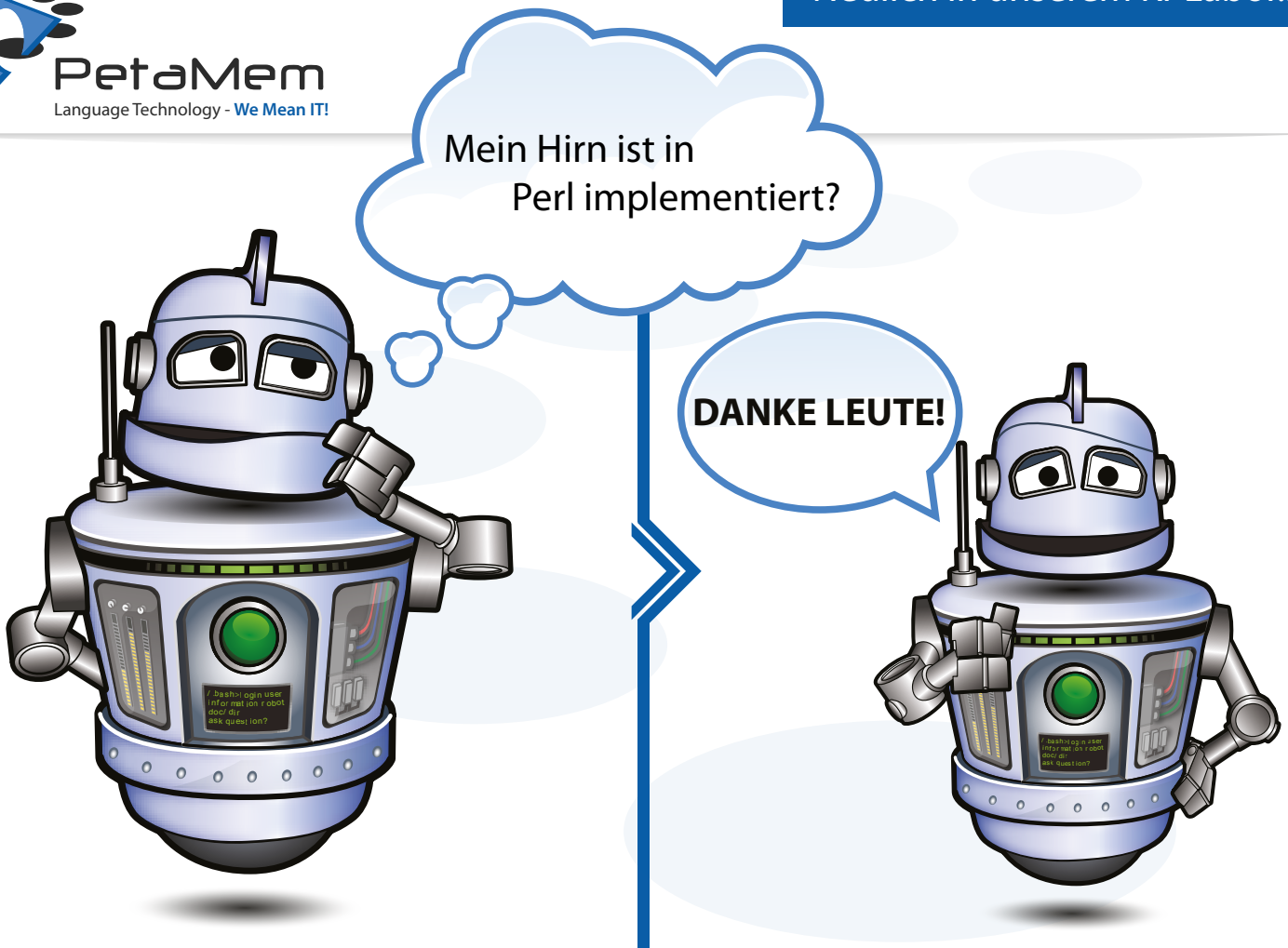
German Perl Workshop

Berlin, 13.03. – 15.03.2013

PetaMem
Language Technology - We Mean IT!

Neulich in unserem KI-Labor...



PetaMem GmbH Flurstr. 78, 90765 Fürth www.PetaMem.com info@petamem.com

Vorwort

Die Berliner Perlmonger freuen sich, dass Du jetzt dieses Heft in den Händen halten kannst. Spät haben wir uns entschieden, den 15. Deutschen Perlworkshop in Berlin auszurichten. Wir sind uns sicher, dass es gelungen ist ein spannendes Programm zusammenzustellen, in denen die unterschiedlichsten Themen behandelt werden. Bitte hilf uns, diese Tagung zu einem vollen Erfolg zu machen indem Du Dich hilfsbereit gegenüber Teilnehmenden und Organisierenden zeigst. Zeige Nachsicht, wenn etwas nicht sofort toll funktioniert und sei begeistert, wenn etwas unerwartet gut ist. Vorfälle auf anderen Tagungen haben uns veranlasst den Code of Conduct der O'Reilly Conferences zu adadptieren:

Wir möchten, dass sich die TeilnehmerInnen wohlfühlen und möchten daher einen Workshop frei von Diskriminierung und Sexismus bieten. Natürlich soll der Spass nicht auf der Strecke bleiben, aber im Zweifel lieber einmal mehr darüber nachdenken, ob ein Witz angebracht ist oder nicht. Bei Problemen könnt ihr euch immer an uns wenden.

Drei spannende Tage in Berlin wünscht Euch das Orga Team des Deutschen Perlworkshop 2013

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

Thomas Klausner

App::TimeTracker, Metaprogramming & Method Modifiers

App::TimeTracker ist ein überaus praktisches Command Line Tool zur Zeiterfassung. App::TimeTracker hat auch eine wunderschöne Website mit tollen Animationen, die die Verwendung demonstrieren: http://timetracker.plix.at

In diesem Vortrag stelle ich App::TimeTracker kurz vor, um dann einige interessante Implementierungsdetails herzuzeigen: Moose, MooseX::Storage, MooseX::Getopt, triggers, dynamisches Anlegen und Anwenden von Rollen und Klassen, diverse Method Modifiers, und viel Perl dazwischen...

Jan Gehring

Von Test nach Live mit Rex

Testsysteme und Produktivsysteme unterscheiden sich sehr oft von einander was zu unerwarteten Fehlern führen kann. Dieser Vortrag zeigt wie man mit Hilfe von Rex eine identische Test- und Liveumgebung aufbauen kann.

Rex ist ein in Perl geschriebenes Open-Source-Tool für die Automatisierung von Applikations- und Konfigurations-Rollouts sowie zum Managen ganzer Infrastrukturen und Cloud-Umgebungen, wie z. B. OpenNebula oder Amazons AWS-Diensten. Im Vergleich zu vielen anderen Lösungen benötigt Rex keinen Agenten auf dem Zielsystem, da die Kommunikation über SSH stattfindet. Alle Aufgaben, die ausgeführt werden sollen, werden in einem strukturierten "Rexfile" in Perl definiert.

Dieser Vortrag wird zuerst auf die Basics von Rex eingehen. Es wird gezeigt wie man sich mit Hilfe von Rex/Boxes eine saubere Entwicklungsumgebung aufbauen kann und diese dann nach den eigenen Bedürfnissen anpassen kann. Im laufe des Vortrags wird ein Rexfile exemplarisch weiterentwickelt um damit auch später die Livesysteme deployen zu können.

Links

http://rexify.org/ - Projektwebseite

Steffen Ullrich <sullr@cpan.org>

HTTP Traffic Inspection und Manipulation mit APP::HTTP_Proxy_IMP

Bio

Der Autor entwickelt seit 1996 in Perl. Seit 2001 entwickelt er bei der genua mbH Hochsicherheitsfirewalls und seit 2011 beschäftigt er sich im Rahmen des von der BMBF gefürderten Projektes Padiofire mit der Verbesserung der Sicherheit im Web 2.0.

Abstract

IMP (Inspection and Modification Protocol) ist eine Schnittstelle zur Anbindung von Analyseverfahren an Analysesysteme wie z.B. Proxies oder IDS. Sie ermöglicht eine unabhängige Entwicklung und Optimierung von Analyseverfahren und Analysesystemen sowie einen einfache Nachnutzung einer Analyseidee in verschiedenen Systemen. App::HTTP_Proxy_IMP stellt ein HTTP-Proxy dar, welcher über die IMP Schnittstelle Inspektion und Modifikation der Datenströme erlaubt.

Einführung

`Net::IMP` implementiert die IMP Schnittstelle in Perl (siehe entsprechenden Vortrag). `Net::IMP::HTTP` baut darauf auf und implementiert Datentypen für HTTP-Verbindung und HTTP-Request. Letzterer umfasst folgende Datentypen

- Request bzw. Response Header

- Request bzw. Response Content

Dieser Datenstrom umfasst den reinen Content, d.h. insb. sind sowohl das Framing des Inhalts durch chunked Encoding sowie evtl. vorhandene Content-Encodings von gzip bzw. deflate entfernt.

- Daten

Hier handelt es sich um Daten, wie sie z.B. nach einem CONNECT Request bzw. nach einem Upgrade auf Websockets (noch nicht implementiert) entstehen.

`App::HTTP_Proxy_IMP` implementiert einen Proxy, der IMP Plugins diese Schnittstelle zur Verfügung stellt und etwaige Änderungen am Header oder Inhalt weiterreicht. Das Plugin selber braucht sich nicht um chunked Encodings, Content-length, Content-encoding, persistente Connections o.ä. zu kümmern, sondern kann sich voll auf die eigentliche Analyse konzentrieren.

Stand der Implementierung

Die Implementierung ist eventbasiert und nutzt AnyEvent. Die daraus resultierende Performance sollte für kleine Gruppen ausreichend sein.

Neben der Implementierung des Proxies wurde die Net::IMP::HTTP-Request Schnittstelle auch in einen Proxy der genugate Firewall eingebaut.

Als Plugin stehen zur Zeit Implementierungen (teilweise nur Proof of Concept) für die Verhinderung von CSRF-Angriffen, Ersetzen von Antworten mit lokalen Varianten, Umschreiben der Ziel-URL, Loggen von Formularen, Drehen von übertragenen Bildern etc zur Verfügung.

Max Maischein

Zweifaktor Authentifizierung mit Auth::GoogleAuthenticator

Übersicht

Das Perl-Modul Auth::GoogleAuthenticator bietet die Möglichkeit, neben einem einfachen Kennwort noch einen weiteren, dynamischen Zugangscode als Sicherungsmechanismus einzurichten. Der folgende Artikel beschreibt die Problemstellung, die API des Moduls und einige Aspekte der Speicherung von Zugangsdaten mit Hinblick auf die Verwendung von OATH-Zugangscodes.

Problemstellung

Jede Anwendung mittlerer Komplexität steht vor dem Problem der Zugriffskontrolle. In vielen Situationen lässt sich der Zugriff auf die Anwendung über das Betriebssystem steuern. Gerade bei Anwendungen mit Zugriff über das Netzwerk ist meistens eine Zugangskontrolle innerhalb der Anwendung nötig. Diese wird oft über eine Kennwortabfrage implementiert, um sicherzustellen, dass nur befugte Personen Zugang erhalten. Andere Möglichkeiten sind zum Beispiel SSL-Zertifikate, die auf dem Nutzerrechner im Browser installiert werden, diese sichern aber nur die Identität der Maschine, nicht die Identität der Person ab.

Zugangskennwörter haben mehrere Probleme, die den Nutzen als Zugangskontrolle einschränken. Zugangskennwörter werden an andere Nutzer weitergegeben. Diese Kenntnis kann nur durch einen sofortigen Kennwortwechsel zurückgenommen werden. Nutzer, die Ihr Zugangskennwort weitergeben sind nicht genug sensibilisiert um einen solchen sofortigen Kennwortwechsel durchzuführen. Weiter werden Zugangskennwörter im allgemeinen nicht geändert oder die Nutzer verwenden ein einheitliches Zugangskennwort über verschiedene Anwendungen hinweg. Hier führt die Kenntnis über ein Zugangskennwort direkt zur Kenntnis von Zugängen zu mehreren Anwendungen.

Die traditionelle Reaktion auf die Probleme sind Regeln zur regelmässigen Kennwortänderung. Diese Regeln verhindern, dass ein einmal bekannt gewordenes Kennwort über einen längeren Zeitraum hinweg nutzbar ist. Die Kombination aus einem herkömmlichen Kennwort und einem dynamischen Zugangscode ist eine Alternative, mit der das Auswechseln der Zugangsdaten vereinfacht werden kann.

