Ausgabe 3/2014 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Messaging
mit RabbitMQ

SSL mit Perl
einfach aber sicher

Lisp in Perl
Erwachsen werden durch Interpreterbau

Nr 31

Swiss Perl Workshop 2014

Friday, 5. and Saturday, 6. September 2014

Venue: Flörli Olten

Language: English

Meet Perl hackers from Switzerland and other countries

•

Learn from and be inspired by talks

•

Chat and socialise during breaks and attendees dinner

Register today

•

Submit your talk

•

Become a Sponsor

www.perl-workshop.ch

Spiele für die Gemeinschaft

Vergnügen und Arbeit miteinander zu verbinden ist wirklich schön und wenn es in ein Spiel mit einem kleinen Wettkampf mündet dann ist das motivierend. Wer mit Perl programmiert und seinen Code nicht verstecken möchte hat vielfältige Möglichkeiten in einen kleinen Wettkampf mit anderen CPAN-Autoren zu treten. So führt Neil Bowers Buch darüber wer wie lange am Stück regelmäßig neue Module bzw. neue Versionen von Modulen auf CPAN lädt. Unter *http://neilb.org/cpan-regulars/* findet man die Listen. Dort kann man sehen, dass (Stand 26. Juni 2014) z.B. BARBIE seit 100 Tagen jeden Tag etwas auf CPAN lädt oder Dave Rolsky seit 162 Monaten jeden Monat.

Man kann auch versuchen unter die Top-10 der CPAN-Testers zu kommen... Dazu braucht man zwar eine Weile, aber wenn man mit dem CPAN-Testen anfängt kann man ganz schnell ein paar Plätze gut machen. Genau wie bei den CPAN-Regulars muss man gar nicht an die Spitze zu gelangen, aber ein Blick auf die Top-Leute hilft vielleicht den inneren Schweinehund zu überwinden und sich regelmäßig an die Aufgabe zu setzen.

Als CPAN-Autor kann man auch in das CPANTS-Spiel einsteigen. Unter *http://cpants.cpanauthors.org/* wird die "Kwalitee" eines Moduls berechnet. Es ist immer wieder schön wenn bei einem eigenen Modul wieder ein Feld mehr grün wird.

Mit Questhub.io (*https://questhub.io/welcome)* gibt es sogar ein Portal auf dem man sich seine eigenen Spiele/Aufgaben stellen kann. Man kann diese auch mit anderen teilen. Und wenn man seine Aufgaben erfüllt bzw. erledigt kann man von anderen Spielern auch Punkte bekommen. Eigene Spiele kann man auch anderen als Vorlage bereitstellen. Eine solche Vorlage ist es z.B. die "Changes"-Dateien von Modulen in ein bestimmtes Format zu bringen.

Bei den hier gezeigten Spielen geht es in erster Linie nicht darum "der/die Beste" zu sein, sondern der Gemeinschaft zu helfen. Wenn man neue Module auf CPAN lädt oder Distributionen testet, dann bringt man einen echten Mehrwert für die Community und macht die Perl-Welt noch besser.

Einen kleinen Spielvorschlag hätte ich noch: Versuchen für jede Ausgabe des Communitygetriebenen Perl-Magazins einen Artikel - egal welcher Länge - beisteuern. Siehe auch *http://bit.ly/1eeVfG7*

Jetzt aber viel Spaß bei der neuen Ausgabe von $foo.

Renée Bäcker

Die Codebeispiele dieser Ausgabe können mit dem Code

185lm6d

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

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue31

IMPRESSUM

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

INHALTSVERZEICHNIS

ALLGEMEINES

MODULE

NEWS

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

Renée Bäcker

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

Yanick Champoux

Mark Hofstetter

ist seit etwa 15 Jahren in der IT und hauptsächlich in Sachen Perl, Postgres, PHP und Oracle unterwegs. Hauptsächlich ist er bei der Universität Wien im Rahmen der Domainverwaltung tätig. Nebenbei arbeitet als selbständiger Entwickler und Trainer in und um Wien (http://www.trust-box.at) Wenn er gerade nicht programmiert, seine Kinder hütet, Stechmücken jagt (http://www.gelsenbekaempfung-leithaauen.at/), Spare Ribs testet (http://www.ripperl.at/) oder politisch tätig ist, kommt er tatsächlich ein paar mal im Jahr zu Paragleiten (wo man am Berg auch wieder nur ITler trifft)

Herbert Breunung

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

Wolfgang Kinkeldei

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

Thomas Klausner

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

Steffen Ullrich

Steffen Ullrich (SULLR) arbeitet seit 17 Jahren vorwiegend mit Perl. Seit 2001 ist er bei der genua mbh angestellt und beschäftigt sich dort mit der Erforschung und Entwicklung von Firewalls für Bereiche mit hohen Sicherheitsanforderungen.

Seit 2006 ist er Maintainer von IO::Socket::SSL und hat noch viele weitere Module auf CPAN, die sich überwiegend mit Netzwerkprotokollen und -sicherheit beschäftigen.

Mirko Weitermeier

Mirko Westermeier ist wissenschaftlicher Mitarbeiter an der WWU Münster und arbeitet dort unter Anderem für das Committee on European Computing Education. Er benutzt Perl für die viele kleinen und großen Programmieraufgaben im Alltag, unter anderem in der astronomischen Datenverarbeitung.

PerLisp auf Github: *https://github.com/memowe/perlisp*

Persönliche Website: <*http://mirko.westermeier.de*

Mark Hofstetter

Geo::Heatmap

Geographische (oder noch besser geographisch vernetzte) Daten sind einer der momentanen Hypes. Sehr viele Daten die anfallen sind - zumindest zusätzlich - geographisch zuordenbar. Jeder User der auf meine Website zugreift ist über seine IP Adresse – näherungsweise – lokalisierbar.

Jeder Kunde hat eine physische Wohnadresse, eine Handy App liefert Geodaten und so weiter und so fort ... die Datenmengen und -quellen können also schier unendlich sein.

Hat man eine überschaubare Zahl an Punkten so kann man sie meist direkt visualisieren, zum Beispiel mit einer der vielen Möglichkeiten, die die Google Maps API [*https://developers.google.com/maps/*] liefert.

Als überschaubar würde ich eine Größenordnung von bis zu etwa 1000 Punkten bezeichnen. Darüber hinaus wird die Übermittlung der Punkte über Javascript (entweder direkt oder über AJAX Calls oder ähnliches) zeitraubend und somit nicht mehr benutzerfreundlich.

Alternativ kann man mit graphischen Overlays arbeiten auf denen die Punkte unabhängig von der Anzahl dargestellt werden können. Die überlagerten Grafiken können vorab gerendert oder nur gecached sein, somit ist die Zahl der Punkte kein limitierender Faktor mehr.

Das Problem

Mehrere hunderttausend Punkte sollen in ihrer geographischen Dichte-Verteilung in Google Maps dargestellt werden. Konkret geht es um die Verteilung der Herkunft der (End)Kunden von nic.at [*http://www.nic.at*] der Domainvergabe für .at Domains.

Auf dem „Markt“ der freien Software war nichts zu finden das genau diese Anforderung erfüllt hätte, und somit habe ich mich entschlossen das Problem selbst anzugehen, mit den von mir bevorzugten Werkzeugen nämlich perl, postgres und postgis. Daraus resultierte das Paket Geo::Heatmap [*http://search.cpan.org/~hofstettm/Geo-Heatmap-0.18*].

Ein Live Demo ist unter *http://www.trust-box.at/dev/gm/GoogleMapsHeatmap/www/GoogleMapsHeatmap.html* zu finden.

Die Verwendung

Zuallererst müssen die Daten in einer Datenbank zur Verfügung stehen, die es erlaubt die Daten aus eine Längen- und Breitengradbox abzufragen, ich beschränke mich hier in meiner Beschreibung auf postgres/postgis.

Die einzelne Datenpunkte müssen geografisch verortet sein, und diese Information muss zum Beispiel in einer `geometry` Spalte gespeichert sein. Dann ist es nämlich möglich SQL Queries folgender Form zu machen:

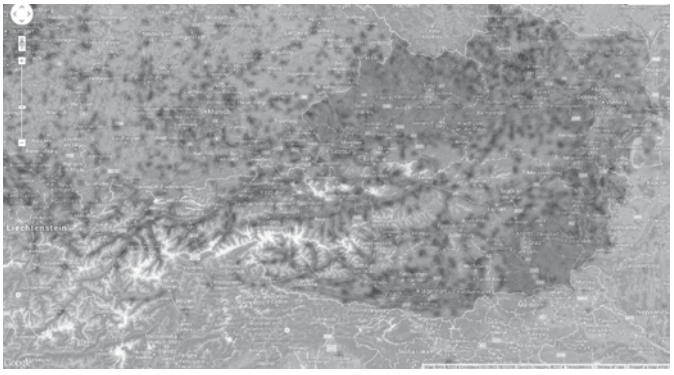


Abbildung 1: Screenshot der Live-Demo

```
select ST_AsEWKT(geom) from geodata
  where geom &&
    ST_SetSRID(
        ST_MakeBox2D(
            ST_Point($r->{LATN},
                     $r->{LNGW}),
            ST_Point($r->{LATS}, $r->{LNGE})
        ),4326))
```

um an die Datenpunkte für eine Google Map Tile zu gelangen. Es ist also nicht notwendig die Daten zu aggregieren, sondern man kann direkt mit den Einzeldaten arbeiten.

Die eigentliche Verwendung ist dann zweigeteilt, ein HTML Teil (siehe Listing 1).

Der in diesem Fall das fcgi Programm *hm.fcgi* mit den Google Maps Tile Koordianten – also x,y und den Zoomlevel - aufruft. Der Rest ist Google Map API Javascript Boilerplate Code. In dem Programm *hm.fcgi* wird dann die eigentliche „Arbeit“ getan:

Die Benutzung des Moduls ist recht einfach und nur ein Punkt ist heikel – doch dazu gleich `mehr`. Das Modul erwartet

• eine Funktionsreferenz (return_points), die aus den Google Tile Koordinaten eine Arrayref of Arrayrefs returniert

• eine Palette wie die Werte zu Farben konvertiert werden sollen

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
    <style type="text/css">
      html { height: 100% }
      body { height: 100%; margin: 0; padding: 0 }
      #map-canvas { height: 100% }
    </style>

    <script type="text/javascript"
        src="https://maps.googleapis.com/maps/api/js?key=<youapikey>&sensor=true">
    </script>

    <script type="text/javascript">
        var overlayMaps = [{
          getTileUrl: function(coord, zoom) {
            return "hm.fcgi?tile="+coord.x+"+"+coord.y+"+"+zoom;
          },
          tileSize: new google.maps.Size(256, 256),
          isPng: true,
          opacity: 0.4
        }];

        function initialize() {
          var mapOptions = {
            center: new google.maps.LatLng(48.2130, 16.375),
            zoom: 9
          };

          var map = new google.maps.Map(
            document.getElementById("map-canvas"),
            mapOptions
          );

          var overlayMap = new google.maps.ImageMapType(overlayMaps[0]);
          map.overlayMapTypes.setAt(0,overlayMap);
        }

        google.maps.event.addDomListener(window, 'load', initialize);
    </script>
  </head>
  <body>
    <div id="map-canvas"/>
  </body>
</html>
```

Listing 1

```
use FCGI;
use DBI;
use CHI;
use FindBin qw/$Bin/;
use lib "$Bin/../lib";

use Geo::Heatmap;

my $cache = CHI->new(
  driver => 'File',
  root_dir => '/tmp/domainmap'
);
our $dbh = DBI->connect(
  "dbi:Pg:dbname=gisdb",
  'gisdb', 'gisdb',
  {AutoCommit => 0},
);

my $request = FCGI::Request();

while ($request->Accept() >= 0) {
  my $env = $request->GetEnvironment();
  my $p   = $env->{'QUERY_STRING'};

  my ($tile) = ($p =~ /tile=(.+)/);
  $tile =~ s/\+/ /g;

  my $ghm = Geo::Heatmap->new();
  $ghm->palette('palette.store');
  $ghm->bin(8); # default value
  $ghm->zoom_scale( {
    1 => 298983,
    2 => 177127,
    ...
    17 => 2,
    18 => 0,
  } );

  $ghm->cache($cache);
  $ghm->return_points( \&get_points );

  my $image = $ghm->tile($tile);
  my $length = length($image);

  print "Content-type: image/png\n";
  print "Content-length: $length \n\n";
  binmode STDOUT;
  print $image;
}

sub get_points {
  my $r = shift;
  my $sth = $dbh->prepare(
    qq(select ST_AsEWKT(geom) from geodata
            where geom &&
              ST_SetSRID(ST_MakeBox2D(
                  ST_Point($r->{LATN},
                           $r->{LNGW}),
                  ST_Point($r->{LATS},
                           $r->{LNGE})
              ),4326))
  );
  $sth->execute();

  my @p;
  while (my @r = $sth->fetchrow) {
    my ($x, $y) = ($r[0] =
                ~/POINT\((.+?) (.+?)\)/);
    push (@p, [$x ,$y]);
  }

  $sth->finish;
  return \@p;
}
```

Listing 2

• einen Cache für die gerenderten Bilder – um genauer zu sein ein CHI Cacheobjekt [*http://search.cpan.org/~haarg/CHI-0.58/lib/CHI.pm*]. Der Cache wird zum Ablegen der berechneten Tile Grafiken verwendet bzw. zum Aufbau derselben.

• eine *bin* (ein divisor von 256), der die Zahl der Felder pro Achse angibt in die ein Google Tile geteilt werden soll – defaultmäßig 8, ein Wert der hübsche Resultate produziert.

• und das bei weitestem komplexeste: einen Hash (zoomscale) der die maximalen Zahl von Werten in den jeweiligen Zoomstufen enthält.

Die Skalierung kann leider nicht dynamisch erfolgen, weil sie ja konstant innerhalb jeder Zoomstufe sein muss - unabhängig vom Ort den man gerade betrachtet. Somit hat das Modul keine Chance zu wissen wie viele Daten es maximal pro Tile gibt, ergo muss man vorher feststellen was die maximale Dichte seiner Daten in einer bestimmen Zoomstufe ist. An dieser Stelle muss man etwas postgis Magie verwenden. Ich habe der Distribution ein Skript beigefügt welches einem diese Aufgabe leichter macht (max_points_per_tile_zoom.pl) welches über der postgis Funktion `ST_GeoHash` die Tiles mit den meisten Daten herausfindet – und somit die Eingabewerte für *zoomscale*.

```
with geohash as (
  select ST_GeoHash(geom::geometry, 5)
                                  st_geohash,
    geom
  from geodata where
    not(
        St_X(geom) < -180 or
        St_Y(geom)< -90 or
        St_X(geom) > 180 or
        St_Y(geom) > 90
    )
)
SELECT ST_Extent(geom) as extent
  FROM geohash
  where st_geohash in
    (select st_geohash from
      (select st_geohash, count(*) c
        from geohash
        group by st_geohash
        order by c desc limit 2
      ) max_geohash
    )
  )
```

Wie oft man die Skala neu berechnen muss hängt natürlich von der Dynamik der Änderung der Daten ab. Die Daten werden logaritmisch skaliert auf eine 500 Farben umfassende Palette (siehe Abbildung 2).

Unter der Haube

Wie werden aus einzelnen Punkten nun Dichtekarten/Heatmaps? Wenn wir ein Google Maps Overlay erstellen wollen, bekommen wir von der Google API die Information welches Tile gerade angezeigt wird [*http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/*.

Sieht man sich zum Beispiel Wien in der Zoomstufe 7 an (da geht sich Österreich knapp ganz aus) so liegt Wien in dem Tile 139,88,7 (x,y,zoom). Diese Google Tile Koordinaten müssen nun in echte geographische Koordinaten umgerechnet werden. Dies geschieht mit einem von John D. Coryat geborgten Codeteil (USNaviguide_Google_Tiles).

Damit werden aus den obigen „Google-Koordinaten“ folgende echte Koordinaten errechnet:

```
'LATN' => '48.458352',
'LATS' => '47.991760',
'LNGW' => '14.062500',
'LNGE' => '14.762878'
```

(siehe Abbildung 3)

Mit diesen Koordinaten kann man nun in die Datenbank gehen und sich alles Punkte innerhalb dieses Tiles holen (siehe erstes SQL Statement). Der Tile wird dann nochmals durch *bin*geteilt und je nach der Zahl der Punkte(skaliert mit der Zoomscale) in dem jeweiligen*Korb* wird aus der palette ein Farbwert geholt.

Mit diesem Farbwert wird nun ein Quadrat in eine png Grafik gezeichnet (mittels Imager [*http://imager.perl.org/*]) . Wollte man harte Kästchen als Heatmap, dann wäre man an dieser Stelle fertig. Da wir aber hier noch „bluren“, d.h. wir wollen die harten Quadrate verwischen.

Damit man die Grafik übergangslos bluren kann braucht man noch alle acht benachbarten Tiles (in Wirklichkeit bräuchte man natürlich nur einen schmalen Rand).

Wird also ein Tile benötigt wird das Tile selber und seine acht Nachbarn erstmals berechnet und in einem Cache (CHI-Objekt) abgelegt. Nach dem bluren wird das eigentlich benötigte Tile aus dem 3x3 Quadrat wieder ausgeschnitten und in einem separaten Teil des Cache‘ abgelegt.

Wird im nächsten Schritt das Nachbartile angefragt, so müssen nur noch drei Tiles neu gerechnet werden und danach die Bluroperation durchgeführt werden.

Wird ein schon einmal gerechnetes Tile abgefragt kommt es natürlich gleich komplett aus dem Cache.

Möchte man also das Modul dazu bringen die Overlay Grafiken neu zu rechnen muss man den Cache entleeren. Es ist eigentlich nur sinnvoll den Cache komplett zu entleeren, da es andernfalls zu optischen Anschlussfehlern zwischen den Tiles kommen kann.

Der erste User, der einen bestimmten Kartenabschnitt abfragt, muss unter Umständen mit längeren Wartezeiten leben, da die Tiles ja nur on demand gerechnet werden. Möchte man auch das vermeiden, so kann man zum Beispiel aus seinem Cache oder seinem Webserverlog die am häufigsten abgefragten Kartenabschnitte auslesen und nach einem Cachereset genaus diese Abschnitte gleich erneut berechnen lassen.

Abbildung 2

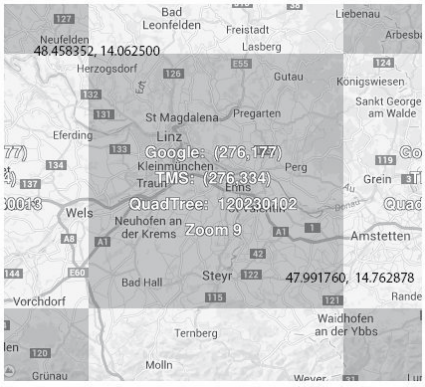


Abbildung 3

Fazit

Mit einigen wenigen perl Zeilen lässt sich ein Visualisierungslayer bauen, der gegenüber diversen Javascript Lösungen einen enormen Vorteil hat: Er ist unabhängig von der Zahl der zugrundeliegenden Datenpunkte, da nur fertige Grafiken an den Browser ausgeliefert werden.

Zusätzlich bietet das Modul ein Fundament für andere grafische Präsentationsformen von geografischen die mit Hilfe von Perl!

Über Verwendungsberichte beziehungsweise Verbesserungsvorschläge würde ich mich sehr freuen!

Wolfgang Kinkeldei

Messaging mit RabbitMQ

Wozu Messaging?

Sobald Anwendungen und Systeme beginnen zu wachsen, verteilt oder entkoppelt werden müssen, entstehen Notwendigkeiten zur Kommunikation zwischen den Teilsystemen. Erschwert werden solche Vorhaben dann, wenn verschiedene Systeme oder Programmiersprachen im Einsatz sind oder zusätzliche Skalierungs- oder Qualitäts-Anforderungen im Spiel sind oder gar zeitaufwändige oder Ressourcen schluckende Vorgänge an das Auftreten bestimmter Ereignisse gebunden werden sollen.

An dieser Stelle sind Messaging-Systeme einsetzbar, die den kommunikativen Teil der Arbeit erledigen. Beim Nachrichten-Austausch kann man grob verschiedene Muster der Kommunikation unterscheiden:

Einrichtungs-Kommunikation (One Way)
Das einfachste Muster verbindet einen Sender und einen direkt bekannten Empfänger. Jede Nachricht des Senders landet direkt beim Empfänger. Es findet keine Quittierung statt. Mögliche Anwendungen sind zum Beispiel das Erzeugen von Log-Einträgen, Mailversand oder die Erzeugung von Mitteilungen an einen Benutzer.

Veröffentlichen und Abonnieren (Publish-Subscribe)
Bei diesem Muster der Kommunikation stellt typischerweise ein Sender eine Nachricht bereit, die für eine nicht bekannte aber möglicherweise große Anzahl von Empfängern gedacht ist. Die Empfänger erhalten die Nachricht, reagieren entsprechend, quittieren den Erhalt der Nachricht aber nicht. Zum Einsatz bringen könnte man dieses Muster zum Beispiel zur Benachrichtigung aller Benutzer eines Systems oder der Aufforderung mehrerer Systeme zur erneuten Synchronisierung oder sonstiger Aktivitäten.

Anfrage und Antwort (Request-Reply)
Hier sendet ein Partner genau einem Gegenüber eine Nachricht und wartet auf den Erhalt einer entsprechenden Antwort. Gerne wird dieses Muster auch als Entfernter Methodenaufruf (Remote Procedure Call) bezeichnet. Die Abfrage von in entfernten Systemen gespeicherten Informationen lassen sich mit diesem Muster gut bewerkstelligen. Eine mögliche Anwendung wäre der Login-Vorgang eines Benutzers oder die Abfrage von Benutzer-Daten zu einer Person bei einem dafür vorgesehenen Server.

Warteschlangen (Work Queues)
Dieses Standard-Muster erlaubt einem Absender eine Nachricht in einer Warteschlange abzulegen. Ein oder mehrere Arbeits-Stationen erhalten jeweils eine Nachricht aus der Warteschlange und bearbeiten diese. Dieses Muster ist zum Beispiel zur Berechnung von Thumbnails, der Aufbereitung von Datenauslieferungen als ZIP Archive oder der Erstellung von Druck-Dokumenten geeignet. Durch die Hinzunahme weiterer Bearbeitungsstationen ist eine schnelle Skalierung möglich.

Sehr oft finden sich auch Mischformen aus diesen Mustern, z.B. Warteschlangen mit Versendung einer Rückantwort. Oder die Muster werden um zusätzliche Eigenschaften erweitert, z.B. Abonnieren mit Filterung um ausschließlich für einen Kommunikationspartner interessante Nachrichten zu erhalten. Aber im Wesentlichen sind diese Muster immer erkennbar.

Die Qual der Wahl

Sucht man nach Messaging Systemen, wird man sehr schnell fündig. Sowohl quelloffene als auch kommerzielle Systeme mit den unterschiedlichsten Merkmalen sind verfügbar und erschweren die Entscheidung für das die Anforderungen am besten erfüllende System. Vor einiger Zeit standen wir vor der gleichen Entscheidung und haben uns vier vorab ausgewählte Systeme etwas genauer angesehen. Notwendiges Kriterium war die Ansteuerbarkeit von Perl aus. Wünschenswert waren zusätzliche Bindings für andere Programmiersprachen sowie die Möglichkeit Client und Server unter OS-X, Linux und Windows betreiben zu können. Zum Vergleich traten bei uns an:

ØMQ - eigentlich nur eine Bibliothek, allerdings mit großem Umfang

Gearman - der Klassiker für Perl Entwickler wenn es um Warteschlangen geht

Resque - auffällig durch einfache Installation und ein nettes Web Frontend

RabbitMQ - der etwas schwergewichtigere Alleskönner für hohe Ansprüche

ØMQ näher betrachtet

Bei ZeroMQ (kurz ØMQ) handelt es sich eigentlich nur um eine Bibliothek. Diese ist in C geschrieben und relativ klein. Das zentrale Element bei ØMQ ist ein Socket, der vergleichbar mit denen aus der Netzwerk-Programmierung sind. Unterschied ist die Fehlerbehandlung sowie eine einfache darin verborgene Warteschlange, sowie verschiedene Einstellmöglichkeiten, die Kommunikation nach den oben beschriebenen Mustern ermöglichen, wenn man Sender- und Empfänger-seitig jeweils bestimmte Paarungen von Socket-Typen verwendet. Damit hat man mit ØMQ einen Baukasten in der Hand, der das Zusammenstecken geeigneter Kommunikations-Mechanismen dezentral ermöglicht.

Daher gibt es auch kein Web Frontend, denn das würde eine zentrale Steuerung und Verwaltung erfordern.

Gearman -- der Klassiker

Gearman -- ursprünglich in Perl geschrieben (inzwischen sind Teile davon in C), ist ein zentraler Server, der als Ablageort für Aufträge in Warteschlangen dient.

Typischerweise legt ein Klient einen Auftrag auf dem Gearman Job Server ab. Ein oder mehrere Arbeitsstationen registrieren sich beim Job Server mit den Fähigkeiten, bestimmte Jobs abarbeiten zu können. Entsprechend ihrer Fähigkeiten werden Aufträge dann an einzelne Stationen verteilt. Ein solcher besteht typischerweise aus einem eindeutigen Namen sowie Parametern.

Standardgemäß gibt es kein Web Frontend für Gearman, es genießt aber einen guten Ruf.

Resque

Eher zufällig sind wir auf Resque gestoßen, weil ein netter Mensch eine Perl Adaption dafür geschrieben hat. Resque ist sehr leichtgewichtig und besteht eigentlich nur aus einer Bibliothek (wahlweise für Perl oder Ruby), mit der die zentral in einer Redis-Datenbank gespeicherten Warteschlangen bedient werden.

Durch das schöne Web Frontend hat man eventuell hängende Arbeits-Rechner schnell isoliert und behält einen Überblick über alle offenen Aufgaben. Das Web-Frontend allerdings ist nur für Ruby verfügbar und erfordert etwas Einarbeitung in die Ruby-Werkzeugkiste.

RabbitMQ

Hinter RabbitMQ steckt ein komplexeres Gebilde, das großen Wert auf Skalierbarkeit, Hochverfügbarkeit und Sicherheit legt. So lässt sich die zentral verwaltete Nachrichtenzentrale problemlos auf mehrere Server verteilen und ist damit redundant ausgelegt. Durch zahlreiche Plugins lassen sich Erweiterungen aktivieren, die sogar den Zugriff von Web Browsern (mit entsprechender JavaScript Bibliothek) auf den RabbitMQ Server erlauben. Anbindungen existieren für alle gängigen Programmiersprachen.

Die Konfiguration kann wahlweise programmatisch, via Kommandozeile oder über die Web-Oberfläche vorgenommen werden. Diese ist übersichtlich und zeigt alle derzeit laufenden Vorgänge sowie beteiligten Kommunikationspartner an.

Wir haben uns dafür entschieden, unser nächstes Projekt mit RabbitMQ anzugehen. Nachrichten werden hierbei aus einer Web Abwendung ausgelöst, auf einem RabbitMQ Server unter Linux verwaltet und auf einer OS-X Maschine mit diversen Programmen von Adobe und viel Perl aus den Daten der Nachrichten dann druckfertige PDF Dokumente erzeugt.

RabbitMQ installieren

Die Installation von RabbitMQ ist extrem einfach. Für Windows existieren auf der Homepage des Projekts herunterladbare Installer, die RabbitMQ als Windows Service installieren. Für die meisten Linux Distributionen existieren fertige Pakete und für OS-X stehen durch homebrew oder MacPorts ebenso einfach zu installierende Versionen bereit. Länger als 15 Minuten dauert die Installation typischerweise nicht.

Nach der Installation sollte man einmalig das `Management`-Plugin aktivieren, was mit dem Kommando `rabbitmq-plugins` ausgeführt wird.

```
$ rabbitmq-plugins enable rabbitmq_management

The following plugins have been enabled:
  rabbitmq_management_agent
  rabbitmq_management
Plugin configuration has changed.
Restart RabbitMQ for changes to take effect.
```

Und damit kann es losgehen. Über *http://localhost:15672/* ist die Administrations-Oberfläche des eben installierten RabbitMQ Servers erreichbar und dient fortan als Überwachungs-Zentrale oder zur händischen Konfiguration.

Umgang mit RabbitMQ

Bevor man mit RabbitMQ los legen kann, muss man sich mit zahlreichen Begriffen anfreunden. Die wichtigsten sind:

AMQP
Das Advanced Message Queueing Protocol ist ein Binär-Protokoll, welches von RabbitMQ zur Kommunikation mit allen Beteiligten verwendet wird. Dank seiner binären Natur sind Bibliotheken, die dieses Protokoll nachbilden leider nicht ganz einfach zu verstehen und der Datentransfer über die Netzwerkleitung wenig aussagefähig. Dafür ist der Netzwerk-Transfer sehr kompakt.

Exchange
Sollen Nachrichten an einen RabbitMQ Server gesandt werden, müssen diese bei einem Exchange eingeliefert werden. Ein Exchange trägt einen sprechenden Namen und besitzt neben diversen Parametern einen Typ, der später Einfluss auf die Verarbeitung der Nachricht hat. Je nach Typ wird ein mit einer Nachricht wahlweise angegebener Routing-Schlüssel oder einer mehrerer optionaler Header für die Zustellungs-Entscheidung benutzt oder ignoriert.

Queue
Bei einem Exchange eingelieferte Nachrichten werden abhängig vom Typ des Exchange sowie Routing-Regeln an eine oder mehrere Queues geliefert. Eine in einer Queue liegende Nachricht kann nur von einem Empfänger gelesen werden. Verbinden sich also mehrere Empfänger mit derselben Queue, erhalten sie jeweils unterschiedliche Nachrichten. Einzige Ausnahme ist, wenn die Bearbeitung einer Nachricht nicht abgeschlossen werden konnte, dann verteilt RabbitMQ dieselbe Nachricht nochmals an einen anderen Empfänger.

Binding
Sowohl vom Typ des Exchange als auch von Routing-Parametern anhängig wird festgelegt, welche Nachricht(en) in welche Queue(s) gelangen. Der Mechanismus dazu wird bei RabbitMQ als Binding bezeichnet und ist sehr vielseitig konfigurierbar.

Message
Eine Nachricht besteht aus einer beliebigen Anzahl an Bytes. RabbitMQ interessiert sich nicht für den Inhalt einer Nachricht. Zusätzlich kann ein Routing-Schlüssel z.B. `website.render.thumbnail` mit angegeben oder beliebige Header beigefügt werden. Abhängig vom Typ des Exchange und den Einstellungen im Binding wird eine Nachricht an entsprechende Queues geleitet.

Wo und wie die Konfiguration der Nachrichtenverarbeitung festgelegt wird, ist Geschmacksache. Neben einer händischen Anlage der diversen Komponenten über die Web Oberfläche können Kommandozeilen-Werkzeuge zur Konfiguration herhalten oder eine als JSON Datei vorbereitete (bzw. ausgelesene) Konfiguration eingespielt werden. Ebenfalls denkbar ist die Erweiterung der Produzenten und Konsumenten um die Erzeugung der notwendigen Bestandteile.

Die folgenden Programmbeispiele konfigurieren jeweils den Teil der Nachrichtenverarbeitung, der für den jeweiligen Kommunikationspartner von Interesse sind. Der Sender ist für die Erstellung des Exchange zuständig, der Empfänger kümmert sich um die Queue und das Binding.

RabbitMQ und CPAN

Auf CPAN ist eine ganze Reihe von Distributionen zu finden, die es erlauben, mit RabbitMQ zu sprechen. Die Lager teilen sich grob in Distributionen, die auf einem der verfügbaren Event Loops aufsetzen und nicht ereignisgesteuerte konventionell programmierte Bibliotheken. Für den Anfang wollen wir uns einmal auf die konventionellen Technologien beschränken.

Bei diversen Versuchen habe ich nachfolgende Distributionen ausprobiert:

Net::Thumper
Von der API her relativ einfach (leider manchmal auch eingeschränkt) ist `Net::Thumper`. Es ist pure-Perl, erfordert keine der asynchronen Event-Loops und läuft problemlos. Einzige Hürde ist, dass der Autor auf `Net::AMQP` zurückgreift und dafür die Lage einer xml-Datei im Dateisystem zur Instanziierung des Moduls als Parameter erforderlich ist. Grund dafür ist, dass Net::AMQP anhand dieser XML-Datei alle für die Kommunikation relevanten Parameter für AMQP ausliest und damit die Datenpakete für den Netzwerk-Transfer erzeugen oder dekodieren kann. Damit ist in jedem Fall sicher gestellt, dass AMQP korrekt implementiert wird, denn es werden keinerlei hart codierte magischen Werte in die Quelltexte gepackt, sondern aus der offiziellen XML-Datei gelesen (*http://www.rabbitmq.com/resources/specs/amqp0-9-1.xml*).

Einschränkend muss ich allerdings darauf hinweisen, dass Net::Thumper nicht mit virtuellen Hosts auf RabbitMQ Servern umgehen kann, der virtuelle Host `< "/" >` ist leider hart codiert. Insofern wird es für größere praktische Einsätze vermutlich nicht ausreichend sein, will man die Quelltexte unverändert lassen.

Net::RabbitMQ
`Net::RabbitMQ` ist zum Teil in C programmiert, implementiert AMQP ohne die Protokollbeschreibung in Form der XML-Datei zu verarbeiten. Es ist etwas umfangreicher vom Funktionsumfang.

Net::AMQP::RabbitMQ
Auf der Basis von Net::RabbitMQ aufgebaut und erweitert ist `Net::AMQP::RabbitMQ`. Der Autor gibt als Grund die Verweigerung der Annahme von Erweiterungen an.

Erste Gehversuche

Ein einfacher Produzent, der lediglich Textnachrichten versenden soll, ist relativ schnell zusammengebaut. Praxistauglich ist dieses Vorgehen natürlich nicht, doch der Weg vom kurzen Text zu JSON serialisierten Objekten ist nicht weit. `MooseX::Storage` lässt grüßen!

Für die nachfolgenden Beispiele nutze ich `Net::AMQP::RabbitMQ`. Der Verständlichkeit wegen sei noch erwähnt, dass AMQP sogenannte Channels dazu verwendet, über eine TCP-Verbindung via Multiplexing mehrere logische Kanäle zwischen Klient und Server abzubilden. Solange man nicht mehrere logische Verbindungen zu einem Server unterhalten muss, braucht man sich keine Gedanken darüber zu machen.

```
#!/usr/bin/env perl
# Producer
use strict;
use warnings;
use Net::AMQP::RabbitMQ;

our $CHANNEL = 1;

# Verbindungsaufbau
my $mq = Net::AMQP::RabbitMQ->new;
$mq->connect('localhost', {});
$mq->channel_open($CHANNEL);

# Einrichtung eines "Posteingangs"
$mq->exchange_declare($CHANNEL,'render',
    { auto_delete => 0 });

# Versand einer Nachricht
$mq->publish($CHANNEL, '', 'Hello rabbit',
    { exchange => 'render' });
```

Allerdings haben wir noch keine Queue und kein Binding zu dem eben definierten Exchange erzeugt. Jeder Aufruf des obigen Scripts erzeugt und verwirft sofort die Nachricht,

da RabbitMQ keine Regeln kennt, wie mit der Nachricht zu verfahren ist. Erst wenn wir unser zweites Programm erzeugen und starten, werden die notwendigen Einstellungen im RabbitMQ Server vorgenommen und keine Nachrichten gehen mehr verloren -- selbst wenn sie nicht abgeholt werden. Starten wir das Programm ein zweites Mal können wir sogar schön die Verteilung der Nachrichten auf unsere zwei Terminal-Fenster mitverfolgen.

