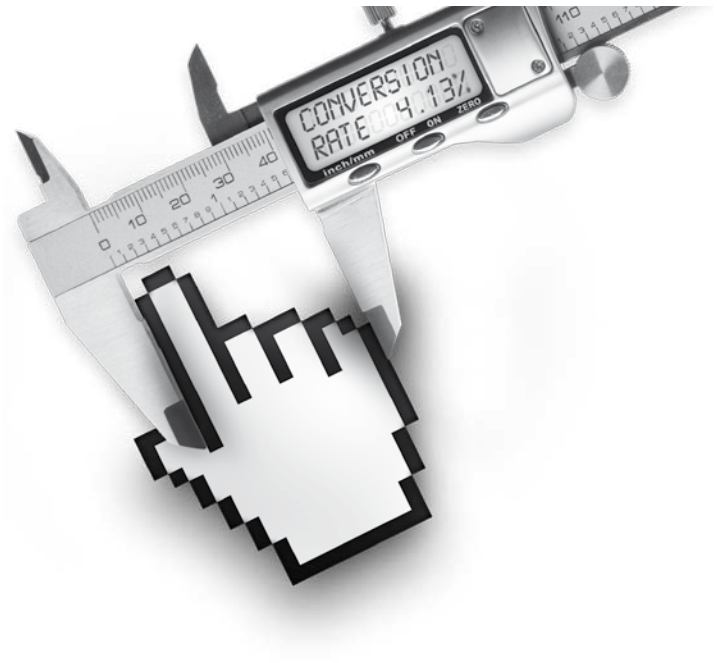
Listing 1

Sicherheitshinweis

Gesalzene Hashes sind, unabhängig vom gewählten Hashingalgorithmus, unter Sicherheitsapekten grundsätzlich genau so zu behandeln, wie Klartextpasswörter. Auch *Salted Hashes* können durch Brute-Force- oder Wörterbuchattacken angegriffen werden.

NETslave

QUALITYCLICK



$_=~ /Programmierer/i;

Perl Web-Entwickler (m/w)

für unseren Standort in Berlin

Die NetSlave GmbH ist ein junges, inhabergeführtes Online-Software Unternehmen in Berlin. Unser Hauptprodukt ist die Affiliate Marketing Software QualityClick, die es unseren internationalen Kunden ermöglicht eigene Inhouse Partnerprogramme leistungsfähig zu betreiben.

Beschreibung & Anforderungen

- Sie verfügen über ausgeprägte Kenntnisse in Perl
- Sie haben Erfahrung in der Administration von Linux-Systemen
- Der Umgang mit MySQL und Datenbankstrukturen ist Ihnen bestens vertraut
- mod_perl ist eine bekannte Umgebung für Sie
- JavaScript-, CSS- und XHTML-Erfahrung sind wünschenswert
- Umgang mit XML- und SOAP-Schnittstellen wäre vorteilhaft

- Sie kommunizieren sicher in Deutsch und besitzen gute Englischkenntnisse
- Sie verfügen über starke analytische Fähigkeiten
  und haben Freude an unkonventionellen Problemlösungen
- Kreativität, Engagement, Zielstrebigkeit und Qualitätsbewusstsein

Bewerbungsunterlagen bitte an: **jan.bischoff@netslave.de**

NETslave

NetSlave GmbH
Simon Dach Str. 12
10245 Berlin
Germany

Tel: +49 (0)30 94408-730
Fax: +49 (0)30 96083-706
E-Mail: mail@netslave.de
Web: **www.netslave.de**

CPAN News

File::HashCache

Bei Webanwendungen gibt es immer wieder das Problem, dass an JavaScript- oder CSS-Dateien etwas angepasst wurde, diese Änderungen aber nicht sofort beim Benutzer ankommen, weil der Browser die Datei cached. Hier hilft es, nach der Änderung den Namen der Datei anzupassen. Um nicht hunderte Vorkommen der Einbindung des Skripts anzupassen - wobei man dabei garantiert das eine oder andere Vorkommen vergisst -, kann man das Generieren der Datei automatisieren.

