2014 GERMAN PERL WORKSHOP

26.-28.03. / HANNOVER

```
Social::Event->new(
   sponsored_by => 'XING'
);
```

Vorwort

Vor 15 Jahren fand der erste Deutsche Perl-Workshop statt, das war im Februar 1999 in Sankt Augustin bei Bonn, noch vor der ersten YAPC im Juni desselben Jahres in Pittsburgh. Es war also die allererste Grassroots-Veranstaltung für Perl. Daraus hat sich ein lebhaftes internationales Biotop von YAPCs, Workshops, Hackathons usw. entwickelt, ca. 25 bis 30 pro Jahr.

2014 hat es der Perl-Workshop erstmals in die Messestadt Hannover geschafft. Hier gibt es so viele motivierte Perl-Anwender, dass sich genug davon zu einem lebendigen und dauerhaften Hannover.pm zusammengefunden haben. Ohne sie und Renée und Max von Frankfurt.pm gäbe es diesen 16. Workshop nicht. Ein riesiges Dankeschön weiterhin unseren zahlreichen und großzügigen Sponsoren, ohne deren Unterstützung wir diese Veranstaltung nicht hätten realisieren können. Last but not least herzlichen Dank an alle Vortragenden, ohne euch würde der Workshop nur drei Minuten statt drei Tage dauern.

Offizieller Hashtag: #gpw2014

Wir danken den folgenden Sponsoren, ohne deren Unterstützung der Workshop nicht möglich wäre

IMPRESSUM

Herausgeber:	Frankfurt Perlmongers e.V. c/o Max Maischein Röderbergweg 167 60385 Frankfurt
Redaktion:	Renée Bäcker
Anzeigen:	Renée Bäcker
Layout:	Katrin Bäcker
Auflage:	100 Exemplare

Inhaltsverzeichnis

Boris Däppen

Philosophische Auseinandersetzung mit Software-Tests

Abstract

Testen wird in der Perl-Gemeinschaft gross geschrieben.
Aber was machen wir eigentlich wenn wir das t-Verzeichnis mit Code füllen?
Mit Zuhilfenahme aktueller Theorien aus der Darmstä8dter Technikphilosophie versuche ich hier einen Vorschlag zur Orientierung zu geben.
Das Versprechen: Nachher schreibt man bessere Tests - hoffentlich.

Über den Autor

Boris Däppen lernte Perl im Umfeld der Finanzdienstleister in Zürich kennen und vertiefte seine Kenntnisse der Sprache später bei perl-services.de.
Nach seinem Master in Technik und Philosophie an der TU Darmstadt arbeitet er als Informatik-Pädagoge und Berufsschullehrer in der Schweiz.
Diverse Artikel im $foo-Magazin und kleinere CPAN-Module gehören zu seinen Beiträgen zur Perl-Community.

Mathias Weidner

Das Modul WebService::Leanpub

Abstract

Dieser Vortrag beschreibt das genannte Modul und seinen Einsatz. Mit diesem Modul ist es möglich das Web-API von Leanpub in Perl-Programmenoder auf der Kommandozeile zu nutzen.
Es ist vor allem für Autoren bei Leanpub interessant und für ihre Helfer bei der Buchproduktion.

Was ist das Leanpub-API

Das Leanpub API ist ein Web-API, beschrieben in *https://leanpub.com/help/api*.

Für die meisten Funktionen benötige ich einen API-Schlüssel, diesen kann ich als Autor bei Leanpub bekommen. Mit dem API-Schlüssel kann ich nur Bücher bearbeiten, deren Autor ich bin.

Ein weitere wesentliche Komponente ist der sogenannte Slug, das ist der Teil der URL hinter *https://leanpub.com/*, der den Pfad zum Buch ausmacht.
So ist zum Beispiel die URL des Buches *Using the Leanpub API with Perl*, *https://leanpub.com/using-the-leanpub-api-with-perl*, und der Slug zu diesem Buch ist `using-the-leanpub-api-with-perl`.

Nebenbei: Sollte jemand auf Grund dieses Vortrages dieses Buch erstehen wollen, rate ich dazu, E-Mail-Benachrichtigungen für Neuausgaben abzustellen, weil ich das Buch zum Testen des APIs verwende.

Was kann ich damit machen?

Mit dem API kann ich

- Vorschauen für das ganze Buch erzeugen, für Teile des Buches oder einzelne
Dateien,
- das Buch veröffentlichen,
- eine Zusammenfassung des Buches bekommen,
- die Verkaufsdaten abfragen,
- Rabattcoupons verwalten.

Wie verwende ich das Perl-Modul?

Der Dreh- und Angelpunkt des Moduls ist ein Objekt vom Typ `WebService::Leanpub`, das für Aktionen rund um genau ein Buch verwendet wird. Aus diesem Grund gebe ich beim Aufruf von `new()` den API-Key und den Slug an.

```
use WebService::Leanpub;

my $wl = WebService::Leanpub->new(
    $api_key,
    $slug,
);
```

Mit diesem Objekt rufe ich verschiedene Methoden auf, je nachdem, was ich machen möchte. Diese Methoden geben als Antworttext den Text des Web-APIs aus (JSON-Format), den ich ausgeben oder interpretieren kann.

Das Erzeugen vollständiger und partieller Vorschauen ist sehr einfach, dafür gibt es je eine Methode. Welche Dateien für die Vorschau herangezogen werden, steht in den Dateien *Book.txt* beziehungsweise *Subset.txt*, Autoren bei Leanpub

wissen, wie diese Dateien auszusehen haben.

```
$pv = $wl->preview();

$pv = $wl->subset();
```

Eine Vorschau für eine einzelne Datei zu erzeugen, ist etwas komplizierter, weil die Datei via POST-Request an das Web-API gesendet wird. Das folgende Beispiel zeigt, wie eine Datei geöffnet wird, ihr Inhalt in eine skalare Variable eingelesen und dann zum Web-API geschickt wird.

```
if (open(my $input, '<', $filename)) {
  local $/;
  undef $/;
  my $content = <$input>;
  close $input;

  $pv = $wl->single({
    content => $content
  });
}
```

Ein Buch zu veröffentlichen ist einfacher, in der Hashreferenz `$opt`, die ich als Argument angebe, kann ich mitteilen, ob die Leser eine E-Mail bekommen sollen und was darin stehen soll. Details stehen in der Handbuchseite des Moduls.

```
$pv = $wl->publish( $opt );
```

Da das Erzeugen einer Vorschau und das Veröffentlichen lediglich durch das Web-API gestartet werden und dann asynchron innerhalb von Leanpub weiter laufen, will ich vielleicht wissen, wie der Stand der Dinge ist. Dazu kann ich den Status des letzten Jobs abfragen:

```
$pv = $wl->get_job_status();
```

Um die Buchzusammenfassung zu erhalten, rufe ich die Funktion `summary()` auf. Das Web-API benötigt hierfür keinen API-Key, das Perl-Modul für `new()` schon.
Will ich wissen, wie die Ausgabe ohne API-Key aussieht, kann ich beim Aufruf von `WebService::Leanpub->new()` einen falschen API-Key angeben.

```
$pv = $wl->summary();
```

Für die Verkaufsdaten gibt es zwei Methoden.Die eine liefert mir eine Zusammenfassung, die andere die Daten zu den einzelnen Verkäufen. Die zweite Funktion liefert mir die Daten zu den 50 letzten Verkäufen.
Will ich ältere Daten haben, gebe ich an, die wievielte Seite ich sehen möchte.

```
$pv = $wl->get_sales_data();

$pv = $wl->get_individual_purchases( { } );

$pv = $wl->get_individual_purchases( {
  page => 2
} );
```

Rabattcoupons kann ich erzeugen, ändern und auflisten. Details stehen in der Modul-Dokumentation.

```
$pv = $wl->create_coupon(\%lopt);

$pv = $wl->update_coupon(\%lopt);

$pv = $wl->get_coupon_list()
```

Ich will nichts programmieren

Das brauche ich auch nicht (mehr), denn mit dem Kommandozeilenprogramm `leanpub`, das in der Distribution enthalten ist, kann ich das Web-API recht einfach in Makefiles nutzen.
Das Programm nutzt `Getopt::Long` und `Pod::Usage`, kann also einen kurzen Hilfetext oder die komplette Handbuchseite ausgeben, wenn man es darum bittet.

```
$ leanpub --help
Usage:
     leanpub [options] command \
        [command options]
...
```

Generell teile ich dem Programm beim Aufruf mit einem Befehl mit, was ich will. Die Befehle heißen in etwa, wie die

Methoden.
Vor dem Befehl kommen globale Optionen, dahinter Befehlsoptionen. Die Handbuchseite ist da sehr ausführlich.

```
  Options:
    -api_key=key
             Provide the Leanpub API key to be
used for all actions.
    ...
    -really State that you really intend to do
the command (e.g. publish).
    ...
    -slug=your_book
            Provide the book's slug.
    ...
```

Von den globalen Optionen sind `-api_key` und `-slug` so wichtig, dass ich sie nicht immer eingeben muss. Nämlich genau dann, wenn ich sie in einer Konfigurationsdatei namens *.leanpub* angegeben habe.

```
$ cat .leanpub
# configuration for leanpub
#
api_key = my_api_key_from_leanpub
slug    = using-the-leanpub-api-with-perl
```

Etwas kurios mag vielleicht die Option `-really` anmuten. Also wirklich? Ich hätte diese auch `-do_as_I_say` nennen können, aber das erschien mir zu lang.
Diese Option ist nur beim Befehl `publish` wirksam und genau dann ist sie auch erforderlich.
Das ist so etwas wie eine Notbremse, damit man nicht unbeabsichtigt das Buch veröffentlicht, wenn es eigentlich noch nicht bereit ist.

Fazit

Das Modul `WebService::Leanpub`, genauer gesagt, das darin enthaltene Kommanodzeilenprogramm `leanpub` ist bei mir zusammen mit einem Makefile bei der Buchproduktion im täglichen Einsatz.
Es fügt sich nahtlos in meinen Workflow ein, der aus einem Editorfenster, einer Shell und einem PDF-Betrachter besteht.
Ich vermeide damit die Unterbrechung (und mögliche Ablenkung) durch das Umschalten zum Webbrowser.

Über den Autor

Mathias Weidner ist seit vielen Jahren Systemadministrator für Linux und Netzwerke in Wittenberg. Vorher arbeitete er in der Softwareentwicklung, zuletzt an Serversoftware für diverse Protokolle.

Nebenbei schreibt er Bücher und Perl-Module. Letztere sind am einfachsten auf CPAN *http://search.cpan.org/~mamawe/* zu finden.

Jürgen Peters

Statische Typsysteme in funktionalen Sprachen

Einleitung

Im laufe der letzten 20 Jahre haben funktionale Programmiersprachen sehr mächtige Typsyteme entwickelt, die das Ziel haben, den Programmierer bei der Erstellung von Code hoher Qualität zu unterstützen. Diese Entwicklung hat leider weitestgehend abseits vom Mainstream statt gefunden. Dadurch kennen die meisten Entwickler Sprachen wie Haskell oder OCaml nur vom Namen her und Sprachen wie Coq und Idris sind weitestgehend unbekannt.

Dabei sollten wir die aktuellen Entwicklungen aufmerksam verfolgen, da sich hier nicht nur akademisch interessante Theorien, sondern vor allem auch praktisch hilfreiche Werkzeuge finden, die sich so in prozeduralen oder Objekt orientierten Sprachen bisher kaum verbreitet haben.

Dem Anfänger sei noch als Rat mit auf den Weg gegeben, sich nicht von unbekannten, mathematischen Fachbegriffen einschüchtern zu lassen. Sicher nutzen die hier aufgelisteten Sprachen ein mathematisches Fundament, das sehr komplex ist. Die Nutzung dieser Konzepte und Sprachen ist aber nicht schwieriger zu erlernen als die bereits bekannten Sprachen. Insbesondere Haskell ist meiner Meinung nach aufgrund der guten Auswahl an Fachbüchern auch Programmieranfängern zu empfehlen. Der erfahrene Programmierer sollte sich allerdings darauf einstellen, an einigen Punkten umdenken zu müssen. Es sollte auf jeden Fall darauf geachtet werden zu vermeiden, in funktionalen Sprachen so zu entwickeln, wie man dies aus prozeduralen Sprachen gewohnt ist.

Die nachfolgende Auflistung ist eine Auswahl von Sprachen, die aus meiner Sicht besonders lohnenswerte Anschauungsobjekte darstellen.

Haskell

Haskell ist insbesondere aufgrund der Stabilität und großen Nutzerbasis interessant. Es existiert ein mit dem CPAN vergleichbares Repository an Programmbibliotheken und gute, stabile Compiler die performanten Code generieren.

Zum Einstieg kann man sich unter *http://www.haskell.org/platform/* die Haskell Platform herunterladen. Das ist gewissermaßen das rundum sorglos Paket mit allem was man zu Anfang braucht. Die beste Einsteigerlektüre stellt wohl das Buch *Learn You a Haskell for Great Good!* dar, welches unter *http://learnyouahaskell.com/* auch kostenlos gelesen werden kann. Die Haskell-Bücher aus dem O'Reilly Verlag sind auch empfehlenswert. Gute Bücher auf deutsch habe ich leider noch keine gefunden.

Coq

Coq ist keine Programmiersprache im eigentlichen Sinne, sondern ein s.g. Proof Assistant. Das bedeutet, dass Coq in der Hauptsache dazu entwickelt wurde, mathematische Beweise zu entwickeln und automatisch zu prüfen. Dies erlaubt eine sehr strenge Überprüfung des in Coq geschriebenen Codes, macht das Entwickeln in Coq aber auch um einiges Umständlicher als man es gewohnt ist.Coq kann selbst keine ausführbaren Programme erzeugen, erlaubt aber den Export von Code in andere Sprachen wie Haskell oder OCaml.

Weitere Informationen über Coq erhält man auf der offiziellen Webseite unter *http://coq.inria.fr/*. Eine gute Einführung zu Coq als Proof Assistant bekommt man in *Software Foundations*, das unter *http://www.cis.upenn.edu/~bcpierce/*

sf/ gelesen werden kann. Eine Einführung mit Schwerpunkt auf praktische Softwareentwicklung bietet *Certified Programming with Dependent Types* welches unter *http://adam.chlipala.net/cpdt/* gelesen werden kann.

Idris

Idris ist für mich die aktuell interessanteste unter den jungen Programmiersprachen. Es stellt den Versuch dar, Dependent Types wie man sie aus Coq oder Agda kennt in eine universelle Programmiersprache zu integrieren. Der Ansatz sieht sehr vielversprechend aus. Es sei aber darauf hingewiesen, dass sich Idris aktuell in der Entwicklung befindet und noch oft Änderungen erfährt, die die Kompatibilität mit altem Code brechen.

Der Stand der Dokumentation ist noch sehr lückenhaft. Solide Kenntnisse zumindest der Grundlagen von Haskell oder OCaml sind dringend angeraten.
Wobei Haskell hier mit Sicherheit die hilfreicheren Vorkenntnisse liefert, da Idris selbst in Haskell geschrieben ist.

Auf Grundlage dieser Voraussetzungen kann man sich am besten erst einmal das Tutorial durchlesen, welches sich wie alle weiteren zu Anfang interessanten Informationen auf der Seite *http://www.idris-lang.org* finden.

Über den Vortragenden

Ich schreibe seit über 20 Jahren Computerprogramme, seit 14 Jahren mache ich das auch professionell und in Perl. Nachdem ich von der Entwicklung, die Perl6 nahm etwas enttäuscht war, begann ich vor c.a. drei Jahren mal wieder andere Sprachen genauer unter die Lupe zu nehmen. Anfänglich etwas enttäuscht von dem Einheitsbrei der verbreiteteren Sprachen bin ich bei Haskell hängen geblieben und sehe die rein funktionalen Sprachen mittlerweile als vielversprechendste, aktuelle Entwicklung auf dem Gebiet der Programmiersprachen.

