Ausgabe 4/2014 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Websockets mit Mojolicious
Das Hypethema im Bereich Webanwendungen

PowerShell
Was die Windows PowerShell mit Perl verbindet

WWW::Mechanize::PhantomJS
Web-Automation

Nr 32


$foo
Perl-Magazin
Perl und Win32
Datenbanken
Perl 5.10
Perl
Magazin
Testen mit Perl


$foo
Perl-Magazin
Perl 5.10 Advanced Regular Expressions
Perl::Critic
FAQ mit Catalyst
Portrait


$foo
PERL MAGAZIN
Perl 6
AppleTerl
LDAP
Nr 03


$foo
PERL MAGAZIN
Moose
Caching in Webapplikationen
VDEP mit Net::Sip
Nr 04


$foo
PERL MAGAZIN
Nr 05


$foo
PERL MAGAZIN
Threads
TWiki
SANE Backend
Nr 06


$foo
PERL MAGAZIN
Scaffolding
Debugger Einführung
Nr 07


$foo
PERL MAGAZIN
CPAN::Mini
Google Summer of Code
Logging für Perl-Programme
Nr 08


$foo
PERL MAGAZIN
CGI::IDS
Videomixer in Perl
F*EX
Nr 09

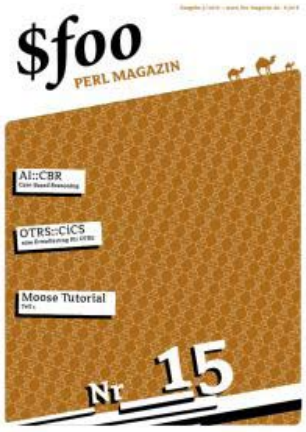

$foo
PERL MAGAZIN
Nr 10


$foo
PERL MAGAZIN
Nr 11


$foo
PERL MAGAZIN
DPath ist nicht XPath!
Methoden-Signaturen
WxPerl-Tutorial
Nr 12


$foo
PERL MAGAZIN
Performance großer Webseiten
Visualisierungen
Lotus Notes/Domino
Nr 13


$foo
PERL MAGAZIN
XMLSocket Klasse
Devel::REPL
Komplexe Datenstrukturen mit Data::Dumper
Nr 14


$foo
PERL MAGAZIN
AI::CBR
OTRS::CiCS
Moose Tutorial
Nr 15


$foo
PERL MAGAZIN
Zentyal
Simple DirectMedia Layer
How-To: App::perlbrew
Nr 16

VORWORT

Danke, danke, danke!

Nun ist es soweit - die allerletzte Ausgabe von $foo liegt vor Ihnen. 8 Jahre Perl-Magazin. Und es wird Zeit nochmal abschließend "Danke" zu sagen. Danke an Sie, denn ohne Leser ist ein Magazin ziemlich sinnlos. Ohne Leser macht man sich keine Arbeit für einen Artikel zu recherchieren und Code auszuprobieren.

Einige Leser sind im Laufe der Zeit dann auch mal zum Autor geworden und haben für ein Anwachsen des Perl-Wissens gesorgt. Es war toll wenn ich selbst in "meinem" Magazin lesen und lernen durfte. Deshalb ein Dank an alle Autoren, die ich hier nicht alle aufzählen kann. Aber es waren insgesamt 70 Autoren die mir geholfen haben die 50-60 Seiten pro Ausgabe mit Leben zu füllen.

Insgesamt wurden in den 32 Ausgaben von $foo über 340 Artikel geschrieben - in den unterschiedlichsten Bereichen. Ich denke, da war für jeden etwas dabei.

Ich habe seit den Anfängen 2007 viel dazugelernt. So waren die ersten beiden Ausgaben vom Layout her nicht so gut und speziell in der ersten Ausgabe habe ich etliche Fehler gemacht, die unter Magazinmachern wohl zu den "Todsünden" gehören. Nach etlichen Stunden bei //SEIBERT/MEDIA wurde die zweite Ausgabe schon deutlich besser und seit der Ausgabe 3 bin ich zufrieden mit dem Layout.

Heute sehe ich Printprodukte mit deutlich anderen Augen als vor 8 Jahren.

Ich habe auch gelernt, dass es für kleine Magazine unmöglich ist in Buchhandlungen zu kommen weil der Presse-Grosso für eine Zeitschrift wie $foo viel zu teuer ist.

Dank $foo habe ich viele neue und interessante Leute kennengelernt, ich hoffe dass ich den Kontakt auf Konferenzen halten kann.

Jetzt bleibt mir - mit einem lachenden und einem weinenden Auge - ein letztes Mal viel Spaß bei der Lektüre zu wünschen.

Renée Bäcker

Die Codebeispiele dieser Ausgabe können mit dem Code

pm3r9t

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/issue32

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

NICHT - PERL

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.

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.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996.

Websites:

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

Frank Fuhrmann

Seit über 10 Jahren arbeite ich nun bereits in den Bereichen Systemadministration, System Engineering, DevOps und IT-Consulting. In dieser Zeit bekam ich mit vielen verschiedenen Technologien vor allem im Bereich der Linux-basierten Web-Plattformen zu tun, auf die ich mich mittlerweile auch spezialisiert habe. Das heißt, dass Linux (diverse Distribution), BSD, Solaris, Apache, Tomcat und MySQL zu meiner täglichen Arbeit einfach dazu gehören. Daneben gibt es natürlich noch Dinge wie CI-Umgebungen, Monitoring (bevorzugt auf Basis von Nagios), Backups, Loadbalancer uvm., die mich im Alltag begleiten. Meine bevorzugten Programmiersprachen sind Perl und PHP, aber auch Java, C/C++ und Bash sind mir geläufig. Mit Assembler und BASIC habe ich durch private Projekte auf Basis von Raspberry Pi und RiscOS immer mal wieder zu tun.

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.

Max Maischein

Max Maischein ist Baujahr 1973 und studierter Mathematiker. Seit 2001 ist er für die DZ BANK in Frankfurt tätig und betreut dort den Fachbereich Operations und Services im Prozess- und Informationsmanagement.

Renée Bäcker

Websockets mit Mojolicious

Websockets sind ein Hypethema im Bereich Webanwendungen. Mit Websockets können "Echtzeitanwendungen" umgesetzt werden. Ein typisches Beispiel ist der Chat. Wir wollen aber eine kleine Monitoringseite für einen Server schreiben.

Aber was sind Websockets überhaupt?

Websockets ist ein Protokoll, das entworfen wurde, um eine bi-direktionale Verbindung zwischen einer Webanwendung und einem Client zu ermöglichen. Das Protokoll basiert auf TCP und wurde in HTML5 integriert. Mittlerweile unterstützen auch alle modernen Browser Websockets. Möchte man aber auch ältere Browser bedienen, so muss man sich um ein Fallback kümmern. Die Fallbacks sollen hier aber kein Thema sein, nur ein kleiner Tipp dazu: Mit socket.io kann man sich der umständlichen Arbeit mit verschiedenen Fallbacks entledigen. Socket.io erkennt automatisch, welche Alternative verwendet werden kann, wenn Websockets nicht möglich sind.

Möchte man in einer Webanwendung dem Client immer wieder aktuelle Daten übermitteln, muss man mit bisherigen Standardmitteln entweder mit *Polling* oder *Long Polling* arbeiten. Dabei hält der Server eine Verbindung sehr lange offen wenn keine Daten vorliegen - anstatt einfach eine leere Antwort zu schicken. Wenn dann Daten für den Client vorliegen, schickt der Server sie und schließt damit den HTTP/S-Request ab.

Mit dem *Long Polling* kann man vermeiden, dass der Client in sehr kurzen Abständen beim Server nachfragen muss ob irgendwelche Daten vorliegen. Jede Anfrage bringt einen gewissen Overhead mit, weil die Verbindungen aufgebaut werden und etliche Informationen mitgeschickt werden müssen.

Mit Websockets wird die Verbindung einmal aufgebaut und bleibt dann bestehen. So können sich Client und Server gegenseitig zu jeder Zeit Nachrichten zuschicken. Websockets funktionieren auch über Domaingrenzen hinweg - im Gegensatz zu AJAX-Requests.

Technisch gesehen in ein Websocket-Request nichts anderes als ein aufgewerteter *GET*-Request. Der Ablauf einer Websocket-Verbindung ist in Abbildung 1 dargestellt.

Werfen wir mal einen Blick auf die Header, die beim Verbindungsaufbau mitgeschickt werden. Im Folgenden ist die Anfrage des Clients zu sehen:

```
User-Agent: Mozilla/5.0 [...] Firefox/29.0
Upgrade:            websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: ZUu04F2KlNt5ScTibzVdyg==
Pragma:             no-cache
Origin:             http://localhost:3000
Host:              localhost:3000
Connection:         keep-alive, Upgrade
Cache-Control:          no-cache
Accept-Language:    de,en-US;q=0.7,en;q=0.3
Accept-Encoding:    gzip, deflate
Accept:             text/html
```

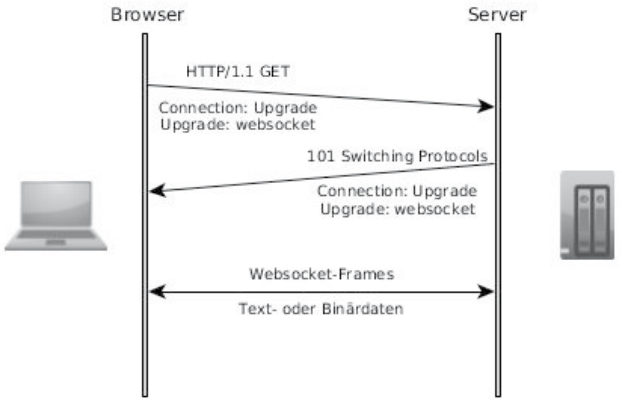


Abbildung 1: Websocket-Verbindung

Ein paar Begriffserklärungen hierzu:

• Origin
Feld muss mitgeschickt werden, Server kann damit Cross-Domain Anfragen handeln

• Sec-WebSocket-Key
zufälliger 16-Byte-Wert (Base64 kodiert)

• Sec-WebSocket-Protocol
optional, Unterstützung von Subprotokollen. Mittlerweile gibt es eine ganze Reihe von möglichen Subprotokollen, z.B. *WAMP* (*http://wamp.ws*) oder *AMQPWSB10* (Websocket Transport für AMQP 1.0).

• Sec-WebSocket-Extension
optional, Protokoll Erweiterung von Client unterstützt.

Was weiterhin in den Headern auffällt sind zwei Felder:

```
Upgrade:     websocket
Connection:  keep-alive, Upgrade
```

Wie oben schon kurz erwähnt wird die Websocketverbindung mit einem GET-Request aufgebaut. Das *Upgrade*-Feld wurde mit HTTP/1.1 eingeführt. Ein weiteres Einsatzgebiet für das Feld ist die Verwendung von *Transport Layer Security* (TLS). Der Client beginnt also mit einer normalen Klartext-Anfrage, die später auf eine neuere HTTP-Version oder ein anderes Protokoll umgestellt wird.

Prinzipiell können mehrere mögliche Protokolle angegeben werden:

```
Upgrade:  HTTP/2.0, SHTTP/1.3, IRC/6.9
```

Der Server kann dann prüfen welche Protokolle er versteht und kann dann umswitchen (wie, werden wir bei den Headern der Antwort sehen).

Diese Upgrade-Angabe gilt aber immer nur für die aktuelle Anfrage, deswegen muss das Schlüsselwort *Upgrade* im *Connection*-Feld auftauchen.

Die Header der Antwort sind in Listing 1 zu sehen.

Und der Status-Code ist `HTTP/1.1 101 Switching Protocols`.

Auch hier erst ein paar Begriffserklärungen:

• Sec-WebSocket-Accept
- Sec-WebSocket-Key verknüpft mit einer GUID verknüpft werden -> SHA1-Hash -> Base64
- Stellt sicher (Client), dass der Server die Anfrage verstanden hat.
- Ein falscher Wert wird als serverseitige Ablehnung verstanden

• Sec-WebSocket-Protocol
optional. Max. 1 Eintrag aus der Auswahl vom Client

• Sec-WebSocket-Extension
optional

Ist die Verbindung aufgebaut, kann man mit den Websockets arbeiten. Diese haben ein zwei Attribute, die Informationen zu den Websockets beinhalten. Das ist zum einen das Attribut `readyState`, das Informationen zum Status der Verbindung gibt. Das zweite ist ein Attribut, auf das nur lesend zugegriffen werden kann.