Elemente der Authentifizierung

Zugangskontrollen kennen drei Elemente:

• **Wissen** - Dinge, die nur die Person weiss. Idealerweise ist dies ein Kennwort, welches ausschließlich für die Anwendung verwendet wird. Die sogenannten ``Sicherheitsfragen'' wie die Frage nach dem Lieblingstier oder Geburtsnamen der Mutter können möglicherweise leicht aus anderen Quellen ermittelt werden und taugen daher nicht zur Sicherung.

• **Haben** - Dinge, die sich im Besitz der Person befinden. Ein Türschlüssel ist ein bekanntes Beispiel. Wichtig ist, dass sich der Gegenstand nicht leicht kopieren lässt.

• **Sein** - eine Eigenschaft, die die Person auszeichnet. Zum Beispiel die Sprache und Intonation oder ein Fingerabdruck oder Iris-Muster können als ein solches Element gelten. Ein problematischer Aspekt ist das Fehlen der Austauschbarkeit der Merkmale. Sobald die Signatur eines Fingerabdrucks öffentlich und reproduzierbar ist, kann dieser Fingerabdruck nicht mehr als Element genutzt werden.

Der Aufwand, die Elemente ``Haben'' und ``Sein'' für eine Anmeldung über das Netzwerk zu implementieren, ist oft hoch. Insbesondere Fingerabdruckleser oder DNA-Scanner sind nicht allgemein verfügbar.

Implementationsmöglichkeiten

Das Element des vorhandenen Gegenstands hat als Element zusätzlich zum Kennwort verschiedene Implementationsmöglichkeiten. Diese eignen sich auch für die Übertragung über ein potentiell unsichere Verbindung zwischen dem Nutzer und der Anwendung.

Die erste Methode, dieses Element zu implementieren ist das One Time Pad, eine Liste von Zugangscodes, wie sie ein Codebuch oder eine iTan-Liste für das Online-Konto ist. Diese Liste wird dem Nutzer auf sicherem Weg ausgehändigt. Jeder Zugangscode ist nur für eine einzige Transaktion gültig. Durch die Eingabe des richtigen Zugangscodes weist der Nutzer den Besitz der Liste nach.

Eine andere Möglichkeit ist die Verwendung eines zweiten, getrennten und sicheren Nachrichtenkanals, zum Beispiel die Telefonverbindung des Nutzers. Über diesen Kanal kann der Zugangscode zum Beispiel als SMS übermittelt werden. Durch die Eingabe der empfangenen SMS weist der Nutzer den Besitz des Mobiltelefons nach.

Die dritte Möglichkeit, die im folgenden betrachtet wird, ist ein sogenanntes Token, ein kleiner Computer, der abhängig von der Tageszeit und anderen Parametern eine Ziffernfolge anzeigt. Durch Eingabe dieser Ziffernfolge weist der Nutzer den Besitz dieses Tokens nach. Solche Token werden zum Beispiel von RSA (``RSA Token''), Blizzard (``Battle.net Authenticator'') und YubiCo (``Yubi Key'') verkauft.

Google Authenticator

Die Open Source Anwendung ``Google Authenticator'' ist eine Anwendung für Android und iGeräte, die die Erzeugung von dynamischen Zugangscodes analog der von Tokens implementiert. Die Erzeugung geschieht anhand eines bei Einrichtung ausgetauschten Geheimnisses. Zu keinem Zeitpunkt ist eine Netzwerkverbindung zwischen dem Gerät und der Anwendung oder Google nötig.

Die Idee hinter Google Authenticator ist, aus der Uhrzeit und einem vorher beiden Seiten bekannten geheimen Code einen Zugangscode zu berechnen. Wenn beide Seiten dieselbe Uhrzeit und den selben geheimen Code kennen, kommen sie zum selben Ergebnis. Der im Programm verwendete Algorithmus ist OATH (RFC 6238) der Initiative for Open Authentication. Mit `Authen::OATH` gibt es auf CPAN ein Modul, welches diesen Algorithmus implementiert und die selben Berechnungen in Perl durchführt, die auch auf dem Handy durchgeführt werden.

Aus der Funktionsweise des Algorithmus ergibt sich auch eine wesentliche Schwachstelle. Der geheime Code muss auf beiden Seiten im Klartext vorliegen. Damit kann bei einem Informationsleck der Zugangscode von jedem Angreifer ebenfalls errechnet werden. Allerdings macht es Google Authenticator einfach, den geheimen Code gegen einen neuen geheimen Code auszutauschen.

Ablauf im Programm

Für den Anwender läuft die Einbindung von Google Authenticator in ein Programm recht einfach ab:

- Anmelden mit Username und Kennwort
- Einrichten der Zwei-Faktor Anmeldung
  Der Anwender fotografiert den angezeigten QR Code
- ... später ...
- Der Anwender meldet sich an mit seinem Usernamen, seinem Kennwort und der sechsstelligen Zahl, die auf dem Handy angezeigt wird.

API von Auth::GoogleAuthenticator

Die folgenden Aufrufe werden durch das Modul zur Verfügung gestellt:

Konfiguration in der Anwendung

Für jeden Nutzer muss ein separater, geheimer Schlüssel erzeugt werden. Der Konstrutor von `Auth::`

`GoogleAuthenticator` nimmt einen entsprechenden Parameter:

```
# Secret pro Nutzer
my $auth = Auth::GoogleAuthenticator->new(
    secret => 'test@example.com'
);
```

Einrichtungsdaten für den Nutzer

Um dem Nutzer die Schlüsseldaten mitzuteilen, stellt die Klasse mehrere Methoden zur Verfügung. `->registration_key` liefert eine Zeichenkette zurück, die bei der manuellen Eingabe des Schlüssels verwendet werden kann.

```
print "Registration key " .
   $auth->registration_key() . "\n";
```

Am bequemsten für den Nutzer ist die Ausgabe eines QR-Codes für den Schlüssel. Mit der Gerätekamera kann das Bild eingescannt und der Code direkt an Google Authenticator übergeben werden. Das Bild wird im PNG Format zurückgegeben.

```
print $auth->registration_qr_code()
```

Achtung! Dieses Bild ist ein Sicherheitsrisiko und sollte nicht gecached werden. Es enthält den gemeinsamen Schlüssel für den Nutzer im Klartext.

Überprüfen einer Anmeldung

Um zu überprüfen, ob die eingegebenen Anmeldungsdaten zu den Schlüsseldaten passen, gibt es die Methode `->totp()`, die den erwarteten Zugangscode zu einem Schlüssel und einem Zeitpunkt zurückliefert.

```
print "Erwarteter OTP Zugangscode " .
   $auth->totp() . "\n";
```

Falls das Auseinanderdriften der Uhrzeit zwischen Computer und Handy zu Problemen führt, bietet es sich an, das Sicherheitsfenster für den Zugangscode aufzuweiten und für einen Zeitraum auch noch den zuletzt verfallenen Zugangscode und den nächsten Zugangscode zu akzeptieren.

```
# Allow for up to one minute of
# clock difference
print $auth->totp( time-30 ), "\n";
print $auth->totp(  ), "\n";
print $auth->totp( time+30 ), "\n";
```

Die Verifikation kann mittels der Methode `->verify()` auch direkt vorgenommen werden:

```
my $verified = $auth->verify( $user_input )
        ? 'verified'
        : 'not verified';
print "$verified\n";
```

Relevante Module

`Auth::GoogleAuthenticator` verwendet die folgenden Module:

- `Authen::OATH` - Implementation des OATH Algorithmus
- `Imager::QRCode` - zur Erzeugung von QRCode Bildern
- `Dancer` - für die Beispielanwendung

Links

Das Programm ``Google Authenticator'' kann über den Google Play Store installiert werden.

Quellcode: *http://code.google.com/p/google-authenticator/*

Das Programm ist auch für iGeräte verfügbar.

Martin Becker <mhasch@cpan.org>

Number-Crunching mit Perl

Bio Martin Becker

Martin Becker ist Mathematiker und entwickelt Software beim IT-Dienstleister ScanPlus. Er wohnt in Blaubeuren am Südrand der Schwäbischen Alb.

Außer mit Perl verbringt er gerne Zeit mit seiner Familie, Perkussionsinstrumenten, Liegefahrrädern und Tango.

Abstract

Perl ist eine vielseitige Sprache, aber für extrem rechenintensive Anwendungen nicht schnell genug. Zum Glück gibt es Erweiterungen, die Perl zu mehr Rechenleistung verhelfen. Einige davon werden in diesem Vortrag kurz vorgestellt.

Es sind dies Schnittstellen zu einschlägigen externen Bibliotheken (GMP, PARI/GP) und das Projekt PDL (Perl Data Language). Damit wird -- bildlich gesprochen -- aus dem Taschenmesser eine Kettensäge, wenn auch kein Bulldozer.

Links

• http://gmplib.org/
Homepage - The GNU Multiple Precision Arithmetic Library

• http://pari.math.u-bordeaux.fr/
Homepage - PARI/GP

• http://pdl.perl.org/
Homepage - Perl Data Language

• http://search.cpan.org/~pjacklam/Math-BigInt-1.997/
CPAN Distro - Math::BigInt

• http://search.cpan.org/~turnstep/Math-GMP-2.06/
CPAN Distro - Math::GMP

• http://search.cpan.org/~sisyphus/Math-GMPz-0.36/
CPAN Distro - Math::GMPz

• http://search.cpan.org/~sisyphus/Math-GMPq-0.35/
CPAN Distro - Math::GMPq

• http://search.cpan.org/~sisyphus/Math-GMPf-0.35/
CPAN Distro - Math::GMPf

• http://search.cpan.org/~ilyaz/Math-Pari-2.01080605/
CPAN Distro - Math::Pari

• http://search.cpan.org/~chm/PDL-2.4.11/\
CPAN Distro - PDL

Die Delticom AG ist Europas führender Internet-Reifenhändler und wurde 1999 in Hannover gegründet. Das Unternehmen bietet Privat- und Geschäftskunden in 125 Online-Shops in 41 Ländern ein beispiellos breites Sortiment aus Reifen, ausgesuchten Pkw-Ersatzteilen und Zubehör, Motoröl und Batterien.

MIT GRIP(S) ZUM ERFOLG

Wenn Sie mit uns weiter wachsen wollen, dann bewerben Sie sich bei uns als

Softwareentwickler (m/w) Perl

Ihre Aufgaben:

- Entwurf und Entwicklung von sicheren, skalierbaren Webanwendungen auf Basis von LAMP
- Optimierung und Erweiterung von hauseigenen Shop- und Backoffice-Systemen
- interne Abstimmung, Planung und Implementierung eigener Projekte

Ihr Profil:

- mehrjährige Berufserfahrung als Softwareentwickler (m/w) im Bereich Perl
- Erfahrung mit Webentwicklung und Webtechnologien (XHTML, JavaScript, AJAX, SOAP, XML)
- Kenntnisse in der objektorientierten Entwicklung
- sicherer Umgang mit MySQL
- gutes Englisch in Wort und Schrift

Idealerweise ergänzt durch:

- Grundkenntnisse im Umgang mit Linux-Systemen
- Erfahrung mit CVS-Systemen wie SVN oder Git, mit Unit Testing, Testautomatisierung und/oder Selenium sowie im Umgang mit dem CPAN
- Interesse an Performance-Optimierung

Ihre Einstellung:

Sie zeichnen sich aus durch hohe Motivation, Einsatzbereitschaft und großes Engagement. Sie verfügen über eine schnelle Auffassungsgabe sowie eine systematische und analytische Arbeitsweise. Zielorientiertes Handeln ist für Sie selbstverständlich. Als Teamplayer sind Sie flexibel, gewissenhaft und kontaktfreudig. Sie haben Spaß am Lösen von Problemen und legen Wert auf wiederverwendbaren Code und sorgfältige Code-Dokumentation.

Was Sie von uns erwarten dürfen:

Wir bieten Ihnen einen sicheren Arbeitsplatz mit anspruchsvollen und spannenden Aufgaben in einem modernen, schnellen Unternehmen mit flachen Hierarchien. Ein gutes Betriebsklima ist für uns ebenso selbstverständlich wie Ihre Flexibilität am Arbeitsplatz. Sie entscheiden selbst, welches Betriebssystem und welche IDE Sie verwenden möchten und können Ihre Arbeitszeiten flexibel gestalten.