Wenn ich nicht Programmiere, spiele ich E-Gitarre oder engagiere mich im Vorstand des Hackerspace Bielefeld e.V.

Herbert Breunung

GCL in Beispielen

Abstract

WxPerl-Programme sind schnell und sehen sehr gut aus. Doch sie zu schreiben ist umständlicher als nötig. Einige Module, die ich gerade als Teil mehrerer, kleiner Projekte entwickle, sollen Linderung bringen.

Problemanalyse

1. Sinnlose Tipparbeit bei Widgeterzeugung
2. Fiese Fallen um die eigene App abzuschieÃŸen
3. Layouts werden sehr schnell unhandlich

Wx ist in C++ geschrieben, also Objekte soweit das Auge reicht, selbst wenn es unpraktisch ist.Außerdem gibt es wie in Perl positionale Parameter, blöd ist nur, dass der einzig Interessante an vierter oder fünfter Stelle kommt. Daher sind viele Angaben sinnfreie Defaultwertaufrufe.
Einige entscheidende Parameter wie Callbacks müssen in einem zweiten, dritten oder vierten Aufruf nachgereicht werden, obwohl sie sinnigerweise bei der Initialisierung angegeben werden.

Die Parameterüberprüfung sollte strenger sein, da manche akzeptierten Angaben die Ereignissteuerung auflaufen lassen. Auch muss man sich um unnötige Details kümmern, wie Eltern-Kind-Beziehungen der Widgets oder deren Sichtbarkeit, die sich eigentlich aus der Logik des Layouts ableiten lassen.

Auch beim Zusammenklöppeln einer GUI-Oberfläche aus den vorher erstellten Widgets, gibt es oft nur eine oder zwei effektive Informationen je gut ausgefüllter Codezeile.
Vor allem weil an einer guten Optik viel geschraubt wird bevor sie strahlt, wird das sehr schnell lästig.

Lösungsansatz

Da das Ziele recht groß ist und bereits andere scheiterten, zerlegte ich mir den Aufgabenberg in erträgliche Hügel, die einzeln im CPAN erscheinen.

Einzelne Widgets die mehr können gehören in den Namensraum für Pure-Perl-Module: als Wx::Perl::*. Eine Zuckerschicht über konventionelle Widgets ist Wx::Perl::Smart.
Zum Schluss wird GCL::Wx übergestülpt werden, um eine schöne MVC-Abstraktion zu haben und vielleicht sogar eine Tk übergreifende API, um aus Perl-Datenstrukturen (geladen aus JSON oder YAML oder programmgeneriert) zu einer Oberfläche zu kompilieren.

Dieser Vortrag wird zumeist von Wx::Perl::Smart handeln.

Einzelne Widgets

Den Anfang machten aber einzelne Widgets wie:

- Wx::Perl::RadioGroup
- Wx::Perl::DisplaySlider
- Wx::Perl::DawMap

Zum Beispiel fasst `Wx::Perl::DisplaySlider` einen Slider und ein Textfeld zusammen, welches den gewählten Wert Anzeigt. Da beides auf einem Wx::Panel per Sizer angebracht ist, kann es wie ein einzelnes Widget behandelt werden. Änderungen im TextCtrl oder des Sliders lösen den selben Callback aus wie ein Verschieben des Sliders (nach beenden der Verschiebung) und auch sonst ist die API wesent-

lich einfacher als normale WxPerl Widgets. `new` hat zwar 6 Parameter, aber darin ist nichts Überflüssiges und zudem alles enthalten, was man später brauchen könnte um das Widget zu definieren, wie Min-, Max- und Default-Werte und der Callback. Die Prüfungen der Parameter sind strenger und unlogische Angaben werden abgewiesen oder bereinigt. Die sonstige API ist sehr einfach (GetValue/SetValue und ähnliches) und folgt dem üblichen WxPerl Benennungs-Schema.

Auch sonst ist es ein normales Widget (Kind von Wx::Window), das keine Sonderbehandlung verlangt. Da es nur einen einen Abstand zwischen den Teilwidgets gibt, aber keinen rundherum, hat der Nutzer noch volle gestalterische Freiheit. Das Panel unter den Teilwidgets vermeidet zudem das optische Problem von durchscheinenden Untergrundflächen, weil Slider wie alle anderen Knöpfe halbtransparent sind.

Neben dem einfachereren Umgang kapselt das Widget auch etwas an Layout und Verhalten von ansonst zwei zusammenhängenden Widgets in einem. Das vereinfacht das Hauptprogramm, womit man sich auch Ziel 3 nähert.

Wx::Perl::Smart

Die drei Ziele ergeben sich aus der Umformulierung der anfänglich gelisteten Probleme:

1. knappe und vollständige Widgeterzeugung mit einem `new`
2. babysichere Widgets und Apps
3. Layout kann leicht überblickt und verändert werden

Wx::Perl::Smart::Util

Ist einfach die kleine Werkzeugkiste, die einen Umgang wie mit `Scalar::Util` fordert und einige der Umständlichkeiten von WxPerl ausgleicht. Sei es mit einem `is_widget`, welches mir prüft ob ein Parameter eine brauchbare Referenz enthält, informativeren Lognachrichten oder `wxcolor`, welches aus fast allen halbwegs sinnigen RGB-Angaben oder Farbnamen ein `Wx::Colour` erzeugt.

Wx::Perl::Smart::WidgetFactory

Meist benötigt man die üblichen kleinen Widgets wie Buttons, Textfelder u.s.w. welche hiermit erzeugt werden, nur halt kürzer mit intelligenten Parameterlisten, benannten Paramtern oder Strings welche die Widgets definieren. Der Name ist zawar recht lang, aber man sollte sich eine WidgetFactory-Objekt erzeugen lassen, und von dem aus die Methode `make_widget` aufrufen. Bei der Erzeugung des Objekts kann man auch einige schöne defaults angeben, wie auch einen default-parent was mehrere böse Fehlerquellen verhindert. Zwar muss man beim Bau des Layouts das wieder umdefinieren, aber das sollte man entweder eh machen, um flexiblen Code zu bekommen oder man benutzt die folgenden Module, welche das automatisch tun.

Wx::Perl::Smart::Sizer

Das Modul entstand aus dem Traum, einfach einen Array an Widgets zu übergeben, und ich bekomm einen Sizer, der all diese Widgets in der Reihenfolge enthält. Das spart eine Portion Schreibarbeit. Seit dem fand ich auch heraus, dass wenn ein Element des Arrays wiederum ein Array ist, es als Anordnung von Widgets quer zur ursprünglichen Richtung der Anordnung verstanden werden kann.
Das hat den den netten Effekt, dass solche Definitionen sich im Kopf viel leichter zu optischen Anordnung übersetzen lassen als WxPerl-Code, da sich diese Perl-Datenstrukturen im Code schön tabellarisch anordnen lassen.

Seit dem wuchs das Modul zu einem Universalen Layout-Mechanismus, da letzlich auch Notebooks (Reiterleisten) und ähnliches nur Mittel der Anordnung sind. Außerdem kann zusätzliche Abstände, Label oder Trennlinien mit sinnigen und einfachen Angaben einfügen um so schnell und tatsächlich leicht änderbar Layouts zu erzeugen.

Wx::Perl::Smart::Panel

Dieses Modul hat fast die identische API, denn es setzt auf dem Sizer auf und setzt es auf ein Panel, mit all den Vorteilen, welche schon bei `Wx::Perl::DisplaySlider` beschrieben wurden. Ausserdem sorgt das Modul dafür, dass die Kindwidget tatsächlich sichtbar sind und auch die korrekten Eltern haben. Dies sind meiner Erfahrung nach die zwei einfachsten, aber häufigsten Fehlerquellen.

Wx::Perl::Smart::Frame

.. setzt wiederum auf dem `Smart::Panel` auf, beinhaltet aber auch eine WidgetFactory dessen default-parent schon auf den `Frame` gesetzt ist. Darüber hinaus hilft es bei der Verwaltung von der Widgets und Label-Strings und nimmt einem die üblichen 3 Zeilen bei der Erstellung eines `Wx::Frame` (Hauptfenster eine App) ab. Diese verwalteten Wid-

gets können auch `Smart::Panel` sein, welche einem Teil des Layouts tragen und mit ihrem Namen im Gesamtlayout auftauchen. Das macht komplexe Oberflächen wesentlich umgänglicher.

GCL::Wx

Obwohl das noch reine Planung ist, wird wohl der Syntax dieser Markup-Sprache zur definition von GUI sich an den Fähigkeiten von `Wx::Perl::Smart::WidgetFactory` und `Wx::Perl::Smart::Sizer` orientieren. Oder anders gesagt: mit der Entwicklung und Feinjustierung der von diesen Modulen akzeptierten Syntax ist das Gros der Arbeit für einen `Wx`-Backend von GCL getan.

Autor

Herbert Breunung tauchte etwa 2004 in die Perlszene ein und bedroht seit dem jeden Entgegenkommenden seinen Editor namens Kephra zu benutzen. Dazu hat er oft Gelegenheit, denn er ist Moderator im größten deutschen Perlforum und Besucher aller deutscher Perl-Workshops seit 2005 und YAPC::EU's seit 2007 auf denen er manchmal auch Vorträge hält.

Neben Perl-Software schreibt er auch viel anderes Zeugs, nämlich Tutorien in diversen Wikis, als auch Artikel für das Perl-Magazin, die Perlzeitung, das freie Magazin und nicht zuletzt heise online.

Sebastian Willing

Log Everything... und dann?

Abstract

Immer wieder wird "Log everything" gepredigt: Möglichst viele Daten sollen protokolliert werden, damit sie bei Bedarf später ausgewertet werden können. Doch genau diese Auswertung ist es, die bei der Forderung nach "Log everything" häufig vernachlässigt wird.

Aus der Praxis

Ich arbeite derzeit in einem seit langer Zeit gewachsenem Projekt, bei dem noch vor zwei Jahren sehr unterschiedlich geloggt wurde. Die Vereinheitlichung der Debugging- und Fehlermeldungen führte wieder zum Problem der Auswertung. So ist ein System entstanden, dass nicht nur ausführlich loggen, sondern auch mit großen Volumina umgehen kann.

5 Bausteine

Unsere Lösung besteht aus 5 Bausteinen

Generierung

Wir verarbeiten Nachrichten aus vielen Quellen. Für absichtliche Debug-und Fehlermeldungen wurden zwei Funktionen geschaffen, die neben der eigentlichen Meldung viele nützliche Metadaten transportieren, z.B.:

- Gearman-Job-Daten bzw. HTTP-Daten
- Environment %ENV und @ARGV
- Stacktrace
- DBI-Fehlermeldungen und letztes SQL-Statement
- $! und $@
- Prozessdaten

Transport

Eine Nachricht von beliebigen Servern und auch aus gerade sterbenden Prozessen zu übermitteln, ist nicht so trivial, wie es sich anhört.
Um die teilweise sehr großen Nachrichten zu übermitteln, werden diese in je nach Transportweg unterschiedlich große Pakete zerlegt und nach dem Transport wieder zusammengesetzt. Für den Transport kommen Syslog, Gearman und HTTP zum Einsatz.

Datenbank

Die Anforderungen für Entwicklungs- und Livesystem weichen stark voneinander ab, also wurde auch die Speicherlösung modular aufgebaut. Für die Entwicklungssysteme wird SQLite eingesetzt, im Livesystem arbeitet eine MySQL-Datenbank.

Klassifizierung

Neben der Möglichkeit, manuelle Regeln für die Gruppierung von Nachrichten zu definieren, gibt es eine automatische Gruppierung. Dazu werden die Meldungen (ohne Metadaten) zunächst normalisiert. Diese Routine entfernt beispielsweise Zeitstempel, IP- und Email-Adressen, Referenzen (HASH0x123456), Leerzeichen und Zeilenumbrüche, um am Ende einen bei ähnlichen Meldungen identischen String zu generieren.

Erledigung

Die einzelnen Meldungsgruppen werden schließlich in einer Weboberfläche von einem Entwickler bewertet. Sie können ignoriert, mit anderen zusammengefasst und halbautomatisch in Tickets überführt werden. Laufen bei einem offenen Ticket neue Meldungen auf, so werden diese dem Ticket automatisch hinzugefügt. Größere Mengen neuer Meldungen werden zusammengefasst, um das Ticket nicht zu überfluten. Erst wenn das Ticket geschlossen wurde, taucht die Meldungsgruppe wieder in der zu-bearbeiten-Liste im Webinterface auf. Hierzu wurde eine Schnittstelle zu dem im Projekt ohnehin eingesetzten Ticketsystem entwickelt.

Über mich

Jahrgang 1979, erste Programmiererfahrungen ca. 1988 auf DATAPOINT 6600. Über Databus, DB/C und verschiedene Basic-Dialekte kam ich 1999 zu Perl und bin dort bis heute geblieben. Neben der Arbeit bin ich Familienvater, Blogger, Buchautor und seit einigen Monaten begeisterter Ingress-Spieler.

Mein Blog beschäftigt sich mit allen Themenbereichen meines Lebens:

http://www.pal-blog.de

Kerstin Puschke

Entwicklungsprozess und Architektur einer gewachsenen Webanwendung

Abstract

Skalieren muss nicht nur die Anwendung und ihre Architektur - auch der Entwicklungsprozess verändert sich mit der Größe des Entwicklungsteams und der Komplexität der entwickelten Anwendung.
Der Vortrag gibt einen Einblick in die Architektur hinter XING, dem sozialen Netzwerk für berufliche Kontakte mit 14 Millionen Mitgliedern.

Wie beeinflussen sich Architektur und Entwicklungsprozess, und wie sieht die Zusammenarbeit aus in einem mehr als 80-köpfigen Engineeringteam, das auf mehrere Standorte in Europa verteilt ist und sich einer heterogenen, über mehr als zehn Jahre gewachsenen Architektur gegenübersieht?

Neben der Architektur der Plattform werden auch Entwicklungstools und -umgebungen vorgestellt, die den Entwicklungsprozess von der Idee bis zum Release begleiten.

Architektur der Plattform

Herzstück der Plattform xing.com ist eine über die letzten zehn Jahre gewachsene Perlanwendung mit ca. 600K Zeilen und bis zu 30M externen Zugriffen pro Tag. Als Teil einer verteilten Architektur kommuniziert diese synchron und asynchron mit über 15 Ruby on Rails Anwendungen und weiteren Komponenten.

Entwicklung

Von mehreren europäischen Standorten aus arbeitet ein etwa 80-köpfiges Engineeringteam an Entwicklung und Betrieb der Plattform. Zur Entwicklungsinfrastruktur gehören automatisiert erstellte und konfigurierte virtualisierte Entwicklungsumgebungen, verteilte Versionskontrolle und eine Continous Integration Infrastruktur. Das Spektrum an Releasezyklen reicht von wöchentlich bis zu Continuous Deployment.

Bio Kerstin Puschke

Kerstin Puschke ist Softwareentwicklerin in Hamburg. Gemeinsam mit einem großartigen Team arbeitet sie hauptsächlich am Backend von XING, dem sozialen Netzwerk für berufliche Kontakte mit 14 Millionen Mitgliedern.

Julian Knocke

HTTP effizienter nutzen mit WWW::Curl::UserAgent

Abstract

Webseiten und REST APIs werden über das Protokoll HTTP abgewickelt. Mit Parallelität und der Wiederverwendung bestehender TCP-Verbindungen (KeepAlive) können große Mengen an Anfragen schneller abgearbeitet werden. Durch eine Callback basierte Schnittstelle wird der Programmcode bei gleichzeitigen Anfragen optimal durchlaufen und so Zeitverluste durch IO minimiert. Mit Vergleichen zu bestehenden User Agents wie `LWP::UserAgent`, `LWP::Parallel::UserAgent`, `Mojo::UserAgent` und `WWW::Curl::Simple` können Vor- und Nachteile einer Neuimplementation auf Basis von cURL beschrieben werden.