```
#!/usr/bin/env perl
# Consumer
use 5.010;
use strict;
use warnings;
use Net::AMQP::RabbitMQ;

our $CHANNEL = 1;

# Verbindungsaufbau
my $mq = Net::AMQP::RabbitMQ->new;
$mq->connect('localhost', {});
$mq->channel_open($CHANNEL);

# Einrichtung "Warteschlange"
$mq->queue_declare($CHANNEL, 'render',
       { durable => 0, auto_delete => 0});
$mq->queue_bind($CHANNEL, 'render' =>
'render', '');

# Bearbeitungs-Schleife
say 'waiting for messages...';
$mq->consume($CHANNEL, 'render');
while (1) {
    my $message = $mq->recv();
    say "Received: $message->{body}";
}
```

Notwendige Verbesserungen

Damit aus den bisherigen Beispiel-Programmen praxistaugliche Lösungen werden, sind noch ein paar kleine Erweiterungen vorzunehmen.

Verbindungsmanagement
Lang laufende Netzwerk-Verbindungen bergen stets das Risiko, durch Störungen, eine Firewall zwischen den Kommunikations-Partnern oder gar den Server beendet zu werden. Insofern müssen lang laufende Prozesse für stabile Verbindungen zum Server sorgen und abgebrochene Verbindungen gegebenenfalls wieder hergestellt werden. Zusätzlich kann man den Heartbeat-Mechanismus, den RabbitMQ bietet, ebenso einsetzen, um die Wahrscheinlichkeit für Verbindungsabbrüche zu reduzieren.

Quittierung
Speziell bei Nachrichten, die eine Bearbeitung nach sich ziehen, sollte man über auftretende Fehler wärend der Bearbeitung nachdenken. Eine Queue kann so eingestellt werden, dass eine Nachricht quittiert werden muss und erst dann als erfolgreich bearbeitet betrachtet wird.

Vermittlung
Abhängig von Quelle und Ziel der auszutauschenden Nachrichten muss man sich Gedanken über die zu erstellenden Exchanges, Queues sowie das Routing Regelwerk machen. Da die Anforderungen jedes Projektes unterschiedlich sind, gibt es keine generelle Vorgehensweise. Als Anfang kann vermutlich dienen, von einer Exchange ausgehend, Nachrichten über Routing-Schlüssel oder Header gezielt an Queues zu verteilen. Je nach auftretenden Limitierungen oder Sonderwünschen lassen sich dann gezielt Erweiterungen vornehmen.

Überwachung
Zwar steht RabbitMQ unter dem Stern, besonders stabil zu sein, Kontrolle ist dennoch besser als reines Vertrauen. Auf der Projekt-Seite von RabbitMQ finden sich Links zu einem Nagios-Plugin für RabbitMQ. Hierbei lassen sich diverse Metriken abfragen, die Aussagen zum Gesundheitszustand zulassen.

Ausfallszenario
Da ein Messaging-System zur Entkoppelung eingesetzt wird, können relevante Vorgänge vermutlich nicht ohne erfolgen. Je nach Bedeutung für einzelne Teile des Systems muss man sich also geeignete Fehler- oder Mitteilungs-Szenarien einfallen lassen.

Wir haben bei unserem z.Z. entstehenden Projekt eine einfache Bibliothek im Einsatz, die auf obigen Mustern basiert und den Nachrichtenversand zum RabbitMQ Server sowie die Verarbeitungslogik auf Konsumenten-Seite kapselt. Auf diese Weise gestaltet sich der Nachrichtenversand relativ einfach, indem lediglich eine Methode aufzurufen ist. Sämtliche Implementierungsdetails bleiben dem Anwendungs-Entwickler verborgen.

Falls jemand ebenfalls Erfahrungen diesbezüglich gesammelt hat, wäre ein weiterer Artikel oder ein Blog-Eintrag herzlich willkommen!

MODULE

Renée Bäcker

Unicode, Sortierung und mehr...

Mit dem Thema Unicode, Zeichensätze etc. kann man ganze Bücher füllen und wenn man der Kinderstube entwächst muss man sich mit dem Thema "Sprache" wieder neu befassen. Spätestens dann wenn man Software entwickeln möchte. Denn dann fällt einem auf, dass es in anderen Sprachen ganz andere Zeichen gibt und dass man Zeichen ganz unterschiedlich darstellen kann.

Das fängt schon bei der Wahl der Zeichenkodierung an. Da werden für die Zeichen unterschiedlich viele Bytes benötigt (siehe Listing 1).

Wenn man da nicht aufpasst, bekommt man schon bei einfachsten Perl-Programmen Probleme:

```
$ perl -E 'say length "Ä"'
2
```

Möchte man in diesem Fall sagen, dass der Perl-Quelltext UTF-8 enthält, muss man `use utf8;` verwenden:

```
$ perl -Mutf8 -E 'say length "Ä"'
1
```

Aber das soll hier gar nicht Thema dieses Artikels sein, denn darüber hat Moritz Lenz in der Ausgabe 5 von $foo schon einen sehr guten Artikel geschrieben. In diesem Artikel soll es um andere Probleme gehen.

Da ist zum einen das Problem, dass *é* nicht unbedingt gleich *é* ist. Da hilft jetzt auch das Putzen der Brille nicht, denn hier auf Papier sehen die beiden Zeichen identisch aus. Das Zeichen kann aber einmal ein

```
é: U+00E9 (LATIN SMALL LETTER E WITH ACUTE)
```

oder ein

```
e + ´: U+0065 + U+0301
     (LATIN SMALL LETTER E und
      COMBINING ACUTE ACCENT)
```

sein.

```
$ perl -E 'binmode STDOUT, ":utf8"; \
    say "\x{0065}\x{0301}"'
e´
$ perl -E 'binmode STDOUT, ":utf8"; \
    say "\x{00e9}"'
é
$ perl -E '
    say "\x{00e9}" eq "\x{0065}\x{0301}"
        ? 1
        : 0'
0
```

Und nicht immer hat man Einfluss darauf, wie die Daten aus der Quelle kommen. In so einem Fall muss man sich um eines kümmern: Normalisation. Der Unicode-Standard definiert verschiedene Normalisationsformen, die zwei, die am häufigsten verwendet werden, sind *Composed* und *Decomposed*.

Die zusammengesetzte Form (*Composed*) von é ist also das *U+00E9* und die getrennte Form (*Decomposed*) ist das *U+0065 + U+0301*.

Kann man sich nicht sicher sein, in welcher Form der String auftaucht, muss man für den Vergleich normalisieren. Dafür gibt es `Unicode::Normalize`, das unter anderem die Methoden `NFC` und `NFD` bereitstellt. Wie man vielleicht erahnen kann, liefert `NFC` die *Composed*-Form und `NFD` die

```
Codepoint | Zeichen | ASCII | UTF-8          | Latin-1 | ISO-8859-15 | UTF-16
U+0041    | A       | 0x41  | 0x41           | 0x41    | 0x41        | 0x00 0x41
U+00c4    | Ä       |  -    | 0xc4 0x84      | 0xc4    | 0xc4        | 0x00 0xc4
U+20AC    | €       |  -    | 0xe3 0x82 0xac | -       | 0xa4        | 0x20 0xac
U+c218    | 수      | -     | 0xec 0x88 0x98 | -       | -           | 0xc2 0x18
```

Listing 1

Decomposed-Form. Für den Vergleich müssen nur beide Varianten in die gleiche Form gebracht werden und schon funktioniert es:

```
$ perl -MUnicode::Normalize -E '
    say NFD("\x{00e9}") eq
        NFD("\x{0065}\x{0301}")
      ? 1
      : 0'
1
```

Wunderbar, Problem gelöst. Und schon stoßen wir auf das nächste Problem: Die Sortierung. Auch wenn unterschiedliche Sprachen die gleichen Zeichen verwenden heißt das noch lange nicht, dass die Reihenfolge der Zeichen gleich ist. Gehen Sie mal zu Personen aus Deutschland und Schweden und lassen Sie mal die zwei Worte *zorn* und *örtchen* sortieren. Sie werden überrascht sein, denn die schwedische Person wird - anders als die deutsche Person - das *ö* nach dem l<z> einsortieren.

Wenn wir also eine Anwendung haben, die von Benutzern unterschiedlichster Herkunft verwendet wird und Strings irgendwie sortiert dargestellt werden müssen, dann müssen Sie solche Regeln beachten. Solche Sortierregeln nennt man Collation. Dazu gibt es die einen Algorithmus, den *Unicode Collation Algorithm*, der in dem Perl-Modul `Unicode::Collate` umgesetzt ist. Und für die Umsetzung der Sprachspezifischen Sortierregeln gibt es `Unicode::Collate::Locale`.

```
use strict;
use warnings;
use Unicode::Collate::Locale;

my @letters = qw(z ä);
my @reversed = reverse @letters;

my $german = Unicode::Collate::Locale
                 ->new(locale => 'de_DE');
my $swedish = Unicode::Collate::Locale
                 ->new(locale => 'sv_SE');

for my $letters (\@letters, \@reversed) {
  print "Original: @$letters\n";

  my @german = $german->sort(@$letters);
  print "German: @german\n";

  my @swedish = $swedish->sort(@$letters);
  pring "Swedish: @swedish\n\n";
}
```

Das die Ausgabe

```
Original: z ä
German: ä z
Swedish: z ä

Original: ä z
German: ä z
Swedish: z ä
```

bringt.

Die Verwendung des Moduls hat aber einen Nachteil: Es ist relativ langsam. Eine etwas schnellere Variante ist `Unicode::ICU::Collator`:

```
use strict;
use warnings;
use utf8;
use Unicode::ICU::Collator;

binmode STDOUT, ":utf8";

my @letters = qw(z ä);
my @reversed = reverse @letters;

my $german = Unicode::ICU::Collator
                 ->new('de_DE');
my $swedish = Unicode::ICU::Collator
                 ->new('sv_SV');

for my $letters (\@letters, \@reversed) {
  print "Original: @$letters\n";

  my @german = $german->sort(@$letters);
  print "German: @german\n";

  my @swedish = $swedish->sort(@$letters);
  print "Swedish: @swedish\n\n";
}
```

Das die gleiche Ausgabe wie oben erzeugt.

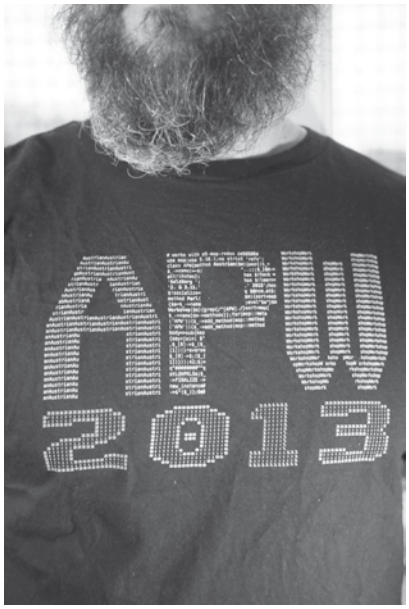
Thomas Klausner

MOPfuscation

Für den Österreichischen Perl Workshop 2013 in Salzburg wollten wir wiedermal ein T-Shirt produzieren. Bei einem Orga-Meeting trug Nick ein uraltes Shirt vom allerersten Austrian Perl Workshop (2004), auf dem eine selbst-referenzielle Obfuscation gefeatured wird. Mitten in meinen Überlegungen hat Stevan Little `p5-mop-redux` announced, seinen zweiten Versuch, ein MOP (Meta Object Protocol) für Perl 5 zu bauen. Also habe ich beschlossen eine MOPfuscation zu versuchen, also eine Obfuscation, die Meta-Programmierung verwendet (und so das selbstreferenz-Thema des 2004-er T-Shirts wieder aufzunehmen).

`p5-mop-redux` findet man hier: https://github.com/stevan/p5-mop-redux. Hier gibt es eine sehr gute Einführung in `p5-mop`: http://blogs.perl.org/users/damien_dams_krotkine/2013/09/p5-mop.html, inklusive eine Installationsleitung basierend auf `perlbrew` und `cpanminus`.

Das T-Shirt



Das P

Für uns relevant ist nur das P:

```
# works with p5-mop-redux ce50586a
use mop;use 5.18.1;no strict 'refs';
class APW{method Austrian($m){map{($_=
$_->name)=~s;              ^..;;x;$_}$m->
attributes};                 has $!hack =
'Salzburg ';                 has $!learn=
'2. & 3.11.'                 .' 2013';has
$!socialize=                 q $@ncm.at$;
method Perl(                 $m){sort+map
{$a=$_->name                 ;eval"$a"}$m
->attributes}               ;42.;method
Workshop($m){grep{/^[APW]./}sort+map{
$_->name}$m->methods}};for(mop::meta
('APW')){$_->add_method(mop::method
->new(name=>$",
body=>sub{map
{say+join( $"
,$_[0]->$_($_
[1]))}reverse
$_[0]->$;($_[
1])}));42;$;=
q^########^^q
atLQHPKLSa;$_
->FINALIZE ->
new_instance#
->$"($_)};###
```

Da vermutlich nicht auf den ersten Blick ersichtlich ist, was dieser Code ausspuckt, zeige ich hier mal den Output:

```
Austrian Perl Workshop
2. & 3.11. 2013 @ncm.at Salzburg
hack learn socialize
```

Alles, was hier ausgegeben wird, steht auch genau so im Source Code. Es handelt sich hier also nicht um eine (langweilige) "Daten-Verschlüsselungs"-Obfuscation, sondern um eine richtige "Wie-verdammt-noch-mal-hat-er-das-gemacht"-Obfuscation. Um die Aufklärung dieser Frage zu vereinfachen, zeige ich hier den Code in einer schön(er) formatierten Variante, die auch auf der Rückseite des T-Shirts zu sehen ist:

```
use mop; use 5.18.1; no strict 'refs';
# use p5-mop-redux ce50586af1d

class APW {
 has $!hack        =  'Salzburg';
 has $!learn       =  '2. & 3. 11 2013';
 has $!socialize   =  '@ncm.at';

 method Austrian ($m) {map
  ($_=$_->name)=~s;^..;;;$_}$m->attributes}
 method Perl     ($m) {sort+map{
  $a=$_->name;eval"$a"}$m->attributes}
 method Workshop ($m) {grep{/^[APW]./
  }sort+map{$_->name}$m->methods}
}

for(mop::meta('APW')){$_->add_method(
   mop::method->new(name=>$",body
=>sub{map{say+join($",$_[0]->$_(
   $_[1]))}reverse$_[0]->$;($_[1])}));
$;=q^########^^q atLQHPKLSa;$_->
   FINALIZE->new_instance->$"($_)};
```

Zeile für Zeile...

Und das gehen wir jetzt mal im Detail durch...

```
use mop; use 5.18.1; no strict 'refs';
# use p5-mop-redux ce50586af1d
```

Zuerst laden wir mal `mop` und ein aktuelles Perl. `no strict 'refs'` ist in der finalen Version nicht mehr notwendig, aber anscheinend hab ich vergessen, es rauszunehmen, und wollte dann das P nicht noch mal neu "zeichnen".

```
class APW {
   ...
}
```

Mittels `class` wird eine Klasse namens `APW` definiert. `APW` steht natürlich für Austrian Perl Workshop.

```
has $!hack        =  'Salzburg';
has $!learn       =  '2. & 3. 11 2013';
has $!socialize   =  '@ncm.at';
```

`has` definiert Attribute dieser Klasse, relativ ähnlich wie `Moose`, nur kompakter. `$!` ist ein sog. Twigil (also ein doppeltes Sigil, wie `$`, `@` etc in Perl genannt werden). Twigils gibt's auch in Perl 6, und `p5-mop` verwendet dieselben Twigils. `$!hack` zum Beispiel macht `hack` zu einem privatem Attribut der Klasse `APW`. D.h. man kann auf dieses Attribut nur in Methoden dieser Klasse zugreifen, und nicht von außerhalb. Praktischerweise kann ich den Attributen auch gleich defaults zuweisen.

```
method Austrian ($m) { ... }
method Perl     ($m) { ... }
method Workshop ($m) { ... }
```

`method` definiert überraschenderweise eine Methode. Ein sehr cooles Feature von `p5-mop` ist, dass man in der Methodendefinition gleich eine Parameterliste definieren kann, die dann innerhalb der Methode ohne mühsames entpacken von `@_` zur Verfügung steht. Ebenso automatisch kann jede Methode auf `$self` zugreifen.

Alle drei Methoden bekommen jeweils ein Argument, `$m`. Dieses ist das Meta-Objekt der Klasse `APW` (dazu später mehr).

```
method Austrian ($m) {map{
 ($_=$_->name)=~s;^..;;;$_}$m->attributes}
```

Die Methode `Austrian` besteht aus einem `map`-Statement, das über das Meta-Objekt die Liste der definierten Attribute durchgeht. Im Code-Block des `map`-Statements passiert folgendes: `$_` ist das jeweilige Attribute-Objekt. Auf dieses kann ich `name` aufrufen, was z.B. `$!hack` zurückgibt. Damit überschreibe ich `$_` (eine nur in Obfuscations empfohlene Praxis..) und ersetze mittels der Regex `s/$..//` die ersten beiden Zeichen (`$!` mit nichts. `Austrian` liefert also eine Liste der "schönen" Attributnamen, zB "hack", "learn", "socialize".

```
method Perl ($m) {sort+map{
  $a=$_->name;eval"$a"}$m->attributes}
```

In der Methode `Perl` gehe ich wieder die Attribut-Liste mit einem `map` durch. Diesmal wird aber der Name jedes Attributes in `$a` gespeichert, welches danach mittels `eval` ausgeführt wird. Das `+` zwischen `sort` und `map` wird von Perl ignoriert, was in Obfus recht praktisch sein kann. `Perl` gibt also eine sortierte Liste der Attribute-Werte zurück, also "2. & 3.11. 2013", "@ncm.at" und "Salzburg".