Haben wir Ihr Interesse geweckt?

Dann senden Sie uns bitte Ihre aussagekräftigen Bewerbungsunterlagen, vorzugsweise per E-Mail an: bewerbung@delti.com. Wir bitten ausdrücklich um Angabe Ihrer Gehaltsvorstellung pro anno.

Delticom AG
Britta Knoche
Brühlstraße 11
30169 Hannover

DELTICOM
Aktiengesellschaft

Stefan Hornburg

Wir tanzen um die Welt

American Spaces

Die über 850 "American Spaces" weltweit fördern das Verständnis für die Kultur, Politik und den Lifestyle der USA.

Wir unterstützen die "American Spaces" bei der Beschaffung von Materialien (Bücher, DVD) und stellen die Infrastruktur für den Zugriff auf öffentlichen und kommerziellen Datenbanken zur Recherche bereit.

Anwendungen

Die im folgenden vorgestellten Anwendungen sind alle mit Dancer realisiert. Dancer unterstützt dabei die einfache und schnelle Entwicklung und bietet mit seinem umfangreichen Ökosystem (Plugins, Hooks, Engines) eine hohe Flexibilität. Außerdem ist die Dancer-Community sehr aktiv und hilfsbereit.

Die Daten befinden sich in einem LDAP-Verzeichnis und in mehreren PostgreSQL-Datenbanken.

Dashboard

Startpunkt für alle Anwendungen, stellt Single Sign-On zur Verfügung.

http://americanspaces.state.gov/

eShop

Der Onlineshop für die "American Spaces" dient zur Beschaffung von diversen Materialien wie Bücher, DVD, CD, Publikationen.

Daten und Bilder werden teilweise von externen Quellen eingebunden, wie z.B. ISBNdb.com und LibraryThing.

Die shopspezifischen Daten werden in einer PostgreSQL-Datenbank verwaltet.

https://eshop.state.gov/

eLibraryUSA

eLibraryUSA gibt Besuchern der American Spaces Zugriff auf Informationen, die Amerikaner in öffentlichen Bibliotheken finden können.

Die nicht öffentlichen Resourcen werden durch ezProxy zur Verfügung gestellt. ezProxy greift zur Authentifizierung der Benutzer auf Single Sign-On mittels HTTPS-Request zu.

http://elibraryusa.state.gov/

Training

Support für Training (Registrierung, Hotel, Material, Kalender) inklusive Räumlichkeiten in Wien.

https://americanspaces.state.gov/training

Adminstration

Die Daten zu den American Spaces und den Benutzern werden mit einer einfachen Administrationsoberfläche bearbeitet. Die Verantwortlichen vor Ort können für ihren jeweiligen American Space Benutzer anlegen und pflegen als auch Informationen über den Space aktualisieren.

LDAP

Unser LDAP-Verzeichnis beinhaltet die Hierarchie der American Spaces (Region, Land, Ort) und die Benutzer für unseren Anwendungen. Außerdem speichern wir Zusatzinformationen zu den American Spaces, wie z.B. Typ, Homepage, Jahr der Eröffnung, geographische Länge und Breite.

Es dient ebenso zur Authentifizierung mittels Single Sign-On, dabei wird die Emailadresse als eindeutiger Benutzername verwendet.

Der Zugriff auf das LDAP-Verzeichnis erfolgt mittels `Dancer::Plugin::LDAP`. Dieses Plugin stellt einfache Methoden bereit, um Einträge zu erstellen, bearbeiten und umzubenennen.

```
ldap->quick_insert(
    "uid=racke@linuxia.de,ou=people,"
        . ldap->base(),
              {givenName => 'Stefan',
               lastName => 'Hornburg',
               dosInstitute => 'IRC Berlin',
               dosRole => 'Administrator',
               c => 'Germany',
               l => 'Berlin',
              );
```

`Dancer::Plugin::LDAP` kümmert sich um die korrekte Behandlung von UTF-8 und wandelt automatische Attribute ohne Wert in eine Arrayreferenz mit 0 Elementen um. Da kaum ein LDAP-Attribute einen leeren String als Wert akzeptiert, würde sonst ein Fehler ausgelöst.

Single Sign-On (SSO)

Das Dashboard hat einen SSO-Server integriert und erledigt An- und Abmeldung sowie das Anlegen und Löschen von Benutzerkontos.

Cookies

Wir verwenden TLD-Cookies. Diese können wie folgt mit Dancer gesetzt werden:

```
cookie 'sso.username' => 'racke@linuxia.de',
       domain => '.state.gov',
       expires => config->{session_expires};
```

Zusätzlich wird ein verschlüsseltes Token vom SSO-Server erzeugt und mitgeschickt.

Passwort Policy

Um die Sicherheit der verwendeten Passwörter zu erhöhen, wird folgende Policy eingesetzt:

Ein Passwort besteht aus mindestens 8 Zeichen, in denen ein Klein- und ein Großbuchstabe, eine Ziffer und ein anderes Zeichen vorkommen muß.

Außerdem wird das Passwort auf typische Muster (1234, qwertz), Wiederholungen und "Leet" (4tw, m3h, ph34r) geprüft.

Data::Transpose::PasswordPolicy

Email-Validierung

Beim Anlegen der Benutzerkonto wird die Emailadresse mit `Email::Valid` überprüft. Dabei wird außer auf syntaktischen Fehlern in der Emailadresse (racke.linuxia.de) auch überprüft, ob für die Domain ein DNS-Eintrag (A oder MX) existiert.

Entwicklung

Lokale OpenLDAP-Instanz

Dazu erstellen wir zunächst eine lokale `slapd.conf`. Dann legen wir das `db`-Verzeichnis an und kopieren `DB_CONFIG` aus dem Verzeichnis */usr/share/slapd* hinein.

Jetzt importieren wir eine LDIF-Datei mit den Livedaten:

```
/usr/sbin/slapadd -f slapd.conf live.ldif
```

Und starten unsere lokale Instanz:

```
/usr/sbin/slapd -u 1000 -g 1000 -h
   ldap://127.0.0.1:9009 -f slapd.conf
```

Email Redirect

Um nicht versehentlich Emails an echte Benutzer zu versenden, werden diese zum jeweiligen Entwicker umgeleitet mit `Email::Sender::Transport::Redirect` und folgender Dancer-Konfiguration:

```
 plugins:
   Email:
     transport:
       Sendmail:
         redirect_address: racke@linuxia.de
```

Deployment

Wir betreiben Nginx als "Reverse Proxy" und benutzen Starman als Plack-Backend.

Hier die entsprechende Konfiguration (vereinfacht) für einen virtuellen Host (siehe Listing 1.)

```
server {
    listen 80;
    server_name elibraryusa.state.gov;
    root /home/dancer/Elib/public;

    location / location / {
        try_files $uri @proxy;
        access_log off;
        expires max;
    }

    location @proxy {
        proxy_set_header Host $http_host;
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass       http://localhost:5001;
    }
}
```

Listing 1

Ulrich Habel, <me@rhaen.pm>

Dependencies and Deployments

Deployments

Moderne System Architekturen sind oftmals derartig komplex, dass eine Vielzahl von verschiedenen Entwicklungs- und Qualitätssicherungsinstanzen vor der eigentlichen Produktionsumgebung verwandt wird. Der Vorgang des Auslieferns einer Applikation ist also kein Sonderfall mehr im Rahmen der Entwicklung, sondern wird dadurch zu einer regelmässig wiederkehrenden Tätigkeit. Da menschliche Aktionen jedoch nicht beliebig wiederholbar bei gleichbleibender Qualität sind und die Fehlertoleranz beim Ausspielprozess minimiert werden soll, bietet sich eine Form der Standardisierung und Automatisierung an.

Zu der Anforderung der Ausspielung eigener Software kommen eine Vielzahl weiterer Anforderungen hinzu. Die Server und die eingesetzten Betriebssysteme sollen updatefähig bleiben, eigene Perl Interpreter sollen zum Einsatz kommen wenn die Perl Interpreter der Linux Distribution beispielsweise zu alt geworden sind. Module, die nicht in den Distributionspaketen enthalten sind, sollen innerhalb des Perl Codes der eigenen Applikation verwandt werden und müssen daher auf den Systemen installiert werden.

Auch wenn dieses Problem hinlänglich bekannt erscheint, gibt es immer noch keine belastbare Standardlösung für dieses Problem. Jede Operationsabteilung frickelt ihren eigenen Kram zusammen, der in dem definierten Rahmen oftmals gut funktioniert. Eine Abweichung von dem Rahmen wirft oft Probleme auf, die nicht ohne weiteres zu lösen sind. Bevor an dieser Stelle nun jedoch die bedingungslose Kapitulation eingereicht wird, lohnt es sich das Problemumfeld in kleinere Bereiche aufzuteilen und sich dort die einzelnen Anforderungen anzusehen, zu bewerten und Lösungen zu finden.

Perl - der Interpreter

Viele Linux Distributionen verwenden Perl als essentieller Bestandteil für systemeigene Scripte. Aus diesem Grund wird die Version des Perl Interpreters unmittelbar an das Release der Linux Distribution gekoppelt. Die Linux Distributionen, die auf Red Hat Enterprise basieren, tun sich hier besonders negativ hervor. Das aktuelle Red Hat Enterprise Linux 6.3 liefert die Perl Version 5.10.1 aus. In Anbetracht der Tatsache, dass dieses Release von Perl bereits mehr als 5 Jahre zurückliegt, muss man sich fragen ob man diese Linux Distributionen überhaupt für einen produktiven Einsatz im Perl Umfeld verwenden sollte. Modernere Frameworks oder Module lassen sich mit derart geologisch stabilen Versionen oftmals nicht oder nur unter einem hohen Aufwand zum Laufen bekommen. Beispiel gefällig? Der Versuch das Perl Modul `MooX::Options` zu verwenden scheitert an einem aktuellen `Test::More` - damit beginnt der Spaß. Kurzum, der Austausch des Perl Interpreters scheint eine absolut zwingende Vorraussetzung zu sein, Perl als Sprache wieder nutzbar zu machen.

Windows Benutzer haben es an dieser Stelle durchaus einfacher. Mit ActiveState Perl und Strawberry Perl stehen stets aktuelle Perl Versionen zur Verfügung, die einfach installiert werden können. Der nachfolgende Abschnitt ist daher ausschliesslich den unixoiden Systemen gewidmet.

Beim Austausch eines Perl Interpreters sind ein paar Dinge zu beachten, damit das Upgrade nicht in einem Volldesaster endet. Der systemeigene Perl Interpreter ist grundsätzlich tabu! Jegliche Veränderungen an diesen Teilen führen schneller zur Neuinstallation des Betriebssystems als man annehmen könnte. Gegen das Aufbauen eines Perl Toolchains in einem eigenen Verzeichnis bietet sich an. Da die Reihenfolge in der Umgebungsvariablen `PATH` den Pfad `/usr/local` vor die systemeigenen Pfade wie `/usr/bin` oder`/bin` stellt, ist dieser Ort ebenfalls tabu. Gegen die Installation in einem

Verzeichnis wie beispielsweise `/opt` oder auch `/usr/pkg` spricht hingegen nichts. Mittels der Environmentvariable `PATH` kann der Pfad des Users inviduell gesetzt werden und anstelle des systemweiten Perl Interpreters wird der selbst kompilierte Perl Interpreter verwandt. Offen bleibt jedoch die Frage welche Methodik sich für die Installation eines eigenen Perl Interpreters anbietet. Generell stehen verschiedene Optionen zur Verfügung:

- Selbstkompilieren aus den Perl Sourcen
- Verwendung eines alternativen Package Managers (wie NetBSD *pkgsrc*)
- Verwendung von perlbrew

Selbstkompilieren aus den Perl Sourcen

Generell stellt das Selbstkompilieren aus den Sourcen von Perl auf modernen Unix Systemen keine besondere Herausforderung dar. Ein modernder gcc Compiler und das Tool make sind ausreichend, um Perl selber zu kompilieren. Für eine einzelne Maschine mag dies ein durchaus taugliches Szenario darstellen, für den Einsatz in einem Rechenzentrum mit vielen, verschiedenen Servern und Linux Distributionen beginnt dieses Szenario jedoch schnell zu einem Albtraum zu werden. Wenn sich unter die Linux Distributionen noch andere Betriebsysteme mischen wie beispielsweise Solaris ist der Albtraum Realität geworden und der System Administrator wird zum Perl Interpreter Dompteur.