APIs als Bottleneck

Die Trennung der Datenhaltung und Kapselung domänenspezifischer Logik führt dazu, dass ein Großteil der Zeit, die für die Bearbeitung eines Web-Requests notwendig ist, damit verbracht wird, auf APIs zu warten. Bei XING kann man diesen Effekt durch eine Auswertung der spezifischen Anteile der Antwortzeiten leicht nachvollziehen. Dort kommt als interne API eine HTTP basierte REST Schnittstelle zu Einsatz. Um die Auslieferungsgeschwindigkeit zu steigern, sollte also insbesondere die Zeit optimiert werden, in der REST API antwortet.
Das kann zum einen durch eine Optimierung von Verbindungen zu den API Servern geschehen und zum anderen durch eine effiziente parallele Ausführung.

Bestehende UserAgents

Bei XING wurden bisher die UserAgents `LWP::UserAgent` für sequentielle und `WWW::Curl::Simple` für parallele Anfragen verwendet. Als wir Untersuchungen bezüglich der Handhabung von TCP-Verbindungen durchführten, stellten wir fest, dass der KeepAlive Status in `LWP::UserAgent` als experimentell beschrieben ist und `WWW::Curl::Simple` gar kein KeepAlive ermöglicht. Mit KeepAlive ist es seit HTTP 1.1 möglich, die TCP-Verbindung für weitere Anfragen offen zu lassen und somit den Overhead durch den TCP-Verbindungsaufbau zu sparen. Auch mit `LWP::Parallel::UserAgent` war KeepAlive nicht möglich. Einzig `Mojo::UserAgent` konnte parallele Verbindungen mit KeepAlive handhaben, ist aber aufgrund der großen Abhängigkeiten nicht näher in Betracht gezogen worden.

Wenn man sich die Schnittstellen der verschiedenen UserAgents anschaut, welche auch parallele Ausführung ermöglichen, sind diese sehr inhomogen:

```
# LWP::Parallel::UserAgent
$ua->register( HTTP::Request->new(
    GET => "http://example.org/"
) );
$ua->wait;
```

```
# WWW::Curl::Simple
$ua->add_request( HTTP::Request->new(
    GET => "http://example.org/"
) );
$ua->perform;
```

```
# Mojo::UserAgent
$ua->get( "http://example.org/" => sub {} );
Mojo::IOLoop->delay->wait;
```

Mit der Erfahrung, die wir in Ruby mit `Typhoeus` gemacht haben, lässt sich jedoch auf Basis von cURL eine effiziente Umsetzung erreichen. Das Modul `WWW::Curl::Simple` hätte jedoch eines strukturellen Umbaus bedurft, so dass dieser in einem neuen Modul `WWW::Curl::UserAgent` umgesetzt wurde.

WWW::Curl::UserAgent

Mit den Methoden `request` und `add_request` nimmt `WWW::Curl::UserAgent` eine sequentiell bzw. parallel auszuführende Anfrage entgegen. Wird ein Request der Queue hinzugefügt, müssen ebenso Handler übergeben werden, welche im Erfolgs- oder Fehlerfall aufgerufen werden:

```
my $request = HTTP::Request->new(
    GET => 'http://search.cpan.org/'
);
```

```
# Dieser Request wird direkt ausgeführt:
$response = $ua->request($request);

# Folgender Request kann nebenläufig
# ausgeführt werden:
$ua->add_request(
    request    => $request,
    on_success => sub {
        my ( $request, $response ) = @_;
        print $response->content;
    },
    on_failure => sub {
        my ( $request, $error_msg,
            $error_desc ) = @_;
        die "$error_msg: $error_desc";
    },
);
$ua->perform;
```

Ein Benchmark mit 500 sequentiellen Requests unter Fedora 19 gegen eine interne REST API mit durchschnittlich 5ms Antwortzeit ergab die Ergebnisse wie in Listing 1 zu sehen.

```
+-------------------------------+-----------+------+------+------------+------------+
|          User Agent           | Wallclock |  CPU |  CPU |  Requests  | Iterations |
|                               |  seconds  |  usr |  sys | per second | per second |
+-------------------------------+-----------+------+------+------------+------------+
| LWP::UserAgent 6.05           |    18     | 1.08 | 0.29 |    27.8    |    365.0   |
+-------------------------------+-----------+------+------+------------+------------+
| LWP::Parallel::UserAgent 2.61 |    19     | 1.11 | 0.30 |    26.3    |    354.6   |
+-------------------------------+-----------+------+------+------------+------------+
| WWW::Curl::Simple 0.100191    |    94     | 0.66 | 0.34 |     5.3    |    500.0   |
+-------------------------------+-----------+------+------+------------+------------+
| Mojo::UserAgent 4.83          |    10     | 1.31 | 0.08 |    50.0    |    359.7   |
+-------------------------------+-----------+------+------+------------+------------+
| WWW::Curl::UserAgent 0.9.6    |    10     | 0.61 | 0.05 |    50.0    |    757.6   |
+-------------------------------+-----------+------+------+------------+------------+
```

Listing 1

Werden dabei jeweils 5 Requests parallel ausgeführt, also ebenso 500 Anfragen, jedoch nur 100 Messungen, ergibt sich das wie in Listing 2 zu sehen.

Mit `LWP::Parallel::UserAgent` und `WWW::Curl::UserAgent` konnten durch die parallele Ausführung Geschwindigkeitsgewinne festgestellt werden.
`WWW::Curl::Simple` und `Mojo::UserAgent` scheinen deutlich länger zu blockieren als notwendig, so dass die parallele Ausführung sogar langsamer ist, als die Sequentielle.

Dieser Benchmark zeigt somit die Wirksamkeit der Optimierungen und kann auch selbst mit *https://github.com/xing/curl-useragent/blob/master/tools/benchmark.pl* nachvollzogen werden.

```
+--------------------------+-----------+--------+--------+------------+------------+
|          User Agent      | Wallclock |   CPU  |   CPU  |  Requests  | Iterations |
|                          |  seconds  |   usr  |   sys  | per second | per second |
+--------------------------+-----------+--------+--------+------------+------------+
| LWP::Parallel::UserAgent |      9    |   1.24 |   0.28 |     55.6   |     65.8   |
+--------------------------+-----------+--------+--------+------------+------------+
| WWW::Curl::Simple        |    860    | 256.47 | 217.15 |      0.6   |      0.2   |
+--------------------------+-----------+--------+--------+------------+------------+
| Mojo::UserAgent          |    301    |   1.69 |   0.31 |      1.7   |     50.0   |
+--------------------------+-----------+--------+--------+------------+------------+
| WWW::Curl::UserAgent     |      3    |   0.47 |   0.06 |    166.7   |    188.7   |
+--------------------------+-----------+--------+--------+------------+------------+
```

Listing 2

Tobias Leich

Modulversionierung in Perl6

Abstract

Ein Traum jedes Paketierers!

Perl 6 bricht die Tradition dass man nur eine Distribution eines Modules effektiv installieren und nutzen kann.
Ich werde zeigen wie Module installiert werden, und wie man aus einem Programm heraus verschiedene Versionen desselben Moduls nutzt. Sei es, weil ein Feature aus einer alten Version besser funktioniert, oder weil man in einem anderen Programmteil einen Fork des Modules nutzen möchte.

Über den Autor

Tobias bewegt sich mit kurzer Unterbrechnung seit nunmehr 16 Jahren im Perl Umfeld, doch erst seit ungefähr sechs Jahren ist er auf der Seite der Modulautoren.
Und dies geschah wie vielerorts aus der Not heraus: Das Modul was benutzt werden wollte war in schlechtem Zustand. Auf diese Weise widmete sich Tobias mehrere Jahre lang dem SDL-Modul, um eine nutzbare Bibliothek für Spiele und dergleichen bereitzustellen und selbst zu nutzen.

Durch die schmerzhaften Erfahrungen die sich mit C-Bibliotheken, XS und Threads ergaben, erkannte er einen Lichtblick am Horizont genannt Perl 6.Und genau diesem Lichtblick widmet er den größten Teil seiner Freizeit.

v3.0 v2.0 v1.0 - Release!

Kennt ihr das auch? Ihr aktualisiert Module über CPAN oder apt aber aus irgendeinem unerfindlichen Grund wird trotzdem die alte Modulversion benutzt anstatt der neuen? DWIM ist das nicht, aber was will man denn überhaupt? Es gibt meines Erachtens zwei Ansätze ein Modul zu laden: Erstens, man will das neueste was auf dem System ist, oder zweitens, man will eine ganz bestimmte Version, möglicherweise die eines bestimmten Autors.

```
use Foo:ver<v1.2>;
use Bar:ver(v2.0..*);
use Baz:ver(Any):auth<github:FROGGS>;
```

Okay, soweit so gut. Aber nicht jede Distribution stellt nur Module bereit. Was ist, wenn ein ausführbares Skript bereitgestellt wird? Das erste was im PATH gefunden wird wird geladen, wie kann ich also ein Skript einer bestimmten Version ausführen?

Dies und weitere Details werde ich in einem kurzen Vortrag vorstellen.

Links

Repository - *https://github.com/rakudo/rakudo/tree/eleven*
Spezifikation - *http://perlcabal.org/syn/S11.html*
Blog - *http://usev5.wordpress.com/*
Rakudo Perl 6 - *http://rakudo.org/*

Herbert Breunung

Perl6 in Kontext

Abstract

Noch mehr als Perl 5, ist Perl 6 eine kontextabhängige Sprache. Und da der Kontext oft von Operatoren geprägt wird, geht es im vierten Teil vor allem um Perl 6 - Operatoren.

Was ist eigentlich Perl 6

Perl 6 ist DIE große Aufräum- und Verbesserungsaktion für Perl 5. Das bedeutet in jeder kleinen Ritze herrscht Ordnung und strenge Regeln, auch wenn nach außen vieles noch locker perlig daherkommt.

`say`, Smartmatch und Moose mit Rollen sind vielleicht bekannt, aber Perl 6 umfasst noch wesentlich, wesentlich mehr.

OOP Innereien

Mittel der Ordnung ist die allumfassende Objekthierarchie. Auch die hier vorgestellten Ops sind nur Methoden der Typenklassen, und die Operaden nur die Parameter.

Sigil Kontext Ops

Wenn man Kontext hört, denkt man vielleicht an so etwas wie `wantarray`, oder vielleicht den mysteriösen goatse-Op, die es beide in P6 nicht gibt. Wesentlich simpler lassen sich die vertrauten Kontexte erzwingen, weil die bekannten Sigils jetzt auch als Operatoren funktionieren, aber auch eine lange, ausführliche Schreibweise haben.

```
$()     @()     %()     &()
item()  list()  hash()  code()
```

Natürlich gibt es noch einen flat() Kontext, der sich verhält wie Perl 5, wo alle Listen gleich aufgerollt werden.

Type Kontext

Vielleicht noch größere praktische Bedeutung haben Bool, Numeric- und String-Kontext. Es ist genau das wonach es klingt und sie lauten:

```
?               +               ~
```

wobei ! und - die Verneinung der ersten beiden sind. ? werten also einen Ausdruck zu einem `Bool::True` oder `Bool::False` aus, wie es etwa nach einem `if` geschieht. Sie sind aber noch zu weit mehr einsetzbar.

```
+@array == @array.elems
```

Da etwa das Shiften von Werten im numerischen Kontext passiert heißen sie jetzt

```
+<      +>
```

Überhaupt gibt es jetzt eine Reihe an Ops deren erste Hälfte anzeigt in welchen Kontext beide Operanden Konvertiert werden und der die zweite Hälfte sagt was dann getan wird.

```
?&      ?|      ?^
+&      +|      +^
~&      ~|      ~^
```

Metaops

Das ähnelt den Metaoperatoren, die ebenfalls eine Anweisung sind, wie der folgende oder enthaltene Operator anzuwenden ist. So neu ist das nicht, da jeder halbwegs sattelfeste Perlschreiber weiß, dass ein Op mit = davor selbstzuweisend und mit `!` das Gegenteil bedeutet. (Wie beim forcieren des Bool Kontext)

```
  !      =      []      [\]      <<      >>
  R      S      X       Z

reduce() triangle()crosswith()zipwith()
```

Die neuen Metaops sind letztlich Kurzschreibweisen für Dinge die man in der funktionalen Programmierwelt schon länger kennt und die man in Perl 6 auch ausschreiben kann (TIMTOWTDI). Kreuzprodukt (X) und Reißverschluss(Z) gibt es auch als gleichnamige, normale Operatoren, wo sie mit der gleichen Reihenfolge der Abarbeitung lediglich Listen generieren.

Junctions

Bei den Junctions werden auch um mehrere Werte in einer Variable gespeichert, allerdings mit ihrer logischen Verknüpfung. Hat man schon eine Liste und will die nur noch verknüpfen, nimmt man einen der unteren Ops.

```
  |      &      ^
any()  all()  one()
```

Junctions sind ein Werkzeug um die Ausdrücke nach einem `if` kompakt zu halten:

```
if $result = 4 | 2 | 33 { ...
if $result = any( @l )
# entspricht  $result ~~ @l
```

Aber so richtig fortgeschritten ist es, wenn ich in eine geschickt benannte Variable die junktive Verknüpfung mehrer anderer Variablen stecke und während ein Algorithmus läuft der Daten kaut immer mal wieder Frage, ob die Verknüpfung (vielleicht noch in Verbindung mit ein paar anderen Regeln) jetzt das gewollte Ergebnis liefert. So etwas liest sich mit Junctions fast wie ein Gedicht.

Werterhaltende Ops

Wer mag sie nicht, die guten alten Kurzschlussoperatoren. Man erkennt sie am doppelten Symbol und man weiß, dass sie den Kontext des Operanden nicht verändern, sondern einen ausgewählten Wert unverfälscht zurückgeben.

```
  &&     ||     //
```

Kein Wunder das der tertiäre Op jetzt so ähnlich geschrieben wird:

```
  ??   !!
```

ENDE ???

Natürlich werden noch weit mehr Operatoren im Vortrag behandelt, aber sie lassen sich auch nachlesen auf *http://tablets.perl6.org/appendix-a-index.html*.

Autor

Herbert Breunung tauchte etwa 2004 in die Perlszene ein und bedroht seit dem jeden Entgegenkommenden seinen Editor namens Kephra zu benutzen. Dazu hat er oft Gelegenheit, denn er ist Moderator im größten deutschen Perlforum und Besucher aller deutscher Perl-Workshops seit 2005 und YAPC::EU's seit 2007 auf denen er manchmal auch Vorträge hält.

Neben Perl-Software schreibt er auch viel anderes Zeugs, nämlich Tutorien in diversen Wikis, als auch Artikel für das Perl-Magazin, die Perlzeitung, das freie Magazin und nicht zuletzt heise online.

Ralf Peine

Porf Praxis

Einleitung

Anfang letzten Jahres stand ich vor der Aufgabe, eine Liste mit Hashes als Tabelle in verschiedenen Formaten auszugeben. Dazu musste es doch Perl-Module auf dem CPAN geben. Gibt es auch, mehr als 5000 Treffer für "Report". Aber diese Reports sind spezialisiert für bestimmte Aufgaben und stellen meistens nur ein Ausgabeformat zur Verfügung.

Für "Report & Framework" gibt es genau einen Treffer: C<Data::Report>, letzte Änderung am 17.08.2008, Version 0.10. Eine kurzer Blick in die Doku des Frameworks zeigt, dass es Klassen für die Plugins verwendet, die die Formate definieren. Der dort gewählte Ansatz kann zu sehr langsamer SW führen, Informationen über die Performance gibt es nicht.