```
method Workshop ($m) {grep{
 /^[APW]./}sort+map{$_->name}$m->methods}
```

Auch die Methode `Workshop` verwendet `map`, diesmal allerdings, um via `methods` alle Methoden durchzugehen. `$_->name` gibt die Namen der Methoden an `sort` und dann weiter an `grep`. Hier werden nur Methodennamen durchgelassen, die mit A, P oder W beginnen. `Workshop` liefert also die sortierten Methodennamen "Austrian", "Perl", "Workshop"

Der nächste Code-Block ist noch ein wenig kompakt, hier mal eine etwas lesbarere Variante:

```
for (mop::meta('APW')) {
    $_->add_method( mop::method->new(
        name => $",
        body => sub {
            map { say+join($",$_[0]->
              $_($_[1])) }
                reverse $_[0]->$;($_[1])
        }
    ));
    $;=q^########^^q atLQHPKLSa;
    $_->FINALIZE->new_instance->$"($_)
};
```

`mop::meta('APW')` ermittelt das Meta-Objekt für die Klasse `APW`. Dieses Meta-Objekt wird dann später an die div. Methoden als `$m` übergeben. Der `for`-Loop ist nur dazu da, das Meta-Objekt auf nicht allzu offensichtliche Art in `$_` zu stopfen.

Auf das Meta-Objekt rufe ich jetzt `add_method` auf. Mit dieser Meta-Methode kann ich zur Laufzeit eine neue Methode in der Klasse anlegen (ein sehr praktisches Feature von Meta-Programmierung!). Die neu angelegte Methode wird mittels `mop::method->new` definiert. Sie bekommen (via `name`) einen Namen, der in `$"` gespeichert ist. `$"` wird nirgendwo definiert, was auch nicht notwendig ist, da es sich um eine der netten / kryptischen `predefined variables` handelt, die in `perlvar` nachgelesen werden können. `$"` ist der `LIST SEPERATOR` und per default ein Space. Die Methode heisst also `" "` (ja, das geht!).

Die Funktionalität der Methode wird in einer Coderef festgelegt, die mittels `body` gesetzt wird. Was genau hier passiert schauen wir uns später an, jetzt machen wir mal hier weiter:

```
$;=q^########^^q atLQHPKLSa;
```

Anscheinend wird hier etwas an `$;` zugewiesen. Aber was? Ersetzen wir also mal die quote-Operatoren durch normale Anführungszeichen:

```
$;='########' ^ 'tLQHPKLS';
```

Das erste Quoting mit `q^...^` sollte ja noch relativ klar sein, das zweite verwendet allerdings ein eher fragwürdiges Feature von `q`. Wenn nämlich nach `q` ein Space folgt, können beliebige Zeichen als quote-Zeichen verwendet werden. In diesem Fall also das `a`. Die beiden Strings werden nun mit `^` (also XOR) bitweise vermanscht, was 'Workshop' ergibt. `$;` ist (so wie `$"`) eine predefined variable und damit global. Wir werden sie in Kürze wiedersehen.

Die letzte Zeile ruft nun auf das Meta-Objekt `FINALIZE` auf. Das war zum Zeitpunkt der Erstellung dieser Obfu noch notwendig, sollte aber eigentlich automatisch passieren, ist aber ein geringer Preis, den ich für die `p5-mop`-Coolness gerne zahle. `new_instance` legt nun eine neues Objekt an, auf welches wir die Methode `$"` mit dem Argument `$_` aufrufen. `$"` ist die dynamisch angelegte Methode, `$_` ist weiterhin das Meta-Objekt der `APW` Klasse.

Nun wenden wir uns also dem `body` der dynamisch angelegten Methode `$"` zu:

```
map { say+join($",$_[0]->$_($_[1])) }
  reverse $_[0]->$;($_[1])
```

Ich ersetze mal `$_[0]` und `$_[1]` durch ordentliche Variablen:

```
my ($self, $meta) = @_;
map { say join($", $self->$_($meta)) }
  reverse $self->$;($meta)
```

Nun sieht man ein wenig besser, was passiert: Ich rufe auf `$self` (also die Instanz von `APW`) die Methode auf, deren Name in `$;` gespeichert ist (also 'Workshop') und übergebe als Parameter das Meta-Objekt. Die Methode `Workshop` liefert (wir erinnern uns) alle Methoden der Klasse, die mit A, P, oder W beginnen. Diese Liste wird mittels `reverse` umgedreht und an `map` übergeben. Innerhalb des Codeblocks von `map` ist `$_` nun der jeweilige Methodenname, der nun aufgerufen wird und dessen Rückgabewerte mit `$"` (Space) gejoined und dann ausgegeben werden.

Ich rufe also die Methoden `Workshop`, `Perl` und `Austrian` auf, die, wie wir weiter oben gesehen haben, die Methodennamen, die in den Attributen gespeicherten Werte und die Attributenamen ausgeben, also:

```
Austrian Perl Workshop
2. & 3.11. 2013 @ncm.at Salzburg
learn socialize hack
```

Perl6

Auf der Rückreise vom Perl Workshop von Salzburg nach Wien habe ich die MOPfuscation nach Perl6 portiert. Der folgende Code sollte mit dem schon etwas alten Commit c95b6d95f06f5f3b39a379dff50dd68bc46e481a funktionieren:

```
class APW {
  has $!hack      = 'Salzburg';
  has $!learn     = '2. & 3. 11 2013';
  has $!socialize = '@ncm.at';

  method Austrian { self.^attributes.map(
    { .name.substr(2) }) }
  method Perl     { self.^attributes.map(
    { .name.eval }) }
  method Workshop { self.^methods.grep(
    { .name ~~ /^<[APW]>/ }).sort }
}