Hier hilft `File::HashCache`. Auf Basis des Dateiinhalts wird der Dateiname erweitert und in einem "Cache"-Verzeichnis gespeichert.

```
use File::HashCache;

my $hc = File::HashCache->new(
    cache_dir => '/var/www/js/cache',
);

my $hashed_path = $hc->hash(
    '/var/www/js/test.js'
);
```

Der Aufruf der `hash`-Methode muss in jedem Request stattfinden, denn dann überprüft das Modul ob sich die Datei geändert hat und generiert gegebenenfalls eine neue Datei.
Die Integration in Template::Toolkit könnte so aussehen, wie in Listing 1 dargestellt.

Das Modul bietet aber noch mehr Möglichkeiten. Man kann das Modul auch so verwenden, dass beim Erzeugen der gecachten Dateien das JavaScript bzw. das CSS komprimiert wird:

```
use JavaScript::Minifier::XS qw(minify);
my $hc = File::HashCache->new(
    cache_dir   => 'js',
    process_js  => \&minify,
    process_css =>
        \&CSS::Minifier::XS::minify,
);
```

```
$template->process('template.tt2', {
    script => sub {
        my $path = $hc->hash($_[0]);
        "<script src=\"js/$path\" type=\"text/javascript\"></script>\n";
    } } ) || die $template->error();

# And in your template.tt2 file:
#    [% script("myscript.js") %]
# which will get replaced with something like:
#    <script src="js/myscript-708b88f899939c4adedc271d9ab9ee66.js"
#            type="text/javascript"></script>
```

Listing 1

Git::Repository::Log

git ist ein verteiltes Versionskontrollsystem (*VCS*), das auch bei der Perl-Kernentwicklung verwendet wird. Der Vorteil gegenüber älteren - zentralen - Systemen wie *svn* oder *CVS* ist, dass man auch unterwegs Änderungen *committen* kann, egal ob man eine Internetverbindung hat oder nicht. Sobald man wieder online ist, kann man die gesammelten Änderungen an Remote-Repositories *pushen*.

Auch wenn bei *git* jede Kopie des Repositories als "Hauptrepository" dienen kann, wird es bei den meisten Projekten ein zentrales Repository geben, in dem alle Änderungen gesammelt werden.

In manchen Fällen muss man aus einem Perl-Programm heraus mit einem Git-Repository arbeiten - z.B. das Anlegen neuer Repositories, Logs abfragen etc. Entweder man nutzt direkt die git-Befehle und greift sich die Daten mittels `qx//` oder ähnlichen ab, oder man nimmt ein Modul. Eines der Module ist `Git::Repository`.

```
# neues git repo erzeugen
Git::Repository->run(
     init => $dir,
);
$r = Git::Repository->new(
   work_tree => $dir,
);

# mit bestehendem git repo arbeiten
$repo = Git::Repository->new(
    work_tree => '/local/git-repo/',
);

# mit bestehendem git repo arbeiten
$repo = Git::Repository->new(
    git_dir => '/local/git-repo/.git',
);

chdir $git_dir;

my $add_output = Git::Repository->run(
    add => $file,
);

my $commit_output = Git::Repository->run(
   commit =>
    '-m'       => $file,
    '--author' => $author,
    $file
);
```

`$dir` ist ein beliebiger Pfad. `Git::Repository` selbst ist so ausgelegt, dass es theoretisch auf allen Plattformen funktioniert. Aber das Modul benutzt zum Absetzen der git-Befehle das Modul `System::Command` und das hat Probleme mit Windows. Philippe Bruhat, der Autor beider Module, ist für Hilfe dankbar.