Außerdem war mir die Anwendung der Frameworks/Reports, die ich mir angesehen habe, zu kompliziert und nicht Perl-like.

Deshalb startete ich mit der Entwicklung eines eigenen, allgemeinen, offenen Report-Frameworks: Perl Open Report Framework, kurz PORF, ist der Arbeitsname. (ORF ist schon durch irgendeinen östereichischen Fernsehsender belegt.)

```
use Report::Porf::Framework;  # "use" zaehle ich nicht als Statement

# --- Report erzeugen lassen ---
my $report_framework = Report::Porf::Framework::Get();             # 1.
my $report           = $report_framework->CreateReport($format);   # 2.

# --- Spalten konfigurieren, die Daten liegen als Hash vor ---

# --- n = 3 für 3 Spalten ---
$report->ConfigureColumn(-header => 'Vorname',  -value_named => 'Prename' ); # lang
$report->ConfCol        (-h      => 'Nachname', -val_nam     => 'Surname' ); # kurz
$report->CC             (-h      => 'Alter',    -vn          => 'Age'     ); # minimal

# --- Konfiguration abschlieÃŸen, das ist notwendig ---
$report->ConfigureComplete();                                      # 3.

# --- Daten ausgeben ---
$report->WriteAll($person_rows, $out_file_name);                   # 4.

# --- Fertig ! ---
```

Aktuell liegt die Version 0.950 für euch zum Download auf meiner Homepage bereit.

Falls jemand ein Report-Framework mit ähnlichen Eigenschaften kennt, informiert mich bitte. Dann kann ich mir die Weiterentwicklung von PORF eventuell sparen.

Einen Report in 4+n Statements konfigurieren und ausgeben

Einen Report mit 4+n Statements zu konfigurieren und die Ergebnisse in eine Datei zu schreiben, ist das Ziel von PORF und mit der aktuellen Version bereits performant umgesetzt (siehe Listing 1).

Erste Tests mit Kollegen ergaben, dass sie die API von PORF genauso leicht verständlich finden wie ich. Im einzelnen:

0. Use

Es existieren bislang keine weiteren Abhängigkeiten zu anderen Perl-Modulen, bis auf einige absolute Basis-Module wie z.B. `FileHandle` und `Exporter`. Auch das Report-Modul selbst benötigt kein use, weil kein `Report->new()` durch den Anwender aufgerufen werden muss (darf).

1. Framework erzeugen

Mit Anweisung 1 holt man sich ein vorkonfiguriertes Report-Framework ab. Es ist sehr wichtig, sich nicht selbst eines mit "new" zu erzeugen, denn die Konfiguration des Frameworks und der Reports erfordert einige Arbeit in der "Framework-Factory".

2. Report erzeugen

Das Framework kann jetzt unverändert (Out Of The Box) verwendet werden, um einen Report im Format `$format` zu erstellen. Als Formate sind aktuell "Text", "HTML" und "CSV" untersützt, weitere sind geplant. Vielleicht schaffen wir es ja im Rahmen des Perl-Workshops, die Report-Konfiguratoren für Wikis und LaTeX zu erstellen.
Zur Konfiguration des Reports werden die Klassen `HtmlReportConfigurator`, `TextReportConfigurator` und `CsvReportConfigurator` aus `Report::Porf::Table::Simple` verwendet, die den Report gesteuert durch das Framework unsichtbar im Hintergrund vorkonfigurieren. Man kann beliebige eigene Report-Konfiguratoren - auch für neuen Formate - erstellen und in einer eigenen Frameworkinstanz bereitstellen, ohne das Framework selbst irgendwie ändern zu müssen.

`CreateReport($format)` liefert eine Instanz der Klasse `Report::Porf::Table::Simple` zurück, die einen einfachen tabellarischen Report im geforderten Ausgabeformat erzeugen kann. Es handelt sich hier immer um dieselbe Klasse, unabhängig vom gewünschten Format. Das vereinfacht die Anwendung sehr, denn so fängt man sich keine (unerwünschten) Abhängigkeiten zu weiteren Klassen/Modulen ein.

1..n Spalten des Reports konfigurieren

Je nach Geschmack und Erfahrung kann man zwischen kurzen und langen Bezeichnern wählen. Hier ist die minimale Konfiguration angegeben: Überschrift und Zugriff auf den Wert einer Zelle (Cell). Ein Datensatz für eine Datenzeile (Row) liegt hier als Hash vor. Arrays, Klassen und freie Zugriffe werden auch unterstützt.

3. Konfiguration abschließen

Der Abschluss der Konfiguration mit `$report->ConfigureComplete();` ist notwendig, da es viele Möglichkeiten gibt, die anschließende Ausgabe durchzuführen. Außerdem kann man die Konfiguration ab jetzt nicht mehr verändern. Der Versuch, noch einmal `$report->ConfigureColumn(...);` aufzurufen, endet in einer Warnung.

4. Daten in Datei schreiben

Wählt man das Text-Format und schreibt man die Daten in eine Datei, erhält man das Ergebnis, das in Listing 1 gezeigt wird.

Man kann die Datenausgabe auch zeilenweise oder sogar zellweise durchführen lassen oder sich nur die Ergebnisse als String abholen, andernfalls hätte das Framework den Titel "Offen" nicht verdient.

```
*============+============+============*  # Fette Trennlinie
|   Vorname  |  Nachname  |    Alter   |  # Überschriften
*------------+------------+------------*  # Normale Trennlinie
| Vorname 1  | <Zelle>    | 7.69230769 |  # Eine Datenzeile mit Zellen
| Vorname 2  | Name 2     | 15.3846153 |
| Vorname 3  | Name 3     | 23.0769230 |
| Vorname 4  | Name 4     | 30.7692307 |
*============+============+============*
```

Eine Zeile als String liefert:

```
my $line = $report->GetRowOutput($data_ref);
```

Aber Achtung, falls man Trennzeilen definiert hat, kann $line auch mehrere Zeilen getrennt mit '\n' enthalten.

Konfigurationsmöglichkeiten

Wechsel des Ausgabeformats

Um die Tabelle als HTML auszugeben, belegt man `$format` einfach mit "HTML" statt "Text", und führt den Code nocheinmal durch.

Es bietet sich an, alle Zeilen bis auf Anweisung 4., Daten schreiben, in eine eigene sub zu packen, ein Beispiel ist in Listing 2 zu sehen.

Dann kann man die Daten als HTML oder Text ausgeben mit

```
my $report = CreateAgeReport($format);
$report->WriteAll(
    $person_rows,
    $out_file_name
);
```

und einen zweiten Report mit

```
$report->WriteAll(
    $person_rows_2,
    $out_file_name_2
);
```

Konfiguration der Spalten

Bis hierher hätte man das alles noch irgendwie mit `join(...)` erledigen können. Aber einige der folgenden Konfigurationsmöglichkeiten für die Spalten sind damit nicht mehr leicht implementierbar.

```
sub CreateAgeReport {
    my $format = shift;

    my $report_framework = Report::Porf::Framework::Get();              # 1.
    my $report           = $report_framework->CreateReport($format);    # 2.

    $report->ConfigureColumn(-header => 'Vorname',  -value_named => 'Prename' ); # lang
    $report->ConfCol        (-h      => 'Nachname', -val_nam     => 'Surname' ); # kurz
    $report->CC             (-h      => 'Alter',    -vn          => 'Age'     ); # minimal

    $report->ConfigureComplete();                                       # 3.

    return $report;
}
```

Layout-Optionen

```
-header  -h   constant: Text
-align   -a   constant: (left|center|right)
                        (l   |   c   |    r)
-width   -w   constant: integer
# Der Text wird gekürzt/verlängert
-format  -f   constant: string für sprintf
-color   -c   constant / sub {...}
```

Die `sub {...}` ermöglicht eine bedingte Einfärbung durch die Daten auf eine einfache Art. Hier zeigt sich einmal mehr die "Magie des EVAL {}" (siehe 2. Vortrag). Wie das genau funktioniert, erläutert das zweite Beispiel.

Nicht alle Optionen sind in jedem Format verfügbar. Unbekannte Optionen werden einfach ignoriert.

Datenzugriff

Daten liegen in Perl typischerweise als Array, Hash oder Instanz einer Klasse vor. Oder auch irgendwie anders. Für jeden Datentyp gibt es eigene, komfortable Zugriffsmethoden durch den Report. Es ist sogar möglich, sie miteinander zu kombinieren (falls die Daten das hergeben...)

GetValue Alternative 1 --- ARRAY

```
my $prename = 1;
my $surname = 2;
my $age     = 3;

$report->ConfigureColumn(
  -header        => 'Vorname',
  -value_indexed => $prename
); # long
$report->ConfCol        (
  -h      => 'Nachname',
 -val_idx       => $surname ); # short
$report->CC(
  -h      => 'Alter',
  -vi     => $age
); # minimal
```

GetValue Alternative 2 --- HASH

```
$report->ConfigureColumn(
    -header      => 'Vorname',
    -value_named => 'Prename'
); # long
$report->ConfCol(
    -h       => 'Nachname',
    -val_nam => 'Surname',
); # short
$report->CC(
    -h  => 'Alter',
    -vn => 'Age'
); # minimal
```

GetValue Alternative 3 --- OBJECT

```
$report->ConfigureColumn(
    -header       => 'Vorname',
    -value_object => 'GetPrename()'
); # long
$report->ConfCol(
     -h       => 'Nachname',
     -val_obj => 'GetSurname()'
); # short
$report->CC(
    -h  => 'Alter',
    -vo => 'GetAge()'
); # minimal
```

GetValue Alternative 4 --- Free

```
$report->ConfigureColumn(
    -h     => 'Vorname',
    -value => '"Dr. " . $_[0]->{Prename}'
);
$report->ConfCol(
    -h => 'Nachname',
    -val => sub {
        return $_[0]->{Surname};
    };
);
$report->CC(
    -h => 'Alter (Monate)',
    -v => '(12.0 * $_[0]->GetAge())'
```

```
);
```

Der Report erzeugt sich in jedem Fall eine anonyme sub, ähnlich wie bei 'Nachname' und 'Alter' in Alternative 4. In der aktuellen Implementierung ist das ein mehrstufiger Prozess. Man sollte sehr darauf achten, hier keinen Syntaxfehler in den "value"-Optionen einzubauen. Die Fehlersuche kann sich sehr schwierig gestalten. Das ist ein Nachteil des Frameworks, der sich nur vermeiden lässt, wenn man die Performance und den Komfort drastisch reduziert.

Zur leichteren Problemanalyse kann man sich Trace-Informationen ausgeben lassen mit:

```
$report->SetVerbose(3)
```

liefert für die Spalte 'Alter' z.B.

```
#------------------------------
-h = Alter
-vn = Age
### eval_str = sub { return $_[0]->{Age}; }
### ref(sub_ref) CODE
width 10
 ### eval_str = sub { return  ConstLength-
Left(10, $value_action->($_[0])) |; }
### ref(cell_action) CODE
### AddCellOutputAction: CODE(0x285251c)
```

Weitere Beispiele

Bedingte Einfärbung

Für alle Personen aus der Liste, die noch nicht mindestens 18 Jahre alt sind, soll die Alter-Zelle rot eingefärbt werden. Dazu benötigt man eine sub {} für die Konfiguration der Zellfarbe:

```
$report->ConfigureColumn(
   -header  => 'Age',
   -width   => 15,
   -align   => 'Right',
   -format  =>  "%.3f years",
   -color   =>  sub {
        return $_[0]->{Age} >= 18 ?
               "":
               '#EECCCC';
   },
   -vn      => 'Age',
);
```

Außerdem wird mit "-format" noch die Anzahl der Nachkommastellen auf 3 beschränkt. Man beachte, dass man in der Sub Zugriff auf die/den gesamten Datensatz/-zeile hat, sodass auch Bedingungen mit mehreren Parametern als Input möglich sind.

Spezialanzeige für Werte

In diesem Beispiel wird die Augenanzahl eines Würfels im HTML-Report auch in einer zusätzlichen Spalte als Grafik angezeigt:

Count	Throws	Dices
1	1	
2	6	
3	10	
4	11	
5	12	
6	13	
7	0123456789ab	??

Abbildung 1: Tabelle mit Würfeln

Dazu benötigt man zunächst 10 Bilder, die 0 bis 9 Augen auf einer Würfelseite anzeigen. Je nach Wert des Wurfs wird dann das entsprechende Bild in der HTML-Tabelle angezeigt.

Dazu wird eine spezielle -value Option verwendet:

```
sub {
   return $dices_to_image->(
       $_[0]->{'Dices'}
   );
}
```

Insgesamt erhält man:

```
$report->ConfigureColumn
    (-header => 'Dices', -a => 'C',
```

```
      -value => sub {
          return $dices_to_image->(
              $_[0]->{'Dices'}
          );
        }
      ) if $report->IsFormat('HTML');
```

Für alle, die noch nicht so häufig mit dem `sub {}` Befehl gearbeitet haben: `$dices_to_image` ist eine Referenz auf die aufzurufende Funktion, und die Variable wird automatisch vom erzeugenden Code in die anonyme `sub{}` importiert, inklusive aller anderen bekannten Variablen und Funktionen. Eine solche 'Sub' wird dann als 'Closure' bezeichnet.

`$_[0]` ist die erste Variable aus der Argumentenliste, die für die -value option immer mit dem auszugebenden Datensatz belegt ist, hier also mit der Wurfnummer und den erzielt(en) Wert(en) des Würfel-Wurfs:
`$_[0]->{'Dices'}` enthält eine Abfolge gewürfelter Werte als String.

Das nachgestellte if sorgt dafür, dass diese zusätzliche Spalte nur erzeugt wird, falls es sich um einen HTML-Report handelt.

`$dices_to_image` wird folgendermaßen definiert:

```
my $dices_to_image = sub {
  my $throws = shift; # Dice throw values

  my $result = '';
  my $number_html_code;

  foreach my $number (split (//, $throws)) {
    if ($number =~ /^\d$/) {
      $number_html_code =
           "<img src='dice_$number.jpg'/>"
    }
    else {
      $number_html_code = "?";
    }
    $result .= $number_html_code;
  }

  return $result;
};
```

Bis auf die erste Zeile eine Übungsaufgabe für den Perl-Fortgeschrittenen-Kurs. Also eine kleine Fingerübung für erfahrene Perl-Programmierer. Zurück liefert diese Sub eine Abfolge von <img ...> HTML-Befehlen, die im Browser die Bilder mit den Würfelaugen in der gewünschten Reihenfolge ausgibt.

Ein direkter Aufruf einer "normalen" Sub wäre auch möglich, aber durch die Verwendung einer anonymen Sub geht man allen Problemen bei der Namensauflösung aus dem Weg, die andernfalls entstehen könnten.

Gleichzeitig Verwendung mehrerer Frameworks

Im etwas größeren Beispiel "Fussball" im gleichnamigen Ordner werden mehrere Frameworks gleichzeitig erzeugt und durch die Aufrufoptionen wird - eventuell - eines ausgewählt. Dadurch ist es möglich, das Layout (z.B. Farben) sowie begleitende Texte zu verÃ€ndern, ohne bei der Programmierung der Listen auch nur an Varianten zu denken.

Beispiele starten

Wie man die Beispiele startet, steht in demo.txt, das wie die Beispiele im Ordner Practice abgelegt ist.

Auf die Erstellung von Start-Scripts habe ich bewusst verzichtet, denn bis auf ein zusätzliches

```
-I../lib
```

wird einfach nur Perl aufgerufen.

Vortrag

Ich werde verschiedene kleine und große Beispiele zeigen und die verschiedenen Anwendungsmöglichkeiten von PORF erläutern, vom Einsatz in kleinen Scripts bis zu großen Applikationen.