Verwendung eines alternativen Package Manager (pkgsrc)

Das NetBSD Projekt hat auf der Grundlage der FreeBSD ports seit vielen Jahren ein stabiles und skalierbares Paketverwaltungssystem zur Verfügung. Die Software kann ähnlich wie bei den Ports aus einem Baum mit Kategorien aus den Sourcen kompiliert werden. Alternativ dazu kann die Software in fertiger Paketform als Binärpaket installiert werden. Die Zahl der zur Verfügung stehenden Pakete beträgt aktuell rund 15.000 Pakete. *pkgsrc* ist auf verschiedenen Plattformen und Betriebssystemen verfügbar, zu den großen Plattformen gehören NetBSD, DragonFly BSD, Linux und Solaris. *pkgsrc* bietet viele verschiedene Vorteile gegenüber dem Selberkompilieren. Die Software Paketverwaltung wird von einem großen Projekt gepflegt, Security Advisories werden vom NetBSD Projekt auch für *pkgsrc* herausgegeben und können automatisiert verarbeitet werden. Das NetBSD Projekt hat eine lange Tradition im Perl Umfeld und verfügt im aktuellen Source Tree (-current) immer über die aktuellste Version des Interpreters.

Verwendung von perlbrew

Perlbrew oder auch App::perlbrew ist ein Perl Script, welches das Kompilieren des Perl Interpreters vereinfacht. Es zeigt die zur Verfügung stehenden Perl Versionen von www.perl.org an, lädt die ausgewählte Version herunter und kompiliert sie. Der Benutzer kann danach entscheiden welchen der verschiedenen Perl Interpreter er verwenden möchte. Durch perlbrew ist es möglich verschiedene Perl Versionen mit voneinander getrennten Library Verzeichnissen zu verwenden. perlbrew ist seit einiger Zeit der de-Facto Standard wenn es um das Kompilieren von Perl Interpretern geht und ist weit verbreitet. Durch die Verwendung von Puppet kann die Installation von perlbrew und die Installation von Perl Interpretern automatisiert werden. Um die Installation und den Betrieb mit Puppet und perlbrew zu vereinfachen, wurde auf GitHub ein eigenes Repository eröffnet.

Puppet Module puppet-perlbew (https://github.com/rafl/puppet-perlbrew)

Das Projekt führt derzeit einen Rewrite des gesamten Codes durch - eine Beteiligung ist definitiv erwünscht!

Perl Module

Die Installation von Perl Modulen ist definitiv eine Herausforderung auf jedem System. Das Hochladen oder Kopieren von ganzen Modul Bäumen mittels copy, rsync oder ftp gehört zu den Schauergeschichten der System Administration. Mit diesen Verfahren geht jegliche Nachvollziehbarkeit über Versionen, Installationen oder auch der Historie verloren. Die fertigen Perl Module, die in Paketform der jeweiligen Distribution beiliegen, sind oftmals hoffnungslos veraltet und das Updaten eines einzelnen Paketes zieht meist ein "Paketmikado" nach sich. Man wackelt an einem Paket und der ganze Oberbau gerät ins Wanken.

Auch hier haben sich mittlerweile einige Entwicklungen etabliert und der Toolchain hat sich im Laufe der Zeit weiter verbesssert. Das Tool `cpanm` oder `cpanminus` ist eine Vereinfachung des `CPAN` Moduls. Viele Funktionen stehen nicht oder nur eingeschränkt zur Verfügung, ein direkter Vergleich zu `CPAN` erscheint daher nicht sinnvoll. `cpanm` konzentriert sich besonders auf die Installation von Perl Modulen. Durch den abgespeckten Funktionsumfang lassen sich mittels

cpanm auch oftmals noch Installationen auf kleineren Servern, die die Anforderungen an verfügbares RAM von CPAN nicht mehr befriedigen können, problemlos durchführen.

cpanm bietet dabei nicht nur Unterstützung für Pure Perl Module, also Module, die komplett in Perl geschrieben sind, sondern auch für XS Module. XS Module implementieren Teile der Funktionalität in C. Sie müssen für das jeweilige Hostsystem kompiliert werden.

Auch im Bereich der Module kann auf pkgsrc von NetBSD zurückgegriffen werden. Die Maintainer des Projekts versuchen mit den aktuellen Entwicklungen Schritt zu halten, dennoch kann nicht immer garantiert werden, dass die Module in den aktuellen Versionen vorliegen. Das gleiche Problem wird wie oben beschrieben vermutlich für alle Distributionen gelten.

Windows Benutzer haben in diesem Umfeld noch mit einem ganz anderen Problem zu kämpfen: der Portierbarkeit. Nicht alle Perl Module sind geeignet, dass sie auf Windows kompiliert werden können. Strawberry Perl und ActiveState Perl sind jedoch beide in der Lage aus den Sourcen Module zu kompilieren.

Insbesondere ActiveState Perl tut sich mit vorgefertigten Perl Modulen hervor. Durch eine integrierte Build Umgebung wird der Pool der fertigen Perl Module stets auf einem aktuellen Stand gehalten. Bei dem Release des Perl Moduls Mojolicious, welches sehr häufig Releases durchführt, ist das gut zu beobachten.

Das Puppet Modul puppet-perlbrew wird in den zukünftigen Versionen die Unterstützung von cpanm weiter ausbauen und verbessern.

Eigene Entwicklungen

Eigene Entwicklungen in der Programmiersprache Perl, sei es Applikationen oder Bibliotheken, die Funktionalitäten zur Verfügung stellen, sollten grundsätzlich so geschrieben werden als ob sie auf CPAN zur Verfügung gestellt werden könnten. Der Toolchain, der hinter den Systemen PAUSE und CPAN steht ist einfach schon zu lange im Einsatz und bietet einen guten Startpunkt für eigene Entwicklungen Dazu gehört auch der disziplinierte Einsatz der Toolchains für die Erstellung einer Modul Distribution: Test::More, ExtUtils::MakeMaker oder Module::Build.

Wenn das eigene Modul oder die eigene Applikation Abhängigkeiten zu anderen Modulen aufweist, sollten diese innerhalb des Moduls aufgezeigt werden. Nur so ist die korrekte Installation, die erfolgreiche Ausführung der Tests sowie die vollständige Funktionalität sichergestellt.

Tricks aus dem Bereich Legacy Perl

Perl selbst eine große Tradition auf einer Vielzahl von Systemen. So beinhaltet Perl selbstverständlich neben den bisher aufgezählten Varianten eigene Implementierung zur Installation von Perl Modulen. Diese Legacy Verfahren haben jedoch auch im Jahr 2013 immer noch ihre Berechtigung und können das Leben eines Perl Entwicklers vereinfachen.

Ein brauchbares Verfahren um Module schnell und einfach zu installieren sind die sogenannten bundles. Diese bieten sich vor allem an einen definierten Stand schnell wieder zu erreichen beziehungsweise diesen festzuhalten.

Ein bundle ist nichts anderes die Aufzählung von Modulen innerhalb Contents Sektion in einer POD/PM Datei. Beispiel 1 soll das näher illustrieren.

Der Hinweis wie das Package zu installieren ist befindet sich im Quelltext selbst. Auch zum Neuinstallieren einer Workstation oder für die Migration der Umgebung auf eine zweite Umgebung bietet sich dieses Verfahren an. Wenn nicht bekannt ist welche Module installiert waren, kann die autobundle Funktion von CPAN.pm verwandt werden.

```
snowflake:~ rhaen$ perl -MCPAN -eautobundle
[...]
Wrote bundle file
    /Users/rhaen/.cpan/Bundle/
         Snapshot_2013_02_20_00.pm
```

Dieses von autobundle generierte Package kann auf dem anderen Rechner ganz einfach mit Hilfe des CPAN Moduls installiert werden.

```
perl -MCPAN -e „ Snapshot_2013_02_20_00“
```

```
package Bundle::Personal::rhaen;

our $VERSION = '0.1';
1;
__END__
=head1 NAME

Bundle::Personal::rhaen - Alles was ich zur Entwicklung brauche

=head1 SYNOPSIS

perl -MCPAN -e 'install Bundle::Personal::rhaen'

=head1 CONTENTS

Moo
MooX::Options
IO::All
[...]

=head1 DESCRIPTION

Dieses Package enthält alle Module, die ich zur Entwicklung benoetige.
```

Listing 1

Leider weist dieses Verfahren ein paar Nachteile auf. Zum einen werden alle Perl Module, die installiert wurden, erfasst. Das kann mitunter zu einer imposanten Anzahl von Modulen führen und die Installation zieht sich ein wenig. Auf der Maschine des Autors summierte sich das zu 1987 Modulen. Zum anderen ist die Reihenfolge in der die Liste abgearbeitet wird nicht immer optimiert für die Reihenfolge der Installation. Module, die in der Liste weiter unten aufgeführt sind, werden erneut versucht zu installieren auch wenn sie als Abhängigkeit schon aufgetaucht sind. Dies hat die eine oder andere Nachfrage zur Folge, insbesondere bei optionalen Modulen (*„Module a recommends b“*).

Eine Lösung für dieses Problem kann die Verwendung des Modules *Bundlefly* sein. Rocco Caputo hat das Problem durchaus erkannt und stellt zwei Helferlein zur Verfügung, die die Reihenfolge herausfinden sollen und damit die Installation optimieren können. Die beiden Helferlein heissen `smartcpan` und `smartbundle` und nutzen das Modul `Graph::Directed` zur Optimierung.

Sideproject puppet-perlbrew

Der folgende Abschnitt stellt ein Projekt, welches sich zur Aufgabe gemacht hat mit Hilfe eines Konfigurationsmanagement Systems Perl Interpreter zu installieren vor. Zu finden in das Projekt auf GitHub unter der folgenden URL.

Puppet Module puppet-perlbew (https://github.com/rafl/puppet-perlbrew)

Schnellkurs Puppet

Puppet ist ein Konfigurationsmanagement System welches sich hervorragend für die Verwaltung von großen Rechnergruppen eignet. Puppet führt dabei eine eigene Domain Specific Language ein, die Konfigurationszustände beschreibt. Ein lokaler Agent prüft den beschriebenen Konfigurationszustand gegen den tatsächlichen Zustand eines Systems und führt bei Abweichungen die erforderlichen Änderungen aus. Damit eignet sich Puppet einerseits als eine Art Monitoring über Servicezustände und als Automationswerkzeug, um Rechnergruppen beliebiger Größe in einen definierten Zustand zu versetzen und diesen zu halten. Ein Beispiel für eine solche Konfiguration kann beispielsweise das Einrichten des NTP Daemons sein.

```
service { 'ntpd':
  ensure => running,
}
```

Diese Zeilen sorgen dafür, dass ein ntp Daemon auf dem System gestartet ist und läuft. Sollte der Daemon nicht laufen wird Puppet diesen starten, egal ob es dafür ein Script aus `/etc/init.d`, das Kommando `service` oder das Kommando `systemctl` verwendet. Die Abstraktion wie der Dienst zu starten ist, übernimmt dabei Puppet. Das Interface welches von Benutzerseite zu konfigurieren ist, bleibt immer gleich.

Puppet verfügt über ein großes Ecosystem rund um Module, die das Werkzeug um neue Funktionen erweitern.

Das puppet-perlbrew Modul

Die Hauptfunktionalität des Moduls besteht darin einen Perl Interpreter mit Hilfe des Programms perlbrew zu installieren und zu verwalten. Alle dafür erforderlichen Schritte werden automatisiert, so dass lediglich die Puppet Konfiguration für das Einbinden des Modules vorgenommen werden muss. Desweiteren wird das Modul die Installation von Perl Modulen auf Basis des erstellten Toolchains anbieten.

Verwendung des Modules

Die konkrete Konfiguration befindet sich zum aktuellen Zustand noch im Entwurf. Das Verwenden des Moduls sieht zum aktuellen Zeitpunkt wie folgt aus.

```
# Initialisieren der Klasse
class { 'perlbrew': }

# Installation eines Perl Interpreters
perlbrew::build { '5.16.2': }
```

Das Initialisieren der Klasse übernimmt dabei die Installation des erforderlichen Compiler Toolchains und die Installation des perlbrew Scripts und stellt für die nachfolgende Funktion perlbrew::build einige Variablen zur Verfügung. Die Funktion perlbrew::build übernimmt das Einrichten der erforderlichen perlbrew Verzeichnisstruktur und den entsprechenden Aufruf des perlbrew Scripts. Selbstverständlich können die Klassen beim Aufruf parametrisiert werden, damit eine individuelle Anpassung an das System und die eigenen Bedürfnisse durchgeführt werden kann.