Man hat auch die Möglichkeit, `run` als Objekt-Methode und nicht als Klassenmethode aufzurufen. Dann ist das `chdir` in das Verzeichnis des Repositories nicht notwendig:

```
my $r = Git::Repository->new(
    work_tree => $git_dir,
);

my $add_output = $r->run(
    add => $file,
);
```

Wenn man `run` als Klassenmethode aufruft und sie *nicht* im Verzeichnis des Repositories aufruft, bekommt man die Fehlermeldung *fatal: Not a git repository (or any of the parent directories)*

Diejenigen, die sich mit *git* auskennen, sehen, dass `Git::Repository` die `git`-Befehle 1:1 benötigt.

```
my $commit_output = Git::Repository->run(
    commit =>
    '-m'   => 'fixed annoying bug',
    '-a',
);
```

entspricht dem git-Befehl `git commit -m 'fixed annoying bug' -a`. Das Modul ist also nicht wirklich eine Abstraktion. Warum also das Modul nutzen? Zum einen bietet es die Möglichkeit, Plugins zu schreiben. Zum anderen existiert die Methode `command`. Diese wird so benutzt wie die `run`-Methode, allerdings liefert es nicht einfach den Ausgabestring von `git` zurück, sondern ein Command-Objekt. Das holt nicht die komplette Ausgabe auf einmal. Das ist vorteilhaft wenn man Befehle nutzen möchte, die große Ausgaben produzieren (z.B. `git log`) und somit viel Speicher benötigen würden.

```
my $cmd = $r->command(
    log => '--pretty=oneline',
    '--all',
);

my $log = $cmd->stdout;
while (<$log>) {
      ...;
}

$cmd->close;
```

Und hier kommt dann auch das Modul zum Tragen, das hier eigentlich vorgestellt werden soll: `Git::Repository::Log`. Das ist ein solches Plugin, das man für `Git::Reposi-`

tory schreiben kann. Das Modul bietet einen Iterator und erstellt aus den einzelnen Log-Meldungen Objekte, so dass man bequem auf die Informationen der Log-Meldung zugreifen kann.

```
# load the Log plugin
use Git::Repository 'Log';

# get the log for last commit
my ($log) = Git::Repository->log( '-1' );

# get the author's email
print my $email = $log->author_email;
```

Möchte man einen Stream von Logmeldungen haben und auswerten, sollte mit dem Iterator gearbeitet werden:

```
use Git::Repository::Log::Iterator;

# use a default Git::Repository context
my $iter =
   Git::Repository::Log::Iterator
     ->new('HEAD~10..');

# or provide an existing instance
my $iter =
   Git::Repository::Log::Iterator->new(
     $r, 'HEAD~10..' );

# get the next log record
while ( my $log = $iter->next ) {
    print $log->author_email,"\n";
}
```

Data::Section

Manchmal ist es sehr praktisch, feste Textteile in den __DATA__-Bereich zu schreiben. Gerade wenn es kleine Textteile sind lohnt es sich nicht, diese in Dateien oder gar eine Datenbank zu schreiben. Nur oft genug hat man das Problem, dass man gerne mehrere Textteile in den Bereich schreiben möchte. Hier hilft dann Data::Section.

```
package MyConfig;
use Data::Section -setup;

# get_config('prod');
# get_config('dev');

sub get_config {
    my ($class, $type) = @_;

    my $data = $self->section_data(
                        $type || 'dev' );
    my %config = map{
       split /\s*=\s*/ }split /\n/, $data;
    return %config;
}

__DATA__
__[ dev ]__
host = localhost
db = test

__[ prod ]__
host = test.de
db = prod
```

App::MojoSlides

Perl-Konferenzen. Perl-Themen. Warum dann nicht eine Präsentationssoftware, die in Perl geschrieben ist. Joel Berger nutzt bei `App::MojoSlides` - wie der Name schon ahnen lässt - Mojolicious. Aus Mojolicious-Templates werden die Folien generiert, wobei für die Templates eigene Hilfsfunktionen existieren, um das Verhalten an LaTeX-Beamer anzunähern.

Für Syntaxhighlighting wird `PPI` genutzt und für das Basislayout wird Bootstrap eingesetzt (wobei man das natürlich anpassen kann).