Eigenschaften

Im folgenden möchte ich noch die wesentlichen Eigen-

schaften erläutern, die ich so bei anderen Report-Modulen nicht gefunden habe:

Offen

Es ist leicht möglich, eigene Varianten von Report-Konfiguratoren zu erstellen oder neue Formate zu unterstützen. Auch die gleichzeitige Verwendung von verschiedenen Konfiguratoren für dasselbe Format ist durch die Möglichkeit, mehrere Framework-Instanzen zu verwenden, sehr einfach zu realisieren.

Unabhängigkeit von anderen Modulen

Bewusst verwendet (Standard)-PORF z.B. keine HTML-Funktionen aus CPAN-Modulen, damit das System wirklich offen bleibt. Der Anwender kann selbst entscheiden, welche Module er in eigenen/adaptierten Report-Konfiguratoren verwendet.

Performance

Auf meinem alten Win-XP Laptop mit 1 GByte Hauptspeicher, 800 MHz AMD 64 Bit Single-Core und Perl 5.14.2 konnte man zwischen 50.000 und 100.000 Zellen (nicht Zeilen!) pro Sekunde in eine Datei auf die Festplatte schreiben.

Auf aktuellen Systemen können mehr als 200.000 Zellen pro Sekunde erzeugt werden, für 1 Mio Zellen werden damit weniger als 5 Sekunden benötigt.

Zweck und Eingrenzung der Funktionalität

PORF wurde entwickelt, um Massendaten leicht, komfortabel, flexibel und schnell ausgeben zu können, bevorzugt in Listenform. Die Konfiguration soll generisch und Perl-Like funktionieren.

Datenbeschaffung und -Verarbeitung sind nicht Bestandteil des Frameworks und sollen es auch nicht sein. PORF kümmert sich nur um Formatierung und Ausgabe, ist also ein komfortables "Printing" Modul.
Diagramme und längerer Text können von anderen Tools besser bereitgestellt werden und sind daher auch kein Bestandteil von PORF. Aber da man den vollen Durchgriff auf die Reports hat, kann man Teile (Tabellen) erstellen, die man in anderen Dokumenten verwenden/importieren kann.

Wie Porf intern aufgebaut ist und warum, wird in meinem 2. Vortrag *"The Magic Of Eval"* vorgestellt.

Ausblick

Zur Zeit werden nur einfache Listen unterstützt. Mehrzeilige Ausgaben, mehrzeiliger Aufbau mit Verbundzellen, Visitenkarten-Layouts bis hin zur freien Platzierung von Werten auf der Seite sind angedacht.

Aber in der Erstellung einer einfachen, konsistenten API steckt noch viel Arbeit. Wer in irgeneiner Weise mitarbeiten möchte, ist dazu herzlich eingeladen.

Weitere Konfiguratoren für Wikis und LaTeX sind in Vorbereitung.

Autor und Kontakt

Ralf Peine, Jahrgang 1965

Dipl. Mathematik 1991
Software-/Tool-/CM-Architekt

Renesas Electronics Europe GmbH

Ich programmiere seit ca. 20 Jahren mit wachsendem Vergnügen mit Perl, beherrsche aber auch viele andere Sprachen wie C/C++/C#, VB, PHP, Lisp, Java-Script, HTML, XML... Von GroÃŸrechner-SW bis zur Microcontroller-Digitaltechnik (wo ich mich jetzt bewege) habe ich schon so manches entworfen und entwickelt.

Meine private Web-Domain, dort könnt ihr euch PORF herunterladen:

* http://www.jupiter-programs.de/
* http://www.jupiter-programs.de/prj_public/porf/index.htm

Ralf Peine

Report::Porf::Framework

Einleitung

Siehe PORF Praxis

Teil I: Vorstellung

Siehe PORF Praxis

Teil II: Vom Script zum Framework

Evolution der SW Entwicklung mit Perl im Schnelldurchlauf

Anhand von animierten Folien werden die verschiedenen Stufen in der SW-Entwicklung mit Perl, die alle ihrer Berechtigung haben, beleuchtet:

Perl Scripts

Input - Operation - Output

Integration - Operation (das ist neu!)

Nach Ralf Westphal
http://blog.ralfw.de/2013/04/software-fraktal-funktionale.html

Lose Kopplung

Lose Kopplung mittels Closures hat jeder schon einmal angewendet, der z.B. ein GUI mit Perl-TK programmiert hat.

Teil III: Implementierung (old/new Style)

Ableitungen

Ableitungen setze ich nicht mehr ein, denn Ableiten erzeugt jede Menge Ärger und ist noch dazu absolut überflüssig! Viele gute Programmierer hatten das schon im letzten Jahrtausend erkannt.

Ich erkläre, warum das so ist.

Komposition

Statt abzuleiten, kann man besser die Komposition wählen und Objekte aus anderen zusammensetzen.

Zum Abschluss von Teil III stelle ich vor, aus welchen Klassen ich PORF komponiert habe.

Teil IV: Higher Order Perl

Bisher habe ich immer nur Daten in Instanzen von Objekten gespeichert.

Warum eigentlich nicht Subs (Funktionen) oder Closures in Instanzen von Objekten speichern?

Wie und warum ich das in PORF anwende, wird in diesem Kapitel vorgestellt sowie die Funktionsweise und Anwendung von Closures.

Teil V: The Magic Of Eval

Für die performante Ausgabe von Zellen und Zeilen ist die einfache Verwendung von Closures nicht ausreichend, doch mit `eval()` wird alles ganz einfach:

Zunächst werden in der Methode `ConfigureColumn(...)` alle für eine Spalte angegebenen Optionen ermittelt und daraus der Perl-Code als String erzeugt, der die Ausgabe durchführt und (fast) nicht mehr weiter optimiert werden kann.

Mittels `eval("...")` wird dann aus dem String Perl-Code erzeugt. Das ist wie Magie! Aber sehr leicht zu lernen.

Schritt für Schritt wird vorgeführt, wie PORF den String erzeugt, sodass hinterher jeder Zuhörer in der Lage sein sollte, ebenfalls mit `eval()` Perl-Code zu erzeugen.

Teil VI: Zusammenfassung

Wie immer gebe ich am Schluss noch eine Zusammenfassung des gesamten Vortrags auf so wenigen Folien wie möglich.

Autor, Kontakt, Downloads

Ralf Peine
Jahrgang 1965

Dipl. Mathematik 1991
Software-/Tool-/CM-Architekt

Renesas Electronics Europe GmbH

Ich programmiere seit ca. 20 Jahren mit wachsendem Vergnügen mit Perl, beherrsche aber auch viele andere Sprachen wie C/C++/C#, VB, PHP, Lisp, Java-Script, HTML, XML... Von Großrechner-SW bis zur Microcontroller-Digitaltechnik (wo ich mich jetzt bewege) habe ich schon so manches entworfen und entwickelt.

Meine private Web-Domain, dort könnt ihr euch PORF herunterladen:

* *http://www.jupiter-programs.de/*
* *http://www.jupiter-programs.de/prj_public/porf/index.htm*

Anmerkungen zu PORF könnt ihr in meinen Blog schreiben unter:

http://blogs.perl.org/users/jpr65/2013/05/perl-open-report-framework-0901-released.html

Meine aktuellen Favoriten in der SW-Entwicklungsmethodik sind

* Testdriven Development (nie mehr anders!)
* Operation und Integration
  (*http://blog.ralfw.de/2013/04/software-fraktal-funktionale.html*)

"So einfach wie möglich, aber nicht einfacher" (Albert Einstein)

Have Fun Using PORF

Ich würde mich freuen, von euren Erfahrungen mit PORF zu hören und wünsche euch viel Spaß mit Perl :))

Ralf.

Tobias Leich

Slangs in Perl6

Abstract

Bei diesem Talk möchte ich Slangs vorstellen, im Besonderen den Perl 5 Slang.

Es geht also darum andere Sprachen lexikalisch in Perl 6 einzubetten, ohne diese in irgendwelche Quotes stecken zu müssen um sie zu evaluieren.

Es wird gezeigt was alles möglich ist, welche Slangs es zur Zeit gibt, wie die Zukunftsaussichten sind und wo mitgemacht werden kann.

Über den Autor

Tobias bewegt sich mit kurzer Unterbrechnung seit nunmehr 16 Jahren im Perl Umfeld, doch erst seit ungefähr sechs Jahren ist er auf der Seite der Modulautoren.
Und dies geschah wie vielerorts aus der Not heraus: Das Modul was benutzt werden wollte war in schlechtem Zustand. Auf diese Weise widmete sich Tobias mehrere Jahre lang dem SDL-Modul, um eine nutzbare Bibliothek für Spiele und dergleichen bereitzustellen und selbst zu nutzen.
Durch die schmerzhaften Erfahrungen die sich mit C-Bibliotheken, XS und Threads ergaben, erkannte er einen Lichtblick am Horizont genannt Perl 6. Und genau diesem Lichtblick widmet er den größten Teil seiner Freizeit.

Sub LANGuageS - Slangs

Okay, wir kennen alle here- und nowdocs. Diese sind ganz nützlich wenn es darum geht größere Textblöcke einzubetten. Wenn wir mit diesem Text aber noch mehr vorhaben stoßen wir schnell an Unbequemlichkeiten.

```
use Inline C;
greet('Ingy');
greet(42);
__END__
__C__
void greet(char* name) {
  printf("Hello %s!\n", name);
}
```

Hier ein klassisches Inline::C Beispiel. Für C mag das ganz angemessen sein, da wir da im Grunde nur Funktionen deklarieren und aufrufen können, da ist Lexikalität nicht das vorherrschende Thema.

Bei Perl sieht das aber anders aus. Wir wollen auf Variablen zugreifen die in äußeren Blöcken deklariert wurden. Ebenso wollen wir aber auch auf den engstmöglichen Raum eingrenzen, sodass angrenzender Code nicht davon beeinflusst wird. Perl 6 bietet da einen sehr eleganten Ausweg. Wir können nicht nur Module in einem lexikalischen Bereich einbinden, sondern diesen Bereich auch noch weiter verändern. Wir können ihn so weit verändern das er nach und nach zu einer anderen Sprache wird.

```
use v6;
class Foo {
    method greet(:$name = 'Peter') {
        use v5;
        print "Hallo " . $name
    }
}
Foo.greet(:name<Hans>)
```

Ob diese Sprache uns nun hilft eine gegebene Problemstellung effektiver zu lösen, oder ob diese Sprache eine Art Migrationspfad für bestehenden Code darstellt, liegt nur in der Art und Weise begründet, wie wir dieses Feature nutzen.

Links

Repository - *https://github.com/rakudo-p5/v5*
Test Status - *https://github.com/rakudo-p5/v5/blob/master/STATUS-m.md*
Blog - *http://usev5.wordpress.com/*
Rakudo Perl 6 - *http://rakudo.org/*

Steffen Winkler

Testen von Binärdaten

Vorstellung einiger Module zum Testen von Binärdaten

Steffen Winkler - 15. GPW - Hannover - 26.03.2014

Module

- Test::BinaryData
- Test::Bits
- Test::HexString
- Test::HexDifferences

Test::BinaryData

```
is_binary
    "\x00 A",
    "\x00 B",
    'binary data';

not ok 1 - binary data
#   Failed test 'binary data'
#   at ...
# have (hex)               have           want (hex)               want
# 002041------------------ . A          ! 002042------------------ . B
```

Test::BinaryData (ref)

```
is_binary
    "\x00 A",
```

```
    [ qw( 00 20 41 ) ],
    'binary data (ref)';

not ok 2 - binary data (ref)
#   Failed test 'binary data (ref)'
#   at ...
# have (hex)               have           want (hex)               want
# 002041------------------ . A          ! 002042------------------ . B
```

Test::BinaryData beschränken

```
is_binary
    'AA' x 512,
    'AB' x 512,
    'binary data (attr)',
    {
        columns   => 79,
        max_diffs => 2,
    };

not ok 3 - binary data (attr)
#   Failed test 'binary data (reduced)'
#   at ...
# have (hex)               have           want (hex)               want
# 414141414141414141414141 AAAAAAAAAAAA ! 414241424142414241424142 ABABABABABAB
# 414141414141414141414141 AAAAAAAAAAAA ! 414241424142414241424142 ABABABABABAB
# …
```

Test::Bits

```
bits_is
    "\x01\x02",
    [ qw( 0b1 0b11 ],
    'bits';

not ok 4 - bits
#   Failed test 'bits'
#   at ...
# Binary data begins differing at byte 1.
#   Got:    00000010
#   Expect: 00000011
```

Test::HexString

```
is_hexstr
    "\x01ABC",
    "\x01ABCD",
    'test name';

not ok 5 - hex string
#   Failed test 'hex string'
#   at ...
#   at bytes 0-0xf (0-15)
#   got: | 01 41 42 43 .. .. .. .. .. .. .. .. .. .. .. .. |.ABC            |
#   exp: | 01 41 42 43 44 .. .. .. .. .. .. .. .. .. .. .. |.ABCD           |
```

Test::HexDifferences

```
eq_or_dump_diff
    "\x01ABC",
    "\x01ABCD",
    'diff';

not ok 6 - diff
#   Failed test 'diff'
#   at ...
# +---+----------------------------+---+----------------------------+
# | Ln|Got                         | Ln|Expected                    |
# +---+----------------------------+---+----------------------------+
# |  1|'0000 : 01 41 42 43 : .ABC  |  1|'0000 : 01 41 42 43 : .ABC  |
# |   |                            *  2|0004 : 44          : D      *
# |  2|'                           |  3|'                           |
# +---+----------------------------+---+----------------------------+
```

Test::HexDifferences

```
dumped_eq_dump_or_diff
    "\x01ABC",
    <<'EOT',
0000 : 0140414243 : .ABC
0004 : 44         : D
EOT
    'diff dump';

not ok 7 - diff dump
```

```
#   Failed test 'diff dump'
#   at ...
# +---+----------------------------+---+----------------------------+
# | Ln|Got                         | Ln|Expected                    |
# +---+----------------------------+---+----------------------------+
# |  1|'0000 : 01 41 42 43 : .ABC  |  1|'0000 : 01 41 42 43 : .ABC  |
# |   |                          * |  2|0004 : 44          : D      *
# |  2|'                           |  3|'                           |
# +---+----------------------------+---+----------------------------+
```

Test::HexDifferences Optionen

```
eq_or_dump_diff
    "\x01\x23\x45\x67\x89\xAB\xCD\xEF"
    . "\x01\x23\x45\x67"
    . "\x89\xAB\xCD\xEF",
    "\x01\x23\x45\x67\x89\xAB\xCD\xEF"
    . "\x01\x23\x45\x67"
    . "\x89\xAB\xCD\xEF"
    . "\xFF",
    {
        address => 0x4000,
        format  => <<"EOT",
%a : %N %4C : %d\n%1x%
%a : %n %2C : %d\n%*x
EOT
    },
    'diff format';

not ok 8 - diff format
#   Failed test 'diff format'
#   at ...
# +---+---------------------------------------+---+---------------------------------------+
# | Ln|Got                                    | Ln|Expected                                 |
# +---+---------------------------------------+---+---------------------------------------+
# |  1|'4000 : 01234567 89 AB CD EF : .#Eg....|  1|'4000 : 01234567 89 AB CD EF : .#Eg....  |
# |  2|4008 : 0123 45 67 : .#Eg               |  2|4008 : 0123 45 67 : .#Eg                 |
# |  3|400C : 89AB CD EF : ....               |  3|400C : 89AB CD EF : ....                 |
# |   |                                       |  4|4010 :      FF    : .                    *
# |  4|'                                      |  5|'                                        |
# +---+---------------------------------------+---+---------------------------------------+
```

Test::HexDifferences Default-Format

```
"%a : %4C : %d\n"
```

oder komplett geschrieben

```
"%a : %4C : %d\n%*x"
```

Wenn mehr die Daten länger sind als das Format, greift wieder das default Format.