Historie

Das `puppet-perlbrew` Modul wurde ursprünglich von Florian Ragwitz unter dem Namen `puppet-module-perlbrew` entwickelt und folgt der Puppet DSL von früheren Versionen. Mittlerweile hat sich jedoch Puppet auch deutlich weiterentwickelt, es wurden Veränderungen an der DSL vorgenommen und das perlbrew Modul ist mittlerweile technisch auf einem alten Stand. Um jedoch auch in Zukunft dieses Modul nutzen zu können, wird aktuell ein Rewrite des gesamten Codes auf Basis der bisherigen Implementation durchgeführt. Der Rewrite verfolgt durchaus ehrgeizige Ziele. Der Code wird mit einer umfangreichen Testsuite auf Funktion überprüft, der Code soll den aktuellen Empfehlungen des Puppet Projekts entsprechen und eine benutzerfreundliche Dokumentation soll erstellt werden. Das Modul wird auf GitHub gehostet und verwaltet. Das Wiki gibt einen Einblick über die Modul Internas, den Ablauf des Rewrites und die Verwendung des Moduls. Der Issue Tracker enthält eine Übersicht über Fehler, künftige Funktionalitäten und dient maßgeblich der Steuerung. Am einfachsten erreicht man die aktuellen Beteiligten im IRC Channel #puppet auf irc.perl.org oder via EMail.

Fazit des Autors

Perl verlässt insbesondere mit der Einbindung in Puppet den Status der ungeliebten Schattensprache. Es gibt keinen Grund sich mit Perl zu verstecken. Vielleicht muss sich jedoch die Deployment Kultur von Perl verändern. Anstatt den total veralteten Perl Interpreter des Systems zu verwenden, sollten moderne Werkzeuge wie perlbrew, mit oder ohne Puppet Unterstützung die wiederholbare Installation und Konfiguration übernehmen. Perl bietet ein umfangreiches Testframework als Bestandteil der Sprache welche bei jeder Modulinstallation genutzt wird. Der Autor setzt daher auf den Ansatz des Kompilierens des Perl Interpreters auf den Zielhosts mit Unterstützung von Puppet. Die Automation dieses Vorgangs sichert die Wiederholbarkeit bei gleichbleibener Qualität. Die Installation von Modulen wird mittels `cpanm` und der Funktionalität im Puppet Modul ebenfalls automatisiert. So kann in der Entwicklung als auch in der Produktion auf definierte, aktuelle Stände von Code zugegriffen werden.

Verweise und Referenzen

Alle im Text genannten Links können sich im Laufe der Zeit ändern. Die Kontakt Adresse des Autors ist *me@rhaen.pm*/*mailto:me@rhaen.pm*.

- NetBSD pkgsrc http://www.pkgsrc.org
- Puppet http://www.puppetlabs.com

Auf Twitter ist der Autor unter dem Nicknamen @rabenfeder zu finden, neues aus der Welt rund um Puppet und Perl findet sich auf seinem Blog: *http://www.pkgbox.de*.

Steffen Winkler

Was hat join mit map zu tun? Welche Freunde hat \Q

Abstract

2012 hatte ich einen Vortrag ausgearbeitet, der hieß "Vom Spaghetti-Code zur wartbaren Software". Ich hatte viel zu viel Material gesammelt. Den 2. Teil der Sammlung beinhaltet dieser Vortrag. Und einige neu gesammelte Dinge sind natürlich auch dabei.

Boolean ist etwas Besonderes (1/5)

```
perl -e 'use Devel::Peek; Dump 1 == 1'
```

und

```
perl -e 'use Devel::Peek; Dump 0 == 1'
```

Devel::Peek schaut in den Bauch einer Perl-Datenstruktur.

Boolean ist etwas Besonderes (2/5)

```
SV - Scalar value
Flags
 IOK       -> int gültig
 NOK       -> numeric gültig
 POK       -> pointer auf String gültig
 READONLY -> 0 == 1 = 1 geht eben nicht
             zu schreiben
 IV = 1    -> int ist 1
 NV = 1    -> numeric ist 1
 PV        -> es liegt ein string auf
              Adresse 0x...
              mit dem Wert "1" mit \0
              abgeschlossen
CUR = 1    -> Der string hat eine Länge von 1
LEN = 1    -> Die gesamte Datenstruktur hat
              eine Länge von 8 Bytes
```

Boolean ist etwas Besonderes (3/5)

Für viele, die aus anderen Programmiersprachen kommen, ist schon mal völlig unklar, wie ein Scalar gleichzeitig mehrere gültige Werte (String, Numeric, Integer) beinhalten kann.

"https://metacpan.org/module/Scalar::MultiValue" ist ein Modul, mit dem man das aktiv machen kann.

Boolean ist etwas Besonderes (4/5)

```
|| && and or
```

macht das nicht.

Hier wird der rechte Teil der Anweisung nicht boolsch evaluiert.

Deswegen ergibt

```
 50 && 100
```

100 und nicht nur einen wahren Boolean.

Boolean ist etwas Besonderes (5/5)

Dual-Value numerisch verwenden

```
$boolean = !! $any;
$numeric = 0 + $boolean;
$string  = q{} . $boolean;

0 + !! 'bla'
0 + !! ( $result && $result->event )
```

Verstehen von: "Warum hier"

```
my $is_y = $x eq 'y' ? 1 : 0;
print $is_y;
```

An der Schnittstelle "Numeric output" konvertieren.

```
my $is_y = $x eq 'y';
print 0 + $is_y;
```

map verstanden? (1/2)

Jedes Element kommt als $_ (Alias) der Reihe nach im map-Block an.

Das Ergebnis der letzten Anweisung ist der Wert, welcher weitergereicht wird.

```
@destination
    = map {
        my $name = "${_}suffix";
        $name;
    } @source;

@destination = map { "${_}suffix" } @source;
```

map verstanden? (2/2)

```
@destination
    = map {
        $_ if defined;
    } undef, 2;
# is: q{}, 2
```

grep verstanden?

Das Ergebnis der letzten Anweisung wird boolsch verarbeitet

- wahr: Element wird weitergeleiet.
- falsch: Element wird nicht weitergeleitet.

```
@destination
    = grep {
        $_ < 10;
    } @source;
```

grep als map geschrieben

```
  @destination
      = map {
          $_ < 10
          ? $_     # 1 Element weiterleiten
          : ();    # 0 Elemente weiterleiten
      } @source;
```

aus 1 mach 2 oder 0

```
%destination
    = map {
        defined
        # 2 Elemente weiterleiten
        ? ( $_ => undef )
       : ();   # 0 Elemente weiterleiten
    } @source;
```

Was hat join mit map zu tun?

Es sammelt wieder ein.

```
$string
    = join
        ', ',
        map {
            m{ ( \S+ ) }xmsg;
        } (
            'foo bar',
            'baz',
        );
# foo, bar, baz
```

Wann for(each), map, while?

map => for(each)

```
map {
   print "$_\n";
} @array

for my $element (@array) {
    print "$element\n";
}
```

for(each) + push => map

```
my @destination;
for my $element (@source) {
    push @destination, $element x 2;
}

my @destination
    = map {
        $element x 2;
    } @source;
```

for(each) passt hier nicht

```
my @destination;
for my $element (@source) {
    push @destination,
        { $element => shift @source };
}

my @destination;
while ( my ($key, $value) =
             splice @source, 0, 2 ) {
    push @destination, {$key, $value};
}
```

grep als map Beispiel

```
@destination
    = map {
        "$_\n";
    }
    grep {
        length;
    }
    grep {
        defined;
    } @source;

@destination
    = map {
        defined && length
        ? "$_\n"
        : ();
    } @source;
```

Die Listenreferenz

gibt es nicht.

Der Begriff taucht im Netz meist in Verbindung mit Perl auf, wie z.B. hier: http://www.ims.uni-stuttgart.de/~zinsmeis/Perl/material/ho_referenzen_040123.pdf

```
$ref = \@array; arrayref
$ref = [ 1 .. 10 ];
$ref = \( 1 .. 10 ); # \10
```

Code in Strings ausführen

```
"code ${ \( $obj->method ) } in string"
"code @{ [ $obj->method ]  } in string"
```

Methoden aufrufen

```
$obj->method
$obj->$string
$obj->$coderef
$obj->${ \"get_$name" }
$obj->${ \( call_a_sub ) }
```

\Q und Freunde

```
my $chars   = '.,[{';
my $string = "Weiter mit $chars foo.";
$is_match = $string
              =~ m{ \QWeiter mit\E }xms;
$is_match = $string
            =~ m{ \s+ \Q$chars\E \s+ }xms;

\Q => quotemeta
\L => lc
\l => lcfist
\U => uc
\u => ucfirst
\E => Ende
```

Multiplizieren

```
'abc' x 2; # abcabc
( 'abc' ) x 2 # ( 'abc', 'abc');
```

Die Regex ist zu schnell zur Hand

index, rindex, substr

```
$index  = index  'aa bb cc', 'bb'; # 3
$index  = rindex 'aa bb cc', 'bb'; # 3
$substr = substr 'aa bb cc',  3, 2; # 'bb'
$substr = substr 'aa bb cc', -5, 2; # 'bb'
$string ~~ [ qw( a b c ) ];

no warnings qw(numeric);
$numeric = -'123.456-'; # -123.456
```

Gleich und völlig unterschiedlich

require, do, shift, eval

```
# version check, not use 5.010 for
# features
require 5.010;
require PackageName;

do { expr1; expr2; };
do 'include.pl';

shift
shift @ARGV
shift @_

eval { expr1; expr2; }
eval $string;
```

Wann die oder confess? (1/2)

die

```
# Error at [file] line [line].
die 'Error';
die "Error\n";     # Error
die 'Error', "\n"; # Error
# Died at [file] line [line].
die;
die "\n";          # \n
```

Wann die oder confess? (2/2)

confess

```
confess 'Error', 123;
# Error123 at [file] line [line].
# \tPackage::sub(parameterlist)
# at [file] line [line]
confess 'Error', "\n";
# Error
#  at [file] line [line].
```

Woher kommt confess?

```
use Carp qw(confess);
use Moose; # imports confess
```

Die Krankheit von File::Slurp

binmode

```
# ok
my $bytes = read_file(
    'foo/bar/filename',
    binmode => ':raw',
);

# not ok
my $unicode = read_file(
    'foo/bar/filename',
    binmode => ':encoding(cp1252)',
);
```

Die Heilung (1/2)

Path::Class

```
use Path::Class qw(file);
my $unicode
    = file( qw( foo bar filename ) )
        ->slurp(
        iomode => '< :encoding(cp1252)' );
```

Die Heilung (1/2)

Path::Class und nicht File::Spec

```
use Path::Class qw(dir file);
my $unicode
     = file(
        dir('./')->subdir( qw( foo bar ) )
             ->absolute,
         'filename',
     )
     ->slurp( iomode =>
        '< :encoding(cp1252)' );
```

anstatt File::Find

Path::Class::Rule

```
use Path::Class::Rule;

my @html_filenames
    = Path::Class::Rule
        ->new
        ->file
        ->name( qw( *.htm *.html ) )
        ->all( $dir );
```

method und function

as syntax

```
use syntax qw(method function);

fun foo ($parm) {
    return $parm;
}

method bar ($param) {
    return $self->baz($param);
}
```

Web ohne Template (1/4)

oder das HTML-File selbst ist es.

Designertauglich und mit W3C-Validator validierbar.

```
...
  <title></title>
  ...
  <tr class="z-result">
      <td class="z-name">dummy name</td>
```

Web ohne Template (2/4)

Laden

```
use HTML::Zoom;

my $zoom
    = HTML::Zoom
    ->from_file('example.html');
```

Web ohne Template (3/4)

Verarbeiten

```
$zoom = $zoom
  ->replace_content( title => 'Example' )
  ->select('.z-result')
  ->repeat([
   map {
     my $result_ref = $_;
     sub {
       return $_->replace_content(
         '.z-name' => $result_ref->{name},
       )
     }
   } @{$results_ref}
 ]);
```

Web ohne Template (4/4)

Ausgeben

```
print $zoom->to_html;
```

Siehe Verzeichnis "example" in https://metacpan.org/release/Data-Page-Pagination

Alexander Kluth

Klein, schnell, schön - mit Curses den eigenen Scripten eine schöne Oberfläche verleihen

Abstract

Wer kennt das nicht: Man braucht mal eben ein kleines Tool, ein simples Script, um eine Aufgabe zu erfüllen. Dafür die Shell hernehmen? Das ist nicht unser Ding: Wir machen so etwas schnell und einfach in Perl.