Folgende Werte sind für das Attribut erlaubt:

0: Verbindung wurde noch nicht aufgebaut
1: Verbindung aufgebaut, Kommunikation möglich
2: Befindet sich im *closing handshake*
3: Verbindung geschlossen oder nicht möglich

Da in der Regel Timeouts für die Verbindungen existieren, sollte man immer wieder den Status der Verbindung prüfen und ggf. neu mit dem Server verbinden.

```
Upgrade:              websocket
Server:               Mojolicious (Perl)
Sec-WebSocket-Accept: /3TMv7qMvMVspKwbO9nq2cYDiCk=
Date:                 Thu, 05 Jun 2014 22:46:55 GMT
Content-Length:       0
Connection:           Upgrade
```

Listing 1

Da die Kommunikation asynchron ist, muss man mit Events arbeiten um auf Nachrichten etc. reagieren zu können. Websockets kennen vier solcher Events:

• open
Mit einem Callback auf `Socket.onopen` kann man darauf reagieren wenn die Verbindung aufgebaut wurde.

• message
Mit `onmessage` kann man mit den Nachrichten arbeiten.

• error
Sollen Fehler behandelt werden, muss man auf mit einem Callback für `Socket.onerror` darauf reagieren.

• close
Wird die Verbindung geschlossen, wird das `close`-Event gefeuert. Dafür kann ein Callback für `Socket.onclose` definiert werden.

Ansonsten gibt es noch zwei Methoden: Mit `send` werden Nachrichten an den Server geschickt und mit `close` die Verbindung beendet.

Auf Clientseite muss alles mit JavaScript gemacht werden, auf der Serverseite gibt es viele Möglichkeiten. In diesem Artikel werden die Websockets mit Mojolicious behandelt. In Mojolicious ist ein Handling von Websockets nativ enthalten. Man braucht keine zusätzlichen Module oder Plugins.

Unsere Beispielanwendung soll nur relativ klein sein. Auf einem Server soll eine kleine Mojolicious-Anwendung sein, weswegen wir `Mojolicious::Lite` einsetzen. Der Code der Aktionen funktioniert genauso aber auch in einer größeren Mojolicious-Anwendung.

Besucht der Benutzer die Startseite der Anwendung bekommt er gleich ein paar Informationen aufgelistet. Um die Beispielcodes klein zu halten, wird hier nur eine reine Textdarstellung genommen und nicht irgendwelche schönen Grafiken.

Beginnen wir also mit einem `Mojolicious::Lite`-Gerüst:

```
#!/usr/bin/perl

use Mojolicious::Lite;

# more code comes here

app->start;
```

Als nächstes eine einfache Seite, auf der die Infos ausgegeben werden. Dazu brauchen wir eine Route und ein Template. Auf ein Layout verzichten wir hier der Einfachheit halber.

Folgender `__DATA__`-Bereich muss dann hinzugefügt werden:

```
__DATA__

@@monitor.html.ep
<!DOCTYPE html>
<html>
 <body>
   <table>
     <tr>
       <td>CPU-Load:</td>
       <td id="load"></td>
     </tr>
     <tr>
       <td>Uptime:</td>
       <td id="up"></td>
     </tr>
     <tr>
       <td>Freeram:</td>
       <td id="free"></td>
     </tr>
     <tr>
       <td>Totalram:</td>
       <td id="total"></td>
     </tr>
   </table>
   <input type="text" name="info_input"
          id="info_input" /><br />
   <div id="info"></div>
 </body>
</html>
```

Und wir brauchen noch eine Route.

```
get '/' => sub {
   shift->render( 'monitor' );
};
```

Soweit, so gut. In Abbildung 2 sieht man wie die Seite dann aussieht.

Bis jetzt ist noch nichts mit Websockets im Spiel. Das werden wir erst nach und nach einführen. Wie bereits erwähnt ist auf Clientseite JavaScript notwendig. Für die angenehmere Arbeit - wenn auch hier nicht unbedingt notwendig - verwenden wir hier jQuery. Außerdem brauchen wir noch Java

```
<head>
  <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"
     type="text/javascript"></script>
  <script src="/foo.js"></script>
</head>
```

Listing 2

vaScript, das die Websocket-Verbindung initiiert und die Nachrichten vom Server auswertet.

Das Template bekommt also noch einen `head`-Bereich (Listing 2)

Das *foo.js* ist unser eigenes JavaScript (Listing 3). Der Code kommt auch in den `__DATA__`-Teil mit einem vorangestellten `@@foo.js`. Hinweis: Sollen "statische" Inhalte mit Mojolicious::Lite aus dem `__DATA__`-Bereich ausgeliefert werden, dürfen die nur einen Dateisuffix (z.B. *.js*) haben. Man kann also nicht

```
@@jquery.min.js
...
```

machen, da das von Mojolicious::Lite als Template interpretiert werden würde.

In der Variablen `ws` wird die Websocket-Verbindung gehalten. Wenn das Dokument fertig geladen ist, wird mit `ws = new WebSocket( ws_url );` diese Verbindung aufgebaut. Danach kommen die Eventhandler. Im Falle eines Fehlers soll einfach eine Meldung auf der (JavaScript-)Konsole ausgegeben werden. Wenn die Verbindung aufgebaut ist, soll ebenfalls eine Meldung in der Konsole auftauchen.

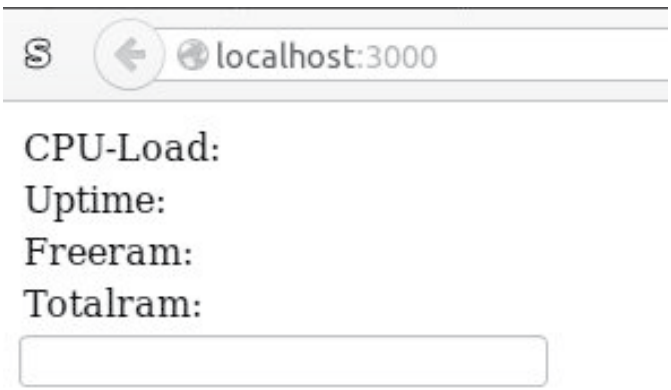


Abbildung 2: Eine einfache Seite für die Monitoringausgabe

Der wichtigste Handler ist hier der `onmessage`-Handler. Hier steht das drin, was gemacht werden soll wenn eine Nachricht vom Server kommt. Hier kommt einfach eine JSON-Struktur (siehe Listing 4) an, wovon die ganzen Informationen einfach in der HTML-Tabelle angezeigt werden soll.

Dann gibt es auf der Seite noch ein Eingabefeld. Sobald dort etwas eingegeben wird, wird die Nachricht an den Server geschickt (über die Methode `send` in dem `bind`). Der Server schickt diese Meldung an alle Clients. Das kann dann dazu verwendet werden, den Kollegen mitzuteilen, dass man neue Software auf dem Server installiert.

Kommen wir zum interessanteren Teil - der serverseitige Code. Wir brauchen jetzt eine Route, das von dem `ws = new WebSocket( ws_url );` angesteuert wird. Als Schlüsselwort steht hier `websocket` zur Verfügung.

```
var ws;

$(document).ready(function() {
   ws = new WebSocket( 'ws://server:3000/' );

   ws.onerror = function() {
       console.log( "an error occured" );
   };

   ws.onopen = function() {
       console.log(
           "ws connection established" );
   };

   ws.onmessage = function(msg) {
       var res = JSON.parse( msg.data );

       if ( res["info"] ) {
         $('#info').text( res["info"] );
         return;
       }

       $('#load').text( res["info"] );
       $('#up').text( res["info"] );
       $('#free').text( res["title"] );
       $('#total').text( res["title"] );
   };
});

$('#info_input').bind( 'change', function() {
  ws.send( $('#info_input').val() );
});
```

Listing 3

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

Auf eines ist hier zu achten: Diese Route muss vor dem `get '/' => sub {}` kommen. Da der Verbindungsaufbau für Websockets nur ein *GET*-Request ist, würde bei der falschen Reihenfolge eben diese `get`-Route zuschlagen.

In der Aktion an sich müssen wir zuerst einen Stream erzeugen, weil die Verbindung ja offen bleiben muss. Mojo kommt mit einer eigenen Eventloop, die wir hierfür verwenden können:

```
Mojo::IOLoop->stream(
    $self->tx->connection
)->timeout( 300 )
```

Für die aktuelle Verbindung (`$self->tx->connection`) wird dieser Stream erzeugt. Damit dieser nicht gleich wieder geschlossen wird, setzen wir ein Timeout von 300 Sekunden.

Da die Nachrichten aus dem Eingabefeld an alle Clients geschickt werden sollen, müssen wir uns die Clients merken. Dazu brauchen wir einen "globalen" Hash. Hier `%clients`.

```
my %client;
websocket '/' => sub {
  my $self = shift;
};
```

Für das Merken der Clients verwenden wir folgenden Code:

```
my $id = $self->tx =~ /(0x\w+)/ && $1;

$client{$id} = $self;
```

Als nächstes müssen wir die Nachrichten entgegennehmen und an alle Clients schicken.

```
$self->on( message => sub {
    my ($ws, $msg) = @_;

    for my $client_id ( keys %client ) {
        my $json = Mojo::JSON->encode({
           info => $msg,
        });

        $self->app->log->debug( $json );
        $client{$client_id}->send( $json );
    }
});
```

Wenn die Clients die Verbindung schließen, sollte diese aus der Liste der Clients gelöscht werden:

```
$self->on( finish => sub {
    delete $client{$id};
});
```

Das hier das Event nicht *close* heißt, wie es bei den Websockets genannt wird, liegt daran, dass hier die Mojo-eigene Eventloop verwendet wird. Dort heißt das Event *finish*.

Bleibt noch, den Clients regelmäßig die Monitoringinformationen zu schicken. Das soll in dieser Anwendung alle drei Sekunden passieren. Hier erzeugen wir mit `Mojo::IOLoop` einen Timer. Hier wird die Nachricht aber nicht an alle Clients geschickt sondern immer nur an den Client der die Verbindung aufgebaut hat.

```
my $scheduler =
 Mojo::IOLoop->recurring( 3 => sub {
   my $si   = sysinfo;
    my $json = Mojo::JSON->encode({
      uptime   => $si->{uptime},
      load     => $si->{load1},
       freeram  => $si->{freeram},
       totalram => $si->{totalram},
    });

    $self->app->log->debug(
       "Sending data: ".$json);
    $self->send( $json );
});
```

Was noch fehlt: Bis jetzt wird die Verbindung mit dem Timeout unterbrochen. Der Client sollte also regelmäßig etwas über die Leitung schicken, damit die Verbindung erhalten bleibt. Im Code, der zum Download bereitsteht, ist diese Änderung bereits enthalten.

Zum guten Schluss sollen die Websocketverbindungen getestet werden, um sicherzustellen, dass der Server auf bestimmte Nachrichten richtig reagiert und auch regelmäßig Daten schickt. Was hierbei nicht getestet wird ist das JavaScript.

Für das Testen von Websockets, stellt Mojolicious auch einige Methoden bereit - in `Test::Mojo`.

Als erstes sollte man testen, ob eine Verbindung aufgebaut werden kann und ob der richtige Status-Code zurückgeliefert wird. Wir erinnern uns, dass bei einem erfolgreichen Verbindungsaufbau der Status `101` zurückgeliefert wird, weil ein Upgrade des Protokolls vorgenommen wird.

```
use Test::More;
use Test::Mojo;

do './server';

my $t  = Test::Mojo->new();
my $ws = $t->websocket_ok( '/' );

$ws->status_is( 101 );

done_testing();
```

Das `do './server'` ist notwendig weil `Mojolicious::Lite` verwendet wird. Ohne diesen Befehl weiß `Test::Mojo` nicht, welche Anwendung gestartet werden soll. Bei Mojolicious-Anwendungen kann man `new` den Anwendungsnamen übergeben.

Der nächste Schritt ist es, das Schicken der Nachrichten an den Server zu testen und zu prüfen ob der Server auch die richtige Nachricht zurückschickt.

```
$ws->send_ok( 'tester2' );
$ws->message_ok();
$ws->message_is( '{"info":"tester2"}');
```

Mit `send_ok` wird geprüft, ob das Senden an sich fehlerfrei funktioniert hat, während `message_ok` einfach nur prüft, ob eine Nachricht vom Server ankam. Ob diese Nachricht auch richtig aussieht, kann man mit `message_is` prüfen. Weiß man nicht genau wie die Nachricht aussieht, kann man mit `message_like` arbeiten. Wie bei `Test::More` gibt es auch noch die umgekehrten Funktionen. In diesem Fall `message_isnt` und `message_unlike`.