Das Konfigurationsfile (hier: *test_slides.pl*) muss zu einem Hash evaluieren. Die Slides kann man als Templates direkt in den `__DATA__`-Bereich packen. In der Konfiguration muss man dann noch angeben, welche Slides es gibt. Mit `mojo_slides test_slides.pl daemon` kann man dann die Anwendung starten und im Browser ist die Präsentation unter *http://localhost:3000* erreichbar.

```
use Mojo::Base -strict;

my $config = {
    slides          => [qw/start seite2/],
    bootstrap_theme => 1,
};

__DATA__
@@ start.html.ep
% title '$foo - Test';
%= column 6 => begin
%= overlay "1-" => begin
Test 1
% end
%= overlay "2-" => begin
Erst beim zweiten Klick sichtbar
% end
% end

%= column 6 => begin
%= overlay 1 => begin
Spalte 2
% end
% end

@@ seite2.html.ep
% title 'Seite2';
%= p 'test'
```

Regexp::VerbalExpressions

Wer schon immer lesbare Reguläre Ausdrück haben wollte, kann sich mal `Regexp::VerbalExpressions` anschauen. Dort sind für die Elemente von Regulären Ausdrücken entsprechende Methoden vorhanden. Das ist Modul ist ganz nett um Einsteigern Regexe näherzubringen und zu zeigen wie man Regexe aufbauen kann. Ein Ersatz für die puristischen Reguläre Ausdrücke ist es aber nicht, da auch nicht alle Features der Regex-Engine umgesetzt wurden.

```
use Regexp::VerbalExpressions;

# Regex for URLs
my $re = verex
     ->start_of_line
     ->then('http')
     ->maybe('s')
     ->then('://')
     ->maybe('www.')
     ->anything_but(' ')
     ->end_of_line;

if ('https://www.google.com/' =~ $re) {
    print 'We have a correct URL';
}

# ^(?:http)(?:s)?(?:\:\/\/)(?:www\.)?(?:[^\
   ]*)$
print $re;
```

Pod::Markdown

Markdown ist aktuell die Markup-Sprache schlechthin. Viele Wikis und Github "verstehen" Markdown und auch sonstige Programme (z.B. pandoc), die das Format verarbeiten können, gibt es genug. Liegt die Dokumentation für ein Modul in *Pod* vor und wird die Dokumentation in Markdown benötigt, dann hilft `Pod::Markdown`. Dieses Modul ist von `Pod::Parser` abgeleitet, kennt also die gleichen Methoden und Einstellungen. Die wichtigste Methode ist `as_markdown`.

```
#!/usr/bin/perl

use strict;
use warnings;

use Pod::Markdown;

my $parser = Pod::Markdown->new;
$parser->parse_from_file( $0 );
print $parser->as_markdown;
```

NEWS

November 2013

02.-03. Österreichischer Perlworkshop
05. Treffen Frankfurt.pm
Treffen Stuttgart.pm
Treffen Hannover.pm
07. Peking Perl-Workshop
12. Treffen Dresden.pm
09.-10. OpenRheinRuhr
15.-16. YAPC::Brasil
Linuxinformationstage Oldenburg
18. Treffen Erlangen.pm
19. Treffen Hannover.pm
20. Treffen Darmstadt.pm
23. Nordic Perl-Workshop
26. Treffen Bielefeld.pm
27. Treffen Berlin.pm
30. London Perl-Workshop

Dezember 2013

03. Treffen Frankfurt.pm
Treffen Stuttgart.pm
Treffen Hannover.pm
05. Treffen Dresden.pm
06.-08. patch -p1 hackathon in Paris
09. Treffen Ruhr.pm
11. Treffen Niederrhein.pm
16. Treffen Erlangen.pm
17. Treffen Hannover.pm
18. Treffen Darmstadt.pm
19.-21. YAPC::Asia
25. Treffen Berlin.pm
31. Treffen Bielefeld.pm

Januar 2014

06.-10. Linux.conf.au
07. Treffen Frankfurt.pm
Treffen Stuttgart.pm
08. Treffen Niederrhein.pm
14. Treffen Ruhr.pm
20. Treffen Erlangen.pm
21. Treffen Hannover.pm
29. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss. Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perlfoundation.org

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

http://www.Perl.org

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

Perl-Services.de

Programmierung - Schulung - Perl-Magazin
info@perl-services.de