Dieser Beitrag zeigt mit Hilfe von Curses, wie man mit Hilfe von Curses seinen gehackten Tools eine schnieke Oberfläche verleiht und so aus einem kleinen Hack schnell ein robustes und einfach zu verwendendes Programm macht.

Als Beispiel wird ein kleines Bugtracking-System alá Bugzilla/Redmine in Perl nachgebaut - mit Curses, SQLite, Mailing-Versand und Export in verschiedene Formate.

Warum Curses, warum nicht Tk? Warum keine GUI?

Kurze Antwort:
Weil wir nix Klicki-Bunti, wir sind Perl-Hacker die mit der Konsole arbeiten.

Lange Antwort:
Das Ziel ist es nicht, unsere zusammegehackten Scripte mit einer grafischen Oberfläche zu versehen, natürlich mit Tooltips, einem Spalshscreen beim Start und einer Büroklammer am Seitenfenster, die leicht schizophren mit nützlichen Hinweisen aufwartet, damit selbst die Sekträterin vom kaufmännischen Leiter unser Script ausführen kann -

Das Ziel ist, Scripten mit einer simplen aber dennoch effektiven Oberfläche zu versehen, und das mit möglichst wenig Overhead. Auch gilt es die Ängste vor Curses zu nehmen, welche sich vor allem in den mit C vorbelasteten Seelen der Gemeinde verankert haben.

Ist das für jedes Script geeignet?

Beileibe nein - stellt euch vor, ihr schreibt einen Dreizeiler, da eine UI drumzupacken wäre das Gleiche wie Java für die Programmierung eines BIOS zu nutzen. Der Einsatzgebiet betrifft vor allem die Kombination viel genutzter Scripte zu einem größeren Programm oder um einem viel genutzten Script eine intiutivere Oberfläche zu verleihen, welches die Benutzung angenehmer macht und die Darstellung von Daten vereinfacht bzw. verschönert.

Was brauche ich zum Starten?

Zum Starten benötigt ihr in jedem Falle `Curses::UI`, eine kleine Abstrationsschicht über Curses, die es ermöglicht vorgefertigte Widgets (Notebook, Buttons, Listviews etc.) zu nutzen.

Nach der Installation via CPAN ist die Nutzung sehr einfach, in folgendem Beispiel (siehe Listing 1) erstellen wir ein einfaches Fenster mit den berühmt-berüchtigten Worten "Hello World".

Dieses kleine Programm erzeugt ein neues Fenster, schreibt "Hallo Welt!" oben linksin die Ecke
und lässt sich per Tastendruck auf "q" oder STRG+C beenden.

```
#!/usr/bin/perl
use strict;
use warnings;
use Curses::UI;

my $cui = new Curses::UI(-clear_on_exit => 0, -color_support => 1);
my $window = $cui->add('main', 'Window', -fg => 'green');
my $label = $window->add('Hello', 'Label', -text => "Hallo Welt!", -x => 1, -y => 1);

$cui->set_binding(sub {exit 0;}, "\cC", "q");

$cui->mainloop();
```

Listing 1

Wie immer bieten die Manpages hervorragende Dokumentation zu allen Themen, besonders zu den einzelnen Widgets, eine wichtige Ressource.

Welche Widgets beinhaltet Curses::UI?

Eine kurze Auswahl wichtiger Widgets:

- `Curses::UI::Label`

Haben wir oben schon eingesetzt, wird für Labels gebraucht

- `Curses::UI::Menubar`

Eine Menüzeile, welche verschiedene Untermenüs zur Verfügung stellt

- `Curses::UI::Listbox`

Eine ListBox zum Anzeigen von Elementen (wird in dem kleinen Ticketsystem des Vortrags benutzt)

- `Curses::UI::Buttonbox`

Einen oder mehrere Buttons erstellt man mit diesem Widgets

- `Curses::UI::Notebook`

Stellt Tab-Funktionalität zur Verfügung

So etwas wie Layout

Ein direktes Layout-System gibt es bei Curses::UI nicht, was aber nicht weiter tragisch ist - folgendes Beispiel erstellt z.B. drei Panels im Fenster:

```
my $project_list =
  $main->add("Projects", "Listbox",
   -fg => "white",
   -border => 1,
   -x => ($max_width / 2),
   -height => int ($max_height / 2)
);

my $ticket_list =
  $main->add("Tickets", "Listbox",
   -fg => "white",
   -border => 1,
   -y => 0,
   -height => int ($max_height / 2),
   -width => int ($max_width / 2)
);

my $textbox =
  $main->add("Content", "TextViewer",
   -fg => "white",
   -border => 1,
   -y => ($max_height / 2),
   -height => ($max_height / 2)
);
```

Dieses Beispiel erstellt ein dreteiliges Layout mit zwei Boxen im oberen und einer langen Box im unteren Bildschirmrand. Durch diese lassen sich einfach von Haus aus per Tabulator durchnavigieren.

Mit der Methode `values()` bzw. der Methode `text()` können bei den Listboxen bzw. bei dem TextViewer Inhalt hinzugefügt werden.

Eventhandling und Dialoge

Eventhandling und Dialoge sind ebenfalls denkbar einfach. Wie schon im Hello World-Beispiel gezeigt erstellt man per `set_binding()` ein neues Binding. Um zum Beispiel eine Methode bei Tastendruck auf "a" aufzurufen:

```
$cui->add_binding(\&open_help;, "?");
```

Nun ruft das Drücken von ? die Methode `open_help` auf, in welcher wir spontan einen kleinen Hilfedialog definieren:

```
sub open_help {
    $cui->dialog(
       -title => "Hilfe",
       -message => "? - zeigt diese Hilfe\n"
. "q - beendet den Dialog/das Programm"
    );
}
```

Drücken wir nun im Hauptprogramm auf das Fragezeichen öffnet sich ein kleiner aber feiner modaler Dialog.

Ressourcen, Links und Weiterführendes

Wie man sieht ist Curses::UI gar nicht schwer, ganz im Gegenteil. Was man damit alles anstellen kann kann auf *https://github.com/deralex/hardticks/* gesehen werden - dort wird ein kleines Ticketing-System ähnlich Bugzilla/Redmine realisiert mitsamt SQLite-Anbindung, Mailversand und Export in verschiedene Formate.

Den kompletten Vortrag mit Vorstellung des Programms ist unter *https://github.com/deralex/klein-schnell-schoen* einsehbar.

Das Modul *Curses::UI* findet ihr im CPAN eures Vertrauens: *http://search.cpan.org/~mdxi/Curses-UI-0.9609/lib/Curses/UI.pm*.

Steffen Ullrich <sullr@cpan.org>

Traffic Inspection und Manipulation mit IMP

Bio

Der Autor entwickelt seit 1996 in Perl. Seit 2001 entwickelt er bei der genua mbH Hochsicherheitsfirewalls und seit 2011 beschäftigt er sich im Rahmen des von der BMBF gefürderten Projektes Padiofire mit der Verbesserung der Sicherheit im Web 2.0.

Abstract

IMP (Inspection and Modification Protocol) ist eine Schnittstelle zur Anbindung von Analyseverfahren an Analysesysteme wie z.B. Proxies oder IDS. Sie ermöglicht eine unabhängige Entwicklung und Optimierung von Analyseverfahren und Analysesystemen sowie einen einfache Nachnutzung einer Analyseidee in verschiedenen Systemen. Net::IMP implementiert diese Schnittstelle ein Perl und bietet auch passende Analysesysteme.

Problemstellung

Das vom BMBF geförderte Projekt Padiofire beschäftigt sich mit der verbesserten Absicherung von Web Applikationen mit Hilfe von Firewalls, IDS etc. Für dieses Projekt brauchten wir eine Schnittstelle, um die gewünschten Analyseideen an vorhandene Systeme, z.B. Proxies auf der genugate Firewall, anzubinden.

Folgende Anforderungen haben wir an eine derartige Schnittstelle gestellt:

- Damit einmal umgesetzte Analyseideen für möglichst viele Systeme verfügbar sind, muss die Schnittstelle einfach zu implementieren, aber trotzdem flexibel genug für verschiedenste Arten von Analysen sein.

- Die Schnittstelle muss skalieren Insbesondere darf sie nicht blockieren und dadurch bei parallelen Analysen Threads benötigen, sondern muss sich in unsere bisherigen, eventbasierten Umgebungen, gut einfügen.

- Die Schnittstelle muss performant sein, d.h. sowohl geringe Latenzen, wie auch einen hohen Durchsatz bieten. Sie muss auch diesbezügliche Optimierungen der Analyse ermöglichen, d.h. den Overhead für Fälle, bei denen nicht alle Daten einer Verbindung zu analysieren sind (z.B. nur HTTP-Header, aber nicht Body) möglichst gering halten.

- Um eine Analyse von laufenden Verbindungen zu ermöglichen muss die Schnittstelle streamingfähig sein, d.h. die Daten analysieren, sobald sie da sind und nicht erst, wenn alle Daten vollständig sind.

Lösungsidee

Die grundlegenden Ideen von IMP sind

- Daten werden so früh wie möglich an die Analyse weitergereicht, d.h. in Chunks.

- Das Analysesystem puffert die Daten solange, bis es weiß, was es damit machen soll. Auf diese Weise müssen die Daten nur zur Analysekomponente kopiert werden, nicht aber zurück (außer bei Veränderungen).

• Die Antworten von der Analysekomponente erfolgen über einen Callback. Sowohl das Analysesystem wie auch die Analyse selber führen Buch über die Position der Daten im Datenstrom und die Antworten beziehen sich darauf, d.h. erlauben z.B. das Weiterleiten aller Daten bis zur angegebenen Position.

• Antworten können festlegen, was mit zukünftigen Daten gemacht wird. Wenn z.B. die Analysekomponente nicht an den weiteren Daten der Verbindung interessiert ist, kann sie dem Analysesystem erlauben, diese Daten ohne weitere Analyse weiterzuleiten.

Es gibt folgende wesentliche Antworten in IMP

PASS dir,offset
Alle Daten in Richtung `dir` bis Position `offset` können weitergeleitet werden. `offset` kann sich dabei sowohl auf bereits empfangene Daten, wie auch auf zukünftige Daten beziehen. In letzterem Fall brauchen diese garnicht zur Analysekomponente geschickt werden.

REPLACE dir,offset,newdata
Alle Daten von der letzten Position bis zu `offset` werden mit `newdata` ersetzt. `offset` muss sich auf eine Position innerhalb der empfangenen, aber noch nicht weitergeleiteten, Daten beziehen.

PREPASS dir,offset
Daten können bis Position `offset` unverändert weitergeleitet werden, müssen aber trotzdem an die Analysekomponente geschickt werden. `offset` ist normalerweise eine Position in der Zukunft. Anwendungsfälle sind z.B., wenn die Daten nur geloggt werden sollen, oder wenn die Analysekomponente einen State mitführen muss, um zu erkennen, wann erneut Daten kommen, die zu analysieren sind (z.B. bei persistenten HTTP-Verbindungen). Eine weiterer Anwendungsfall ist Daten mit minimaler Latenz weiterzuleiten, sie aber nachfolgend doch noch zu analysieren und die Menge der potentielle gefährlichen Daten durch geschickte ständige Aktualisierung von `offset` akzeptabel zu halten (z.B. bei Real-Time Daten).

Stand der Implementierung

Die Schnittstelle ist im frei verfügbaren Perl-Module `Net::IMP` implementiert. Dort sind auch als Analysesysteme ein Proxy sowie ein Pcap-Filter (dh. Lesen Pcap, Analysieren und Modifizieren, Schreiben Pcap) verfügbar, sowie diverse Analyseplugins für Logging (Session als Pcap loggen) oder Forcieren von Protokollen mittels reguläre Ausdrücke Weiterhin gibt es diverse HTTP spezifische Analysen in `Net::IMP::HTTP` sowie ein dazu passender Proxy in `App::HTTP_Proxy_IMP`.

Darüberhinaus ist die Schnittstelle in einigen Proxies der genugate Firewall bereits implementiert und weitere Implementation folgen. Wir beschäftigen uns ebenfalls damit, die Schnittstelle in den OpenBSD relayd zu integrieren.