Abschließend wird mit

```
$ws->finish_ok;
```

getestet ob die Verbindung ordnungsgemäß geschlossen werden kann.

Jetzt geht es noch darum, die Monitoringnachrichten zu testen.

```
my $monitoring = $t->websocket_ok('/');
sleep 4;
$monitoring->message_ok;
$monitoring->json_message_has('/uptime');
$monitoring->json_message_has('/load');
$monitoring->json_message_has('/freeram');
$monitoring->json_message_has('/totalram');
$monitoring->finish_ok;
```

Da die Monitoringinformationen nur alle drei Sekunden wird erstmal vier Sekunden gewartet. Mit `json_message_has` kann festgestellt werden, ob eine JSON-Struktur ein bestimmtes Element hat. Da wir JSON verwenden, können wir diese Methode verwenden. Die JSON-Nachricht des Servers wird automatisch in eine Perl-Datenstruktur umgewandelt. Als Parameter kann man einen XPath-angelehnten Selector übergeben.

Thomas Fahle

Boilerplatecode mit Import::Into reduzieren

Wenn man mal so darüber nachdenkt, wie oft man als Perlprogrammierer (m/w) etwas wie

```
package Foo;
use strict;
use warnings;
use Package::A qw/x y z/;
use Package::B qw/u v w/;
```

in den Kopfteil eines Moduls per copy and paste eingefügt hat, dann fragt man sich, ob sich dieser Vorgang nicht deutlich vereinfachen lässt.

Leider ist das nicht ganz so einfach.

Import Basics

Wenn ein Modul mit `use` eingebunden wird, dann wird zunächst das Modul mit `require` geladen und anschließend die Methode `import` des Moduls aufgerufen.

```
use Foo LIST;
```

ist also äquivalent zu

```
BEGIN {
   require Foo.pm;
   Foo->import( LIST );
};
```

Die Methode `import` ist für das Ein- oder Ausschalten verschiedener pragmas bzw. Compileroptionen und das Kopieren von Modulen, Funktionen und Variablen in den Namensraum (namespace) des Aufrufers zuständig.

Keine Import Standards

Es gibt keine Standards für die Implementierung der Methode `import` - man kann z.B. `Exporter`, `Sub::Exporter`, `Exporter::Declare` oder `Moose::Exporter` verwenden.

Und dann gibt es ja auch noch pragmas.

Das Schreiben eines eigenen Importers kann daher schwierig bis schmerzhaft sein.

Import::Into als einheitliche Schnittstelle

Import::Into von Matt S. Trout, Graham Knop und Christian Walde bietet eine einheitliche und einfach zu bedienende Schnittstelle für den Import von Packages und pragmas in andere Packages.

Der Vorgang ist nicht ganz trivial, die Autoren erklären die technischen Details aber ausführlich in der Dokumentation des Moduls.

Import::Into Beispiele

Nachfolgend zwei einfache Beispiele, die den Einstieg in **Import::Into** erleichtern sollen.

Weitere Funktionen verrät ein Blick in die Dokumentation des Moduls.

Beispiel Pragmata importieren

In diesem Beispiel soll die Pragmata `strict` und `warnings` aktiviert werden. Weiterhin soll der Pfad zu lokal installierten CPAN-Modulen via `lib` erweitert werden.

Die Funktionalität wird in einem eigenen Modul zwecks Wiederverwendung gekapselt.

Zunächst werden die gewünschten Pragmata geladen und ggf. initialisiert.

Anschließend importiert eine **eigene** `import` Funktion diese mittels `import::into` in den Namensraum des Aufrufers.

```
package MySetup;
use strict;
use warnings;

use lib '/opt/extlib/perl/lib/perl5';

use Import::Into;

sub import {
   my $target = caller;

   strict   ->import::into( $target );
   warnings ->import::into( $target );
   lib      ->import::into( $target );
}

# Return true - it's a package
1;
```

Dieses Modul wird einfach in ein Programm oder Modul mittels `use` eingebunden.

```
#!/usr/bin/perl

use MySetup;
```

oder

```
package MyApp;

use MySetup;
```

Die Pragmata `strict`, `warnings` und `lib` sind nun aktiviert, wie das folgende simple Beispiel zeigt.

```
#!/usr/bin/perl

use MySetup;

print "Perl version $]\n\n";

foreach my $inc (@INC) {
       print "$inc\n";
}

# Yields a warning
my $x = "2:" + 3;
```

Ausgabe des Beispielprogramms :

```
Perl version 5.018002

/opt/extlib/perl/lib/perl5/
   x86_64-linux-gnu-thread-multi
/opt/extlib/perl/lib/perl5
/etc/perl
/usr/local/lib/perl/5.18.2
/usr/local/share/perl/5.18.2
/usr/lib/perl5
/usr/share/perl5
/usr/lib/perl/5.18
/usr/share/perl/5.18
/usr/local/lib/site_perl
.

Argument "2:" isn't numeric in addition
 (+) at ...
```

Beispiel Module importieren

Um Module und deren Funktionen mittels **Import::Into** zu importieren, werden die Module zunächst geladen.

Module, die Funktionen exportieren, werden mit einer **leeren** Importliste geladen. Die leere Importliste soll verhindern, dass Funktionen in das `Setup` Modul importiert werden.

Anschließend werden die gewünschten Funktionen explizit in den Namensraum des Aufrufers importiert.

Objektorientierte Module, die ja keine Funktionen exportieren, werden einfach komplett importiert.

```
package MySetup;
use strict;
use warnings;

use lib '/opt/extlib/perl/lib/perl5';

use Import::Into;
use List::MoreUtils ();
use Encode ();
use Moose;

sub import {
   my $target = caller;

   strict     ->import::into( $target );
   warnings   ->import::into( $target );
   lib        ->import::into( $target );

   List::MoreUtils->import::into( $target,
           'any', 'all', 'apply', 'uniq' );

   Encode         ->import::into( $target,
           'encode', 'decode' );

   Moose          ->import::into( $target );

}

# Return true - it's a package
1;
```

Jetzt sind, wie bereits oben gezeigt, die Pragmata `strict`, `warnings` und `lib` aktiviert.

Weiterhin wurden das Modul `Moose` und die Funktionen `any`, `all`, `apply`, `uniq`, `encode` und `decode` in den Namensraum des Aufrufers importiert.

Erheblich weniger Boilerplatecode

Der am Anfang des Artikels gezeigte Code lässt sich nun zur Wiederverwendung kapseln

```
package MySetup;
use strict;
use warnings;

use Import::Into;

use Package::A qw//;
use Package::B qw//;

sub import {
   my $target = caller;

   strict     ->import::into( $target );
   warnings   ->import::into( $target );

   Package::A ->import::into( $target,
                'x', 'y', 'z') ;
   Package::B ->import::into( $target,
                'u', 'v', 'w') ;
}

# Return true - it's a package
1;
```

und deutlich kürzer anwenden

```
package Foo;
use MySetup;
```

Bei verbesserter Wartbarkeit des Codes sind statt fünf Zeilen nur noch zwei Zeilen Code erforderlich.

Frank Fuhrmann

Einführung in die Automatisierung mit Rex

Da unser Chef-Server auf absehbare Zeit nicht mit der Außenwelt kommunizieren darf, hab ich mir jetzt für unsere EC2-Umgebung die Verwaltung mit Rex in vielen Bereichen automatisiert.

Die Umgebung

In unserem AWS-Account tummeln sich verschiedene Projekte. So etwas ist erfahrungsgemäß eher unüblich, da allein schon zu Abrechnungszwecken eine Account pro Projekt durchaus Sinn macht. Nun, bei uns ist das eben nicht so und so war ich gezwungen einen Weg zu finden, wie man darin etwas Struktur beibehalten kann. Als Mittel zum Zweck fiel meine Wahl auf Rex, das ich schon aufgrund seiner Perl-Basis bevorzuge. Außerdem hat es serverseitig fast keine Anforderungen. Es wird lediglich Perl und ein SSH-Zugang auf den Server gebraucht.

Grundlegendes zu Rex

Rex ist ein Deployment-Tool wie Chef oder Puppet. Es wird also genutzt um Software auf Rechnern zu installieren und Konfigurationen zu verwalten. Im Gegensatz zu Chef und Puppet ist es aber eher dezentral aufgebaut und benötigt, wie bereits erwähnt, auf der Server-Seite lediglich Perl und einen SSH-Zugang.

Grundlegend besteht eine Verwaltungsumgebung mit Rex aus sogenannten Rexfiles... Perl-Modulen, in denen Tasks definiert werden. Mit Hilfe dieser Tasks können dann Aufgaben auf ganzen Gruppen von Servern ausgeführt werden. Zum Aufbau einer mit Rex verwalteten Umgebung verwendet man grundlegend 3 Tools:

- rex – mit diesem Tool werden die Tasks aus den Rexfiles ausgeführt
- rexify – damit wird die Grundstruktur eines Rex-Projekts angelegt
- Editor – die Arbeit wird sehr vereinfacht, wenn man einen Editor zur Verfügung hat, der für Perl-Code gedacht ist

Die Installation der Rex-Verwaltungsumgebung ist allerdings kinderleicht und erfolgt komplett auf der Workstation des Admins. Dazu werden zuerst ein paar Module aus dem CPAN-Repository benötigt:

- YAML
- Net::SSH2
- LWP
- DBI (wenn man Datenbank-Zugriffe braucht)
- JSON::XS (für Jiffybox und OpenStack)
- XML::Simple (für Amazon Cloud und Virtualisierung wichtig!)
- String::Escape (wenn man Server-Gruppen in INI-Dateien definieren will)

Auf Linux- und Unix-Systemen wird dann mit einem simplen Einzeiler installiert.

```
curl -L get.rexify.org |
 perl - --sudo -n Rex
```

Bei Macs, die ihr Perl aus den MacPorts nutzen, ist zu beachten, dass /opt/local/libexec/perl5.XX/sitebin in $PATH aufgenommen werden muss, damit die CLI-Befehle von Rex zur Verfügung stehen.

Die Vorbereitung

"Ordnung ist das halbe Leben" haben mich meine Eltern immer gelehrt. Gut, das schaffe ich in meinem Arbeitszimmer nicht immer, aber in der Server-Administration lege ich bekanntermaßen gesteigerten Wert drauf. Je sauberer ein Netzwerk strukturiert ist umso schneller kann man sich darin einarbeiten und umso effektiver lässt es sich verwalten. Gerade bei Cloud-Infrastrukturen geht sonst auch schnell die Übersicht verloren, was dann in ungenutzten Instanzen und nicht mehr benötigten Daten endet, die unnötig Kosten verursachen.

Plant man eine neue EC2-Umgebung oder will eine vorhandene Umgebung auf in eine saubere Struktur bringen, sollte man sich zuerst für ein Basis-Betriebssystem entscheiden. In den meisten Fällen ist das aktuellste Amazon-AMI als Basis zu empfehlen, da es aus einer halbwegs vertrauenswürdigen Quelle stammt und speziell für EC2 angepasst ist. Was darauf später an Server-Software läuft, spielt für das Basis-System keine Rolle, so dass man getrost alle Instanzen mit Hilfe dieses AMI aufsetzen kann. Das hat verschiedene Vorteile, wobei der größte aber sicherlich darin liegt, dass man System-Updates immer mit dem gleichem Befehl machen kann, was der späteren Automatisierung zugute kommt. Wie das geschieht, werde ich später noch aufzeigen. Ein weiterer Vorteil besteht darin, dass man so eine einheitliche Basis hat, auf deren Eigenheiten sich die zuständigen Sysadmins schnell einstellen können.

Wichtig ist, dass man sich von dem AMI eine Privat-Kopie erstellt. So verhindert man Probleme, wenn das AMI später durch Amazon entfernt wird, und kann mit Hilfe dieser Kopie auch ein für die eigenen Anforderungen angepasstes Basis-AMI selbst pflegen. Das ist in fast allen Fällen sinnvoll, denn es erspart unnötige Installationsarbeiten beim Aufsetzen einer neuen Instanz. Also einfach mal eine t1.micro-Instanz aus dem offiziellen Amazon-AMI booten und sich von dieser Instanz eine neues AMI erstellen. Dies geht schnell und problemlos per Klick über die AWS-Konsole.