  APW.^add_method(" ",method {
     my $s=q~########~~^q^tLQHPKLS^;say
  self."$s"().reverse.map(
     { self."$_"().join(" ") }).join("\n");
  });APW.^compose.new." "();
```

Den Programmablauf habe ich unverändert übernommen. Die gesamte Meta-Programmierung funktioniert in Perl6 relativ gleich wir unter p5-mop. Der massivste Unterschied ist, dass in Perl6 `map` und `grep` von links nach rechts geschrieben werden können.

Zusammenfassung

`p5-mop` ist ein sehr interessantes Projekt, ich hoffe mal, dass es weitergeführt und vielleicht sogar mal in den Perl5 Core aufgenommen wird. Perl 6 ist auch schon sehr fertig, allerdings z.Z. noch zu langsam.

Anmerkung

Dieser Artikel basiert auf meinem Blog-Post "MOPfuscation explained" vom 7.11.2013: http://domm.plix.at/perl/2013_11_MOPfuscation.html

bitmuncher

Bitmunchers Admin-Bausatz zur Automatisierung mehr...

Da immer mal wieder Fragen zu der Software, die ich verwende, an mich herangetragen werden, will ich in diesem Artikel mal auf jene Tools eingehen, die ich für die Administration von Webapp-Server-Umgebungen bevorzuge. Die Verwaltung von Netzwerken auf LAMP- und Tomcat-Basis ist ja in den letzten Jahren das Gebiet geworden, in dem ich die meisten Erfahrungen sammeln konnte. Manche der von mir eingesetzten Werkzeuge und Techniken lassen sich aber auch problemlos in anderen Server-Netzwerken einsetzen. Mein Augenmerk bei diesem Artikel liegt darauf, wie Automatisierungen im Netzwerk die Konsistenz des Systems sicherstellen und die Arbeitsweise des Sysadmins vereinfachen können.

Grundlegend kann man das System-Engineering bzw. die Systemadministration in folgende Bereiche unterteilen:

- Planung
- Aufbau
- Pflege & Wartung
- Dokumentation
- Überwachung
- Auswertung
- Kommunikation

Jeder dieser Bereiche benötigt verschiedene Werkzeuge um sie effizient durchführen zu können.

Planung

Plant man ein komplett neues Netzwerk "from scratch", gibt es viele Dinge, die man bedenken muss. In erster Linie gibt es aber vor allem viele Informationen, die man sinnvoll miteinander verknüpfen muss um ein Gesamtbild des zukünftigen Netzwerks erstellen zu können, in dem alles wie in einem guten Uhrwerk konfliktfrei ineinander greift. Diese Informationen lassen sich nicht immer auf die gleiche Weise visualisieren, so dass man verschiedene Tools für verschiedene Arten von Informationen braucht.

Ausgangspunkt aller meiner Planung ist eine Todo-Liste. Ich verwende dafür zumeist Wunderlist. Diese Liste umfasst alle Informationen, die ich benötige um ein Netzwerk für einen Webapp erstellen zu können, wobei am Ende eine Liste von Use Cases steht, die das Netzwerk abbilden muss. Näheres dazu kann man meinem Blog-Artikel *Anwendungsfall Systemadministration* entnehmen.

Alle mit Hilfe der Todo gesammelten Informationen landen dann recht unstrukturiert in Evernote, einem digitalen Notizbuch. Evernote hat den Vorteil, dass man darin auch Fotos von Whiteboards aus Meetings, PDFs usw. sammeln kann.

Diese Informationen werden dann in einem richtigen Konzept zusammengeführt. Hierfür verwende ich zumeist eine Wiki-Software, die dann später auch gleich für die Doku des Netzwerks verwendet werden kann. Ich bevorzuge dabei ganz klar TWiki und zwar aus verschiedenen Gründen. Einer der Gründe ist, dass mein "Admin-Bausatz" komplett auf Perl basiert. Das heisst, dass alle Tools, die nicht nur von mir allein sondern vom kompletten Team genutzt werden, mit Perl erweitert und verwaltet werden können. Ich habe nämlich bisher nur wenige Admins kennengelernt, die kein Perl beherrschten, so dass diese Programmiersprache als Konsenz im Team immer sehr gut passte. Ein anderer Grund ist, dass TWiki sehr flexibel ist und bei Bedarf sogar für's Ticketing genutzt werden kann. REST-API und umfangreiche Plugin-Schnittstellen ermöglichen später auch eine Automatisierung der Doku, auf die ich an gegebener Stelle noch genauer eingehen werde.

Natürlich fallen auch Informationen wie Datenfluss- oder Netzwerk-Diagramme an. Um diese in eine anständige Form zu bringen, verwende ich Dia. Die damit erstellten Diagramme lassen sich als PNG, JPEG, PDF usw. exportieren, so dass man sie auch ins Wiki einbinden kann.

Aufbau

Steht das Konzept dann endlich, kann es an die praktische Umsetzung gehen. Für Bestellung und Einbau der notwendigen Hardware-Komponenten leisten Todo-Listen wieder gute Hilfe. Sie sorgen dafür, dass nichts vergessen wird. Dabei habe ich mir 4 Listen angewöhnt:

- Notwendig: wird abgehakt, wenn die Komponente bestellt wurde

- Geliefert: wird abgehakt, wenn die Komponente geliefert wurde

- Verbaut: wird abgehakt, wenn die Komponente eingebaut wurde

- Bereit: wird abgehakt, wenn die Komponente bereit ist

Sind alle diese Listen abgearbeitet, weiß ich, dass jede notwendige Hardware bereit ist. Dann geht es an den Aufbau der Use Cases. Damit nun nicht wild Software auf den Maschinen installiert und konfiguriert wird, kommt natürlich ein zentrales Konfigurationswerkzeug zum Einsatz. Weniger bekannt als Chef oder Puppet ist in diesem Bereich Rex. Auch Rex ist natürlich in Perl geschrieben und bietet so ziemlich alles, was man von einem Deployment- und Konfigurationsverwaltungswerkzeug erwartet. Setzt man seine Server von Anfang an auf Basis eines Minimal-Systems mit Rex auf, kann man immer sicherstellen, dass die Use Cases an einer Stelle zentral abgebildet werden. Die einzige Regel, die man dafür im Team durchsetzen muss: Niemals manuelle Änderungen an einem Server. Änderungen prinzipiell nur über Rex. Das ist zwar gerade für Admins, die bereits etwas länger im Geschäft sind, anfangs eine ungewohnte Arbeitsweise, aber sie zahlt sich aus, da sie die Konsistenz der Systeme sicherstellt.

Pflege & Wartung

Und damit sind wir auch schon beim nächsten Arbeitsbereich, den üblichen Wartungsarbeiten. Hält man sich an die genannte Regel, erfolgt natürlich auch das Einspielen von Updates oder das Ändern von Konfigurationen ausschließlich über Rex. Allerdings will man Software ja nicht gerade ungetestet in das Live-System einspielen. So wie die Webapp ja durch eine QA muss, sollte es auch ein Test-Netzwerk für die Systemadministration geben. Dabei bietet es sich an für QA und Sysadmins ein gemeinsames Netzwerk aufzubauen. Dieses sollte applikationsidentisch zum Live-Netzwerk sein. Das heißt es sollte die gleichen Komponenten, die gleiche Software mit den gleichen Versionen usw. einsetzen. Allerdings sollte man natürlich die Komponenten aus Kostengründen alle virtualisieren und entsprechend kleiner dimensionieren.

Wenn es sich um ein reines Linux-Netzwerk handelt, baue ich das Test-System bevorzugt mit OpenVZ nach. Und damit dieses wirklich identisch zum Live-Netzwerk wird, verwende ich die gleichen Rexfiles wie für das Live-Netzwerk, nur mit anderen IP-Adressen. Die DNS-Namen werden dann über einen DNS des Test-Netzwerks passend umgebogen, so dass man ein komplett autarkes System erhält, das genau so funktioniert wie das Live-Netzwerk.

In dieses Test-Netzwerk kann man Updates erstmal einspielen und so sicherstellen, dass alle relevanten Use Cases weiterhin funktionieren. Erst wenn dies erfolgreich geprüft wurde, erfolgt die Live-Freigabe und das Deployment kann auf dem Live-System erfolgen.

Gleiches gilt natürlich auch für Konfigurationsänderungen. Alle Änderungen, die nicht die Ressourcen-Größen betreffen, weil diese sich ja zwischen Test- und Live-System unterscheiden dürfen, müssen zuerst im Test-Netzwerk eingespielt und getestet werden, bevor sie live gehen dürfen. Nur so können Flüchtigkeitsfehler vermieden werden. Fügt man z.B. einen komplett neuen Server in das Netzwerk hinzu und vergisst einen DNS-Record für diesen anzulegen bzw. das notwendige Rexfile dafür anzupassen, wird dies schon im Test-System auffallen, da der Anwendungsfall “DNS-Name auflösen” dort nicht funktioniert.

Um sicherzustellen, dass Änderungen wirklich erst im Test-Netzwerk durchgeführt werden, kann man einfach eine

Synchronisierung der Rexfiles vom Test-Netzwerk zum Live-Netzwerk über ein Skript vornehmen, das die IPs entsprechend umbiegt. So kann das Team Änderungen nur am Rexfile-Repository für das Test-Netzwerk vornehmen und der Teamleiter kann erfolgreich getestete Änderungen dann für das Live-Netzwerk freigeben indem er die Synchronisierung vornimmt. Bei einer solchen Arbeitsweise muss aber sichergestellt sein, dass eine Rufbereitschaft im Notfall auch Änderungen am Live-System vornehmen können muss und dass diese transparent über eine Versionsverwaltung erfolgen. Mir helfen dabei immer ein paar Perl-Skripte. ;)

Dokumentation

Dokumentation ist wohl in jedem Sysadmin-Team ein leidiges Thema. Meist wird vergessen Änderungen zeitnah zu übertragen, manche Dinge werden gar nicht dokumentiert und oft genug werden Aktualisierungen der Doku aus Zeitmangel immer wieder verschoben. Das muss allerdings nicht so sein, wenn in der Planungs- und Aufbauphase die Dokumentation der Use Cases korrekt erfolgte.

Eine optimale Dokumentation hat für mich folgenden Aufbau. Als oberste Kategorie gibt es die Projekte des Unternehmens bzw. die Projekte, die innerhalb des Netzwerks aktiv sind. Als Beispiel könnte man hier mein privates Server-Netzwerk nennen, das aus den Projekten Subnetworx mit zugehörigem Test-Wordpress, dem AdminTestNet, den ORSN-DNS, einigen Webservern, einigen Mailservern und diversem anderen Kram besteht. Jedes dieser Projekte umfasst bestimmte Use-Cases, die es erfüllen muss. So muss bei den DNS-Servern sichergestellt werden, dass sie auf Port 53 erreichbar sind. Sie müssen eine bestimmte Antwortzeit erfüllen. Sie müssen immer eine aktuellen Zonen-Datei für die Rootzone haben. Und so weiter... Diese Use-Cases sind auf der Projektseite vermerkt. Und für jeden Use-Cases gibt es in der Doku wiederum eine Unterseite, in der kurz erklärt wird wie der Use Cases umgesetzt ist. In diese werden auch Doku-Seiten für alle beteiligten Netzwerk-Komponenten verlinkt.

Die Dokumentation der Komponenten kann allerdings vollständig automatisch generiert werden, sofern das Netzwerk ein paar Anforderungen erfüllt. Grundlegend muss es im Netzwerk einen zentralen DNS geben, über den die internen Hostnamen der Server aufgelöst werden können. Ein Skript kann dann nämlich sehr einfach regelmäßig prüfen ob im Netzwerk noch immer die gleichen IP-Adressen, d.h. Komponenten, aktiv sind. Ändert sich etwas am Zustand des Netzwerk, ist also eine IP erreichbar, die bisher nicht aktiv war oder ist eine IP plötzlich nicht mehr erreichbar, kann das Skript automatisch Informationen zu dieser IP sammeln und diese im Wiki vermerken. Als Titel der Seite wird dann der aufgelöste Hostname für die IP verwendet.

Am einfachsten verständlich wird dies wohl am ehesten an einem Beispiel. Wir wissen von einem Netzwerk prinzipiell, innerhalb welcher IP-Ranges es agiert. Wenn nicht, können wir diese aus unserem Rexfile-Repository extrahieren. ;) Ein Perl-Skript kann also problemlos regelmäßig diese IP-Ranges prüfen und nachschauen welche IPs gerade aktiv sind. Kommt eine neue IP hinzu, löst es den zugehörigen Namen mit Hilfe des DNS auf und checkt, ob es bereits eine Wiki-Seite zu diesem Namen gibt. Weiterhin fragt es über einen speziellen Monitoring-Account via SSH die Systemdaten zu dieser IP ab. Welche CPU, wie viel RAM, Festplattenaufteilung, Netzwerkspeicher etc. Alternativ kann man diese Daten auch aus den zugehörigen Rexfiles holen. Ist nun bereits eine Seite zu dem DNS-Namen vorhanden, wird diese aktualisiert, sonst wird eine angelegt und mit entsprechenden Inhalten befüllt. Fehlt hingegen eine IP, wird mit dem Monitoring abgeglichen ob es sich um einen Ausfall oder eine geplante Downtime handelt und ob dies bereits in Bearbeitung ist oder ob die Komponente zu dieser IP komplett entfernt wurde. Entsprechend wird dann darauf reagiert (Seite entfernen oder neuen Scheduler-Durchlauf ansetzen usw.).

Weiterhin empfehle ich, dass in diesen automatisch erstellten Seiten zu den Komponenten des Netzwerks auch gleich die zugehörigen Use Cases verlinkt werden. Auch dies kann automatisiert erfolgen, indem überprüft wird in welchen Use Cases auf die Komponente Bezug genommen wird. Voraussetzung dafür ist allerdings eine gewisse Disziplin der Sysadmins, so dass Regel 2 der Systemadministration eingehalten wird: Verändert sich ein Anwendungsfall, muss dies zuerst in der Dokumentation vermerkt werden. Nur so kann sichergestellt werden, dass die konzeptionelle Dokumentation immer aktuell ist. Das ist auch gar nicht so schwierig durchzusetzen, wenn jeder neue Use Case erst vom Teamleiter in's Konzept eingebracht wird und die Admins sich ausschließlich nach diesem richten.

Kurzum: Hat man sich erstmal einen Auto-Doku-Generator für die eigenen Anforderungen zusammengeschraubt, muss man sich um die Dokumentation der Komponenten im Netzwerk nicht mehr kümmern. Das ist zwar anfangs etwas Zeitaufwand, spart aber in den folgenden Jahren ein Vielfaches der Zeit wieder ein und man hat immer einen aktuellen Überblick über den aktuellen Aufbau des Netzwerks. Unbenutzte Komponenten fallen dann z.B. dadurch auf, dass mit ihnen keine Anwendungsfälle verknüpft sind, was wiederum der Kosteneffizienz zugute kommt. Und ein Blick auf das Datenblatt eines Server zeigt auf welche Use Cases er im Netzwerk beeinflusst bzw. welche für ihn relevant sind. Bei einem Webserver wird sich da beispielsweise der Use Case "Deployment der Webapp" finden, der auf die zugehörige Deployment-Anleitung verweist. Oder man hat den Use Case "Erreichbarkeit der Webapp auf Port 80□ mit Link zum zugehörigen Monitoring, dem Datenfluss-Diagramm usw..

Ist ein Netzwerk wirklich komplett unter der Verwaltung von Rex, kann man natürlich die automatische Dokumentation auch mit dem Deployment verbinden. Allerdings muss dann darauf geachtet werden, dass die Generierung der Doku auch wirklich aufgerufen wird. Die Erfahrung zeigt aber, dass gerade neue Teammitglieder, die mit den automatisierten Vorgängen noch nicht vertraut sind, so etwas gern vergessen, was zu Inkonsistenz in der Doku führen kann. Ich bevorzuge daher eher ein Verfahren, das unabhängig von der Team-Disziplin funktioniert. Dennoch macht es Sinn die Rexfiles zusätzlich auszuwerten. So kann auch dokumentiert werden ob und welche Konfigurationsparameter im Vergleich zum Basis-System geändert wurden, welche Software auf der Komponente installiert ist und in welchen Versionen und vieles mehr. Gibt es zu einer Komponente kein Rexfile, sollte dies umgehend an das Admin-Team bzw. den Teamleiter gemeldet werden, da es einen gravierenden Fehler im Workflow darstellt und ggf. ein Angriff sein kann. Man erhält so auch einen zusätzlichen, wenn auch recht einfachen, Security-Layer, der das Netzwerk auf neue Komponenten überprüft und diese meldet. In der Industriespionage sind nämlich kleine VMs, die nach erfolgreichem Eindringen in ein Netzwerk als zukünftige Backdoor gestartet werden, nicht gerade selten. Oftmals werden sie auch einfach genutzt um über Covert Channel Daten nach außen abzuleiten. Ob nun Workflow-Fehler eines Neulings oder Angriff eines Hackers… man will so etwas nicht in seinem Netzwerk.

Überwachung

Monitoring ist im Gegensatz zur Doku ja eher ein Lieblingsthema vieler Admins. Ich bevorzuge dabei Systeme, die auf Nagios basieren. Wenn ich nicht das Original Nagios zum Einsatz bringe, dann zumeist Icinga oder Centreon. Jedes hat so seine Vor- und Nachteile und eignet sich verschieden gut je nach Größe und Struktur des Netzwerks. Gemeinsam haben sie alle, dass sie mittels Perl-Plugins einfach erweitert werden können. So kann man sich z.B. die aufgetretenen Fehler auch gleich im Wiki vermerken lassen, so dass man sie später an einer zentralen Stelle auswerten kann.

Beim Aufbau des Monitorings sollte man darauf achten, dass man vom üblichen "Überwachen von Komponenten" weg geht. Man überwacht im Optimalfall die Use Cases, die für die Projekte definiert sind. Man überwacht also nicht ob der Datenbank-Server erreichbar ist und in der richtigen Geschwindigkeit antwortet. Stattdessen prüft man ob der Server, der die DB erreichen muss, dies auch wirklich kann. Nur so kann ausgeschlossen werden, dass die Security-Layer zwischen den Komponenten irgendwelche Use Cases unterbrechen. Ein Datenbank-Server kann z.B. für den Monitoring-Server problemlos erreichbar sein, während das Angriffserkennungssystem fälschlicherweise eine Webserver-IP geblockt hat. Sollte zwar bei einer korrekten Konfiguration nicht vorkommen, aber auf das "Sollte" sollte man sich in unserem Job nicht verlassen. Die Überregel, also quasi die Regel 0 der Systemadministration, ist nämlich: Gehe immer vom Worst Case aus. Aber ich glaube das muss man Sysadmins nicht extra erklären.

Der einzige Bereich, in dem dann tatsächlich noch Komponenten überwacht werden, bezieht sich auf die Systemparameter. Wenn ein DB-Server nicht mehr schnell genug reagiert, will ich mich nicht erst auf dem System einloggen müssen um zu sehen was da los ist, zumal dies oft kaum noch möglich ist, wenn ein System gerade überlastet ist. Im Monitoring sollte ich daher sehen können ob es einen ungewöhnlichen Anstieg beim RAM-Verbrauch, Swap-Durchsatz, CPU-Verbrauch oder was auch immer gab und ob nur der DB-Server nicht mehr reagiert oder ob die komplette Kiste nicht mehr erreichbar ist.

Und auch hierbei greife ich gern zu Perl und erstelle mir Skripte, die das Komponenten-Monitoring automatisch konfigurieren. Das lässt sich am besten über die Rexfiles für die

Komponenten machen. Fügt man ein Komponente im Netzwerk hinzu, geschieht dies sowieso über Rex und entsprechend kann man auch gleich die grundlegenden Monitoring-Regeln, wie z.B. die Hardware-Überwachung, mit ausrollen. Der zugehörige Monitoring-Account, der auch vom Doku-Generator verwendet wird, sollte Bestandteil des Basis-Setup eines jeden Servers sein.

Auswertung

Die Auswertung von Informationen umfasst bei der Systemadministration grundlegend 2 Bereiche.

1. Fehleranalyse
2. statistische Auswertungen

Wie bereits beim Monitoring angesprochen, sollte man dieses so aufbauen, dass ein Blick ins Monitoring reicht um die Ursache eines Fehlers zu erkennen. Sind die Use Cases korrekt im Monitoring abgebildet, stellt dies zumeist auch kein Problem dar. Voraussetzung ist allerdings, dass man sich ein paar Nagios-Plugins vorbereitet. Nagios beherrscht nämlich per Default kein Log-Monitoring. Mit Hilfe von NRPE kann man dieses aber dennoch umsetzen. Dazu lässt man ein NRPE-Plugin die Fehlermeldungen aus den Logs extrahieren und reicht sie auf diese Weise an entsprechende Plugins zur Auswertung und Aufbereitung an den Nagios-Server weiter, wo sie dann mit den entsprechenden Use Cases verknüpft werden können und ähnliches. Vorteil dabei ist, dass so auch die Fehlermeldungen von verteilten Systemen zentral zusammengeführt werden können. Ab einer gewissen Größe kommt man dann allerdings auch nicht mehr drum herum einen passenden Indexer für diese Logdaten einzubauen. Wenn ich ihn für Beta-tauglich halte, werde ich meinen Log-Indexer evtl. irgendwann mal veröffentlichen. Aktuell hat er noch sehr viel unschönen Code, den man besser nicht veröffentlicht. ;)

Bei der statistischen Auswertung geht es hingegen zumeist darum ob bestimmte SLA-Auflagen erfüllt wurden. Nagios bietet durchaus Funktionalitäten dafür. Diese kann man allerdings auch so erweitern, dass diese Daten automatisiert in der Doku landen und z.B. auf den zugehörigen Projektseiten verlinkt sind. Der Teamleiter hat damit alle Vorfälle, die zu einem Projekt gehören, auf einen Blick zur Verfügung und dazu auch gleich die statistischen Aufbereitungen, die er sich sonst im Monitoring erst generieren müsste. Wird TWiki dann auch noch als Ticketing-System verwendet, kann er sogar Ursache, Zeitpunkt und Verlauf der Problembehandlung per Klick nachvollziehen. Die Tickets beinhalten ja die Projekt-Bezeichnung und können vom Doku-Generator entsprechend zugeordnet und verlinkt werden.

Abschließende Worte

Man sieht also, dass die Dokumentation, sofern sie denn erstmal weitestgehend automatisiert wurde, zum zentralen Werkzeug für ein Team werden kann. Und mit den richtigen Werkzeugen können durch Automatisierungen auch Fehler vermieden und Probleme schnell analysiert werden. Meine Erfahrung zeigt aber, dass es keine allgemein gültige Lösung für jedes Netzwerk gibt. Man benötigt aber immer Werkzeuge für die Konfigurationsverwaltung, das Deployment, das Testing, das Monitoring und die Dokumentation. Der Rest muss durch individuelle Skripte umgesetzt werden. Will man sicherstellen, dass alle Teammitglieder an allen Komponenten gleich gut mitarbeiten können, sollte das Team sich auf eine Programmiersprache einigen, die dann auch zur Pflicht für alle zukünftigen Team-Mitglieder gemacht wird. Sonst werden einzelne Team-Mitglieder immer wieder erst Zeit benötigen um sich in neue Sprachen einzuarbeiten, was ineffizient ist. Die Auswahl der Tools sollte sich an dieser Programmiersprache ausrichten. Je homogener die Verwaltung gemacht werden kann umso weniger fehleranfällig wird sie und umso schneller finden sich neue Teammitglieder in diese ein. Und dann spielt es auch keine Rolle mehr, ob man es mit einem heterogenen Netzwerk zu tun hat. Natürlich sollte ein Sysadmin in der Lage sein mit jeder beliebigen Programmiersprache nach kurzer Einarbeitung klarzukommen aber zwischen Können und Beherrschen einer Sprache liegen oftmals Welten. Das Beherrschen einer Sprache kommt aber erst durch regelmäßige Anwendung, was wesentlich vereinfacht wird, wenn man nicht ständig zwischen Bash, Perl, C, Java, Python, Ruby, KSH usw. wechseln muss. Natürlich hat auch jede Sprache ihre Vorteile, aber gerade bei der Verwaltung eines Server-Netzwerks kann es höchstens noch Python mit Perl aufnehmen.

Schon seit meiner Anfangszeit als Sysadmin hat sich Perl bei mir immer bewährt. Durch die Module der CPAN-Repositories kann man fast alle notwendigen Funktionalitäten mit relativ wenigen Zeilen Code umsetzen. Daher besteht mein bevorzugter Admin-Bausatz, also quasi mein bevorzugter Management-Layer für ein Netzwerk, aus:

- TWiki für Dokumentation und Ticketing
- Rex für Deployment und Konfigurationsverwaltung
- Nagios für Monitoring

Diese Werkzeuge im richtigen Zusammenspiel ermöglichen den Aufbau von Server-Netzwerken, die beliebig nach oben skalierbar sind und dabei nur minimalen Arbeitsaufwand machen, weil grundlegende Konfigurationen und Dokumentationen vollständig automatisiert erfolgen. Nachteil ist allerdings, dass man in den Aufbau einer passenden Management-Infrastruktur erstmal etwas Zeit investieren muss. Die Rexfiles und Skripte müssen natürlich auch erst geschrieben und getestet werden. Hat man diese Technologien allerdings vollständig in ein Netzwerk und in die Workflows des Admin-Teams integriert, kann ein gleich großes Team eine wesentlich größere Anzahl an Systemen verwalten. Der Teamleiter kann sich dabei voll und ganz auf die fertig aufbereiteten Daten konzentrieren, die für seine Entscheidungen relevant sind, und hat diese an zentraler Stelle im TWiki zur Verfügung, während die Admins dort Informationen zu allen Bereichen des Netzwerks finden, ohne sich erst durch Konfigurationen u.ä. wühlen zu müssen.

Leider ist ein solches Idealnetzwerk in der Praxis nur äußerst selten zu finden. Ich hatte bisher nur die Möglichkeit bei toksta* (gibt's mittlerweile leider nicht mehr) ein ähnliches Netzwerk aufzubauen. Allerdings steckte damals das Konfigurationsmanagement noch in den Kinderschuhen, so dass ich es Großteils selbst umsetzte. Das konnte sich mit Tools wie Rex natürlich nicht messen. Ich hege allerdings die leise Hoffnung, dass auch die Sysadmins auf der Welt irgendwann erkennen, dass es mal Zeit für ein paar Standards wird, weil es unsere Arbeit enorm vereinfachen kann. Eine einheitliche handhabbare Automatisierung von Server-Netzwerken tut dies auf jeden Fall, denn man muss sich nicht ständig in neue Sprachen einfinden, weil mal wieder ein neues tolles Tool auf dem Markt ist, und kann sich voll und ganz auf die eigentliche Arbeit konzentrieren, wobei man sich viel Arbeit durch passende Skripte und Workflows ersparen kann.

Neue SSL-Version muss her? Version im Rexfile anpassen, ins Test-Netzwerk deployen, im Monitoring des Test-Netzwerks prüfen ob alle Use-Cases weiterhin funktionieren, Freigabe durch den Team-Leiter für Live und fertig. Im Hintergrund werden die Datenblätter aller betroffenen Server im Wiki aktualisiert und die neue SSL-Version wird vermerkt, das zugehörige Ticket wird mit allen zugehörigen Projekten-Seiten verlinkt, die Alerts, die beim Update auftraten, werden mit dem Vorgang verknüpft usw.. So etwas lässt sich am saubersten in einem homogenen Management-Layer umsetzen, selbst wenn man ein heterogenes Netzwerk zu verwalten hat.

Yanick Champoux

Instant-REST-API für jede Datenbank

Es ist noch nicht so lang her, dass ich mit ElasticSearch gespielt habe, das die interessante Eigenschaft hat eine REST-API als sein primäres Interface zu haben. Sicher, es ist ein wenig gestelzt und umständlich im Vergleich zu nativen Interfaces. Aber auf der anderen Seite ist es eine nette universelle Art von API. Jede Sprache die einen HTTP-Request absetzen kann, kann mit ElasticSearch kommunizieren und hey es kommt noch schlimmer - selbst `curl` kann es. Es wäre irgendwie cool wenn andere Datenbanken so einen Webservice hätten.

Und dann begann ich nachzudenken...

Haben wir nicht `DBIx::Class::Schema::Loader`, das sich mit einer Datenbank verbinden und sein `DBIx::Class`-Schema automatisch generieren kann?

```
package MyDB;

use parent 'DBIx::Class::Schema::Loader';

...;

# later on

my $schema = MyDB->connect(
    'dbi:SQLite:foo.db'
); # boom, we have our schema
```

Und wenn wir eine `DBIx::Class`-Darstellung eines Schemas haben, können wir es nicht untersuchen und so gut wie alles herausbekommen was es darüber zu wissen gibt?

```
use Data::Printer;

# get all the table names
my @tables = $schema->sources;

# and all the columns of all the tables
for my $table ( $schema->sources ) {
  say "Table $table";
  p $schema->source($table)->columns_info;
}
```

Das ist es wenn wir es händisch machen wollen, aber bedenkt man, dass es bereits `SQL::Translator` gibt, das die meiste Arbeit für uns erledigen kann.

```
use SQL::Translator;

print SQL::Translator->new (
  parser      =>
   'SQL::Translator::Parser::DBIx::Class',
  parser_args => {
      dbic_schema => $schema,
  },
  producer    => 'JSON',
)->translate;
```

Da wir über Webservices reden, wollen wir natürlich alles was hin und her geht mittels JSON übertragen - einschließlich der Datenbankeinträge. Nun, das ist kaum ein Problem wenn wir `DBIx::Class::Helper::Row::ToJSON` verwenden.

So sieht es aus als ob wir die Datenbankseite abgedeckt haben. Und das Webframework? Du wirst wahrscheinlich nicht überrascht sein, dass ich `Dancer` nehme. Wir können nicht nur Serializer und Plugins wie `Dancer::Plugin::DBIC` nehmen, auch die Routen zu setzen ist lächerlich einfach.

```
get '/_tables' => sub {
  return [ schema->sources ];
};
```

Es ist sogar noch eleganter: Erinnere Dich, dass Dancer-Routen zur Laufzeit definiert werden, so dass wir das Schema so untersuchen können wie wir wollen und mit jeder denkbaren Route kommen können.

```
my @primary_key =
  schema->source($table)->primary_columns;
my $row_url = join '/',
  undef, $table, ( '*' ) x @primary_key;

# GET ///
get $row_url => sub {
  my @ids = splat;
  return $schema->resultset($table)
                ->find({
      zip @primary_key, @ids
  });
};

# GET /
get "/$table" => sub {
  my @things = $schema->resultset($table)
                      ->search({ params() })
                      ->all;
  return @things;
};

# create new entry
post "/$table" => sub {
  $schema->resultset($table)
         ->create({ params() });
};
```

Zusätzlicher Bonus: Die Art wie `Dancers params()` die in der Abfrage definierten und im Body der Anfrage serialisierten Parameter zusammenfasst spielt uns in die Hände: Einfache Abfragen können einfach über die URL übergeben werden und die komplizierteren Abfragen können als JSON-Struktur definiert werden.

So, füge alle diese Sachen zusammen und Du erhältst `waack`. Alles was es noch braucht ist ein *dsn*, das auf die richtige Datenbank zeigt (und Zugangsdaten wenn benötigt). Um das zu veranschaulichen wollen wir es mit meiner Digikam SQLite-Datenbank ausprobieren.