Test::HexDifferences Daten-Format

```
%C  - unsigned char
%S  - unsigned 16-bit, endian depends on machine
%S< - unsigned 16-bit, little-endian
%S> - unsigned 16-bit, big-endian
%v  - unsigned 16-bit, little-endian
%n  - unsigned 16-bit, big-endian
%L  - unsigned 32-bit, endian depends on machine
%L< - unsigned 32-bit, little-endian
%L> - unsigned 32-bit, big-endian
%V  - unsigned 32-bit, little-endian
%N  - unsigned 32-bit, big-endian
%Q  - unsigned 64-bit, endian depends on machine
%Q< - unsigned 64-bit, little-endian
%Q> - unsigned 64-bit, big-endian
```

Test::HexDifferences Adress-Format

```
%a  - 16 bit address
%4a - 16 bit address
%8a - 32 bit address
```

Test::HexDifferences ASCII-Format

```
%d - ASCII anzeigen, was nicht geht als .
```

Test::HexDifferences Wiederholung

```
%*x - endlos
%1x - 1 Mal
%2x - 2 Mal
...
```

Test::HexDifferences \n

```
%\n - ignore \n
```

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.

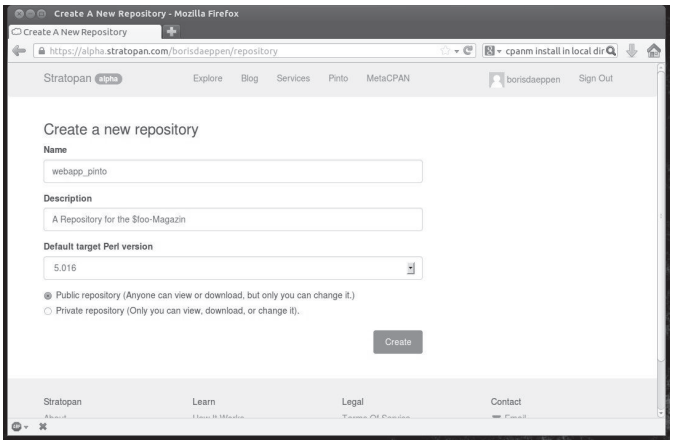


Abbildung 1: Dialog bei Stratopan für das Anlegen eines euen Projekts.

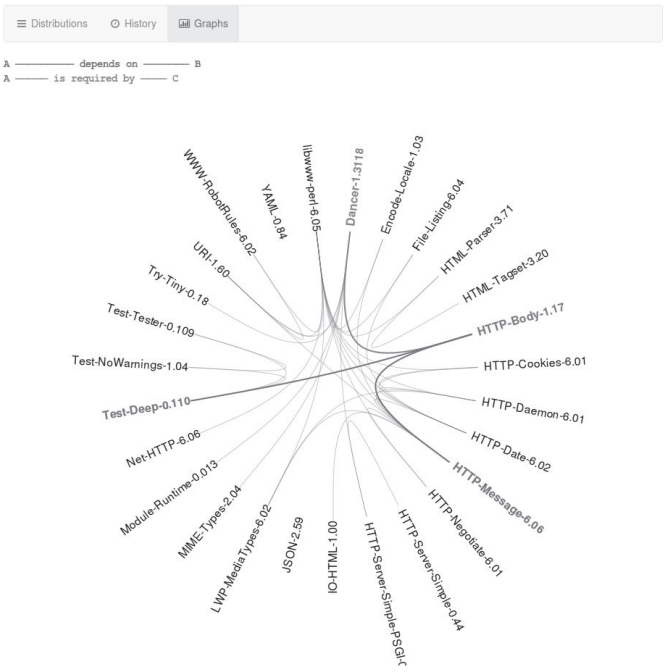


Abbildung 2: Grafik von Stratopan zur Visualisierung der Abhängigkeiten

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

Renée Bäcker

OTRS-Pakete testen und "verifizieren"

Das Thema "Testen" ist ein ganz wichtiges und immer und überall taucht es auf - wenn auch nicht immer ganz so offen. Auch wenn es darum geht, OTRS-Erweiterungen zu programmieren, kommt man ohne Tests nicht aus. Allerdings ist die Kultur des Testens dabei nicht ganz so weit verbreitet wie unter CPAN-Autoren.

Das hat zum einen damit zu tun, dass ein Großteil der Funktionalitäten über die Weboberfläche erreicht wird und OTRS (noch) keine wirklichen Oberflächentests bereitstellt und zum anderen hat es häufig damit zu tun, dass auf Kundenauftrag gehandelt wird und die Erweiterungen nicht "veröffentlicht" wird. Dadurch entfällt ein wenig der Druck Plattformneutral zu programmieren. Man hat ja in der Regel dann vorgeschrieben, für welches Betriebssystem, welche OTRS-Version und welches Datenbanksystem entwickelt wird.

Das ist aber kein idealer Zustand und daran soll sich etwas ändern. Für die Erweiterungen, die auf OPAR - einem Archiv für OTRS-AddOns im Stile des CPAN - geladen werden, soll es zukünftig automatisierte Testläufe geben. Dabei sollen verschiedene Aspekte geprüft werden. Nicht nur Funktionstests sollen laufen, sondern auch Metainformationen sollen gesammelt werden.

Ein kleiner Überblick:

- Einhalten von (OTRS-)Programmierrichtlinien
- Unittests sollen fehlerfrei durchlaufen
- Stabilität und Performanz der OTRS-Installation soll nicht beeinträchtigt werden
- Keine ungewünschte Kommunikation nach außen
- Sind Templates und Funktionalität (auch JavaScript) sauber getrennt
- Bleiben Artefakte nach der Deinstallation übrig

.. und noch viele weitere Sachen

Durch diese Tests und Metainformationen bekommen sowohl die Benutzer als auch die Programmierer jede Menge Hilfe. Die Benutzer sehen, welche Auswirkungen die Installation der Erweiterung auf das eigene System hat und die Entwickler bekommen Hinweise wenn die Erweiterung nicht ganz sauber umgesetzt ist.

Der Ablauf sieht dann so aus:

1. Der Programmierer lädt das Paket auf OPAR hoch
2. Das Paket wird in eine Queue gestellt, in der alle Pakete stehen die überprüft werden sollen
3. Das Paket wird analysiert um zu erfahren welche OTRS-Version(en) benötigt werden
4. Ist das Paket an der Reihe, wird eine Virtuelle Maschine erstellt
5. Auf der Virtuellen Maschine wird die benötigte Software installiert: OTRS, Perl-Pakete, Datenbank, etc.
6. Metainformationen werden gesammelt
7. Unittests von OTRS werden ausgeführt und Zeiten werden gemessen
8. Paket und dessen Abhängigkeiten werden installiert und nach jedem Schritt werden die Unittests von OTRS erneut ausgeführt
9. Pakete werden deinstalliert
10. Alle Ergebnisse werden gepackt und and OPAR geschickt
11. Information an Autor und Darstellung auf der Webseite

In diesem Artikel werden die einzelnen Schritte betrachtet wie sie umgesetzt wurden und wo es evtl. Probleme gab. Das System befindet sich derzeit noch in der Entwicklung. Der meiste Code davon ist auf Github im Account http://github.com/reneeb/ zu finden. Verbesserungsvorschläge bitte dort eintragen.

Die Schritte 1 und 2 sind bereits umgesetzt. OPAR ist eine

Anwendung, die mit Hilfe von Mojolicious umgesetzt wurde. Die Queue ist als einfache Tabelle in der Datenbank umgesetzt, in der ein Cronjob alle x Minuten nachschaut ob es neue Pakete gibt die überprüft werden sollen. Bisher werden einfach Metainformationen aus dem Paket geholt wie Dokumentation, benötigte OTRS-Versionen, Abhängigkeiten, werden Programmierrichtlinien eingehalten. Das Sammeln der Metainformationen passiert im Moment noch auf dem Server, auf dem auch OPAR liegt. Um diesen aber zu entlasten, wird dieser Schritt auf die Virtuelle Maschine ausgelagert. Nur die Information der benötigten OTRS-Versionen wird hier noch gesammelt. Dazu später mehr.

Damit kommen wir schon zu Schritt 4: Für die Prüfung des Pakets wird eine Virtuelle Maschine erstellt. Die Entscheidung fiel zu Gunsten vom JiffyBox, den Cloud Servern von Domainfactory. Eigene Server sollten nicht vorgehalten werden, da auf OPAR (noch) nicht der ganz große Traffic zu finden ist und eigene Server damit zu teuer wären. JiffyBoxen werden hier im Unternehmen immer wieder eingesetzt um Abnahmesysteme für Kundenprojekte aufzusetzen. Damit ist der notwendige Account und das nötige Wissen vorhanden. Außerdem bietet JiffyBox eine nette API, für die es auch Perl-Module gibt (siehe auch $foo Ausgabe 27 - "VM ansteuern mit VM::JiffyBox").

Zum Aufsetzen und konfigurieren der Virtuellen Maschine wird Rex (http://rexify.org) eingesetzt. Damit bleibt alles Perl und Perl-Wissen ist in diesem Unternehmen genug vorhanden. Hinzu kommt noch, dass es für JiffyBox auch schon fertigen Code gibt, der eine solche JiffyBox erstellt.

Mit Rex können solche Maschinen sehr einfach erstellt und konfiguriert werden. Dazu kommen sogenannte *rexfiles* zum Einsatz. Das Ganze erinnert an `make` und die *Makefiles*. Für eine bessere Integration in das Gesamtsystem "OPAR" werden hier aber keine *rexfiles* erzeugt, sondern die Module aus Rex kommen direkt zum Einsatz.

Queueing

Nach dem Auslesen der Queue-Tabelle wird für jeden Eintrag ein fork gemacht:

```
# init fork manager
my $max_processes =
  $self->config->get( 'fork.max' );
my $fork_manager  =
  Parallel::ForkManager->new(
    $max_processes
  );

$logger->trace(
  'init fork manager with max ' .
   $max_processes . ' processes'
);

for my $tmpjob ( @job_info ) {

  # do the fork
  $fork_manager->start and next;

  # create job and run it
  my $job = OTRS::OPR::Daemon::Job->new(
      %{$tmpjob},
  );
  $job->run if $job;

  # exit the forked process
  $fork_manager->finish;
}

$fork_manager->wait_all_children;
```

Im Job selbst wird dann das Paket analysiert um zu erfahren, welche OTRS-Version auf der Virtuellen Maschine überhaupt benötigt wird. Dazu kann man das Paket `OTRS::OPM::Analyzer` verwenden:

```
use OTRS::OPM::Analyzer::Utils::OPMFile;

my $opm_file = '/path/to/package.opm';
my $opm      =
OTRS::OPM::Analyzer::Utils::OPMFile->new(
  opm_file => $opm_file,
);

my @otrs_versions = $opm->framework;
```

VMs einrichten

Für jede dieser OTRS-Versionen wird dann eine Virtuelle Maschine erzeugt:

```
use Rex;
use Rex::Commands::Cloud;

cloud_service 'Jiffybox';
cloud_auth $api_key;

for my $otrs ( @otrs_versions ) {
    my $box = cloud_instance create => {
        image_id => "ubuntu_12_4",
        name     => "test-$otrs",
        plan_id  => $plan_id,
        password => $root_pwd
```

```
    };

    cloud_instance start => $box->{id};
}
```

Auch das könnte man noch parallelisieren.

Im nächsten Schritt muss dann die ganze Software installiert werden. Wenn immer wieder dasselbe gemacht werden muss, kann man sich auch ein Paket für Rex schreiben. Die Schritte zur OTRS-Installation haben sich in den letzten Versionen nicht geändert. Grundsätzlich sieht es so aus, dass zuerst ein User "otrs" angelegt wird mit dem Heimverzeichnis */opt/otrs*.

```
create_user "otrs" =>
  home => '/opt/otrs/',
  groups => [ "www-data" ],
  no_create_home => 1;
```

Danach wird die passende OTRS-Version heruntergeladen, entpackt und nach */opt/otrs* geschoben.

```
my $tmp = "/tmp/otrs-$version.tar.gz";
if ( ! is_file $tmp ) {
    run "wget -0 $tmp $otrs_url";
}

if ( ! is_dir '/opt/otrs/Kernel' ) {
    run "tar zxf $tmp -C /opt";
    unlink $tmp;
}
```

Die Basiskonfiguration muss angelegt werden, die Rechte müssen gesetzt werden und die Datenbank muss erstellt werden.

Für Standardaktionen wie Webserver, Datenbank und Perl-Pakete installieren gibt es schon fertige Befehle für Rex:

```
include qw/Rex::Database::MySQL;
  Rex::Database::MySQL::Admin;
  Rex::Database::MySQL::Admin::User;
  Rex::Database::MySQL::Admin::Schema/;

install "mysql-server";

Rex::Database::MySQL::Admin::
                          Schema::create({
  name => $param->{schema}->{name},
});

Rex::Database::MySQL::Admin::User::create({
  name => $param->{user}->{name},
  host => $param->{user}->{host},
  password => $param->{user}->{password},
  rights => $param->{user}->{rights},
  schema => $param->{schema}->{name}.".*",
});

for my $file ("otrs-schema.mysql.sql",
   "otrs-initial_insert.mysql.sql",
   "otrs-schema-post.mysql.sql") {
 file "/tmp/$file",
 source => "files/db/$otrs_version/$file";

 Rex::Database::MySQL::Admin::execute({
   sql => "/tmp/$file",
   schema => $param->{schema}->{name},
 });

 unlink "/tmp/$file";
}

install "apache2";
```

Damit ist die Virtuelle Maschine in dem Zustand, in dem das große Testen beginnen kann.

Metainformationen zum Paket sammeln

Der nächste Schritt ist das Sammeln der Metainformationen. Auch hier kommt das Paket `OTRS::OPM::Analyzer` zum Einsatz. Es parst zum einen die .opm-Datei -- die Informationen im XML-Format vor -- und zum anderen führt es je nach Konfiguration folgende Aufgaben aus:

- Ist die .opm-Datei wohlgeformt und valide
- Sind Unittests in dem Paket vorhanden
- Ist Dokumentation vorhanden
- Ist es eine Open Source Lizenz, unter dem das Paket steht
- Welche Abhängigkeiten hat das Modul
- Werden System-Aufrufe gemacht
- Hält sich der Programmierer an die Programmierrichtlinien
- Basistests für Templates

Diese einzelnen Aufgaben sind als Rollen für die Hauptklasse umgesetzt und wird entweder auf die einzelnen Dateien des Pakets losgelassen (z.B. die Überprüfung der Programmierrichtlinien) oder auf die .opm-Datei.

Eine Schwierigkeit besteht darin, dass keine Aussage über die Qualität von Dokumentation und Unittests abgegeben werden kann. Jedenfalls nicht automatisiert. Wann ist eine Dokumentation gut, wann nicht? Wenn die Dokumentation in einem Textformat vorliegt kann man vielleicht noch prüfen bzw. raten in welcher Sprache die Dokumentation

vorliegt -- und freie Module für den internationalen "Markt" sollten in Englisch vorliegen. Aber ob die Dokumentation auch tatsächlich die Funktionalität des Moduls beschreibt lässt sich mit (einfachen) Mitteln nicht feststellen. Dazu sind auch zu wenig Informationen im Paket an sich enthalten was es denn macht. Da diese Schwierigkeiten im Moment nicht behoben werden können, wird nur überprüft ob überhaupt etwas vorhanden ist. In der abschließenden Bewertung der Qualität des Pakets haben diese Punkte auch nicht den ganz großen Stellenwert.

Bei OPAR wird -- genau wie beim CPAN -- nicht von Qualität gesprochen, sondern von Kwalitee. Die phonetische Ähnlichkeit soll auch daraufhin deuten, dass es sich nicht um eine absolute Qualität handelt sondern um einen maschinelle Auswertung.