Da alle Systeme nun die gleiche Basis bekommen, muss man dies natürlich auch in Rex irgendwie abbilden. Ich empfehle dafür 2 “Ebenen” an Rexfiles zu nutzen. In der 1. Ebene bildet man jene Dinge ab, die alle Systeme gleich haben. Sie wird unter anderem genutzt um grundlegende Wartungsaufgaben durchzuführen, die bei allen Systemen gleich sind. In der 2. Ebene werden die einzelnen Projekte abgebildet. Man erstellt also in seinem Projektordner (im Folgenden als $PROJECT benannt) ein Rex-Projekt mit Hilfe von rexify.

```
> cd $PROJECT
> rexify ec2
```

Ergebnis dieses Befehls ist ein Unterordner *ec2* in *$PROJECT*, in dem sich eine Datei namens *Rexfile* und der Ordner *lib* befindet. Ein Blick in *$PROJECT/ec2/lib/* zeigt uns weiterhin noch die Datei *ec2.pm*.

Auf die gleiche Weise erstellt man nun noch Rex-Projekte für jedes Projekt im Netzwerk unterhalb des Ordners *ec2*.

```
> cd $PROJECT/ec2
> rexify projekt1
> rexify projekt2
...
```

Auch in diesen findet man wiederum einen lib-Ordner jeweils mit *projekt1.pm*, *projekt2.pm*, ... sowie ein Rexfile. Dass so etwas bei einem ordentlichen Sysadmin in eine Versionsverwaltung gehört, muss hoffentlich nicht extra betont werden, weswegen ich darauf in diesem Artikel auch nicht weiter eingehen will.

Zu der vorgefundenen Struktur, die durch rexify angelegt wurde, gibt es noch ein paar Worte zu sagen. Grundsätzlich gilt, dass in die Rexfiles die Definitionen globaler Werte gehören. Das können Server-Gruppen, Pfade zu SSH-Keys und auch die use-Anweisungen zum Einbinden zusätzlicher Perl-Module sein. In die pm-Dateien, die man in den jeweiligen Unterordnern findet, gehören alle Tasks. Bei Bedarf kann man natürlich weitere Module anlegen.

Machen wir dies einfach an unserem Beispiel fest.

Projektübergreifende Tasks

Wir widmen uns dazu zuerst unsere 1. Ebene, dem Rex-Projekt *ec2*. Hier öffnen wir das zugehörige Rexfile und definieren ein paar grundlegende Dinge, die man zur Verwaltung von EC2 mit Hilfe von Rex immer braucht: den SSH-User, die SSH-Keys, die Access-Keys usw.

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

# der SSH-Benutzer
set user => "ec2-user";

# die folgenden Daten koennen ueber
# die AWS-Konsole
# angelegt bzw. abgerufen werden
private_key '/Pfad/zum/Private-SSH.key';
public_key '/Pfad/zum/Public-SSH.key';
my $access_key = '';
my $secret_access_key = '';

# Definition zum verwendeten
# Cloud-Service und Authentifizierung
cloud_service 'Amazon';
cloud_auth $access_key, $secret_access_key;
cloud_region "ec2.eu-west-1.amazonaws.com";

# Anlegen einer Server-Gruppe aus allen
# Instanzen
set group => "servers" =>
    get_cloud_instances_as_group();

# abschliessend die ec2.pm mit den Tasks
# einbinden
require ec2;
```

Die use-Anweisung am Anfang sorgt dafür, dass uns die Cloud-Funktionen von Rex zur Verfügung stehen. Dadurch werden z.B. Funktionen wie `cloud_service`, `cloud_auth` und `cloud_region` bereitgestellt. Was sonst noch damit möglich ist, erfährt man in der Dokumentation zu `Rex::Commands::Cloud` auf CPAN.org.

Der Benutzer 'ec2-user' ist auf jeder Instanz, die aus dem Amazon-AMI besteht, per Default vorhanden. Er hat volle sudo-Rechte und eignet sich daher bestens für die Systemverwaltung. Paranoide Admins wie ich legen sich natürlich einen eigenen Systembenutzer mit etwas eingeschränkteren Rechten an.

Wie bereits angedeutet, handelt es sich bei `cloud_service`, `cloud_auth` und `cloud_region` um Funktionen, die von `Rex::Commands::Cloud` bereitgestellt werden. Sie dienen dazu bestimmte Werte für die Cloud-Funktionen bereitzustellen. Die Funktion `get_cloud_instances_as_group` aus dem gleichen Modul gibt alle Public Hostnames der Instanzen des Accounts zurück, zu dem die mit `cloud_auth` durchgeführte Authentifizierung passt. Diese Rückgabe erfolgt als Liste, so dass sie von Rex direkt als Server-Gruppe verwendet werden kann.

Server-Gruppen dienen bei Rex dazu bestimmte Tasks für eine bestimmte Menge an Systemen auszuführen. Im obigen Beispiel wird die Gruppe *servers* definiert, die alle Instanzen des EC2-Accounts beinhaltet. Man kann aber natürlich auch selbst die Liste der Server angeben ohne diese durch eine Funktion ermitteln zu lassen.

```
set group => "servers" => '123.123.123.123',
 '123.123.123.124', '123.123.123.125';
```

Zu beachten ist, dass es sich bei den Host-Angaben um jene Hosts handelt, die später bei den Tasks auch für die SSH-Befehle verwendet werden. Ein `ssh user@123.123.123.123` muss also von der Rex-Workstation aus funktionieren.

Die erste Task

Nachdem nun diese grundlegenden Einstellungen getroffen sind, kann man die ersten Tasks vorbereiten. Um die Funktionsweise zu verdeutlichen, sollen einfach mal ein Befehl auf allen EC2-Instanzen ausgeführt werden. Ein Beispiel dafür finden wir auch bereits in der Datei *ec2.pm*.

```
desc "Get uptime of server";
task "uptime", group => 'servers', sub {
   say run "uptime";
};
```

Diese kann man auch bereits aufrufen.

```
> cd $PROJECT/ec2
> rex ec2:uptime
```

Rex wird sich daraufhin auf allen Maschinen mit den angegebenen SSH-Daten einloggen und den Befehl `uptime` ausführen. Der Output des Befehls wird dann auf der Konsole ausgegeben.

Der Befehl rex

Eine kurze Erklärung zum Aufruf einer Rex-Tasks: Der Befehl 'rex' geht prinzipiell davon aus, dass es mindestens ein Rexfile gibt. Dieses wird zuerst geladen. Wird eine Task angegeben, die einen Doppelpunkt im Namen hat, geht rex davon aus, dass die Task aus einem Modul aufgerufen werden soll. Es wird dann geprüft ob im Unterordner 'lib' ein Modul mit diesem Namen zu finden ist und es wird die Task aus diesem Modul ausgeführt. Wird kein Modul angegeben, wird die Tasks ausschließlich im Rexfile gesucht.

Weiterhin kann man die Gruppen-Definition bei der Task auch weg lassen. Dadurch kann man flexibler festlegen ob eine Task ggf. nur auf verschiedenen Host-Gruppen ausgeführt werden kann oder ob ein Task fest an eine Gruppe gebunden ist. Allerdings muss man die Gruppe natürlich auf der Kommandozeile angeben, wenn für eine Task keine Gruppen-Definition existiert. Schließlich muss Rex ja irgendwoher wissen, auf welchen Systemen die Task ausgeführt werden soll. Auch hierzu ein kleines Beispiel.

Wir haben 2 Host-Gruppen:

```
set group => "dbserver" =>
  '123.123.123.123', '123.123.123.124';
set group => "webservers" =>
  '223.223.223.223', '223.223.223.224';
```

Und wir haben 2 Tasks

```
desc "Restart MySQL servers";
task "restart_mysql",
  group => "dbservers", sub {
  run
   'sudo /etc/init.d/mysql-server restart';
};

desc "Reboot a whole cluster";
task "restart_cluster", sub {
  run 'sudo reboot';
}
```

Ruft man nun die Task *restart_mysql* auf, werden garantiert nur die MySQL-Server auf den Servern der Gruppe *dbservers* durchgestartet. Ruft man jedoch *restart_cluster* auf, wird die Admin-Workstation neu gestartet. :D Das liegt daran, dass keine Gruppe für die Task definiert ist. Tasks ohne Gruppe werden per Default lokal ausgeführt. Dafür gibt es implizit die Gruppe *local* in Rex. Sie stellt die Default-Gruppe für alle Tasks dar. Das sollte ein Admin, der mit Rex arbeitet, immer im Hinterkopf behalten.

Will man allerdings alle Webserver-Maschinen richtig rebooten, kann man dies tun indem man ‘webservers’ als Gruppe auf der Kommandozeile angibt:

```
> rex -g webservers restart_cluster
```

Damit wird auf allen Hosts der Gruppe *webservers* der Befehl *sudo reboot* ausgeführt. Man kann mit dem gleichen Parameter auch die für eine Task festgelegte Gruppe überschreiben.

Da selbst in verschiedenen Projekten ggf. ähnliche Systeme zum Einsatz kommen, die sich nur hinsichtlich der Konfiguration und der verwendeten Daten unterscheiden, macht es Sinn die Tasks für diese Systeme in der 1. Ebene *ec2* zu definieren. Ansonsten kann man in dieser Ebene alle Tasks automatisieren, die das verwendete Basis-System betreffen. Diese können dann Aufgaben wie das Deployment von Standard-Umgebungen (Bash-Konfiguration, SSH-User etc.) umfassen oder auch ein simples Update aller Instanzen via `yum`.

```
desc "Run yum update on all machines";
task "sysupdate_all",
   group => "servers", sub {
       run 'sudo yum -y update';
   }
```

Ein `rex ec2:sysupdate_all` bringt so die Systeme aller Instanzen auf den gleichen Stand.

Projektspezifische Tasks

In unseren bereits angelegten Unterprojekten der 2. Ebene kann nun die Software und Konfiguration verwaltet werden, die spezifisch für das Projekt ist. Dort werden also die einzelnen Server-Programme installiert und die zugehörigen Konfigurationen ausgerollt. Außerdem können Webapps u.ä. auf dieser Ebene eingespielt werden.

Nehmen wir als Beispiel ‘projekt1' mit einem Webserver und einer Webapp. Dieser Webserver läuft allerdings auf 5 Instanzen. Wir definieren im Rexfile zu projekt1 dafür zuerst die Hostgruppe *webservers*:

```
set group => "webservers" =>
  "123.123.123.123",
  "123.123.123.124",
  "123.123.123.125",
  "123.123.123.125",
  "123.123.123.126";
```

Auf dieser soll ein Apache installiert werden. Da man bei Amazon-Instanzen üblicherweise mit sudo arbeitet, vor allem wenn man den oben erwähnten ec2-user vom Amazon-AMI für die Systemverwaltung verwendet, muss im Rexfile definiert werden, dass alle Befehle mittels sudo ausgeführt werden sollen. Dazu reicht eine Zeile:

```
sudo TRUE;
```

Nun können wir eine Tasks zur Installation des Apache-Webservers in der *projekt1.pm* definieren.

```
task "install_apache",
  "group" => "webservers", sub {
      install package => "httpd";
  };
```

So einfach ist es, wenn man den Apache aus den Repos verwendet. Als nächstes möchte man vermutlich noch die Konfiguration anpassen. Dazu legt man sie sich in *$PROJEKT/projekt1* in einem Unterordner *files* einfach an.

```
> mkdir $PROJEKT/ec2/projekt1/files
```

Dort legt man die Konfiguration so ab, wie sie später auf der Instanz landen soll und sagt Rex wo diese auf dem Server abgelegt werden soll.