```
$ waack dbi:SQLite:digikam4.db
>> Dancer 1.3124 server 28914 listening\
   on http://0.0.0.0:3000
>> Dancer::Plugin::DBIC (0.2100)
== Entering the development dance\
   floor ...
```

Und nun lass uns `App::Presto` als unseren REST-Client starten.

```
$ presto http://enkidu:3000

http://enkidu:3000> type application/json
```

Als erstes können wir alle Tabellennamen holen.

```
http://enkidu:3000> GET /_tables
[
 "TagsTree",
 "ImageMetadata",
 "Tag",
 "Setting",
 "ImageRelation",
 "ImageTag",
 "ImageProperty",
 "ImageInformation",
 "ImageComment"
]
```

Wir können auch das komplette Schema holen.

```
http://enkidu:3000> GET /_schema
{
 "translator" : {
    "producer_args" : {},
    "show_warnings" : 0,
    "add_drop_table" : 0,
    "parser_args" : {
       "dbic_schema" : null
    },
    "filename" : null,
    "no_comments" : 0,
    "version" : "0.11018",
    "parser_type" : "SQL::Translator::
        Parser::DBIx::Class",
    "trace" : 0,
    "producer_type" : "SQL::Translator::
        Producer::JSON"
 },
 "schema" : {
    "tables" : {
       "ImageRelations" : {
          "options" : [],
          "indices" : [],
          "order" : "12",
          "name" : "ImageRelations",
          "constraints" : [
             {
              "type" : "UNIQUE",
              "deferrable" : 1,
              "name" :
               "subject_object_type_unique",
              "on_delete" : "",
              "reference_fields" : [],
              "fields" : [
                 "subject",
                 "object",
                 "type"
              ],
              "match_type" : "",
              "reference_table" : "",
              "options" : [],
              "expression" : "",
              "on_update" : ""
             }
          ],
...
```

Zu viel? Wir können die Spalten einer einzelnen Tabelle bekommen.

```
http://enkidu:3000> GET /Tag/_schema
{
   "iconkde" : {
      "is_nullable" : 1,
      "data_type" : "text",
      "is_serializable" : 1
   },
   "name" : {
      "is_serializable" : 1,
      "data_type" : "text",
      "is_nullable" : 0
   },
   "id" : {
      "is_nullable" : 0,
      "data_type" : "integer",
      "is_auto_increment" : 1,
      "is_serializable" : 1
   },
   "icon" : {
      "is_nullable" : 1,
      "data_type" : "integer",
      "is_serializable" : 1
   },
   "pid" : {
      "is_serializable" : 1,
      "is_nullable" : 1,
      "data_type" : "integer"
   }
}
```

Frage diese Tabelle mit einer einfachen Bedingung ab...

```
http://enkidu:3000> GET /Tag id=1
[
   {
      "name" : "orchid",
      "icon" : null,
      "id" : 1,
      "pid" : 0,
      "iconkde" : null
   }
]
```

... oder mit etwas komplexerem.

```
$ curl -XGET -H Content-Type:
  application/json --data '{"name":{
  "LIKE":"%bulbo%"}}' http://enkidu:3000/Tag
[
   {
      "pid" : 1,
      "name" : "Bulbophyllum",
      "icon" : null,
      "id" : 32,
      "iconkde" : "/pictures/IMG_0461.JPG"
   },
   {
      "id" : 56,
      "iconkde" : "tag",
      "icon" : null,
      "pid" : 39,
      "name" : "Bulbophyllum ebergardetii"
   },
   {
      "name" : "bulbophyllum",
      "pid" : 564,
      "iconkde" : null,
      "id" : 565,
      "icon" : 0
   }
]
```

Nebenbei: Ich habe bei dem letzten ein wenig geschummelt. Presto sendet keinen *body* bei GET-Requests. Und `Dancer` deserialisiert GET bodies auch nicht. Patches dafür werden heute nacht geschrieben.

Wie auch immer, zurück zur Show. Wir können also einzelne Zeilen über den Primärschlüssel auswählen.

```
http://enkidu:3000> GET /Tag/1
{
   "id" : 1,
   "iconkde" : null,
   "pid" : 0,
   "icon" : null,
   "name" : "orchid"
}
```

Eine neue Zeile erzeugen.

```
http://enkidu:3000> POST /Tag \
    '{"name":"nepenthes","pid":0}'
{
  "pid" : 0,
  "name" : "nepenthes",
  "iconkde" : null,
  "icon" : null,
  "id" : 569
}
```

Und updaten.

```
http://enkidu:3000> PUT /Tag/569 \
    '{"icon":"img.png"}'
{
   "icon" : "img.png",
   "iconkde" : null,
   "pid" : 0,
   "name" : "nepenthes",
   "id" : 569
}
```

Nicht schlecht, oder? Vor allem wenn man bedenkt, wenn Du Dir die Quellen von `waack` anschaust, wirst Du sehen, dass es kaum mehr als 100 Zeilen Code sind. Nimm Dir eine Minute und lass es sacken.

Einhundert Codezeilen. Für einen universellen Datenbank REST-Webservice.

Wenn das nicht auf den Schultern von Giganten steht, dann weiß ich nicht was.

Mirko Westermeier

Lisp in Perl: Erwachsen werden durch Interpreterbau

"Jede Mitteilung geistiger Inhalte ist Sprache."
– Walter Benjamin

Zu meiner Zeit hat eine Laufbahn als Entwickler oder Informatiker oft damit begonnen, HTML zu "programmieren". Die Flamewars, die sich daraus entwickelten, ob HTML denn nun eine Programmiersprache sei oder nicht, waren abendfüllend. Nach einer gängigen Definition nennt man nämlich nur die formalen Sprachen auch Programmiersprachen, die *turing-vollständig* sind, also in der Lage sind, unter der Annahme eines unbegrenzten Speichers eine Turing-Maschine zu emulieren. Mit einer solchen Programmiersprache sind dann alle Berechnungen durchführbar, die allgemein als berechenbar angesehen werden. HTML ist aber nicht in der Lage, entsprechende Berechnungsvorschriften zu formulieren und daher keine Programmiersprache in diesem Sinne.

Jedoch ist HTML ein typisches Beispiel für eine strukturierte Art, Informationen auszutauschen, wie sie dem Programmierer im Alltag ständig begegnen. Überall werden CSV-Dateien gelesen, Websites maschinell auf Aktualisierungen überprüft, JSON-Objekte in den Speicher gelesen oder Shell-Eingaben interpretiert. All diesen Formaten ist gemein, dass sie sich auf klar definierte Art beschreiben lassen, wodurch ihre maschinelle Verarbeitung überhaupt erst möglich wird. Die Mechanismen, die dabei verwendet werden, ähneln teilweise denen aus dem Compilerbau, wenn es also darum geht, einen Programmtext einzulesen und in ausführbaren Maschinencode zu übersetzen oder direkt zu interpretieren.

Manche Menschen nennen eine Programmiersprache dann "erwachsen", wenn sich in ihr auf angenehme Weise ein Interpreter für dieselbe Sprache programmieren lässt, der sich selbst ausführen kann. Mit diesem Hintergedanken kann man es als Meilenstein für die Entwicklung eines Programmierers ansehen, wenn er einen Interpreter für eine Programmiersprache programmiert. Die Implementierungssprache und die interpretierte Sprache sind dabei nicht von zentraler Bedeutung. Viel interessanter sind die verwendeten Methoden und die getroffenen Design-Entscheidungen, denn die daraus gewonnenen Erkenntnisse lassen sich auf viele alltägliche Probleme anwenden. Mit diesem Artikel möchte ich alle Programmierer, die dies noch nicht getan haben, motivieren, einmal einen Interpreter für eine Programmiersprache zu schreiben. Es soll ihr Schaden nicht sein!

Überblick über einen Lisp-Interpreter

Der hier vorgestellte Interpreter erhebt keinen Anspruch auf überragende Effizienz oder Eleganz, kann aber meiner Hoffnung nach ein wenig Inspiration vermitteln, eine eigene Lösung für das gleiche Problem zu entwickeln: Programme formuliert in einer interessanten Programmiersprache auszuführen. Meine Wahl fiel dabei auf Perl als Implementierungssprache, weil sich die Ideen durch die vielfachen Ausdrucksmöglichkeiten der Sprache genau so schreiben lassen, wie sie in meinem Kopf formuliert waren. Außerdem ist Perls Mächtigkeit in String-Verarbeitung im Interpreterbau von großem Vorteil. Die zu implementierende Sprache ist Lisp, einerseits aus Gründen der Ehrfurcht vor diesem Urgestein schierer Eleganz, aber auch weil Lisp mit extrem wenig Syntax auskommt und so das Parsen des Programmtexts sehr einfach wird. Mit Lisp ist hier übrigens kein bestimmter standardisierter Dialekt der großen Lisp-Sprachfamilie gemeint. Ein Beispiel von der Eingabeaufforderung des Interpreters:

```
-> (define (square x) (* x x))
Function: (x) -> (* x x)
-> (square 42)
1764
```

Zentrale Datenstruktur der Programmierung in Lisp ist die Liste (rekursiv definiert über Kopf und Restliste). Wertet man eine Liste aus, so wird versucht, den Listenkopf als Funktion oder Operator aufzulösen und die restlichen Listeneinträge als Argumente zu übergeben. In obigem Code wird eine Quadrierungsfunktion an den Namen `square` gebunden. Der Aufruf in Form einer Liste führt dann zur Anwendung dieser Funktion auf das Argument (hier: die Zahl `42`).

Komplexe Programme lassen sich dann als Kombination von Operatoren und Funktionen zu weiteren Funktionen ausdrücken. Die Ausführung wird jeweils durch Auswertung eines Ausdrucks durchgeführt. Der Interpreter besteht daher im Wesentlichen aus einer Eingabeaufforderung, die bis zur Eingabe eines Beenden-Befehls die Eingaben im Kontext der vorherigen Geschehnisse interpretiert und das Ergebnis der Auswertung ausgibt, dem sogenannten Read-Eval-Print-Loop.

Herzstück des Read-Eval-Print-Loops ist der Aufruf einer `eval`-Methode des Interpreters, die zu einem gegeben Programmtext die generierten Rückgabewerte als Text zurückgibt (Der hier teilweise gekürzt gezeigte Quelltext lässt sich übrigens im *github-Repository|https://github.com/memowe/perlisp* nachlesen):

```
sub eval {
    my ($self, $string) = @_;

    # lex
    my $token_stream =
       $self->lexer->lex($string);

    # parse
    my @exprs =
      $self->parser->parse($token_stream);

    # eval
    my @values = map {
       $_->eval($self->context) } @exprs;

    # return
    return wantarray ? @values :
      shift @values;
}
```

Interpreter's Anatomy

Aus der Struktur der `eval`-Methode wird unmittelbar die Struktur des Auswertungsvorgangs ersichtlich:

• Der **Lexer** wandelt den als String übergebenen Programmtext in eine Folge von sogenannten Token um, also in Objekte, deren Typen durch die formale Beschreibung der Sprache festgelegt sind. Diese Objekte können mit einem Attribut versehen sein. Beispielsweise wird der String `"("` in das Token `LIST_START` übersetzt oder der String `42` in das Token `NUMBER` mit dem Attribut 42, also dem Zahlenwert selbst.

• Auf der Basis der Reihenfolge der Token findet im **Parser** die syntaktische Analyse des Programmtextes statt. Dabei werden zusammengehörige Token in geschachtelte Expression-Objekte zusammengefasst, den sogenannten Syntaxbaum.

• Jeder Knoten des Syntaxbaumes besitzt eine `eval`-Methode, die jeweils auf der Auswertung eventueller Teilausdrücke beruht. Nur bei atomaren Ausdrücken wie etwa Zahlen oder Strings ist der Rückgabewert dieser Methode die Zahl bzw. der String selbst.

Führt man die `eval`-Methode eines solchen Syntaxbaumes aus, wird also der gesamte zu interpretierende Ausdruck schrittweise ausgewertet. Der am Ende resultierende Rückgabewert der Auswertung wird an die Eingabeaufforderung zurückgeliefert.

Komplexe Ausdrücke auswerten

Am besten kann man das oben skizzierte Vorgehen an einem Beispiel nachvollziehen. Unter der Annahme, dass die oben definierte `square`-Funktion bereits bekannt ist, soll der folgende Ausdruck ausgewertet werden:

```
(- (square 42) 22)
```

Der Lexer erzeugt daraus die Tokenfolge:

```
LIST_START, SYMBOL(-),
  LIST_START, SYMBOL(square),
    NUMBER(42), LIST_END, NUMBER(22),
      LIST_END
```

Schließlich baut der Parser den folgenden Syntaxbaum bestehend aus Expression-Objekten auf:

```
          (-)
         /   \
    (square)  (22)
        |
      (42)
```

Aus den im Kontext bekannten Bindungen für – und `square` ist bekannt, was mit den Argumenten geschehen soll. Bei der Auswertung des ersten Operanden von – muss zunächst der Funktionsrumpf von `square` eingesetzt werden:

```
          (-)
         /   \
       (*)   (22)
      /   \
    (42)  (42)
```

Die Auswertung von unten nach oben ergibt schließlich die gewünschte Subtraktion `42 * 42 - 22`. Das Ergebnis `1742` wird zurückgeliefert.

Bei der Auswertung werden jeweils die `eval`-Methoden der Expression-Objekte des Syntaxbaumes aufgerufen, die bei atomaren Ausdrücken besonders einfach sind (nämlich das Expression-Objekt selbst zurückliefern), bei Funktionsaufrufen in Form von Listenauswertungen hingegen die Auswertung verschachtelter Teilausdrücke auslösen (vereinfacht):

```
sub eval {
    my ($self, $context) = @_;

    # get function expression and
    # arguments
    my $fn_expr = $self->car;
    my @args    = @{$self->cdr->exprs};

    # eval function expression
    my $function =
      $fn_expr->eval($context);
```

In diesem Fall muss der ausgewertete Kopf der Liste ein Operator oder eine Funktion sein, also eine `apply`-Methode bieten, um auf Argumente (die ausgewerteten Einträge der restlichen Liste) anwendbar zu sein:

```
 # check apply-ability (duck typing)
 die $fn_expr->to_string .
   " can't be applied.\n"
     unless $function->can('apply');