Jede der Rollen liefert einfach das Ergebnis der Prüfung zurück. Die Ergebnisse aller Prüfungen werden gesammelt und später verarbeitet.

Für die Prüfung ob die .opm-Datei wohlgeformt und valide ist, wurde eine XSD-Datei auf Basis der bekannten Pakete und des Codes im OTRS-Code entwickelt. Gegen diese Schema wird die Datei dann mit Hilfe von `XML::LibXML` und `XML::LibXML::Schema` geprüft:

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

$self->tree( $tree );

# check if the opm file is valid.
try {
    my $xsd = do{ local $/; <DATA> };
    XML::LibXML::Schema->new(
      string => $xsd
    );
  }
catch {
    $self->error_string(
      'Could not validate against XML '
      . 'schema: ' . $_
    );
};
```

Programmierrichtlinien einhalten

Die Programmierrichtlinien werden mit `Perl::Critic` geprüft. Die Regeln dazu stehen teilweise in der Entwicklerdokumentation von OTRS, teilweise sind sie aus dem offiziellen OTRS-Code hergeleitet. Leider gibt es keine kompletten offiziellen Programmierrichtlinien. Für OTRS gibt es auf CPAN auch eine Sammlung dieser Richtlinien in dem Paket `Perl::Critic::OTRS`.

`Perl::Critic` arbeitet mit `PPI` für die statische Analyse des Quellcodes. Damit wird ein Baum aus den einzelnen Token aufgebaut und für jeden dieser Knoten können Regeln implementiert werden. Dazu muss man sich sowohl Positiv- als auch Negativbeispiele für Quellcode aufschreiben. Z.B. soll ein `push` auf `@ISA` nicht erlaubt sein. Was weiterhin erlaubt sein soll ist jedoch das `push` auf `@ISA` eines anderen Pakets und die direkte Zuweisung.

Damit hat man folgende Positivbeispiele:

```
@ISA = qw(ParentClass);
@ISA = ('ParentClass');

my $caller = caller(0);
push @{"$caller\::ISA"}, 'ParentClass';
push(@{"$caller\::ISA"}, 'ParentClass');
```

Und folgende Negativbeispiele:

```
push @ISA, 'Data::Dumper';
push( @ISA, 'Data::Dumper' );
CORE::push @ISA, 'Data::Dumper';
CORE::push( @ISA, 'Data::Dumper' );
```

Die schaut man sich genau an. Was macht `PPI` aus diesen Codestücken und lässt sich daraus eine Regel erstellen? Um sich anzuschauen was `PPI` mit den Codestücken macht, kann man `PPI::Dumper` verwenden:

```
use PPI;
use PPI::Dumper;

my $code = join '', <STDIN>;
my $doc  = PPI::Document->new( \$code );
my $dump = PPI::Dumper->new( $doc );
$dump->print;
```

Ein Ausschnitt aus der Baumstruktur ist in Listing 1. zu sehen:

Nach der Analyse kann man sich überlegen wie die Regel schließlich aussieht. Steht die Regel fest, kann man eine Regel für `Perl::Critic` schreiben. Die Regel für den oben gezeigten Code ist in Listing 2 zu finden. Die typische Regel für `Perl::Critic` setzt 5 Methoden um:

- supported_parameters
- default_severity

```
PPI::Document
  PPI::Token::Whitespace        '    '
  PPI::Statement
    PPI::Token::Symbol    '@ISA'
    PPI::Token::Whitespace      ' '
    PPI::Token::Operator        '='
    PPI::Token::Whitespace      ' '
    PPI::Token::QuoteLike::Words
                          'qw(ParentClass)'
    PPI::Token::Structure       ';'
  PPI::Token::Whitespace        '\n'
  PPI::Token::Whitespace        '    '
  PPI::Statement
    PPI::Token::Symbol    '@ISA'
    PPI::Token::Whitespace      ' '
    PPI::Token::Operator        '='
    PPI::Token::Whitespace      ' '
    PPI::Structure::List        ( ... )
      PPI::Statement::Expression
        PPI::Token::Quote::Single
                            ''ParentClass''
    PPI::Token::Structure       ';'
  PPI::Token::Whitespace        '\n'
  PPI::Token::Whitespace        ' \n    '
```

Listing 1

- default_themes
- applies_to
- violates

Über `supported_parameters` kann festgelegt werden, welche Parameter in der `Perl::Critic`-Konfiguration für diese Regel eingestellt werden können. `Perl::Critic` arbeitet mit verschiedenen Leveln -- also verschiedene Schweregrade -- von Regeln. Welches Level die Regel im Standard hat wird über `default_severity` festgelegt. Um für verschiedene Projekte `Perl::Critic` einsetzen zu können, gibt es die sogenannten *Themes*. Es ist wahrscheinlich, dass für OTRS-Projekte andere Regeln gelten als für Projekte bei anderen Kunden. Teilweise werden die auch genau in die gegengesetzte Richtung gehen. Damit sich solche Regeln nicht in die Quere kommen, können diese mit den Themes aktiviert bzw. ausgeschlossen werden.

Da in diesem Projekt aber nur OTRS-Regeln umgesetzt werden, bekommen die alle den gleichen Level und das gleiche

```
use Perl::Critic::Utils qw{ :severities :classification :ppi };
use base 'Perl::Critic::Policy';

use Readonly;

our $VERSION = '0.01';

Readonly::Scalar my $DESC => q{Use of "push @ISA, ..." is not allowed};
Readonly::Scalar my $EXPL => q{Use RequireBaseClass method of MainObject instead.};

sub supported_parameters { return; }
sub default_severity     { return $SEVERITY_HIGHEST; }
sub default_themes       { return qw( otrs ) }
sub applies_to           { return 'PPI::Token::Word'  }

sub violates {
  my ( $self, $elem ) = @_;

  return if $elem ne 'push' and $elem ne 'CORE::push';

  my $sibling = $elem->snext_sibling;
  return if !$sibling->isa( 'PPI::Token::Symbol' ) and !$sibling->isa(
                                                      'PPI::Structure::List' );

  if ( $sibling->isa( 'PPI::Token::Symbol' ) ) {
      return if $sibling ne '@ISA';
  }
  elsif ( $sibling->isa( 'PPI::Structure::List' ) ) {
      my $symbol = $sibling->find( 'PPI::Token::Symbol' );

      return if !$symbol;
      return if $symbol->[0] ne '@ISA';
  }

  return $self->violation( $DESC, $EXPL, $elem );
}
```

Listing 2

Theme zugewiesen. Die interessanten Methoden sind als `applies_to` und `violates`. `applies_to` muss eine Liste an Knotennamen liefern, auf diese Regel angewandt werden soll. In diesem Fall prüft `Perl::Critic` den Code immer dann auf diese Regel wenn es auf einen Knoten vom Typ *PPI::Token::Word* trifft.

In der `violates`-Methode wird dann die Regel ausprogrammiert. Und hier ist dann auch die Analyse der Positiv- und Negativbeispiele wichtig. Wenn der Code gegen die Regel verstößt, muss eine Beschreibung und eine Erklärung zu der Regel zurückgegeben werden, genauso wie der Knoten der den Regelverstoß ausgelöst hat.

Noch sind nicht alle Regeln für OTRS umgesetzt, aber das folgt im Laufe der Zeit.

Damit sind die ganzen Metainformationen gesammelt und das Paket ist auf die Einhaltung der Programmierrichtlinien überprüft. Damit sind wir beim nächsten Schritt: Die Unittests von OTRS müssen ausgeführt und die Zeiten gemessen werden.

Unittests ausführen und Zeiten messen

OTRS liefert ein Unittestscript mit. Dieses liefert leider kein TAP, das Ergebnis kann also nicht mit den Standard-Perl-Tools ausgewertet werden. Um nicht gleich in das OTRS eingreifen zu müssen, wurde ein Parser für die Ausgabe geschrieben, der die Daten auswerten kann. Das Format mit den meisten Informationen, die OTRS bietet ist das XML-Format.

Das Skript von OTRS erlaubt es auch, nur einzelne Testskripte laufen zu lassen, aber hier soll ja grundsätzlich alles getestet werden.

Zur Zeitmessung ist noch eines zu sagen: Es kann nicht einfach die Zeit der gesamten Testsuite genommen werden, da Pakete auch neue Testdateien hinzufügen. Und es ist nicht überraschend wenn zusätzliche Tests die Gesamtdauer der Testausführung vergrößert. Man würde also eine Warnmeldung bekommen dass das Paket die OTRS-Instanz ausbremst obwohl das gar nicht der Fall ist.

Insgesamt gilt, dass man versuchen muss, möglichst viele Störfaktoren auszuschließen. Die Virtuelle Maschine sollte bei einem neuerlichen Lauf der Testsuite nicht auf einmal unter Volllast stehen, da das die Tests natürlich ausbremst.

```
use Capture::Tiny qw(:all);
use OTRS::Unittest::XMLParser;

my ($out, $err) = capture {
    system '/opt/otrs/bin/otrs
       UnitTest.pl -o XML';
};

my $results =
        OTRS::Unittest::XMLParser->new(
    xml => $out,
);

for my $test ( @{ $result->tests } ) {
    say sprintf "%s -> %s",
           $test->name, $test->duration;
}
```

In OPAR selbst werden nur Informationen darüber benötigt, welche Unittests fehlschlagen und wie groß der Geschwindigkeitsverlust bzw. -gewinn für den jeweiligen Test ist. Diese Daten werden auf der Platte abgelegt und für die finale Auswertung bereitgehalten.

Das Paket und dessen Abhängigkeiten installieren

Ein weitaus größeres Problem ist das Paket und dessen Abhängigkeiten zu installieren. Der Paketmanager von OTRS gibt das nicht her, also wird auch hier neuer Code benötigt. Eines der Probleme ist wie die Abhängigkeiten aufgelöst werden können. Man muss dabei beachten, dass es verschiedene Quellen der Pakete geben kann: OPAR, OTRS oder ein Unterordner von OTRS.

Ein CPAN-Client hat den Vorteil, dass es wirklich ein einziges anerkanntes Repository für Pakete gibt. Das fehlt in der OTRS-Community. Außerdem muss der OTRS-Paketmanager auch berücksichtigen welche OTRS-Version im Einsatz ist und muss bei der Installation von Paketen ggf. Änderungen an der Datenbank vornehmen oder Perl-Code ausführen.

Zusätzliche Schwierigkeit besteht darin, dass es zwei Arten von Abhängigkeiten geben kann: Weitere OTRS-Erweiterungen oder Perl-Module.

Der Ablauf der Installation sieht dann folgendermaßen aus:

1. Hole die Liste der Perl-Abhängigkeiten
2. Installiere die Perl-Abhängigkeiten

3. Hole die Liste der OTRS-Abhängigkeiten
4. Prüfe für jede OTRS-Abhängigkeit aus welcher Quelle sie kommt
5. Prüfe für jede OTRS-Abhängigkeit welches OPM-Paket für die vorliegende OTRS-Version benötigt wird
6. Beginne bei 1. für das neue OTRS-Paket
7. Installiere das OTRS-Paket

Die Liste der Perl-Abhängigkeiten ist in der .opm-Datei zu finden. Um diese zu parsen wird wieder `OTRS::OPM::Analyzer` verwendet.

```
use OTRS::OPM::Analyzer::Utils::OPMFile;

my $opm_file = '/path/to/package.opm';
my $opm      =
OTRS::OPM::Analyzer::Utils::OPMFile->new(
  opm_file => $opm_file,
);
my @perl_deps = grep{
  $_->{type} eq 'CPAN';
} $opm->dependencies;
```

Zuerst wird noch geprüft, ob eine Installation überhaupt notwendig ist:

```
for my $cpan_dep ( @perl_deps ) {
    my $module  = $cpan_dep->{name};
    my $version = $cpan_dep->{version};

    eval "use $module $version" and next;
}
```

Für die Installation der Perl-Abhängigkeiten wird `cpanm` verwendet, das in Schritt 4 installiert wird. Um die Ausgabe von `cpanm` überprüfen zu können, wird die Ausgabe mit Hilfe von `Capture::Tiny` eingefangen:

```
use Capture::Tiny ':all';

my ($out, $err, $exit) = capture {
  system 'cpanm', $module;
};
```

Wenn die Installation des Perl-Moduls fehlschlägt, wird auch die Installation des OTRS-Pakets abgebrochen:

```
if ( $out !~
   m{Successfully installed $dist} ) {
  die "Installation of dependency failed!";
}
```

Damit sind die Perl-Abhängigkeiten abgehandelt. Die OTRS-Abhängigkeiten sind der kompliziertere Part an diesem Schritt. Die unterschiedlichen Quellen bieten jeweils eine XML-Datei an, in der die verfügbaren Pakete aufgelistet sind. Darin sind auch die OTRS-Versionen vermerkt, für die das jeweilige Paket zur Verfügung steht.

Aus den drei bekannten Quellen werden diese Informationen an zentraler Stelle gesammelt:

```
use OTRS::Repository;

my $repository = OTRS::Repository->new(
  sources => [
    'http://ftp.otrs.org/pub/otrs/itsm/
         packages33/otrs.xml',
    'http://opar.perl-services.de/otrs.xml',
    'http://ftp.otrs.org/pub/otrs/packages/
         otrs.xml',
  ],
);
```

`OTRS::Repository` parst diese XML-Dateien und macht die Informationen im Repository-Objekt verfügbar. Danach muss noch die passende OPM-Datei gefunden werden:

```
my $uri = $repository->find(
  name => $name,
  otrs => $otrs_version,
);
```

Mit `HTTP::Tiny` wird die Datei dann geholt:

```
my $response = HTTP::Tiny->new
                          ->get( $uri );
```

Und dann geht das Spiel von vorne los. Nach jeder Installation eines OTRS-Pakets müssen die Unittests wie oben beschrieben ausgeführt und ausgewertet werden. Damit wird sichergestellt, dass Fehler oder Beeinträchtigungen nicht schon durch Abhängigkeiten in das OTRS gebracht werden. Wenn es eine Erweiterung unter mehreren Quellen gibt, dann wird die Installation abgebrochen, da nicht eindeutig ist, welche Erweiterung denn installiert werden soll.

Bei der Installation an sich müssen die in der OPM-Datei enthaltenen Dateien in das OTRS eingespielt und ggf. noch Datenbankänderungen vorgenommen oder Perl-Code ausgeführt werden. Die Änderungen an der Datenbank sind ebenfalls direkt in der OPM-Datei enthalten (siehe Listing 3).

Auch der Perl-Code, der ausgeführt werden soll ist direkt in der OPM-Datei enthalten (siehe Listing 4)

Diese werden mit OTRS-Mitteln in SQL umgesetzt und ausgeführt. Dazu bedient sich das Modul den OTRS-Kernmodulen (Listing 5).

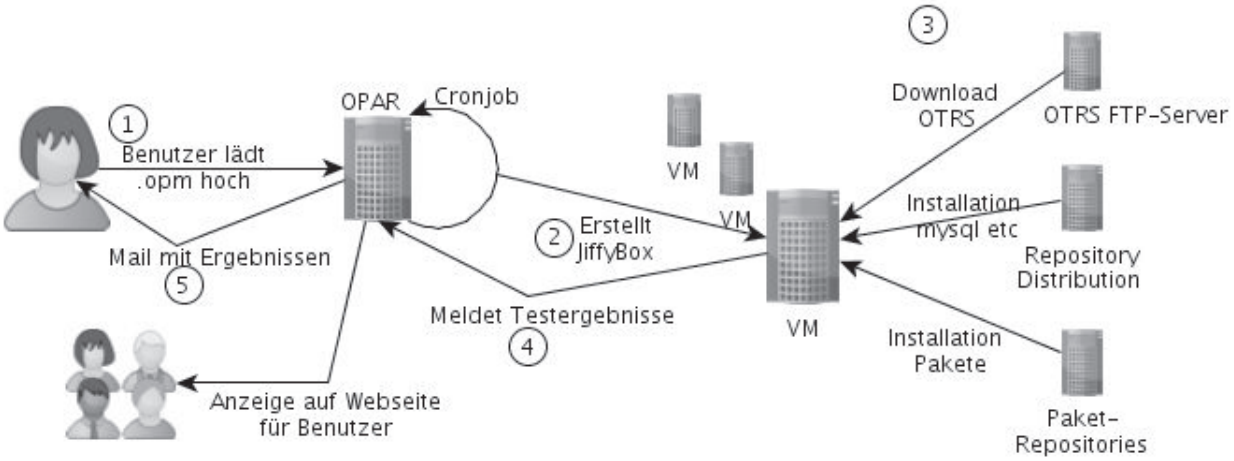
Meldung an OPAR

Die Ergebnisse der ausführlichen Tests müssen auch wieder zurück ins OPAR fließen, um den Autor benachrichtigen und die Ergebnisse auf der Webseite darstellen zu können. Im Laufe der Tests wurden die Ergebnisse immer wieder in ein temporäres Verzeichnis gespeichert. Die ganzen Daten, die in diesem Verzeichnis gesammelt wurden, werden in einem JSON-Objekt gesammelt und an OPAR übertragen. Sollten Auffälligkeiten dabei sein, wird der Autor informiert damit er gegensteuern kann.

Auf der Webseite wird über eine Ampel dargestellt für wie risikoreich OPAR die Installation des Pakets bewertet. Durch einen Klick auf diese Ampel können Interessierte genau sehen was zu der dargestellten Einschätzung führt. Vielleicht ist das die einzelnen Personen uninteressant wenn z.B. die Dokumentation fehlt oder nur in einer anderen Sprache als Englisch vorliegt.