```
task "deploy_apache_config",
"group" => "webservers", sub {
  file "/etc/httpd/httpd.conf",
    source    =>
        "files/etc/httpd/httpd.conf",
    on_change => sub {
        service httpd => "reload"; };
}
```

Will man nun in Zukunft die Konfiguration des Webservers ändern, tut man dies einfach lokal auf der Workstation und aktualisiert sie auf allen Servern mittels `rex projekt1:deploy_apache_config` in einem Schwung.

Für die Webapp verwendet man am besten `Rex::Apache::Deploy`. Dadurch ist eine Versionierung möglich. Die Doku bei CPAN.org erklärt eigentlich alles Notwendige dazu.

Abschließende Worte

Ich hoffe, dass dieser Artikel aufzeigen konnte, dass Rex bestens dazu geeignet ist auf unkomplizierte Weise selbst große Server-Netzwerke zu verwalten. Anders als bei Chef kann man Perl-Code aber an jeder Stelle in Rexfiles und Rex-Modulen einsetzen. Bei Chef muss dazu ein Code-Block definiert werden, der nur sehr eingeschränkt auf den Rest des Rezepts Zugriff hat. Diese in meinen Augen massive Beschränkung gibt es bei Rex nicht. Man kann die volle Flexibilität von Perl einsetzen und dadurch auch problemlos eigene Provider einbinden, die z.B. auf SNMP-Status reagieren oder Webapp-Schnittstellen abfragen können, falls ein Deployment vom Status einer Webapp abhängt. Ein klassisches Beispiel sind Änderungen an der Datenbank während eines Deployments, bei denen keine Schreibzugriffe durch die Webapp stattfinden dürfen. Die Webapp kann über eine einfache Schnittstelle dem Rex-Deployment mitteilen, ob es den notwendigen Status dafür angenommen hat. Ein weiteres Beispiel bei EC2 ist die Definition der Hostgruppen anhand spezifischer Tags an den Instanzen, wodurch eine vollständige Dynamisierung des Netzwerks möglich ist und der Admin sich im Endeffekt nicht mehr um Hostnamen, IP-Adressen u.ä. kümmern muss.

Da Rex nicht umsonst für *Remote EXecution* steht, kann man es auch nutzen um schnell mal bestimmte Befehlsreihen auf Server-Gruppen auszuführen. Dies kann z.B. von Nagios genutzt werden, so dass man auf NRPE oder NCSA verzichten kann.

Ein weiterer Vorteil besteht darin, dass die Entwickler direkten Einfluss auf das Deployment nehmen können. Dazu muss deren Applikation lediglich ein Rexfile mitbringen, mit dem das Deployment gesteuert werden kann. Der Admin kann dabei weiterhin zentral die Host-Gruppen definieren, so dass der Entwickler sich darum nicht kümmern muss. Die Erfahrung zeigt, dass auf diese Weise die Zusammenarbeit zwischen Betrieb und Entwicklung oftmals verbessert werden kann und Fehler, die durch fehlende Kommunikation bezüglich der Deployment-Prozedur verursacht werden, nicht mehr auftreten. Alles in allem kann ich Rex nur empfehlen. Es mag zwar nicht so komplex sein wie Puppet oder Chef, was aber dadurch wieder wettgemacht wird, dass man an jeder Stelle auf Perl zurückgreifen kann. Schließlich ist Rex im weitesten Sinne auch nur eine Sammlung von Perl-Modulen mit einem zugehörigen CLI.

Herbert Breunung

Das Moose Update - Interview mit Stevan Little

Es gab ja nicht viele Änderungen seit Moose 1.0. Bist du heute zufrieden mit dem Stand des Modules oder braucht es noch Funktionen oder Erweiterungen?

Ja, ich bin sehr glücklich über den heutigen Zustand von Moose sowie ein wenig überrascht wie lange es bereits existiert. Moose ist schon acht Jahre alt und ungefähr fünf war ich aktiv an der Entwicklung beteiligt. Mich freut sehr zu sehen, dass es ein Eigenleben bekam.

Ich stimme dem nicht zu, dass sich Moose wenig ändert. An der Oberfläche tat sich wenig, aber darunter gab es große Verbesserungen. Dave Rolsky und Jesse Luehrs verbrachten viel Zeit damit die Interna aufzumöbeln und eine Balance von Geschwindigkeit und Flexibilität zu treffen. Karen Etheridge, die derzeitige Leiterin, ist sehr beschäftigt obskure Fehler zu jagen und leistet großartige Arbeit neue Beitragende zu unterstützen wie Upasana Shukla, die kürzlich echte Ausnahmen-Objekte in Moose einführte. (Anmerkung: die junge Inderin sprach auch darüber auf der YAPC::EU in Sofia. Folien unter *https://speakerdeck.com/upasana20/moose-structured-exceptions*)

Was zusätzliche Funktionalitäten oder Erweiterungen betrifft, so gibt es eine ordentliche Liste an Dingen von denen ich wünschte ich hätte sie nicht zugefügt. Zum Beispiel `augment/inner` war immer verwirrend und seine Benutzung sollte meiner Meinung nach eher als *code-smell* (Anzeichen schlechten Designs) angesehen werden. Auch Methoden-Modifikatoren: Eine Menge Menschen mögen sie, aber nach meiner Erfahrung sollten sie **sehr** selten verwendet werden, wenn überhaupt.

Moose umfasst weit mehr als die Standard-Objektorientierung. Rollen sind nur eine Möglichkeit die Klassenhierarchien völlig verändert. Sollten Programmierer weiter darüber nachdenken, wie man die Konzepte der Objektorientierung verbessern kann?

Ich denke, dass die Objektorientierung als Programmierprinzip ganz gut etabliert ist.

Tatsächlich entlieh ich das meiste Interessante in Moose dem Common Lisp Object System (*CLOS*), dass aus den frühen 1990'ern stammt. Fast alles Weitere kam von Smalltalk, also aus den 80'ern. Das `augment/inner` stammte sogar aus den 70'ern von einer Sprache namens *BETA*. Ich vermute die Rollen sind das Einzige was in diesem Jahrtausend erfunden wurde.

Es ist schon möglich das noch mehr OO-Techniken entdeckt oder erfunden werden. Allerdings neige ich dazu, dass man mehr gewinnt, wenn man Techniken von anderen Sprachen kombiniert. Das lässt sich an neueren Sprachen wie *Scala* oder *Swift* beobachten, welche viele funktionale Ideen vereinen.

Du hättest auch so etwas wie `Object::Tiny` machen können. Hast du gehofft, auch Features wie Signaturen und Typen in die Perl 5-Kultur hineinzutragen, oder ging es dir nur darum, Objekte wie in Perl 6 auch in Perl 5 zu haben?

Ich schuf Moose eigentlich, weil ich wirklich Perl 6-Objekte in Perl 5 haben wollte. Zu der Zeit besaß ich keinerlei Absicht diese bis in den Sprachkern zu tragen, oder kulturelle Barrieren einzureißen. Damals sah ich Perl 6 als Nachfolger von Perl 5 und dachte so etwas wie Moose könnte vielleicht eine Brücke zwischen den zwei Sprachen sein. Heute wissen wir alle, dass es Geschwistersprachen sind, weshalb ich nun versuche Objekte nach der Art von Perl 6 zu Perl 5 hinzuzufügen.

Soweit ich die Meinungen der Leute kenne, ist das *mop* (Metaobjektprotokoll für Perl 5) die am meisten erwartete Funktionalität der kommenden Versionen.

Ich kann das verstehen. *Moose* hat seinen Preis den *Moo* und *Mouse* minimieren wollen, aber es bleibt ein Preis. Ich glaube heutzutage ist es keine unrealistische Erwartung mehr, eine Programmiersprache mit einem anspruchsvollen und funktionsreiches Objektsystem zu haben, dass beinah keine Extrakosten an Speicher und Geschwindigkeit fordert. Der einzige Weg das erreichen zu können, besteht darin das Objektsystem in den Kern zu fügen.

Sollte es das sein (das meist erwartete Feature)?

Ja sollte es. Ich denke nicht, dass es unvernünftig ist, was neue Perlnutzer das erwarten. Die Tatsache, dass wir es nicht bereits haben ist ein Hindernis.

Ich denke es gibt genügend weitere Funktionen, welche die Gemeinschaft realisiert sehen will, doch nicht viele von uns versuchen sich daran. Erst in den letzten Jahren ist die Entwicklung wieder recht in Schwung gekommen.

Nach meinem Verständnis ist mop-redux rein für die Objektorientierung zuständig und kann mit Signaturen oder Ausnahmen beliebiger Module kombiniert werden. Es ist auch mehr ein Fundament, mit welchem andere OO-Module schneller werden und einfacher zu schreiben sind. Ist das soweit richtig?

Ja, der mop-Entwurf adressiert mit Absicht eine Reihe von Funktionen nicht, weil das einzeln und getrennt geschehen sollte. Ausnahmen sind ein perfektes Beispiel für etwas, das unbedingt der Sprache zugefügt werden sollte, aber nicht an eine Objektsystem gebunden. Ich glaub auch jener Ansatz führt letztlich zu einem besseren Design, da er mehr Flexibilität einfordert, was der perligen Philosophie des *TIMTOWTDI* entgegenkommt.

Und falls nicht, was ist das *mop* wirklich?

Nö, das war's. :)

Nur ein Kommentar über den Fortschritt der Arbeit am *mop*. Das github-Projekt *p5-mop-redux* ist eigentlich nur ein Schritt auf dem Weg. Speziell damit konnten wir eine Menge Probleme angehen, denen wir uns mit dem Vorgänger (*p5-mop*) nicht einmal nähern konnten. Und während einiger Fortschritt gemacht wurde, ist es trotzdem an einigen Entwurfsproblemen gescheitert, die auftauchten je näher wir dem Kern kamen. Das ist der Grund, warum ich entschied mehrere Monate damit zu verbringen, die Interna von *perl* und *C/XS* zu studieren. Derzeit schaffe ich am nächsten Entwurf (*p5-mop-XS*), in den ich mein neues Verständnis einbrachte. Am liebsten würde ich verkünden: "Das ist die letzte Fassung!", aber da ich das bereits zweimal sagte, ist es wohl das Beste zu schweigen um die Erwartungen hochzuhalten.

Pugs und `Class::MOP` alleine war eine Riesenarbeit. Welche Hilfe aus der Gemeinschaft hast du bekommen und was war am Ende das Hilfreichste?

Hmm, schwer zu sagen, wie die Leute beim Entwurf helfen können, der zu dem Zeitpunkt aus 80% Denken und 20% Programmieren besteht, nicht zu vergessen das dazu nötige, riesige Wissen über das was bereits geschaffen wurde. Aber es gibt Entwicklerversionen des *mop* (welche `p5-mop-redux` benutzen) auf *CPAN*. Denn während sich die Innereien noch ändern ist die *API* nach außen, also der Syntax rechts stabil. Programme zu schreiben die das nutzen, wäre für mich hilfreich.

Herbert Breunung

Rezension - Medizin

Peter J. Scott
Perl Medic
Addison Wesley, 335 S.
1.Auflage März 2004
ISBN: 8129710692
Softcover: $31.99
gebraucht: $13.99
Epub, Mobi, PDF: $25.59
Kindle: $15.13

Gabor Szabo
Perl Maven Cookbook
perl5maven.com/products 31 S.
ständig aktualisiert
PDF: noch frei

Auch ich möchte mit dieser siebzehnten Folge meine Hand zum Abschied heben und mich für das Interesse und die Anregungen bedanken. Wenigstens aus Rezensentensicht fließt auch etwas Erleichterung ein, da kaum noch wichtige Perlbücher übrig waren und Papier insgesamt nicht mehr so wichtig ist, wie noch vor 10 Jahren. Das liegt aber weniger an der Zellulosefaser, als an dem Umstand, dass sorgfältig Ausgearbeitetes zuweilen schnell altert und man daher verstehen sollte, welche Information am besten wo zu suchen ist.

Darum soll es dieses Mal gehen: um meine persönliche Bestenliste, einige Titel die bisher nach hinten verschoben wurden und ... aktuelles. Denn natürlich entstehen heutzutage Bücher zu Perlthemen.

Neuigkeiten

Damian Conway, der selbsternannte verrückte Wissenschaftler von Imperator Larry Wall, versprach ein Update von *Perl Best Practices* zu erarbeiten, und liefert nun ein Video, in dem er darüber referiert und das bei O'Reilly erschien. [1] Auch Toby Inkster, Mitentwickler und nimmermüder Paukenschwinger für Moose hat ein Buch begonnen - über Perl, Objekte und Rollen. [2] Zu guter Letzt sei der Gabor nochmal erwähnt, welcher derzeit an einem Kochbuch rührt. 32 Seiten klingt nicht nach viel, für die 19 Rezepte zu echten Problemen und mit aktuellen Zutaten verlangt er derzeit gerade einmal eine Registrierung in seine *Perl Maven*-Mailingliste. Der Code ist soweit von mir verstanden gut, schön formatiert und die begleitenden Texte bleiben beim Punkt - also navigieren um die Fallen, die man höchstwahrscheinlich als Anfänger laufen würde. Am besten gefällt mir, dass ab und zu ein Humor durchscheint, der eigentlich nur ein augenzwinkernder Realitätssinn ist. Die Versionsnummer (0.02) zeigt auch das Herr Szabo realistisch genug ist um zu wissen, dass zu einem richtigen Buch noch deutlich mehr gehört. [3]

Nicht so Neues

Die Mehrheit der weniger bekannten Titel, ist nicht empfehlenswert. Das haben meine unerschrockenen Expeditionen fast regelmäßig ergeben. Trotzdem hätte ich zu gerne einmal das *Grundwissen Perl* von Jürgen Schröter von innen gesehen. Der Oldenbourg-Verlag macht ansonsten recht hochwertige Lehrbücher für den akademischen Gebrauch. Überhaupt hat es mich überrascht, dass die Bücher von Jürgen Plate und Udo Müller aus Vorlesungsskripten entstanden sind. Nur hätte eine Zusammenarbeit mit "der Szene" sicher etwas weniger angestaubtes hervorgebracht. Und so viel Aufwand zum bei-

spielsweise in *Perl für Profis*, aus dem heise.de's hauseigenem dpunkt-Verlag floss, so ist diese Sammlung von Artikeln für mich heute nur noch von historischem Interesse. Zwei leicht abgehangene Folianten will ich dennoch darbieten.

Da wäre zum einen *Practical Text Mining With Perl* von Roger Bilisoly, das Wiley 2008 publizierte. Es veranschaulicht Textanalyse anhand kleiner (einfacher) Perl-Programme. Komplexes Parsing lehrt Damian Conway und Tom Christiansen weiß mehr über Unicode als in einen gewöhnlichen Kopf passt. Aber wer seine Perlkenntnisse dazu nutzen will, etwas darüber zu lernen, wie man aus Buchstabensalat Informationen gewinnt und diese auch noch statistisch sinnvoll aufbereitet, der sollte an dem Buch nicht vorbeigehen. Der Autor behauptet zwar, es wären nur die Grundlagen, aber das ist gut so. Zum einen weil es zeigt, dass er vom Fach ist und zum anderen, reicht die dargebotene Portion an Techniken für Anwendungsprogrammierer meist aus. Der Text schult das analytische Auge und für weitere Schritte gibt es sicher weiterführende Literatur. Der Erzählton ist sehr sachlich, aber für die Materie auch recht schlicht und schweift nicht ab, was das Lesen dann doch wieder angenehm macht.

Perl Medic

Ein Werk das ganz bestimmt zu wenig Aufmerksamkeit erfahren hat ist *Perl Medic*. Abgesehen von seiner inhaltlichen Güte ist es sehr liebevoll gemacht. Allein schon die Zeichnung und das Zitat am Anfang jedes Abschnittes haben Charakter, passen zum Thema und stimmen auf kommende und wesentliche Erkenntnisse ein. Das erste Kapitel beginnt mit: *I hate quotations. Tell me what you know.* Und das war ernst gemeint. Es gibt keinen Quatsch, wenig Text, kurze Kapitel, relativ viel Code. Und auch wenn es leider mit Perl 5.8.3 aufhört (aktuelle Errata trotzdem unter [4] einsehbar), so ist es trotzdem empfehlenswert, denn lehrt es vor allem die Systematik und richtige Geisteshaltung um aus einem stinkenden Berg von Code ein halbwegs ansehnliches und handhabbares Programm zu drechseln, ohne dabei den Verstand zu verlieren. Auch das wesentlich neuere Ebuch bietet leider nur den gleichen Inhalt und ist der inoffizielle Nachfolger vom 2001er *Addison-Wesley* - Titel *Perl Debugged*.

Best of

Wie anfangs angedeutet bieten Bücher immer noch etwas, was im Netz sehr selten und fast nie konsistent zu finden ist: das tiefere Verständnis. Zum Beispiel ist *Programming Perl* bestimmt als Hantel geeignet und daher nicht die erste Empfehlung für Beginner. Dennoch ist es die Perl-Bibel. Nicht nur weil Larry maßgeblicher Koautor ist, sondern vor allem weil Larry ein sehr guter Autor ist. Solange man ihm seine Faxen verzeihen kann, ist es eine Freude und Bereicherung an seinen Einsichten teilzuhaben. Aus diesem Verständnis macht Perl zu schreiben macht noch einmal so viel Spaß, weil entgegen anderslautenden Behauptungen die Details von Perl bewusst so ausgestaltet wurden.

Ich halte es für keine falschen Stolz festzustellen, dass eine Reihe Autoren Perlbücher verfassen wie Randal. L Schwartz, Tom Christiansen, cromatic, Simon Cozens, Mark J. Dominus, Damian Conway, Curtis Poe ..., welche über Witz, Sachkenntnis und didaktisches Talent besitzen (natürlich in Abstufungen). Ich hoffe sehr das diese Kultur fortgesetzt wird.[5]

Ohne alles aufzuzählen was in der $foo besprochen wurde - folgende Titel sind besonders empfehlenswert:

Einführung in Perl (das Lama)
kurzweiliger Unterricht, das Nötigste, deutsch, aktuell (5.14)

Intermediate Perl (Alpaka)
Fortsetzung vom Lama, deutsche Übersetzung uralt

Mastering Perl
(dritter, leicht abweichender Teil) nicht übersetzt

Programming Perl (Das Kamel)
Erklärt alles, witzig, umfangreich, 5.14

Modern Perl
kurze Anmerkungen was Stand der Dinge ist, sehr aktuell, kein Lehrbuch

Perl Best Practices
wie man Fehler vermeidet, von 2005, Grundlage von `Perl::Critic`

Effective Perl Programming
kompakte Trainingseinheiten für Sicherheit in Grenzfällen

Beginning Perl
das Nötigste um im Beruf Perl anzuwenden

Higher Order Perl
Funktionale Programmierung mit Perl

Das ist natürlich nur ein Minimum - einzelne Werke fehlen wahrscheinlich zu Unrecht, aber das sind die Klassiker, welche fast alle aktuell gehalten werden oder etwas besonders anschaulich vermitteln. Falls ein Anfänger/Chef fragt - das wäre die Liste, die unter seine Nase gehört. Dass sie der Wikipedia ähnelt ist kein Zufall, sondern entspringt der Tatsache, dass ich mit mir meistens einer Meinung bin. Vielleicht gelingt es uns (der Gemeinschaft) bessere Formen zu finden, gemeinsam gute Dokumentation zu schreiben.

[1] http://shop.oreilly.com/product/110000790.do
[2] http://blogs.perl.org/users/toby_inkster/2014/09/book-report---september-2014.html
[3] http://perlmaven.com/perl-maven-cookbook
[4] http://perlmedic.com/
[5] http://www.amazon.de/gp/product/B004D4YIB6/ref=pd_luc_wl_01_03_t_lh?ie=UTF8&psc=1

Wolfgang Kinkeldei

Was die Windows PowerShell mit Perl verbindet

Sieht man Scripte für die Windows PowerShell, so fallen auf den ersten Blick die mit "$" vorangestellten Variablen-Namen auf. Als Perl-Entwickler fühlt man sich sofort heimisch und will wissen, ob es noch mehr Parallelen gibt. Grund genug, dieser Frage einmal nachzugehen. Dass sich Perl selbst gelegentlich Konzepten von *nix-Shells bedient, darüber sehen wir großzügig hinweg.

Geschichte

Wer unter Windows arbeitet, wird vermutlich für die meisten Anwendungen graphische Werkzeuge gewohnt sein. Die Eingabeaufforderung (cmd.exe) hat sich zugegebenermaßen auch nicht gerade mit Ruhm bekleckert. Mehr als ein Kommando starten oder ein Installations-Script damit zu schreiben, dürfte eher abschreckend sein. Microsoft veröffentlichte gegen Ende 2006 die PowerShell und lieferte diese mit jeder danach veröffentlichten Windows Version mit. Mit Windows 8 wird die Version 4 der Shell ausgeliefert. Sie ist mittlerweile auch fester Bestandteil von Visual Studio und die Windows Server Version 2012 lässt sich anstelle eines graphischen Frontends komplett über eine PowerShell mit entsprechenden Kommandos installieren, betreiben und verwalten. Insofern könnte Microsoft kommandobasierten Bedienungs-Verfahren künftig höhere Bedeutung einräumen.

Kommandos

Von *nix-Shells wurde das Konzept kleiner auf einzelne Aufgaben spezialisierter Programme übernommen. Je nach gewählter Programmiersprache werden diese Cmdlets (implementiert als .NET Klassen), Functions (in der PowerShell Sprache entwickelt) oder Alias (meist *nix-gleiche Abkürzung für ein Cmdlet oder eine Function) genannt. Mittels Pipelines (|) werden diese Programme wie beim *nix Pendant miteinander verbunden. Allerdings transportiert eine Pipe nicht Text, sondern .NET Objekte, was den lesenden Prozessen wesentlich einfacheren Umgang mit den empfangenen Daten ermöglicht. Außerdem bestehen die Namen aller Kommandos jeweils aus einem Verb und einem Substantiv, was relativ schnell die Suche nach bestimmten Kommandos erlaubt. Bestimmte Substantive werden dabei immer für die gleiche Sorte von Entität des Betriebssystems verwendet (z.B. Item für Dateien). Sowohl die Kommandos noch die Schalter und Optionen dürfen wahlweise in Groß- oder Kleinbuchstaben eingegeben werden. Wer mehr lernen möchte, dem sei der Befehl `Get-Help` als Ausgangspunkt empfohlen.

So lassen sich die Kommandos (gefiltert) auflisten:

```
PS> Get-Command -Noun Item
PS> Get-Command -Verb Invoke
```

Wer wie ich Schwierigkeiten beim Einprägen der langen Schalter der diversen Kommandos hat, darf gerne die Vervollständigung in Anpruch nehmen, die wie üblich mittels der Tabulator Taste ausführbar ist. Jedes Kommando stellt seine Schalter zur Vervollständigung bereit und die Vervollständigung hilft sowohl bei Kommandos, als auch Schaltern und Datei-Namen und Pfaden.

Eine Liste von vordefinierten Aliasen erhält man mit:

```
PS> Get-Alias
```

Alle Befehle werden Groß- und Kleinschreibung ignorierend erkannt. Erweitert werden kann der Befehlssatz mittels sogenannter Snap-Ins, die im Wesentlichen eine Sammlung weiterer Befehle darstellen. Zahlreiche github-Projekte so-

wie Web-Sites (z.B. http://poshcode.org) bieten Snap-Ins für diverse Zwecke.

Aber zurück zum eigentlichen Thema: was ist denn nun ähnlich wie bei Perl?

Variablen

Variablen beginnen mit einem Dollar-Zeichen. Allerdings für jeden Datentyp, eine Unterscheidung zwischen Scalaren und z.B. Arrays anhand des Sigils gibt es nicht. Die Leerzeichen in den abgebildeten Kommandos sind wie bei Perl auch nicht notwendig, sie dienen lediglich der Lesbarkeit.

```
PS> $a = "Hello"
PS> $b, $c = "Hello", "World"
PS> $list = 1,2,3,4
PS> $names = "Tom", "Ben", "Rick"
PS> $names[1] # "Ben"
PS> $names[-1] # "Rick"
```

Arrays und anonyme Objekte können in literaler Schreibweise dargestellt werden, hierbei findet dann das "@" Zeichen doch endlich Verwendung:

```
PS> $empty_array = @()
PS> $custom = @{x=1; y=2}
```

Was ebenfalls stark nach Perl riecht, ist die String-Interpolation, leider war man beim Escaping nicht so genau -- vermutlich weil der Backslash durch seine Verwendung in Pfad-Namen eher als "richtiges" Zeichen benötigt wird.

```
PS> $w = "world"
PS> "Hello `"$w`""  # ` = escape Zeichen
```

Filterung

Offenbar waren die Entwickler der PowerShell von Funktionen wie "grep" besonders angetan, denn die Form, in der in Perl anonyme Code-Blöcke verwendet werden, kommt uns gleich bekannt vor. Sogar die Standard Laufvariable heißt gleich! Technisch gesehen werden Objekte durch ein Kommando erzeugt und mit der Pipe an das nächste Kommando geleitet. Jegliche Filter-Kommandos können bei den jeweils transportierten Objekten auf beliebige Methoden und Eigenschaften zugreifen. Das Cmdlet `Where-Object` verwende ich der Kürze wegen in Form des vordefinierten Alias F<where>.

```
PS> Get-Service |
  where { $_.Status -eq "Running" }
```

Selbstverständlich gibt es für Datei-Vergleiche auch passende Vergleichs-Operatoren. Anstelle F<Get-ChildItem> verwende ich das alias F<dir>.

```
PS> dir | where { $_.name -like "*.txt" }
```

Und selbst die Königs-Disziplin von Perl (Reguläre Ausdrücke) hat es in die PowerShell geschafft:

```
PS> dir |
  where { $_.name -match "[A-J].*[.]p" }
```

Nicht ganz korrekt umgesetzt hingegen hat man das Namensraum Trennsymbol (::), den das wird in der PowerShell dazu verwendet, um eine Klassen-Methode anzusprechen, wobei der Klassen-Name (Groß- und Kleinschreibung ist egal) in eckige Klammern geschlossen werden muss.