 # apply
 return $function->apply($context, \@args);
 }
```

Im obigen Beispiel wird diese Methode mehrfach ausgeführt, nämlich immer dann, wenn eine Liste ausgewertet und damit ein Operator oder eine Funktion auf Argumente angewendet werden soll. Also für die Subtraktion mit dem Operator – für den Aufruf der Funktion `square` sowie für die darin angegebene Multiplikation mit dem Operator `*`.

Interessante Fragen

• In PerLisp werden Argumente von Funktionsaufrufen standardmäßig ausgewertet bevor sie der Funktion übergeben werden, PerLisp ist also *strikt*. Nicht-striktes Vorgehen ist aber nicht nur nützlich (wie etwa bei der Kurzschlussauswertung von Perls `or`-Operator, sondern auch absolut nötig: Die bedingte Ausführung mittels eines I<if-then-else>-Konstrukts etwa ist völlig sinnlos, wenn beide Zweige immer ausgewertet werden. Wie diese Unterscheidung konkret realisiert wird, ist eine interessante Frage beim Entwurf des Interpreters bzw. des verwendeten Sprachdialekts.

• Bei der Ausführung vom Rumpf einer Funktion ist relevant, aus welchem Kontext eventuell verwendete Bindungen stammen (statischer bzw. dynamischer Scope). Welche Sichtbarkeitsstrategie bei der Implementierung verfolgt wird, kann starken Einfluss auf die Ausführung bestehender Programme haben.

• In der Einleitung wurde erwähnt, dass es erstrebenswert sein kann, wenn ein implementierter Interpreter sich selbst ausführen kann. Bei unterschiedlichen Sprachen ist das natürlich nicht möglich. Die Frage nach der Selbstausführbarkeit kann dennoch beantwortet werden. PerLisp ist nämlich im Stande, einen einfachen Lisp-Interpreter (mit dynamischem Scope) auszuführen, der sich selbst ausführen kann. Der zugehörige Code ist in der Testsuite von PerLisp verfügbar.

Frisch ans Werk!

Die hier skizzierten Strategien zur Implementierung eines eigenen Interpreters sollen nur als Anhaltspunkt verstanden werden und individuelle Lösungen nicht zu sehr einschränken, daher sind die ausgewählten Code-Schnipsel auch möglichst allgemein. PerLisp zu schreiben war für mich eine sehr lehrreiche Erfahrung, die ich nur weiterempfehlen kann. Ich würde mich über eine kurze Rückmeldung freuen, wenn aus diesem Artikel neue Interpreter entstanden sind. Viel Spaß!

Steffen Ullrich

SSL mit Perl - einfach aber sicher

Das Protokoll SSL (*Secure Socket Layer*) bzw. der Nachfolger TLS (*Transport Layer Security*) ermöglicht eine sichere, verschlüsselte Kommunikation über eine existierende Verbindung. Durch seine Nutzung in HTTPS zur sicheren Kommunikation zwischen Browser und Webserver ist es ein essentieller Bestandteil der heutigen Netzwerkarchitektur und ist auch als Verschlüsselungsschicht bei vielen anderen Netzwerkprotokollen, z.B. beim Transportieren von E-Mail, im Einsatz.

Außerhalb der Browser hat sich OpenSSL als meistgenutzte Implementierung von SSL/TLS (im weiteren einfach nur SSL) etabliert und wird in Skriptsprachen wie Python, PHP, Ruby und auch Perl als Grundlage der SSL-Unterstützung genutzt.

SSL ist ein komplexes Protokoll und die von OpenSSL gebotene Programmierschnittstelle (API) bringt diese Komplexität spürbar zur Geltung. Ein wichtiger Teil dieser API wird in Perl durch `Net::SSLeay` bereitgestellt, ohne jedoch dabei deutlich vereinfacht zu werden.

`IO::Socket::SSL` setzt auf `Net::SSLeay` auf und versucht die Nutzung von SSL drastisch zu vereinfachen, indem es sich weitestgehend analog zu den anderen `IO::Socket`-Modulen verhält. Damit ist die Nutzung von SSL in einer dem Entwickler vertrauten Weise möglich und vorhandene Programme und Module können einfach um eine Unterstützung von SSL erweitert werden. Mit diesen Vorteilen hat es sich als das meistgenutzte SSL-Modul in Perl etabliert.

Dieser Artikel beschreibt die grundlegende Funktionsweise und Probleme von SSL aus Entwicklersicht und wie man trotz der Komplexität des Protokolls mit `IO::Socket::SSL` eine einfache und sichere Nutzung von SSL in eigenen Applikationen gewährleisten kann. Dabei wird erläutert, welche Sicherheitsfunktionen `IO::Socket::SSL` von sich aus bereitstellt, aber auch welche typischen Fehler bei der Nutzung gemacht werden und wie man diese vermeidet.

Grundlagen von SSL

Ziel von SSL ist die sichere Kommunikation zwischen zwei Parteien innerhalb einer etablierten Datenverbindung, also ohne dass jemand die Möglichkeit hat die Daten abzuhören oder zu verändern.

Die grundlegende Idee dabei ist eine Verschlüsselung der Daten. Doch während bei wenigen Kommunikationspartnern die nötigen Schlüssel bei einem gemeinsamen Treffen vorab verteilt werden können, skaliert dieses Verfahren bei der Vielzahl von Kommunikationsbeziehungen im Internet nicht. Daher wird bei SSL der geheime Schlüssel erst beim Etablieren der Verbindung ausgehandelt. Dabei muss sichergestellt werden, dass der Schlüsselaustausch mit dem tatsächlichen gewünschten Kommunikationspartner erfolgt und nicht etwa mit dem Angreifer bei einem *Man-In-The-Middle* Angriff.

Diese Identifikation der Gegenstelle erfolgt bei SSL über Zertifikate, welche vergleichbar mit einem Ausweis sind. Das heißt, sie beschreiben für wen sie ausgestellt wurden und sollten fälschungssicher sein.

Um Fälschungen auszuschließen muss ein derartiges Zertifikat natürlich gründlich überprüft werden. Dieses erfolgt auf folgende Weise:

1. Zuerst wird geprüft, ob das Zertifikat von einer vertrauenswürdigen Zertifizierungsstelle (I<CA>) ausgestellt worden ist. Für diesen Test haben Browser oder Betriebssystem eine Liste vertrauenswürdiger Zertifizierungsstellen hinterlegt und mittels kryptografischer Operationen wird überprüft, ob die Unterschrift des Ausstellers auf dem Zertifikat gültig ist.

Bei der derzeitigen Nutzung im Internet unterschreibt die dem Browser bekannte CA meist nicht direkt die Zertifikate für die Server. Stattdessen gibt es oft mehrere Zwischenzertifikate und durch passende Unterschriften wird eine Kette von dem Serverzertifikat bis zur eingebauten CA gebildet.

Dieser Validierungsschritt setzt natürlich voraus, dass eine entsprechende Liste von CAs vorhanden ist. Während plattformübergreifende Browser eine eigene Liste eingebaut haben, verlassen sich andere Programme auf vorhandene Listen. So wird auf den meisten UNIX/Linux-Systemen eine entsprechende, von OpenSSL unterstützte, Liste mitgeliefert. Auf die Systemzertifikate von Windows hat OpenSSL jedoch keinen Zugriff. Daher kommt es dort immer wieder zu Problemen mit OpenSSL basierten Implementierungen, beispielsweise bei PHP, Ruby, Python or Perl.

2. Danach wird überprüft, ob das Zertifikat abgelaufen ist. Ähnlich wie Ausweise haben Zertifikate eine begrenzte Gültigkeit, d.h. sie müssen regelmäßig erneuert werden. Dadurch wird unter anderem erreicht, dass die Fälschungssicherheit der Zertifikate parallel zu den Möglichkeiten der Angreifer erhöht wird.

3. Durch den Einbruch in einen Server oder das Ausnutzen von Sicherheitslücken wie der Heartbleed-Attacke kann es passieren, dass das Zertifikat kompromittiert wurde und sich der Angreifer mit der fremden Identität ausgeben kann. In solchen Fällen wird das Zertifikat zurückgezogen.

Um diesen Fall zu überprüfen gibt jede CA regelmäßig Listen mit den Seriennummern aller noch nicht abgelaufenen, aber zurückgezogenen Zertifikate heraus (*Certificate Revocation List - CRL*). Um jedoch nicht für die Überprüfung eines einzelnen Zertifikats die oft riesige Liste herunterladen zu müssen, kann über das *Online Certificate Status Protocol* (*OCSP*) bei der CA der Status eines einzelnen Zertifikats direkt erfragt werden.

Auch wenn diese Verfahren in der Theorie sinnvoll erscheinen, so funktionieren sie in der Praxis nur unzuverlässig. So werden CRL nicht benutzt, weil sie einfach zu groß sind. Bei OCSP hingegen kommt es öfter zu sporadischen Problemen bei den Abfragen, die von den Browsern meist ignoriert werden damit eine Verbindung zustande kommt. Auf Grund dieser Mängel verwendet Google Chrome inzwischen sein eigenes Verfahren für "wichtige" Zertifikate, deckt aber nur einen Teil der Zertifikate damit ab.

Die genannten Skriptsprachen unterstützen alle CRL, überlassen es jedoch dem Nutzer, diese bereitzustellen. OCSP dagegen ist überwiegend nicht implementiert bzw. die Schnittstelle so komplex, dass es praktisch nicht nutzbar ist.

4. Wenn über die vorhergehenden Schritte sichergestellt wurde, dass das Zertifikat nicht gefälscht oder kompromittiert wurde, muss noch geprüft werden, ob das Zertifikat für die Gegenstelle ausgestellt wurde mit der man kommunizieren will.

Bei Zugriffen auf einen Web- oder Mailserver geschieht dieses, indem man den Hostnamen des Zieles der Verbindung mit den Hostnamen im Zertifikat vergleicht. Das genaue Vorgehen dabei ist abhängig vom Protokoll und unterscheidet sich insbesondere bei der Behandlung von Wildcards. So sind bei HTTP Wildcards wie `www*.example.com` erlaubt, während bei IMAP nur `*.example.com` erlaubt ist. Wildcards wie `www.*.com` oder gar `*.*.*` sind in keinem Fall erlaubt, werden aber von vielen Implementierungen unterstützt.

Andere Implementierungen, wie z.B. das früher von `LWP` benutzte `Crypt::SSLeay`, nehmen hingegen gar keine Prüfung des Hostnames vor, d.h. diese müsste vom Anwender selber durchgeführt werden.

Es ist offensichtlich, dass ohne eine zuverlässige Identifikation der Gegenstelle zwar eine verschlüsselte Verbindung zustande kommt, diese aber möglicherweise nicht mit dem gewünschten Kommunikationspartner erfolgt.

Leider wird bei vielen Programmiersprachen und Bibliotheken diese Prüfung nicht standardmäßig durchgeführt oder ist nur unvollständig oder fehlerhaft implementiert.

Ein Ziel von `IO::Socket::SSL` ist daher, alle notwendigen Prüfungen standardmäßig und mit einem Minimum an Aufwand für den Entwickler durchzuführen. Diesem Ziel bin ich bei der Entwicklung in den letzten Jahren immer näher gekommen, sodass `IO::Socket::SSL` in den Standardeinstellungen eine bessere Sicherheit bietet, als es bei den meisten anderen Skriptsprachen der Fall ist.

Einfacher Client mit IO::Socket::SSL

```
use IO::Socket::SSL;
my $client = IO::Socket::SSL->new(
 'www.google.com:443');
print $client "GET / HTTP/1.0\r\nHost:
 www.google.com\r\n\r\n";
print <$client>;
```

Auch wenn die Nutzung einfach ist und dem Verhalten anderer `IO::Socket`-Module entspricht, so passiert hinter den Kulissen eine Menge:

• `IO::Socket::SSL` versucht als Basisklasse `IO::Socket::IP` oder `IO::Socket::INET6` zu benutzen, mit Fallback auf `IO::Socket::INET`. Auf diese Weise wird transparent IPv6 unterstützt.

• Es wird eine Liste von Verschlüsselungsverfahren (Cipher) konfiguriert, welche sicherer als die Voreinstellung von OpenSSL ist (seit Version 1.956, 11/2013).

Analog wird die minimal unterstützte SSL-Version aus Sicherheitsgründen auf SSL 3.0 gesetzt (seit 1.70, 5/2012).

• Um mehrere Hostnamen mit unterschiedlichen Zertifikaten hinter der gleichen IP-Adresse zu unterstützen, wird mittels SNI (*Server Name Indication*) der gewünschten Hostname an den Server übermittelt (seit Version 1.56, 02/2012).

• Der Gültigkeitszeitraum des Zertifikates der Gegenstelle wird überprüft und es wird die Vertrauenskette zu den vertrauenswürdigen Zertifizierungsstellen (CA) verifiziert. Dazu werden seit Version 1.951 (7/2013) die auf dem System für OpenSSL installierten CAs benutzt. Gibt es diese nicht, so werden seit Version 1.968 (3/2014) die mit Firefox ausgelieferten Zertifikate über die `Mozilla::CA` Bibliothek installiert und genutzt. Damit funktioniert die Verifikation auch unter Windows.

• Der Hostname des Zertifikats wird verifiziert. Wenn, wie in diesem Beispiel, kein Applikationsprotokoll bekannt ist, so wird zur Validierung ein Verfahren benutzt, welches mit den meisten Protokollen harmoniert. Alternativ kann man über `SSL_verifycn_scheme` das Validierungsverfahren festlegen, zum Beispiel auf `http` oder `imap`.

• Es wird OCSP-Stapling aktiviert, bei der der Server bereits eine OCSP-Antwort mitliefern kann. Liefert der Server keine Antwort mit oder will man auch alle Zwischenzertifikate und nicht nur das Serverzertifikat überprüfen, so muss man dieses derzeit noch manuell machen. Es ist jedoch ganz einfach:

```
$client = IO::Socket::SSL->new(
    PeerAddr => ...,
    SSL_ocsp_mode => SSL_OCSP_FULL_CHAIN
);
my $errors = $client->
   ocsp_resolver->resolve_blocking;
die "OCSP check failed: $errors" if $errors;
```

Einfacher Server mit IO::Socket::SSL

```
my $server = IO::Socket::SSL->new(
    LocalAddr => '0.0.0.0:8080',
    Listen => 10,
    SSL_cert_file => 'server-cert.pem',
    SSL_key_file  => 'server-key.pem',
);
my $client = $server->accept;
...
```

Ein Server mit `IO::Socket::SSL` muss nicht wesentlich komplizierter als ein Server mit anderen `IO::Socket`-Modulen sein. Der wesentliche Unterschied ist, dass man das auszuliefernde Zertifikat angeben muss.

Hinter den Kulissen läuft aber auch hier vieles ab:

• Es werden nur sichere Verschlüsselungsverfahren und SSL-Versionen erlaubt. Der Server wird so konfiguriert, dass er die sichersten Ciphern aussucht, d.h. die Auswahl nicht dem Client überlässt (seit 1.956, 11/2013).

• Es werden die notwendigen Einstellungen vorgenommen, um *Forward Secrecy* anzubieten. Bei Forward Secrecy kann eine mitgeschnittene Verbindung nicht nachträglich entschlüsselt werden, selbst wenn das Zertifikat des Servers

kompromittiert wurde (seit 1.956, 11/2013).

• Mit SNI wird die Möglichkeit geboten, auf einer IP-Adresse mehrere Zertifikate zu betreiben. Hierzu muss bei `SSL_cert_file` und `SSL_key_file` ein Hash mit Zertifikaten bzw. Keys angegeben werden und die passende Auswahl erfolgt entsprechend dem vom Client gewünschten Hostnamen (seit 1.83, 2/2013).

• Die Zertifikate können sowohl im *PEM*-Format, *DER*-Format wie auch als *PKCS#12* vorliegen, das Format wird automatisch erkannt (seit 1.988, 05/2014, davor nur PEM).

• Und natürlich kann auch der Server IPv6, sofern die passenden Perl-Module installiert sind.

Wesentliche Unterschiede zu IO::Socket

Bei einfachem Gebrauch verhält sich `IO::Socket::SSL` wie andere Sockets. Wenn man jedoch mit non-blocking Sockets oder eventbasiert (select,poll) arbeitet, muss man die Eigentümlichkeiten von SSL berücksichtigen. Zu diesem Thema gibt es ein separates Kapitel in der Dokumentation von `IO::Socket::SSL`, welches man unbedingt studiert haben sollte.

Wenn man den klassischen fork- oder threadbasierten Server bauen will, bei dem nach einem accept ein neuer Prozess oder Thread die Kontrolle übernimmt, so verhält es sich bei SSL etwas anders als gewohnt. Während bei einfachen Sockets die Verbindung vollständig vom Betriebssystemkernel angenommen wird, wird bei SSL der notwendige Handshake innerhalb der Applikation ausgeführt. Bei einem naiven accept auf dem SSL-Socket wird bei einem langsamen SSL-Client der SSL-Handshake im Hauptprozess ausgeführt und blockiert damit die Annahme neuer Verbindungen. Um das zu vermeiden, sollte man den Handshake erst im neuen Prozess bzw. Thread durchführen (siehe Listing 1).

Module mit Unterstützung für SSL

Am meisten wird SSL für https benutzt. Das gebräuchlichste Modul hierfür ist LWP, welches seit der Version 6.0 (3/2011) standardmäßig `IO::Socket::SSL` benutzt und seit Version 6.06 (04/2014) auch eine Proxies bei https korrekt unterstützt.

Für das Versenden von Mail (SMTP) wird empfohlen `Net::SSLGlue::SMTP` zusammen mit `Net::SMTP` zu benutzen. Zwar gibt es eine Vielzahl anderer Module, die SSL mit SMTP unterstützen. Diese bieten jedoch oft nicht alle für SSL bei SMTP nötigen Möglichkeiten und scheitern oft an der sicheren Nutzung von SSL.

Mit der Version 1.28 der libnet-Bibliothek wird `Net::SMTP` voraussichtlich direkte Unterstützung für SSL aufweisen. Ähnlich sieht es mit `Net::POP3` und `Net::FTP` aus. Bis dahin stehen auch hier entsprechende `Net::SSLGlue` Module zur Verfügung.

Für IMAP gibt es verschiedene Module, von denen keines perfekt SSL unterstützt und teilweise sogar die Validierung der Zertifikate explizit ausgeschaltet wird. Durch Angabe von `SSL_verifycn_schema => 'imap'` kann man aber zumindest `Mail::IMAPClient` fit genug machen.

Daneben gibt es noch eine Vielzahl weiterer Module. Details dazu finden sich in meinem Vortrag auf dem Deutschen Perl-Workshop 2014.

```
# normales accept of TCP-Ebene
my $server = IO::Socket::INET->new(
    LocalAddr => '0.0.0.0:80',
    Listen => 10
) or die "Listen failed: $!";
my $client = $server->accept;