```
use Moo;
use IO::All;

has manager => (is => 'ro', builder =>  1);
has otrs    => (is => 'ro', required => 1);

sub install {
  my ($self, $path) = @_;

  my $opm < io( $path );

  $self->manager->PackageInstall(
      String => $opm,
  );
}

sub _build_manager {
  my ($self) = @_;

  push @INC, $self->otrs;

  try {
      require Kernel::Config;
      require Kernel::System::Main;
      require Kernel::System::Encode;
      require Kernel::System::Log;
      require Kernel::System::DB;
      require Kernel::System::Time;
      require Kernel::System::Package;
  }
  catch {
      die "Can't load OTRS modules!";
  };

  my %objects =
   ( ConfigObject => Kernel::Config->new );
  $objects{EncodeObject} =
    Kernel::System::Encode->new( %objects );
  $objects{LogObject}    =
    Kernel::System::Log->new( %objects );
  $objects{MainObject}   =
    Kernel::System::Main->new( %objects );
  $objects{DBObject}     =
    Kernel::System::DB->new( %objects );
  $objects{TimeObject}   =
    Kernel::System::Time->new( %objects );

  my $package_object =
   Kernel::System::Package->new( %objects );
  return $package_object;
}
```

```
<DatabaseInstall Type="post">
    <TableCreate Name="ps_quick_close">
        <Column Name="id" Required="true" AutoIncrement="true" Type="INTEGER" PrimaryKey=
                                                                        "true"/>
        <Column Name="close_name" Required="true" Type="VARCHAR" Size="250"/>
        <Column Name="comments" Required="true" Type="VARCHAR" Size="250"/>
        <Column Name="subject" Required="false" Type="VARCHAR" Size="250"/>
        <Column Name="create_by" Required="true" Type="INTEGER"/>
        <Column Name="change_by" Required="true" Type="INTEGER"/>
        <ForeignKey ForeignTable="users">
            <Reference Local="create_by" Foreign="id" />
            <Reference Local="change_by" Foreign="id" />
        </ForeignKey>
</DatabaseInstall>
```




```
<CodeInstall Type="post"><![CDATA[
    my $FunctionName = 'CodeInstall';

    my $CodeModule = 'var::packagesetup::' . $Param{Structure}->{Name}->{Content};

    if ( $Self->{MainObject}->Require($CodeModule) ) {
        # Create new instance
        my $CodeObject = $CodeModule->new( %{$Self} );

        if ( $CodeObject ) {

            # start method
            if ( !$CodeObject->$FunctionName(%{$Self}) ) {
                $Self->{LogObject}->Log(
                    Priority => 'error',
                    Message => "Could not call method $FunctionName() on $CodeModule.pm",
                );
            }
        }

        #error handling
        else {
            $Self->{LogObject}->Log(
                Priority => 'error',
                Message => "Could not call method new() on $CodeModule.pm"
            );
        }
    }
]]>
</CodeInstall>
```

Stefan Hornburg

Dancer und DBIx::Class

Übersicht

DBIx::Class ist mit Sicherheit einer der größten Schätze von "Modern Perl" und bietet schnelle und komfortable Datenbankabfragen.

Ebenso erleichtert Dancer das Erstellen von Webanwendungen mit einer leicht verständlichen Programmierung.

Wie können beide zusammen genutzt werden? Zunächst mit dem DBIC Plugin für Dancer. Mit diesem können mehrere DBIx::Class Schemas innerhalb der Dancer-Anwendung verwenden werden.

Um auch die Dancer-Sessions in der Datenbank zu speichern, habe ich eine Engine für Dancer und DBIC geschrieben.

Außerdem werde ich ein Projekt vorstellen, mit dem man einfach den Inhalt von Datenbanken mittels eines DBIx::Class Schemas editieren kann.

Dancer::Plugin::DBIC

DBIx::Class ohne Dancer Plugin

```
use Interchange6::Schema;

$schema =
    Interchange6::Schema->connect(...);

$schema->resultset('User')->search({..});
```

DBIx::Class mit Dancer Plugin

```
use Dancer::Plugin::DBIC;

schema->resultset('User')->search({..});

resultset('User')->search({..});

rset('User')->search({..});
```

Konfiguration

Im Normalfall verwendet man nur ein Schema in seiner Dancer-Anwendung:

```
plugins:
  DBIC:
    default:
      dsn: dbi:mysql:interchange6
      user: racke
      pass: nevairbe
      schema_class: Interchange6::Schema
```

Es sind aber auch mehrere möglich:

```
plugins:
  DBIC:
    default:
      dsn: dbi:mysql:interchange6
      user: racke
      pass: nevairbe
      schema_class: Interchange6::Schema
    legacy:
      dsn: dbi:mysql:interchange5
```

```
        user: racke
        pass: nevairbe
        schema_class: Interchange5::Schema
```

Das Schema `legacy` wird dann wie folgt angesprochen:

```
use Dancer::Plugin::DBIC;

schema('legacy')->resultset('UserDb')
                ->search({..});
```

Im Gegensatz zu `Dancer::Plugin::Database` bietet das DBIC-Plugin keine automatische Unterstützung für UTF-8. Also ist die entsprechende DBI-Option in der Konfiguration einzutragen, hier für MySQL:

```
plugins:
  DBIC:
    default:
      dsn: dbi:mysql:interchange6
      user: racke
      pass: nevairbe
      schema_class: Interchange6::Schema
      options:
        mysql_enable_utf8: 1
```

Die Optionen für die gängigen Datenbanken in der Übersicht:

- SQLite

```
sqlite_unicode: 1
```

- MySQL

```
mysql_enable_utf8: 1
```

- PostgreSQL

```
pg_enable_utf8: 1
```

Dancer::Session::DBIC

Die Sessionengines werden in Dancer normalerweise transparent für den Anwendungscode in der Konfiguration eingerichtet:

Konfiguration

- session

Name der Sessionengine, hier DBIC

- session_options

Optionen

- session_expires

Ablaufzeit der Session

Das sieht dann z.B. für *Interchange6::Schema* so aus:

```
session: "DBIC"
  session_options:
  dsn: dbi:mysql:interchange6
  user: racke
  pass: nevairbe
  schema_class: Interchange6::Schema
  resultset: Session
  id_column: sessions_id
  data_column: session_data
```

Die Konfiguration kann aber ebenso im Hauptmodul stattfinden:

```
set session => 'DBIC';
set session_options => {schema => schema};
```

Folgendermaßen sieht die Tabelle `sessions` aus, die vom Schema *Interchange6::Schema* (Version 0.015) erzeugt wird:

```
CREATE TABLE `sessions` (
  `sessions_id` varchar(255) NOT NULL,
  `session_data` text NOT NULL,
  `created` datetime NOT NULL,
  `last_modified` datetime NOT NULL,
PRIMARY KEY (`sessions_id`)
) ENGINE=InnoDB;
```

Sitzungsablauf

Beim Überschreiten der erlaubten Ablaufzeit wird die Sitzung ungültig, sie wird jedoch nicht in der Datenbank gelöscht. Dafür ist ein Skript zur regelmäßigen Löschung der abgelaufenen Datensätze erforderlich.

Tabelleneditor

Das Frontend für den Tabelleneditor basiert auf AngularJS, das mittels CRUD/REST auf das mit Dancer realisierte Backend zugreift.

Deshalb besteht das Backend auch nur aus einer schmalen API zur Verarbeitung der Abfragen und zur Authentifizierung.

Beziehungen

Der Tabelleneditor erkennt automatisch Beziehungen zwischen den Tabellen, die im Schema vorhanden. Entsprechende Links werden auf den Seiten eingeblendet.

Die Beziehungen sind wie folgt:

- has_many
- belongs_to
- many_to_many
- might_have

Repository

Das Git-Repository für den Tabelleneditor befindet sich auf Github:

https://github.com/interchange/TableEditor

Dancer2

Lediglich das DBIC-Plugin ist auch für Dancer2 verfügbar.

STRATO

```
dev@elbonia:/$ cd ~
dev@elbonia:~$ ls –la current_work/interesting_tasks
total 8
drwxr-xr-x 2 dev    users 4096 2014-03-26 13:37 .
drwxr-xr-x 3 dev  users 4096 2014-03-26 13:37 ..
dev@elbonia:~$
dev@elbonia:~$ ls current_work/tasks
Display all 13412308 possibilities? (y or n)
dev@elbonia:~$
dev@elbonia:~$ perl /var/tmp/something_new.pl
Something New Magic Script v.1.0.4
searching something new.............................[100%]
entries found.............................[stored in ~/new]
bye!
dev@elbonia:~$ ls new
strato.txt  strato_admin_jobs.txt  strato_c_jobs.txt
strato_js_jobs.txt  strato_perl_jobs.txt
dev@elbonia:~$
dev@elbonia:~$ less new/strato.txt
```

STRATO JOBS

Als einer der größten Hoster der Welt arbeiten wir am Puls des Internets: Mit Websites, Online-Speicher, Servern und Webshops bringen wir unsere Kunden ins Web. Einen Großteil der Software entwickeln wir dafür selbst. Fast alle unsere Entwickler nutzen für ihre Aufgaben Perl, zum Beispiel für unseren Online-Speicher HiDrive.

Für unser Team sind wir auf der Suche nach

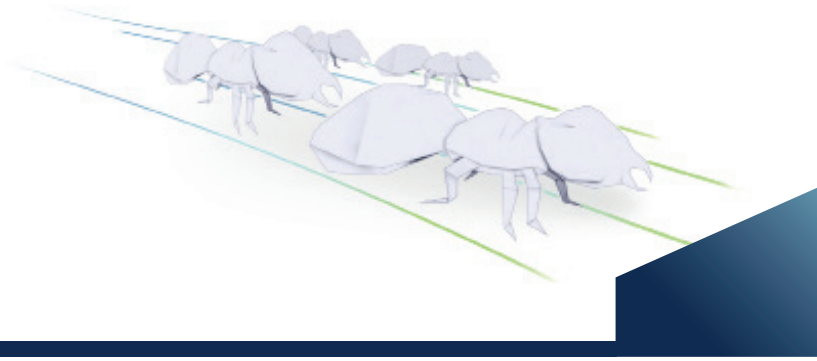
- Experten,
- Kreativen,
- Tüftlern,
- Gesprächigen,
- Zielstrebigen...

Du hast Dich wiedererkannt?
Vielleicht suchen wir gerade Dich.

Offene Stellen unter strato.de/karriere

```
/home/dev/new/strato.txt (END)
```

thinkproject!



ERFOLGREICHE UNTERNEHMEN
BRAUCHEN EIN STARKES TEAM.

JUNIOR/SENIOR WEB-GUI-ENTWICKLER (M/W)

Standort: Berlin | Start: ab sofort

think project! ist eine auf die Anforderungen im Hoch-, Ingenieur-, Industrie- und Anlagenbau ausgerichtete Cloud-Lösung für die unternehmensübergreifende Zusammenarbeit in Projekten (Cross-Enterprise-Collaboration). Zu den insgesamt 20 Segmenten, in denen think project! seine Kunden unterstützt, zählen Generalunternehmen, Energie, Automotive und die öffentliche Hand. think project! ist in 21 Sprachen verfügbar und wird in 40 Ländern in 8.000 Projekten mit insgesamt mehr als 100.000 Nutzern eingesetzt.

Ihr Verantwortungsbereich bei uns:

Unser anspruchsvolles Business-to-Business-Produkt wird seit mehr als dreizehn Jahren unter RedHat enterprise linux auf der Basis von Apache, mod_perl2, PostgreSQL und JQuery entwickelt – vorwiegend mit Open-Source-Software.

- In unserem service-orientierten Applikations-Framework entwickeln Sie für unser Produkt think project! eine neue mobile Benutzeroberfläche sowie die Desktop/Web-Benutzeroberfläche weiter.
- Sie bewerten Trends im Bereich der Web-Technologien mit und integrieren neue Technologien.

Dabei arbeiten Sie eng mit unseren Entwicklern, den Software-Architekten, dem Produktmanagement und der Qualitätssicherung zusammen. Gemeinsam mit diesem 20-köpfigen Team setzen Sie neue Themen eigenverantwortlich um.

Sie passen zu uns, wenn Sie

- eine kundenbezogene, agile Arbeitsweise im Sinne von Test-Driven-Development, Continuous-Delivery und DevOps kennen und bevorzugen;
- sehr gute Kenntnisse und Erfahrung in interpretierten Sprachen (vorzüglich Perl) mitbringen;
- GIT, Buildbot, Bugzilla und Selenium kennen;
- HTML5, REST, UX im Werkzeugkasten haben, oder haben wollen;
- WebApps-GUIs für diverse Plattformen und Browser konzipiert und programmiert haben;
- sich für neue Trends und Frameworks wie SPA, AngularJS, PhoneGap/Cordova interessieren;
- mit hohem Qualitätsbewusstsein ergebnisorientiert arbeiten;
- ein Teamplayer sind, der dennoch eigenverantwortlich und selbständig agiert;
- sehr kommunikationsfähig sind;
- über sehr gute Deutsch- und gute Englischkenntnisse verfügen.

Wir arbeiten in kleinen Teams, um die jeweiligen Anforderungen pragmatisch und im Sinne des Kunden umzusetzen. Sie haben die Chance, ein Unternehmen mitzugestalten, das Ihnen interessante Perspektiven bietet. Es erwarten Sie flache Hierarchien und eine unbürokratische Organisation. Für Ihre engagierte Mitarbeit und Kreativität bieten wir Ihnen ein attraktives, leistungsgerechtes Einkommen.

Bitte schicken Sie Ihre vollständige, aussagefähige Bewerbung ausschließlich per PDF-Datei online über folgendes Web-Formular:

http://community.thinkproject.com/jobs_perl
Bitte nicht per E-Mail senden!

Fragen zum Bewerberformular beantworten Ihnen gerne Frau Milankovic/Herr Leitl, Tel.: +49 89 930 839-300.