```
PS> dir -recurse |
   where { $_.LastWriteTime -gt [DateTime]::
                                        Today }
```

Auch hier bewirkt übrigens die Tabulator-Taste wahre Wunder! Die Eingabe von "[DateTime]::" genügt, ab hier kann man sich bequem durch die zur Verfügung stehenden Methoden tasten. Da jede beliebige .NET Klasse auf diese Weise innerhalb der PowerShell verwendet werden kann, sind die Möglichkeiten, die sich bieten, grandios. So macht Arbeiten in einem Terminal Spaß!

__END__

Wer nicht ganz auf "Klicki-Bunti" verzichten mag, kann gerne einmal probieren, was nachfolgender Befehl bewirkt:

```
PS> Get-Service | Out-GridView
```

Ich hoffe ich konnte den Windows-Anwendern unter uns die Benutzung der PowerShell schmackhaft machen. Für mich gehört dieses Werkzeug inzwischen zum Alltag.

Max Maischein

Web-Automation mit WWW::Mechanize::PhantomJS

Dieser Artikel führt in die Verwendung von WWW::Mechanize::PhantomJS ein. Mit dem Modul werden Informationen automatisiert aus einer Webseite ausgelesen.

Welches Problem?

Automatisches Auslesen von Daten auf Webseiten.

Zum Beispiel die intern entwickelte Webanwendung, die getestet werden soll. Oder das Kinoprogramm des alternativen Kinos, welches leider keine Benachrichtigung per Email anbietet, aber einen Twitter-Account hat.

Als meine Anwendung möchte ich die Daten aus der Notizanwendung "Google Keep" (*https://drive.google.com/keep*) exportieren. Google bietet leider keinen automatischen Export der Daten an. Idealerweise soll die Anwendung alle Notizen aus Google Keep exportieren und die einzelnen Notizen als HTML Dateien ablegen.

Ein sehr gutes Werkzeug um Daten aus einer Webseite auszulesen ist das Modul `WWW::Mechanize`. Es hat allerdings den für diese Situation gravierenden Nachteil, dass es kein Javascript unterstützt. Die Webseite von Google Keep wird komplett mit Javascript aufgebaut und die eigentlichen Inhalte werden durch Javascript nachgeladen, so dass wir eine Alternative zu `WWW::Mechanize` benötigen, die Javascript ausführen kann.

Das Modul `WWW::Mechanize::PhantomJS` ist eine solche Alternative. Statt einen Browser in Perl zu emulieren steuert das Modul den Webkit-Browser `phantomjs` fern. Dadurch läuft Javascript ganz automatisch ab und ich kann auf die erstellte Webseite zugreifen, sobald alle Daten geladen sind.

Der Browser phantomjs

Der Browser `phantomjs` ist ein ungewöhnlicher Browser. Er besitzt keine Ausgabe von Webseiten in ein Fenster auf dem Bildschirm. Stattdessen bietet er für die Eingabe eine Befehlsschnittstelle an, über die Javascript-Befehle geschickt werden. Die aktuell im Browser geladene Webseite kann zur Darstellung als Bilddatei gespeichert werden.

Dadurch dass `phantomjs` kein Bildschirmfenster benötigt, ist der Browser prädestiniert für die Automation und den Test von Webseiten. Der Browser läuft sehr gut auf Server-Rechnern und lässt sich auch problemlos in Cron-Jobs integrieren.

Für die Darstellung der Webseiten greift `phantomjs` auf die WebKit Bibliothek von Apple zurück. Um das benötigte Javascript zu implementieren wird die V8 Bibliothek von Google verwendet.

Installation und Vorbereitungen

Vor den Erfolg haben die Götter den Schweiß gesetzt. Es sind ein paar Vorarbeiten notwendig, um das Modul nutzen zu können:

Installation von phantomjs

Zuerst muss der Browser `phantomjs` installiert werden. Die entsprechenden Pakete sind auf der Webseite (*http://phantomjs.org*) verfügbar. Das Browserprogramm muss sich für den nächsten Schritt im Suchpfad für Programme befinden.

Installation von WWW::Mechanize::PhantomJS

Sobald der Browser `phantomjs` installiert ist, kann das Modul `WWW::Mechanize::PhantomJS` über das CPAN-Tool installiert werden:

```
cpan WWW::Mechanize::PhantomJS
```

Ein erstes Programm

Nachdem die Tests erfolgreich durchgelaufen sind, können wir ein erstes Programm mit dem Modul schreiben. Das Programm öffnet die URL *http://www.perl-magazin.de/* und speichert die Darstellung der Webseite als Bilddatei.

```
#!perl -w
use strict;
use WWW::Mechanize::PhantomJS;

my $url= 'http://www.perl-magazin.de/';
my $mech= WWW::Mechanize::PhantomJS->new();

# URL aufrufen
$mech->get($url);

# Bild holen
my $seite= $mech->content_as_png();

# Bild als PNG-Datei speichern
open my $fh, '>', 'screen.png'
    or die "Konnte nicht in 'screen.png'
      schreiben: $!";
binmode $fh;
print {$fh} $seite;
```

Features von WWW::Mechanize::PhantomJS

Das Modul bietet die von der `WWW::Mechanize` gewohnten Methoden für die Seitennavigation an. Damit können wir neben dem Aufruf einer URL über `->get()` auch Links folgen (`$mech->click()`) oder im Browserverlauf zurück und wieder vorwärts gehen (`$mech->back()` und `$mech->forward()`).

```
my $url= 'http://www.perl-magazin.de/';
$mech->get($url);
my $aktuell= $mech->find_link(
  text_contains => 'Aktuelles Heft');
$mech->click($aktuell);

# Zurück zur Startseite
$mech->back();

# Wieder die Seite des aktuellen
# Hefts darstellen
$mech->forward();
```

Für die Auswahl von Elementen auf Webseiten werden mit der Methode `->selector()` CSS Selektoren unterstützt. Das Modul findet Elemente auch über XPath-Ausdrücke, wenn CSS Selektoren nicht flexibel genug sind.

```
my $url= 'http://www.perl-magazin.de/';
$mech->get($url);
my @links= $mech->selector( 'a' );
for my $link (@links) {

    # Linkziel und -text ausgeben:
    print $link->get_attribute('href'),
          "\n\t-> ",
          $link->get_attribute('innerHTML'),
             "\n"
};
```

Javascript laufen lassen

Um Daten aus der Webseite von Google Keep auszulesen müssen wir Javascript ausführen. Damit laden wir die Daten von Google in die Webseite. Javascript mit `WWW::Mechanize::PhantomJS` auszuführen ist sehr einfach.

```
#!perl -w
use strict;
use WWW::Mechanize::PhantomJS;

my $mech= WWW::Mechanize::PhantomJS->new();

my $javascript=<<'JS';
    " ".join(["Just","another","Perl",
       "Hacker"]);
JS
print $mech->eval_in_page($javascript);
# Just another Perl Hacker
```

Die Methode `->eval_in_page()` führt Javascript Code im Kontext der aktuell angezeigten Webseite des Browsers aus. Dadurch können wir Javascript-Funktionen auf der Webseite aufrufen als hätte sie der Nutzer zum Beispiel durch einen Klick mit der Maus ausgelöst. Wir können aber auch die Ergebnisse beliebiger Javascript Funktionen in Perl weiterverwenden.

Screenshot

Ausschließlich den Browser zu steuern hilft uns nicht weiter, solange wir nicht wissen, welche Webseite gerade dargestellt wird und welche Daten gerade angezeigt werden. Da PhantomJS kein Anwendungsfenster öffnet, in welchem wir nachschauen können, müssen wir zur Fehlersuche und zur Fortschrittsüberwachung darauf zurückgreifen, die aktuelle Webseite als Datei zu speichern und uns dann die Datei anschauen.

Die Methode `->content_as_png` liefert uns die aktuelle Webseite als Bilddatei im PNG Format. Wir müssen die Daten nur noch in eine Datei speichern.

```
#!perl -w
use strict;
use WWW::Mechanize::PhantomJS;

my $mech= WWW::Mechanize::PhantomJS->new();

sub progress {
    my($element)= @_;
    open my $fh, '>', 'progress.png';
    binmode $fh, ':raw';
    if( $element ) {
        print {$fh} $mech->
          element_as_png($element);
    } else {
        print {$fh} $mech->content_as_png;
    };
    close $fh;
};

$mech->get('https://drive.google.com/keep');
progress;
```

Mit der Methode können wir auch eine Bilderserie von Abläufen auf der Webseite erstellen. Wenn einzelne Elemente auf einer Webseite im Fokus stehen, gibt `->element_as_png` die Darstellung eines Elements als Bild zurück.

Ausfüllen von Formularwerten zur Anmeldung

Wir können mit den bisher vorgestellten Methoden den Browser bis hin zur Anmeldung an der Webseite von Google Keep navigieren. Der Zugriff auf die Daten in Google Keep ist mit einer Kombination aus Email-Adresse und Kennwort geschützt. Die Methode `->dump_forms()` gibt uns einen Überblick, welche Formulare mit welchen Feldnamen auf der aktuell dargestellten Webseite verfügbar sind:

```
#!perl -w
use strict;
use WWW::Mechanize::PhantomJS;

my $mech= WWW::Mechanize::PhantomJS->new();
$mech->get('https://drive.google.com/keep');
$mech->dump_forms();
```

Damit erhalten wir zum Zeitpunkt der Erstellung des Artikels folgende Ausgabe:

```
[FORM] <no name>
https://accounts.google.com/ServiceLoginAuth
  [INPUT (hidden)] GALX
  [INPUT (hidden)] continue
  [INPUT (hidden)] followup
  [INPUT (hidden)] service
  [INPUT (hidden)] ltmpl
  [INPUT (hidden)] _utf8
  [INPUT (hidden)] bgresponse
  [INPUT (hidden)] pstMsg
  [INPUT (hidden)] dnConn
  [INPUT (hidden)] checkConnection
  [INPUT (hidden)] checkedDomains
  [INPUT (email)] Email
  [INPUT (password)] Passwd
  [INPUT (submit)] signIn
  [INPUT (checkbox)] PersistentCookie
  [INPUT (hidden)] rmShown
```

Die beiden interessanten Felder sind `Email` und `Passwd`. Mit der Methode `->submit_form()` können wir ein Formular ausfüllen und danach zum Server schicken. Wenn wir also den Zugangsnamen und das Kennwort haben, können wir uns damit anmelden (siehe Listing 1).

Elemente extrahieren

Nach der Anmeldung können wir die einzelnen Notizen sehen. Es fehlt allerdings noch der Zugriff auf den Textinhalt. Hier ist jetzt harte Schweißarbeit angesagt, um in den Daten der Webseite eine Regelmäßigkeit zu erkennen, die sich leicht im Programm umsetzen lässt. Nach intensivem Starren auf die Ausgabe von `->content()` bietet es sich an, alle HTML Elemente mit dem Attribut `@placeholder` zu extrahieren, da diese offenbar die Nutzdaten enthalten. Dieser Ansatz funktioniert allerdings nur, wenn die Listendarstellung statt der Mosaikdarstellung gewählt ist (siehe Listing 2).

```
#!perl -w
use strict;
use WWW::Mechanize::PhantomJS;

my $email= 'max.maischein@gmail.com';
my $password= $ENV{GOOGLE_PASSWORD}
    or die 'Kein Google Kennwort!
         Bitte $ENV{GOOGLE_PASSWORD}
           setzen';

my $mech= WWW::Mechanize::PhantomJS->new();
$mech->get('https://drive.google.com/keep');

print $mech->title,", logging in\n";
$mech->submit_form( with_fields => {
      Email => $email,
      Passwd => $password,
});
progress();

sub progress {
  my($element)= @_;
  open my $fh, '>', 'progress.png';
  binmode $fh, ':raw';
  if( $element ) {
      print {$fh} $mech->
        element_as_png($element);
  } else {
      print {$fh} $mech->content_as_png;
  };
  close $fh;
};
```

Listing 1

Ein schönerer Ansatz wäre es, die eigentlichen Daten abzufragen, aus denen die Webseite die Darstellung erzeugt. Allerdings benötigt das Herausfinden der Datenstruktur und der Abfrage wiederum viel Analyse des Datenverkehrs. Mit Hilfe der eigentlichen Datenstruktur ist es auch möglich, die Daten in beiden Richtungen abzugleichen und zusätzliche Felder wie den Erstellzeitpunkt abzufragen.

Alternativen zu WWW::Mechanize::PhantomJS

Wenn `phantomjs` als Browser nicht funktioniert, gibt es noch weitere Ansätze, um Javascript-lastige Webseiten mit Perl zu analysieren.