Herbert Breunung <lichtkind@cpan.org>

Perl 6 OO

Über den Autor

- aktives Mitglied des Forums perl-community.de seit 2005
- zahlreiche Vorträge auf Konferenzen (Perl und anderes)
- regelmäßiger Autor der Perl-Magazins
- schrieb dort auch über Perl 6
- verfolgt das Projekt ebenfalls seit 2005
- Autor der "Perl 6 Tablets" Dokumentation

Abstract

Perl 6 versucht bei der Objektorientierung, wie bei allem anderen auch, alles zu bieten und noch etwas mehr. Und zwar so, daß es sich miteinander verträgt. Dieser Vortrag stellt die wichtigsten Funktionalitäten und deren Syntax vor. Er soll die Zuhörer in die Lage versetzten, gleich das erste Skript für Rakudo zu schreiben Es werden aber auch die Unterschiede zu Moose gezeigt, um Unsicherheiten auszuräumen und damit eher an Perl 5 Interessierte sich nicht langweilen müssen.

Motivation

Die Objektorientierung in Perl 5.0 hat ihre bekannten Defizite. Die üblichen Dinge, wie Erzeugung von Attributen sind aufwändig oder man benutzte eine der vielen, teilweise untereinander inkompatiblen OO-Module. Für echte private Attribute braucht es echter Anstrengungen. Diese Lage zu verbessern, war eines der hauptsächlichen Anfangsmotivationen der Perl 6-Anstrengungen. Mit der Zeit wurde die Sprache immer komplexer, da alle interessant erscheinenden Ideen mit eingeflochten wurden. Der Hauptgrund warum auch der Entwurf der Sprache solche Zeit beansprucht, liegt im Ziel diese Ideen zu einem harmonischen und ausgereiften Gesamtgefüge zu verbinden.

Generelle Syntaxregeln

Da Perl 6 auf der grünen Wiese neu errichtet wird, konnten Schlüsselworte frei vergeben werden. Eine Klasse beginnt mit `class`, eine Methode mit `method`, anstatt mit `sub`. Dem versucht sich `MooseX::Declare` anzunähern, kann es aber nur teilweise erreichen. Zum einen weil Interpunktionszeichen in Perl 6 in vielen Fällen anders belegt sind. In Perl 6 heißt es `$obj.methode()`, in Perl 5 nunmal `$obj->methode()`, daran kann selbst muß schwer was ändern. Auch innerhalb der Signaturen greift Perl 6 auf Sonderzeichen zurück, um Syntax auszudrücken Zum anderen kennt Perl 6 zum Beispiel (optionale) Datentypen. Wo Moose sich mit:

```
has 'size' => ( isa => Str );
```

behilft und Werte in der Hinterhand prüft (ob sie bestimmten Typen entsprechen), kann Perl 6 ganz locker und knapp typisierte Attribute anmelden:

```
has Str $.size;
```

Eine letzte merkenswerte Regel der Perl 6 Syntax: Jedes Schlüsselwort hat nur eine bestimmte Aufgabe. `package` markiert nur einen Namensraum, `module` ein Modul und `class` immer nur eine Klasse. Falls ein Wort wie `is` in verschiedenen Zusammenhängen auftaucht, dann weil es in Wirklichkeit immer das selbe tut, oder wenigstens die gleiche Idee verkörpert.

Klasse

Wie erwähnt fängt eine Klasse mit `class` an, gefolgt vom Namen, obwohl es auch anonyme Klassen gibt. Dem `class` läßt sich auch ein `my` voranschieben und lexikalisch lokale Klassen zu erzeugen. Der Abschluß dieser Kopfzeile, die meist noch wesentlich mehr Informationen beinhaltet, kann wie in Perl 5 mit einem Semikolon beendet werden. Üblicher ist

es in Perl 6 jedoch, die Klasse mit geschweiften Klammern zu umschließen. Auch das ist ebenfalls mit `MooseX::Declare` möglich.

Attribute

Diese werden wie aus Moose bekannt mit `has` deklariert une ebenfalls durch die automatisch generierte `new`-Methode erzeugt. Da Attribute eigentlich Zugriffsmethoden sind (Getter/Setter), haben sie einen Punkt im Namen (`$.attr`) Attribute mit einem Ausrufezeichen sind privat (`$!attr`) Bei der Definition kann man Attributen neben einem Typen auch eine Wert zuweisen, was sich vor allem bei reinen Gettern empfiehlt. Das kann aber auch noch bei der Objekterzeugung per `new` oder `clone` nachgeholt werden, was den Default-Wert im Objekt überschreibt.

```
my $obj = Klasse.new( :RoAttr<wichtig!>);

my $neuobj =
  $altobj.clone(:RoAttr<viel wichtiger!>);
```

Obacht, Attribute sind von Haus aus reine Getter. Wer schreiben will, muß ihnen das rw (read write) Trait verpassen.

```
has Int $.breite is rw = 22;
```

`is` kann auch Klassen neue Traits (Eigenschaften) geben. Da intern alles eine Objekt ist, steckt viel Macht in diesem kleinen Wort.

Methoden

Neben den zu erwartenden Methoden (`method`), können Klassen auch Submethoden `submethod` enthalten. Diese sind gewöhnliche Methoden, werden jedoch nicht weitervererbt.

Steht for dem `method` ein `multi`, so darf man noch weitere Methoden mit diesem Namen aber unterschiedlicher Signatur hinzufügen. Wird die Methode Aufgerufen, so wird anhand der übergebenen Parameter entscheiden (MMD - multi method dispatch) welcher Kandidat ausgeführt wird oder ob eine Ausnahme geworfen wird.

Vererbung

Da Traits auch nur Klassen sind ist das Schlüsselwort für Vererbung zwischen ihnen das erwähnte `is`. class Excalibur

```
is Raumschiff is Spezialanfertigung { ... }
```

Mehrfachvererbung ist möglich.

Rollen

Um Vererbungshierarchien nicht ausufern zu lassen wurden Rollen eingeführt. Diese sind woanders konzeptionell als Traits oder Mixins bekannt, verhalten sich in Perl 6 etwas anders, daher der eigenständige Name. Rollen sind einfach nur ein Bündel Attribute und Methoden, die einer Klasse oder einem Objekt, auch während der Laufzeit, zugeschoben werden können. Dabei prüft Perl 6 ob die Klasse / das Objekt bereits Methoden und Attribute mit diesem Namen besitzt und wirft im positiven Falle Ausnahmen, welche natürlich abgefangen werden können.

```
role AlarmUhr {
    has DateTime $.timer;

    method Wecken ($food) {
        if ($.timer == DateTime.now) { ...
        }
    }
}
```

Folgende Zeile erweitert die Excalibur:

```
class Excalibur is Raumschiff does AlarmUhr
...
```

Eine spätere Nachrüstung sieht ähnlich aus:

```
$schiff does AlarmUhr;
```

Ermunterung

Alles hier beschriebene ist bereits in Rakudo implementiert und wartet auf freudige Erkundung.

Richard Jelinek

Umrisse einer Perl-Strategie

Abstract

Wir wollen doch alle, dass es Perl gut geht. Oder nicht? Geht es Perl gut bzw. wie ist "gut" definiert? Ich behaupte, es geht Perl nicht so gut wie viele meinen, sicher nicht so gut wie viele möchten und ganz sicher nicht so gut we es gehen *könnte*. Ich behaupte weiterhin, Perl hat "das Zeug" eine populärere Sprache zu werden als Python und PHP zusammen. Analyse der Ist-Popularität und was tun um sie ein "klein wenig" zu steigern sagt dieser Vortrag. Gefallen wird das nicht allen.

Ziel und Weg

Haben wir ein Ziel? Natürlich! Nur ist die Perl-Community sehr gut darin nicht klipp und klar "mit einer Stimme" solche Sachen zu benennen. Haben wir eine Strategie als Weg dieses Ziel zu erreichen? Natürlich nicht! Ovid benutzte in seiner YAPC::EU 2012 Keynote ein Zitat aus dem Schachumfeld: *Unter Spielern gleicher Fähigkeiten gewinnt die schlechte Strategie vor gar keiner Strategie* - Perl hat leider **keine** Strategie. Dafür schlägt es sich noch recht wacker - also müssen Fähigkeiten da sein. Lasst uns diese Fähigkeiten auf den richtigen Weg bringen.

Der Illusion beraubt

Wer findet, der nächste Hackathon, das nächste Feature in perl, das nächste CPAN Modul, die nächte Konferenz, der nächste Vorstand oder Arbeitsgruppe der Perl Foundation oder Perl6 wird Perl *"den Boost geben"* - liegt leider daneben.

Wer auch glaubt, alles sei in Butter, wenn "wir als Community einen Level aufsteigen" (Matt S. Trout) oder das "es doch nicht um Konkurrenz [zwischen Programmiersprachen] gehe" (derselbe), liegt falsch!

Wer allen Ernstes der Ansicht ist, man könnte Manager, Entscheider - also prinzipiell Uneingeweihte - mit *irgendeinem* technologischen Feature von Perl ködern - lebt hinterm Mond.

Wenn Du glaubst, die Mehrzahl der Entwickler adoptiert eine Programmiersprache aus Sympathie- und nicht knallharten ROI-Überlegungen, bist Du ein hoffnungsloses Träumerchen.

Und wer schliesslich glaubt, Perl erlebe eine Renaissance, sollte für sich selbst prüfen, ob er noch ein Strohfeuer von einer Kernfusion unterscheiden kann.

Die Hoffnung geschenkt

Es gibt viele Wunden, in die man bei Perl einen Finger legen kann und wenn man sich reinsteigert, könnte man zu der Auffassung gelangen die Perl-Community macht so ziemlich alles falsch was man nur falsch machen kann... wenn denn das *richtige* Ziel lauten sollte Perl populär zu machen.

Trotz all dessen ist Perl immer noch "Tier A" Programmiersprache und wird es auch bei gleichbleibender Situation noch eine Weile bleiben. Aber will man so dahindümpeln? Wenn nein, besteht Grund zur Hoffnung. Die Sprache hat eine reife Infrastruktur, Community, Technologie und Installationsbasis. Ok, vielleicht ist einiges davon überreif, aber eigentlich

steht Perl im heutigen Wettrennen der Aufmerksamkeits-Ökonomie der Programmiersprachen nicht unweit der Pole Position. Oder sagen wir ... es gibt schlimmere Ausgangssituationen. Die Maßnahmen, daraus Profit zu schlagen, sind fast trivial. Wahrscheinlich machen "wir für Perl" nicht die richtigen und wichtigen Sachen, weil sie uns zu dämlich^H^H^H^H^H^H nicht herausfordernd genug erscheinen.

Konkretismen

Das ist alles zu abstrakt? Natürlich werden wir im Vortrag konkrete Schritte besprechen wie:

• Wenn Perler lesen und schreiben können, warum ist dann perlfaq1 so sch... und wie kann man an diesem (und anderen) Beispielen konkret sehen wie "Newbies" verprellt werden?

• Apropos Newbies: Die Perl Community ist hermeneutisch und Sachen wie "Bring-a-Newbie" lediglich Alibismen. Was wirklich falsch läuft auf den Konferenzen, in der Dokumentation, im IRC, in Perl-Diskussionsforen und MLs.

• Wider das geistige Messietum! Qualitatives vs. quantitatives Wachstum. Die ultimative CPAN Kritik: CPAN mal mit anderen Augen sehen. Keine Sorge - nur kurz, dafür verheerend.

• "Euer Problem ist nicht Technologie. Das Problem seid Ihr." (Klaatu) - Ein wenig Selbstreflexion die vielleicht dem einen oder anderen Zuhörer vertraut sein möge. Konkret: Disziplin vs. Laissez-faire, Kooperation vs. Einzelkämpfer, Wettbewerb vs. Nebenläufigkeit, Böse vs. Gut, Das Imperium vs. die Rebellen, Realität vs. Märchen

• Warum in diesem speziellen Fall das "mach es einfach" nicht funktioniert.

• Konkrete non-tech TODO Vorschläge. Nicht sexy, aber effektiv. TIOBE SEO. Übersetzungen. Sprachspezifikation. Educate, don't advocate.

• Konkrete Fleiß-TODO Vorschläge. Auch nicht sexy, auch effektiv. Konverter, Konnektoren (APIs) - ein wenig die Klebkraft auffrischen. Janitoring.