# SSL-Handshake erst im neuen Prozess
my $pid = fork();
if (!$pid) {
    IO::Socket::SSL->start_SSL($client,
        SSL_server => 1,
        SSL_server_cert => 'cert.pem',
        SSL_server_key  => 'key.pem',
    ) or die "Handshake failed: $SSL_ERROR";
}
```

Listing 1

Typische Fehler bei der Nutzung

`IO::Socket::SSL` existiert bereits seit 2000. Jedoch wurde erst in den letzten Jahren der Fokus darauf gesetzt, standardmäßig soviel Sicherheit wie möglich zu bieten. Dazu wurden viele zu lockere Einstellungen der Vergangenheit verschärft. Um diese Bemühungen nicht zunichte zu machen und durch Fehlkonfiguration die Sicherheit zu gefährden, sollte man in einem aktuellen `IO::Socket::SSL` folgende Sachen besser nicht machen:

• Wie bereits beschrieben ist eine Validierung der Gegenstelle essentiell für eine sichere Ende-zu-Ende Verschlüsselung. Daher sollte man in nahezu keinem Fall die Verifikation mittels `SSL_verify_mode` ausschalten. In den Fällen, wo ein Zertifikat nicht erfolgreich validiert werden kann, sollte man lieber die Ursache dafür ermitteln und dann Teilaspekte der Validierung anpassen:

• Ist das Zertifikat von einer CA unterschrieben, die nicht in der Liste der vertrauenswürdigen Zertifizierungsstellen enthalten ist, so sollte man sie dort aufnehmen oder mittels `SSL_ca_file` bzw. `SSL_ca_path` eine passende Liste konfigurieren.

• Benutzt die Gegenstelle ein selbstsigniertes Zertifikat, so kann man über `SSL_fingerprint` den Fingerprint des Zertifikats hinterlegen, sodass dieser als vertrauenswürdig akzeptiert wird (seit 1.980, 04/2014).

• Entspricht der Hostname im Zertifikat nicht dem Zielhost der Verbindung, so kann man über `SSL_verifycn_name` den bei der Überprüfung erwarteten Hostnamen anpassen.

• Mehrere existierende Module schalten stillschweigend die Validierung von Zertifikaten ab, wenn sie die CA nicht an den erwarteten Stellen vorfinden oder der Nutzer keine CA konfiguriert hat.

Da `IO::Socket::SSL` inzwischen auf allen Systemen einen sinnvollen CA-Pfad voreingestellt haben sollte, ist dieses Vorgehen eher schädlich. Mit `default_ca()` kann überprüft werden, welche Voreinstellungen `IO::Socket::SSL` besitzt und diese wenn nötig auch geändert werden.

• Einige Programme und Module setzen explizit die zu nutzende SSL-Version oder konfigurieren die erlaubten Ciphern, vergessen diese dann jedoch dem jeweiligen Stand der Technik anzupassen. So kann es passieren, das vor 10 Jahren die SSL-Version explizit auf das damals aktuelle TLS 1.0 gesetzt wurde, eine derartige Einstellung jedoch heute die Nutzung von TLS 1.1 und TLS 1.2 unterbindet.

Bei den Verschlüsselungsalgorithmen wiederum gibt es Module, die aus Kompatibilitätsgründen die unterstützten Ciphern mit 'ALL' angeben und dabei außer acht lassen, dass damit auch schlechte Ciphern sowie anonyme Ciphern ohne Zertifikatscheck erlaubt sind.

Außer bei sehr spezielle Bedürfnissen ist es daher besser, die Kontrolle `IO::Socket::SSL` zu überlassen, welches standardmäßig Einstellungen für hohe Sicherheit und hohe Kompatibilität benutzt und diese an den jeweiligen Stand der Technik anpasst.

Aktuelle und zukünftige Entwicklungen

Die letzten Monate haben viele Verbesserungen für `IO::Socket::SSL` gebracht. So wurde Unterstützung für OCSP hinzugefügt, die Dokumentation deutlich überarbeitet und Patches für perl-libnet entwickelt, durch die in der nächsten Versionen von `Net::SMTP`, `Net::FTP` u.a. eine einfache Unterstützung für IPv6 und SSL enthalten ist.

Mit `IO::Socket::SSL::Utils` gibt es ein Modul zur einfachen Manipulation und Erstellung von Zertifikaten und mit `IO::Socket::Utils::Intercept` kann man Zertifikate einfach clonen und neu signieren, so wie es für SSL Interception (für böse oder gute Zwecke) notwendig ist.

Resüme

Auch wenn, im Gegensatz zu Python oder Ruby, Unterstützung für SSL kein integraler Bestandteil von Perl ist, so gibt es mit `IO::Socket::SSL` eine einfache Möglichkeit diese Unterstützung nachzurüsten und auf einfache Weise zu nut-

zen. Die meisten Linux-Distributionen, wie auch Strawberry-Perl unter Windows, liefern das Modul mit aus, allerdings oft in älteren Versionen.

Da viele Verbesserungen erst in den letzten Monaten hinzugekommen sind, ist eine Aktualisierung auf eine neue Version empfehlenswert.

Dankeschön

Ein Dankeschön an alle Nutzer des Modules, die durch Fragen, Bugreports und Patches zur Weiterentwicklung beigetragen haben. Ein besonderer Dank mit der Hoffnung auf eine weiterhin gute Zusammenarbeit geht an die Entwickler von `Net::SSLeay`, auf welchem `IO::Socket::SSL` basiert, sowie an die Entwickler von `LWP` und perl-libnet, die sich offen für Änderungen zur besseren Integration von SSL zeigten.

Weitere Informationen

Ich habe auf dem Deutschen Perl Workshop 2014 einen Vortrag zu diesem Thema gehalten, welcher in vielen Fällen über diesen Artikel hinausgeht. Den Vortrag kann man sich unter *http://www.youtube.com/watch?v=dwVIpkBHwOo* anschauen und die Folien sind unter *http://noxxi.de/pws/2014/pws2014-io-socket-ssl.pdf* verfügbar.

Herbert Breunung

Rezension - Muster

William J. Brown, Raphael C. Malveau
Hays W. McCormick III, Thomas J Mowbray
Anti-Patterns
mitp, 304 S.
2.Auflage Juni 2007
(1998 im Original)
ISBN: 3826617746
Softcover: €29.95

Hort Eidenberger
Elke Michlmayr
Mit Perl programmieren lernen
dpunkt.verlag 278 S.
Juni 2005
ISBN: 3-89864-320-4
Paperback: €29 (9)

Nachdem drei Bücher über Reguläre Ausdrücke besprochen wurden, zu Mustern ganz anderer Art.

Anti Patterns

Das ein Softwareprojekt gerade vor den Baum fährt, sehen Menschen mit Erfahrung meist an bestimmten Mustern - Muster im Code, Muster in der Planung, Muster im Denken der Beteiligten. Sie zu erkennen und rechtzeitig dagegen zu lenken gehört zu den wichtigsten Fähigkeiten überhaupt. Deswegen war ich auf das Buch der vier Autoren neugierig.

Außerdem bin ich sehr kritisch gegenüber der Design-Pattern Bewegung, da Denkschablonen auch behindern können. Oder wie Larry auf der letzten YAPC::EU seine Enkelin frei zitierte: "Es gibt keinen Ersatz für's selber nachdenken, und eine Sache wirklich zu verstehen". Und bei den Anti-Pattern geht es auch darum, was alles bei unsachgemäßer Anwendung der berüchtigten Entwurfsmustern schief gehen kann.

Obendrein kommen einige der besten Bücher der Sparte, wie das in der Ausgabe 2/2011 besprochene "Clean Code" von *Robert C. Martin*, aus dem gleichen Verlag und es hat sogar einen Preis gewonnen. Das sind Gründe genug, froher Erwartung zu sein - doch erfüllt wurde sie nur teilweise.

Der Inhalt hielt zwar ein, war der Umschlag versprach: es gab Ursachen, Symptome, Konsequenzen, Lösungsansätze und Beispiele in Code, Klassendiagrammen und Planungscharts. Im vollen Umfang hilfreich war das jedoch nicht immer. Zum einen hätten einige Passagen klarer und knapper sein können. Zum bot es den bekannten Allgemeinplätzen des gesunden Menschenverstandes reichlich Platz. Ja Papa, Cut'n Paste, Spaghetti-Code oder monolithische Klassen sind schlecht. Wenn "Experten" so etwas sagen möchte ich wenigstens noch etwas Zusatzinfos dazu, die ich nicht bereits wusste. Die gab es in Ansätzen, zum Beispiel als es um das psychologische Phänomen ging, dass Mitarbeiter kurz vor Ende eines Projektes, es in einer unterbewussten Panik vor dem Erfolg sabotieren können, aber allzu oft hatte ich den Eindruck, dass die Absätze mit den Lösungsansätzen hätten länger sein können.

Natürlich setzt der schmale Buchrücken bereits vor dem Kauf das erfreuliche Signal, dass hier nicht im Kreis fabuliert wird. Aber wenn immer mal wieder nur auf Standardansätze

verwiesen wird oder dass man Täuschung durch Marketing dadurch vermeidet, in dem das Management mit Menschen von Fach spricht, dann ist mir das etwas mager. Sicher hätte dieser Ratschlag Hunderte Firmen vor der Pleite gerettet, aber in einem Buch hätte ich dazu noch weiterführende Tipps erwartet - Zum Beispiel wie man solche Gespräche effektiv gestaltet oder wie man das daraus gewonnene Wissen am besten speichert. Denn im Abschnitt der genau das Problem behandelt, dass in der Firma niemand mehr weiß was ein Programmteil genau macht, wird lachs auf Konfigurationsmanagement verweisen.

Im Übrigen wird das Antipattern *Lava Flow* genannt, wenn ein Programm alten, nicht mehr benötigten Code enthält, der jedoch aus Angst etwas zu zerstören und fehlendem Wissen darüber nicht entfernt wird. Die Autoren Empfehlen dagegen als erstens "vor der Entwicklung" eine "passende Architektur" vorzuschreiben. Diese aristokratische Haltung halte ich selbst für ein Antipattern, da in der Praxis sich die Architektur sehr wohl ändern kann und manchmal auch muss und man bei der Planung unmöglich alles voraus sieht.

Die Photoshop-Zeichnungen der eingefügten Comics waren zwar treffend, wirken aber auf mich etwas lieblos. Die Vokabeln aus der Softwaretechnik hätte man im Glossar auch noch erklären können und die äußerst praktische Übersicht der gesamten Antipatterns enthielt nur die Nummern der recht großen Kapitel, nicht jedoch die Seitenzahlen wo sie zu finden sind.

Im Großen und Ganzen ist es dennoch ein gutes, lesbares Buch, welches die wesentlichen Probleme der Softwareentwicklung anschaulich anspricht und aus Sicht der Programmierer, Projektleiter und Management wiedergibt. Es hat einen wesentlich weiteren Blickwinkel als das aus Sicht eines Gruppenleiters geschriebene "IT-Projektmanagement" von Matthias Geirhos, dass in der Ausgabe 4/2012 Teil der Rezensionen war. Durch das spürbare Bemühen sich kurz zu fassen langweilt es nicht, dringt aber auch nirgends tiefer ein. Es ist eher hilfreich um sich eigene Gedanken zu machen wie man Projekte leitet ohne die wesentlichen Erfahrungen von Scheitern selbst wiederholen zu müssen.

Für angehende Programmierer wäre erst einmal eine knappe, praktische Anleitung wie "Art of readable Code" *http://www.heise.de/developer/artikel/The-Art-of-Readable-Code-1664655.html* hilfreich (nicht perlspezifisch) und ist die Karre einmal im Schlamm hilft *Perl Medic http://www.perlmedic.com* welches unter anderem nächstes Mal besprochen wird.

Mit Perl programmieren lernen

Das war eigentlich auch schon der Ausblick für das nächste Mal. Weil sämtliche wesentliche Perlbücher in einem Stapel mit den bisherigen Heften des Perl-Magazins zu finden sind, folgt noch ein kleiner Hinweis auf ein Buch des dPunkt-Verlages von 2005, welche nicht die Aufmerksamkeit bekam, die es verdient.

Wer dieses Heft liest weiß was ein I<if> oder eine Variable ist. Es könnte jedoch eine gute Idee sein, für einen vollständigen Programmieranfänger "Mit Perl programmieren lernen" günstig im Netz kaufen. Denn auch wenn Perl sich erfreulicherweise wandelt und man deshalb noch I<Modern Perl> von cromatic (die neue 2014-Edition) daneben legen sollte, sah ich noch nirgends einen Titel, der sich so viel Zeit nimmt um den Frischlingen eine gründlich eine Orientierung zu geben, was Programmieren eigentlich ist und gleichzeitig Perl lehrt. Nicht gegen das allseits zu Recht beliebte Lama (3/2011), aber das lehrt vor allem die Syntax. Das warum erzählt das Kamel (3/2012), ist aber für die Neuen mit sehr viel Ausdauer.

Das hier empfohlene Buch wurde sorgfältig, einfach aber mit Hintergrundwissen geschrieben, was man in der Kombination leider selten sieht. Auch deshalb hier eine Erwähnung ehrenhalber, selbst wenn es die allermeisten Leser der $foo dafür keine Aufmerksamkeit übrig haben.

CPAN News XXXI

Auf CPAN gibt es immer wieder tolle neue Module oder auch Aktualisierungen von bestehenden Modulen. An dieser Stelle werden wieder fünf neue Module bzw. Versionen der letzten Wochen vorgestellt.

DBG

`DBG` ist eine Sammlung von Debugging-Funktionalitäten. Dies soll die Entwickler bei der Fehlersuche unterstützen. Das Modul nutzt dabei viele andere Module, die sich im Laufe der Zeit bewährt haben. Durch die sehr kurzen Funktionsnamen ist es sehr einfach während der Entwicklung mal schnell ein paar mehr Informationen zu bekommen.

Wichtige Funktionen:

```
trace - trc:

sub foo   { bar() }
sub bar   { trc }
foo();

# TRACE
# 1) main::bar (test.pl:9)
# 2) main::foo (test.pl:8)
# END TRACE
```

Woher kommt eigentlich die Methode? Diese Frage kann `cnm` (codename) beantworten:

```
# LWP::UserAgent::request
cnm $ua->can('request');
```

Warum wird immer mehr Speicher verbraucht? Wird nichts freigegeben? Ein Grund könnten zyklische Referenzen sein. Um diese aufzuspüren, kann die Funktion `cyc` (cycles) verwendet werden (siehe Listing 1).

Aber das Modul hat noch viele weitere Funktionen - ein Blick lohnt sich auf jeden Fall

```
my $a = {};
my $b = { b => $a };
my $c = { c => $b };
$a->{a} = $c;
cyc $a;

# HASH (140416464744656 <- base)
#    HASH (140416464745304 <- 140416464744656)
#       HASH (140416464744992 <- 140416464745304)
#          HASH (140416464744656 <- 140416464744992) -- ref count: 2
```

Listing 1

MySQL::Explain::Parser

Parst das Ergebnis des `EXPLAIN`-Kommandos in MySQL. Mit Explain bekommt man Informationen über die Ausführung einer Abfrage. Auf diesem Weg kann man der Ursache von langsamen Abfragen auf die Schliche kommen. Mit diesem Modul bekommt man die Informationen als Array von Hashes und man kann die Informationen besser durchsuchen; auch als Basis für andere Darstellungsformate kann die geparste Datenstruktur dienen.

```
use utf8;
use MySQL::Explain::Parser qw/parse/;

my $explain = <<'...';
 +----+-------------+-------+-------+---------------+---------+---------+------+------+--------
--+-------------+
 | id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows |
filtered | Extra       |
 +----+-------------+-------+-------+---------------+---------+---------+------+------+--------
--+-------------+
 |  1 | PRIMARY     | t1    | index | NULL          | PRIMARY | 4       | NULL | 4    | 100.00
|             |
 |  2 | SUBQUERY    | t2    | index | a             | a       | 5       | NULL | 3    | 100.00
| Using index |
 +----+-------------+-------+-------+---------------+---------+---------+------+------+--------
--+-------------+
 ...

 my $parsed = parse($explain);
 # =>
 #    [
 #        {
 #            'id'            => '1',
 #            'select_type'   => 'PRIMARY',
 #            'table'         => 't1',
 #            'type'          => 'index',
 #            'possible_keys' => undef,
 #            'key'           => 'PRIMARY',
 #            'key_len'       => '4',
 #            'ref'           => undef
 #            'rows'          => '4',
 #            'filtered'      => '100.00',
 #            'Extra'         => '',
 #        },
```

Devel::Trace::Syscall

Möchte man bei allen Systemaufrufen einen Stacktrace bekommen möchte, kann man `Devel::Trace::Syscall` nehmen:

```
$ perl -d:Trace::Syscall=open uebung1.pl
open("/tmp/test.t", 0x241, 0666) =
                3 at uebung1.pl line 10.
```

Hier wird für jeden `open`-Aufruf ein Stacktrace ausgegeben. In diesem Fall ist das ein `open` direkt im Skript. Wandert der Aufruf in eine Subroutine sieht das so aus:

```
$ perl -d:Trace::Syscall=open uebung1.pl
open("/tmp/test.t", 0x241, 0666) =
   3 at uebung1.pl line 11.
   main::test() called at uebung1.pl line 14
```

CHI::Cascade

Eine Art `make` für Caches bietet `CHI::Cascade`. Damit können Caches in Abhängigkeit gesetzt werden:

```
use CHI;
use CHI::Cascade;

$cascade = CHI::Cascade->new(
   chi => CHI->new(...));

$cascade->rule(
   target  => 'unique_name',
   depends => ['unique_name_other1',
               'unique_name_other2'],
   code    => sub {
       my ($rule, $target_name,
             $values_of_depends) = @_;

       # Now we can calcualte $value
       return $value
   },
   params  => { a => 1, b => 2 }
);

$cascade->rule(
   target  => 'unique_name_other1',
   depends => 'unique_name_other3',
   code    => sub {
       my ($rule, $target_name,
            $values_of_depends) = @_;
       return $value;
   }
);

$value_of_this_target =
  $cascade->run('unique_name');
```

Die erste Regel sagt aus, dass der Cache für *unique_name* von zwei anderen Caches abhängt (`depends`). In der Subroutine, die dem Parameter `code` übergeben wird, wird festgelegt, wie der Wert des Caches berechnet wird. Als Parameter bekommt man die Regel an sich, den Cachenamen und die Werte der Abhängigkeiten übergeben.

```
 $values_of_depends == {
      unique_name_other1 => $value_1,
      unique_name_other2 => $value_2
  }
  $rule->target     eq     $target_name
  $rule->depends    ===    [
    'unique_name_other1',
    'unique_name_other2']
  $rule->dep_values ==
     $values_of_depends
  $rule->params     ==    { a => 1, b => 2  }
```

Der Einsatz des Moduls eignet sich vor allem dann wenn man aufwändige Berechnungen cachen möchte und diese Berechnung in konkrete Einzelschritte aufgeteilt werden kann.

Data::Validator::Recursive

Mit `Data::Validator` kann man wunderbar Daten validieren. Aber leider funktioniert das nur mit flachen Datenstrukturen. Möchte man verschachtelte Strukturen validieren, kommt `Data::Validator::Recursive` in Spiel.

```
use Data::Validator::Recursive;

# create a new rule
my $rule = Data::Validator::Recursive->new(
   foo => 'Str',
   bar => { isa => 'Int' },
   baz => {
     isa  => 'HashRef', # default
     rule => [
     hoge => { isa => 'Str', optional => 1 },
           fuga => 'Int',
       ],
   },
);
```

Hiermit validiert man eine Hashreferenz, wobei der Wert zum Schlüssel *foo* ein String, der Wert zum Schlüssel *bar* ein Integer sein muss. Soweit ist das auch mit `Data::Validator` kein Problem. Dann kommt aber der interessante Teil: Zum Schlüssel *baz* wird wieder eine Hashreferenz gespeichert, für die es eine eigene Regel gibt. Denn zum Schlüssel *hoge* innerhalb dieser zweiten Hashreferenz muss - wenn vorhanden - ein String gespeichert werden.

Somit müsste eine gültige Datenstruktur z.B. so aussehen:

```
 my $input = {
   foo => 'hoge',
   bar => 1192,
   baz => {
       hoge => 'kamakura',
       fuga => 1185,
   },
};
```

Und zur Validierung wird dann noch dieser Code benötigt:

```
 my $params = $rule->validate($input) or
     croak $rule->error->{message};
```

Termine

August 2014

05. Treffen Hannover.pm
Treffen Frankfurt.pm
06. Treffen Niederrhein.pm
07. Treffen Dresden.pm
18. Treffen Erlangen.pm
19. Treffen Hannover.pm
22.-24. YAPC::Europe
23.-24. FrOSCon
27. Treffen Berlin.pm
28.-30. YAPC::Asia

September 2014

02. Treffen Hannover.pm
Treffen Frankfurt.pm
04. Treffen Dresden.pm
05.-06. Schweizer Perlworkshop
10. Treffen Niederrhein.pm
13. Dynamic Languages Conference
15. Treffen Erlangen.pm
16. Treffen Hannover.pm
24. Treffen Berlin.pm
30. Treffen Hannover.pm

Oktober 2014

02. Treffen Dresden.pm
06.-09. Perl::Dancer::Conference
07. Treffen Frankfurt.pm
08. Treffen Niederrhein.pm
09.-10. code.talks 2014
10.-13. Österreichischer Perl-Workshop
14. Treffen Hannover.pm
20. Treffen Erlangen.pm
28. Treffen Hannover.pm
29. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss.

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perlfoundation.org

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

http://www.Perl.org

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

http://perl-academy.de

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