• Konkrete tech TODO Vorschläge. Verdammt sexy, aber weniger effektiv da nicht "Music for the masses". pperl, bessere RegEx Engine (wo man vor der Implementierung nachdenkt), gpu-perl u.a.

Zusammenfassung

• Man kann Perl relativ einfach (wieder) zu einer sehr populären Sprache machen. Einfach, im Sinne von *"die zu unternehmenden Schritte sind kein mystisches Geheimwissen"*.

• Der Aufwand ist durchaus beträchtlich und kann nicht vom Einzelnen gestemmt werden. Auch die Community als solche müsste geschickt vorgehen und Hebelwirkungen (Multiplikatoren) ausnutzen. Also gilt: "mach es einfach" funktioniert nicht "lasst es **uns** machen" schon eher und "lasst uns erreichen, dass andere mitmachen" ist der Königsweg.

• Die wichtigsten Aufgaben mögen als die langweiligsten erscheinen, daher ist Disziplin so wichtig. Es wird kein Wunder geschehen, kein Messias wird kommen. Die ersten Multiplikatoren sind wir. Hier und Jetzt. Eigentlich schon gestern, aber da ist ja noch nix passiert.
eroMQ, AnyEvent & Perl

ZeroMQ (0mq) ist eine asynchrone messaging library, die es sehr einfach macht, verteilte Programme miteinander sprechen zu lassen. (Allerdings ist 0mq keine Job-Queue).

"We took a normal TCP socket, injected it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombarded it with 1950-era cosmic rays, and put it into the hands of a drug-addled comic book author with a badly-disguised fetish for bulging muscles clad in spandex. Yes, ØMQ sockets are the world-saving superheroes of the networking world." -- http://zguide.zeromq.org/page:all

Nach einer Einführung in 0mq Messaging Patterns (REQ-REP, PUSH-PULL, PUB-SUB) zeige ich, wie man mit den Perl bindings von Daisuke Maki (ZMQ::LibZMQ3) und AnyEvent 0mq in Perl verwendet. Danach zeige ich ein zur Zeit von uns (http://www.validad.com/) entwickeltes Module, dass ein perlishers Interface zu 0mq bieten soll: ZMQx::Class.

Thomas Klausner

ZeroMQ, AnyEvent & Perl

ZeroMQ (0mq) ist eine asynchrone messaging library, die es sehr einfach macht, verteilte Programme miteinander sprechen zu lassen. (Allerdings ist 0mq keine Job-Queue).

"We took a normal TCP socket, injected it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombarded it with 1950-era cosmic rays, and put it into the hands of a drug-addled comic book author with a badly-disguised fetish for bulging muscles clad in spandex. Yes, ØMQ sockets are the world-saving superheroes of the networking world." -- http://zguide.zeromq.org/page:all

Nach einer Einführung in 0mq Messaging Patterns (REQ-REP, PUSH-PULL, PUB-SUB) zeige ich, wie man mit den Perl bindings von Daisuke Maki (ZMQ::LibZMQ3) und AnyEvent 0mq in Perl verwendet. Danach zeige ich ein zur Zeit von uns (http://www.validad.com/) entwickeltes Module, dass ein perlishers Interface zu 0mq bieten soll: ZMQx::Class.

Falls das WLAN geht, wird es live demos geben. Wer daran teilnehmen mag, sollte sich die aktuelle Version (3.2) installieren: http://www.zeromq.org/intro:get-the-software

Renée Bäcker

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

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

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

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

Für die Erstellung der Erweiterungspakete liefert OTRS ein Skript mit.

Die Auslieferung an den Kunden kann man per Mail machen.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Also sieht der Ablauf jetzt wie folgt aus:

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

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

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

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

5. Schritt: Auslieferung an den Kunden

Git-Hooks

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

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

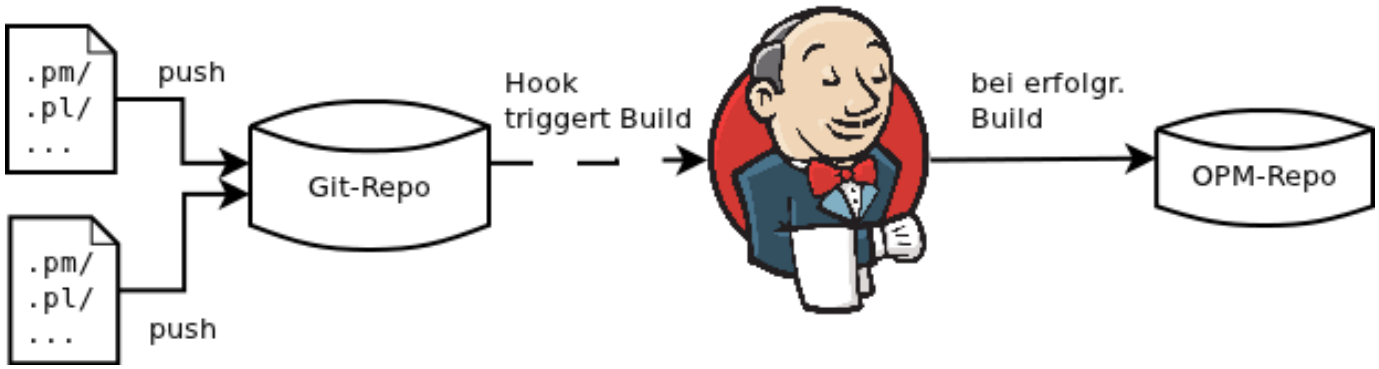


Abbildung 1

```
#!/usr/bin/perl

use strict;
use warnings;

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

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

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

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

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

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

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

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

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

Listing 1

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

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

Der aktuelle Git-Hook ist in Listing 1 zu finden.

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

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

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

ein

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

wird.

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

Jenkins

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

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

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

Konfiguration von Jenkins

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

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

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

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

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

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

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

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

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

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

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

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

Als drittes wird das Paket gebaut

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

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

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

OTRS::OPM::Maker

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

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

Listing 2

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

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

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

Das Programm *opmbuild* besteht nur aus wenigen Zeilen:

```
use strict;
use warnings;

use Getopt::Long;

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

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

Damit hat auch das Modul nur wenige Zeilen:

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

use App::Cmd::Setup -app;

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

our $VERSION = 0.05;

1;
```

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

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

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

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

- build
  opmbuild build ...

- filetest
  opmbuild filetest ...
- sopmtest
  opmbuild sopmtest ...

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

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

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

```
$ opmbuild
Available commands:

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

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

Die Methode sieht dann wie folgt aus:

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

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

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

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

Dafür ist die Methode C<usage_desc> zuständig

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

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

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

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

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

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

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

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

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

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

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

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

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

Auslieferung an den Kunden

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

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

Folgende Routen sind eingerichtet:

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

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

Die letzte URL liefert dann das Paket aus.

Lieferdienst "Mojolicious"

Da es nur wenige Zeilen, die für die Anwendung benötigt werden, wird nur `Mojolicious::Lite` verwendet. Der Quellcode sieht wie folgt aus:

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

plugin 'RenderFile';

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

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

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

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

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

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

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

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

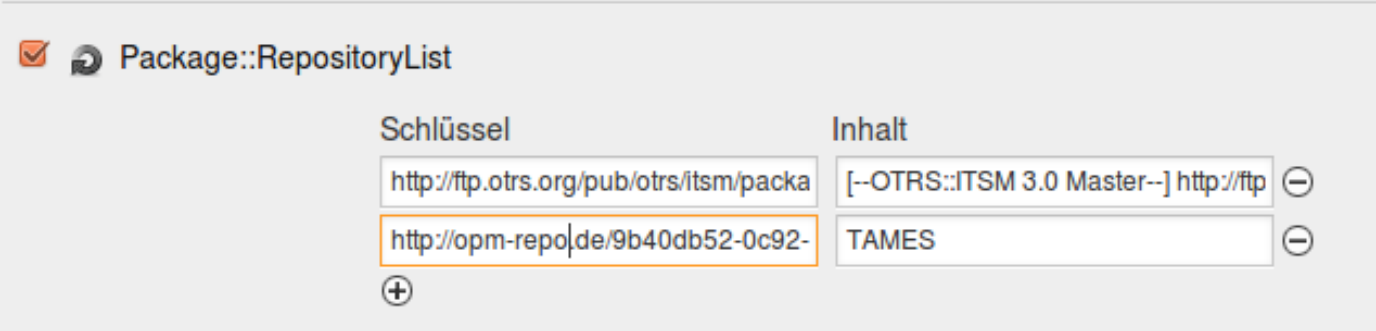


Abbildung 2: Online-Repository in der SysConfig eintragen

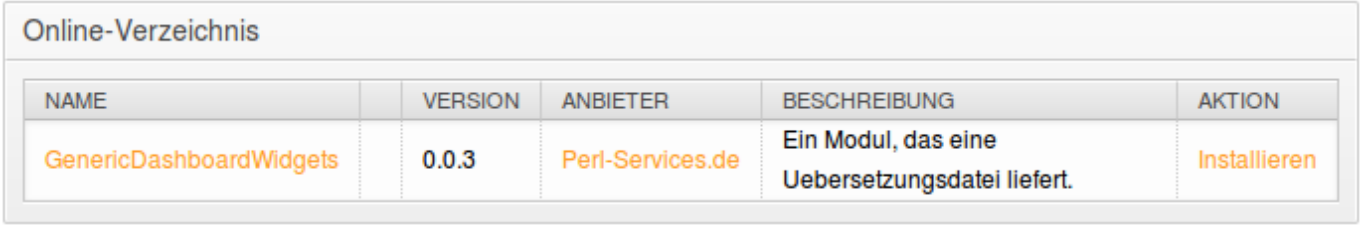


Abbildung 3: Pakete, die über den Paketmanager installiert werden können

Fazit und zukünftige Entwicklung

Mit Perl, Git und Jenkins lässt sich vieles im Release-Prozess vereinfachen und automatisieren. Diese Methode vermeidet viele Fehler und man kann dem Kunden eine bessere Qualität liefern. In diesem Szenario gibt es gerade mit den Möglichkeiten von Jenkins noch ein großes Verbesserungspotential und einiges ist auch schon in Planung. So ist geplant, aus Jenkins heraus VMs mit unterschiedlichen Szenarien (verschiedene OTRS-Versionen, unterschiedliche Datenbanksysteme, etc.) zu erstellen, zu starten und dann die Tests darauf auszuführen.

NOW HIRING!

Booking.com

World's **#1** Online Hotel Reservations Company
...and still growing!

Interested?
www.booking.com/jobs

We need:

Perl Developers, Software Developers, MySQL DBAs,
SysAdmins, Web Designers, Front End Developers

We use:

Perl, Puppet, Apache, MySQL, Memcache, Git, Linux
and much more...

Great location in the center of Amsterdam
Competitive Salary + Relocation Package
International, result driven,
fun & dynamic work environment

thinkproject!

Als Mitglied eines starken Teams
tragen Sie entscheidend
zum Unternehmenserfolg bei.

Junior/Senior Perl-Softwareentwickler (m/w)

Standort: Berlin

Hinter der Marke think project! steht eine expandierende, international erfolgreiche Unternehmensgruppe im Bereich internetbasierter Softwarelösungen. Zu unseren Kunden zählen namhafte Bauherren, Projektentwickler, Projektsteuerer, Planungsbüros und Bauunternehmen mit über 5.000 Projekten in mehr als 40 Ländern.

Ihr Verantwortungsbereich bei uns:
Sie entwickeln und refaktorisieren für unser Web-Collaboration-Produkt think project! Perl-Module in unserem service-orientierten Applikations-Framework. Zu ihren Aufgaben zählt es Trends im Bereich der Web-Technologien mitzubewerten und neue Technologien zu integrieren. 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.

Für unser anspruchsvolles Business-to-Business-Produkt nutzen wir:

- RedHat-Enterprise-Linux
- Apache2/mod_perl2
- PostgreSQL
- git

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 objektorientierter Perl-Entwicklung mitbringen;
- eine Ausbildung oder einschlägige Erfahrung im Bereich Informatik oder Programmierung haben;
- 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. Bei uns haben Sie 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 stellen Sie Ihre vollständige, aussagefähige Bewerbung per PDF-Datei online ein unter:
https://community.thinkproject.com/jobs_perl
Fragen zum Bewerberformular beantworten Ihnen gerne Frau Milanković/Herr Leitl, Tel.: +49 30 92 10 17-00.