```
#!perl -w
use strict;
use WWW::Mechanize::PhantomJS;

my $email= 'max.maischein@gmail.com';
my $password= $ENV{GOOGLE_PASSWORD}
    or die 'Kein Google Kennwort!
       Bitte $ENV{GOOGLE_PASSWORD} setzen';

my $mech= WWW::Mechanize::PhantomJS->new();
$mech->get('https://drive.google.com/keep');

print $mech->title,", logging in\n";
$mech->submit_form( with_fields => {
      Email => $email,
      Passwd => $password,
});

for my $el (
  $mech->selector('@placeholder')) {
    print $el->get_attribute('innerHTML'),
                                        "\n";
};
```

Listing 2

WWW::Mechanize::Firefox

Das Modul `WWW::Mechanize::Firefox` funktioniert sehr ähnlich zu `WWW::Mechanize::PhantomJS` mit dem Unterschied, dass als Browser Mozilla Firefox zum Einsatz kommt. Der wesentliche Unterschied ist, dass der Browser die aktuelle Seite anzeigt und daher ein Fenster öffnet.

Win32::IEAutomation

Das Modul `Win32::IEAutomation` verwendet den Microsoft Internet Explorer um Webseiten zu automatisieren. Wenn eine Webseite auf diesen Browser speziell zugeschnitten ist, ist das Modul oft die einzige Lösung.

Javascript::SpiderMonkey

Das Modul `Javascript::SpiderMonkey` bettet den "Spidermonkey" Javascript-Interpreter von Mozilla direkt als Bibliothek ein. Dadurch kann Javascript ausgeführt werden. Es fehlt allerdings all das, was einen Browser zu einem Browser macht, insbesondere die Funktionalität um HTML darzustellen.

Javascript::Engine

Das Modul `Javascript::Engine` ist ein Javascript-Interpreter, welcher rein in Perl implementiert ist. Dadurch werden keine externen Programme oder Bibliotheken benötigt. Der große Nachteil ist die mangelnde Geschwindigkeit des Moduls.

Links

PhantomJS

Homepage: http://phantomjs.org/

Quellcode: https://github.com/ariya/phantomjs

Programme des Artikels

https://corion.net/articles/www-mechanize-phantomjs/files.html

Renée Bäcker

Mojolicious Tutorial - Validierung

Fast alle Webseiten haben irgendein Formular - und wenn es nur das Kontaktformular ist. Wenn die Daten auf Serverseite verarbeitet werden, sollten diese auf jeden Fall geprüft werden. Nicht nur einmal sind Sicherheitslücken bekannt geworden die wegen mangelnder Prüfung der Daten aufgetreten sind.

In diesem Teil des Mojolicious-Tutorials werden wir genau auf diese Prüfung eingehen. Ich zeige verschiedene Ansätze und werde auch Werbung für zwei meiner Module machen.

Als Basis nehmen wir erstmal diese `Mojolicious::Lite`-Anwendung (siehe Listing 1).

Das erzeugt eine Seite, die so aussieht, wie in Abbildung 1 dargestellt.

Die Probleme

Was kann jetzt alles schieflaufen? Eine ganze Menge! Es gibt einige offensichtliche Sachen, und ein paar weniger offensichtliche. Ich werde jetzt mal auf zwei Angriffsvektoren eingehen.

Abbildung 1: Das Formular

Cross-Site-Scripting (XSS)

Es gibt eine XSS-Lücke, bei der ein Angreifer einem Opfer einfach einen Link zukommen lassen kann und das Opfer holt sich dann Schädlinge auf den Rechner. Dank HTML-Mails fällt dem unbedarften Benutzer noch nicht mal der "komische" Link auf.

Der Link könnte so aussehen:

http://localhost:3000/send?comment=%3C iframe%20style=%22top:0px;%22%20src=%22 http://perl-academy.de%22%20width=%22 100%%22%20height=%22100%%22%3E%3C/ifram%3E

Einfach mal lokal die Anwendung mit `morbo` starten und den Link in den Browser kopieren. Keine Angst, es wird nichts Böses gemacht.

Für diesen einfachen Fall gibt es zwei Ursachen:

- Das Formular kann auch bei *GET*-Requests verarbeitet
- Die Eingaben werden nicht ordentlich überprüft

Abbildung 2: Der Benutzer sieht hier die Perl-Academy.de-Seite

Das Ergebnis sieht dann so aus, wie in Abbildung 2 dargestellt.

Der erste Schritt zur Absicherung ist, nur noch *POST*-Requests für die Formularverarbeitung zuzulassen. Aus

```
any '/send' => sub {};
```

wird dann

```
post '/send' => sub {};
```

Jetzt könnte ein Angreifer aber einen Link auf eine eigene Seite verschicken, die einen *POST*-Aufruf macht:

```
<html>
<body>
 <form action="http://localhost:3000/send"
     method="post" id="test">
     <input type="hidden" name="comment"
       value="<iframe
         src='http://perl-academy.de'>
         </iframe>" />
  </form>
  <script type="text/javascript">
    document.getElementById('test').
                                    submit();
  </script>
</body>
</html>
```

Die Anwendung muss also noch sicherer gemacht werden.

Header-Injection

Ein weiteres Problem liegt darin begründet, dass beim Verschicken der Mail die Daten einfach so an `sendmail` übergeben werden. Der Angreifer kann so den Mailer missbrauchen und Mails aus einer scheinbar vertrauenswürdigen Quelle verschicken.

Der Angreifer könnte als Absender

```
info@perl-services.de
Bcc: test@test.tld
```

übergeben. Man beachte, dass es hier einen Zeilenumbruch gibt und danach eine gültige Zeile für den Mailheader.

Validierung der Daten

Die beiden Beispiel haben gezeigt, dass man um eine Validierung der Daten nicht herumkommt. Es gibt hier zwei Vorgehensweisen:

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

use strict;
use warnings;

use Mojolicious::Lite;

any '/' => sub {
  shift->render('form');
};

any '/send' => sub {
  my $self = shift;

  my $params = $self->req->params->to_hash
      || {};

  open MAIL, '| /usr/bin/sendmail';
  print MAIL 'From: ' . $params->{from} .
      "\n";
  print MAIL "To: " .
      "dummy+test\@perl-services.de\n";
  print MAIL 'Subject: ' .
      $params->{subject} . "\n\n";
  print MAIL $params->{comment};
  close MAIL;

  $self->render( 'sent', p => $params );
};

app->start;

__DATA__
@@form.html.ep
%layout 'default';
<h1>Kontakt</h1>
<form action="/send" method="post">
  Absender: <input type="text"
    name="from" /><br />
  Betreff: <input type="text"
    name="subject" /><br />
  Nachricht: <textarea name="comment"
    rows="5" cols="40"><!--<-->/textarea>
                                      <br />
  <button type="submit">Absenden
    </button>
</form>

@@sent.html.ep
<h1>Vielen Dank!</h1>
Absender: <%== $p->{from} %><br />
Betreff: <%== $p->{subject} %><br />
Nachricht: <%== $p->{comment} %>

@@layouts/defaults.html.ep
<html>
  <body>
    %= content
  </body>
</html>
```

Listing 1

- Blacklisting
- Whitelisting

Beim *Blacklisting* werden "böse" Zeichen rausgefiltert, beim *Whitelisting* werden nur "gute" Zeichen akzeptiert. In der Regel ist es einfacher mit Whitelisting zu arbeiten, weil man sonst garantiert irgendwelche Zeichen beim Blacklisting vergisst und man doch angreifbar bleibt.

Zur Validierung der Daten gibt es verschiedene Plugins und Module und seit Version 4.49 hat `Mojolicious` auch einen eigenen Validierungsmechanismus.

Wir steigen mit einem Modul ein, für das in vielen Frameworks schon Plugins gibt und für Mojolicious gibt es ein veraltetes Modul, weswegen wir es hier direkt einsetzen - `Data::FormValidator`. Danach schauen wir uns den in Mojolicious eingebauten Validierungsmechanismus an. Und zum Abschluss mache ich etwas Werbung für meine eigenen Module.

Bevor wir uns aber mit der Validierung an sich beschäftigen, passen wir unsere Anwendung an, so dass der Benutzer auch etwas von den Fehlern mitbekommt.

Zuerst passen wir die Methode der Formularverarbeitung an:

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

  my $params = $self->req->params->to_hash
      || {};

  my %errors = _validate($self, $params);
  if ( %errors ) {
      $self->stash( errors => \%errors );
      return $self->render('form');
  }

  # [ ... ]
}
```

Und das Template passen wir auch noch an:

```
@@form.html.ep
%layout 'default';
<h1>Kontakt</h1>
% if ( stash('errors') ) {
  <ul>
     % for my $e (
     %     keys %{ stash('errors') } ) {
       <li><%= $e %> ist fehlerhaft</li>
     % }
  </ul>
% }
<form action="/send" method="post">
```

Jetzt müssen wir nur noch die Subroutine `_validate`erstellen und mit Leben füllen.

Überlegen wir uns also erstmal, wie die Bedingungen für die Felder des Kontaktformulars aussehen:

Absender
Ist ein Pflichtfeld, in dem eine valide Emailadresse stehen muss.

Betreff
Ist ein Pflichtfeld, bei dem der eingegebene Text mindestens 3 Zeichen lang sein muss.

Nachricht
Ist ein Pflichtfeld, bei dem der eingegebene Text mindestens 10 Zeichen lang sein muss.

Tritt ein Fehler auf, wird die Mail nicht verschickt, sondern eine Fehlerseite angezeigt (siehe Abbildung 3).

Data::FormValidator

Die Validierung bei `Data::FormValidator` basiert auf sogenannten Profilen. Um die ganzen Möglichkeiten dieses Moduls abzudecken, bräuchte man einen eigenen Artikel. Daher wollen wir hier nur auf dieses eine Beispiel eingehen.

Das Profil ist eine Hashreferenz, in der die Bedingungen für valide Eingaben festgelegt werden.

Abbildung 3: Fehlerhafte Eingaben werden nicht akzeptiert

Zuerst legen wir fest, dass die Felder alle Pflichtfelder sind:

```
my $profile = {
  required => [qw/from subject comment/],
};
```

Weiterhin können im Profil auch die Prüfmethoden definiert werden:

```
constraint_methods => {
    subject => sub{ length $_[1] >= 3 },
    comment => sub{ length $_[1] >= 3 },
    from    => sub{
        Email::Valid->address($_[1]);
    },
}
```

Und in der Methode `_validate` muss dann die Methode `check` von `Data::FormValidator` aufgerufen werden und der Errorhash muss gebaut werden. Die fehlenden Felder können dabei über die Methode `missing` und die ingültigen Felder über die Methode `invalid` des Ergebnisobjekts herausgezogen werden:

```
sub _validate {
  my $c     = shift;
  my $input = shift;

  my %errors;

  my $result = Data::FormValidator->check(
      $input,
      $profile,
  );

  if ( $result->has_missing or
      $result->has_invalid ) {
      %errors = map{
          $_ => 1,
      } $result->invalid,
        $result->missing;
  }

  return %errors;
}
```

Das Modul ist sehr mächtig, weil es auch gut mit Abhängigkeiten ("Wenn das Feld ausgefüllt ist, müssen diese zwei Felder auch ausgefüllt werden") umgehen kann. Und es ist weitverbreitet, gut gewartet.

Mojolicious

Beginnend mit Mojolicious 4.49 (17.10.2013) hat Mojolicious einen (nicht mehr experimentellen) eingebauten Validierungsmechanismus. Man muss dazu die zwei Klassen `Mojolicious::Validator` und `Mojolicious::Validator::Validation` kennen.

Das Vorgehen bei der Prüfung ist folgendermaßen: Zuerst müssen dem Validation-Objekt die Benutzereingaben bekanntgemacht werden. Danach muss man für jedes Feld alle möglichen Prüfmethoden aufrufen. Anschließen muss man prüfen, ob es einen Fehler gab.

Im Standard kenn das Validation-Objekt diese Prüfmethoden:

equal_to
Hiermit ist der Vergleich auf Gleichheit möglich. Der Methode muss man noch den Vergleichswert übergeben.

in
Prüfung, ob die Benutzereingabe einem Wert aus einer Vergleichsliste entspricht.

like
Prüfung mit einem Regulären Ausdruck

size
Hiermit kann die Länge verglichen werden. Der Methode muss man noch eine obere und eine untere Grenze übergeben.

Füllen wir die `_validate`-Methode.

```
sub _validate {
  my $c     = shift;
  my $input = shift;

  my $validation = $c->validation;
  my %errors;

  $validation->input( $input );
```

Mit `my $validation = $c->validation` wird das Validation-Objekt geholt, über `$validation->input( $input );` werden die Benutzereingaben bekanntgemacht.

Danach werden die einzelnen Felder überprüft:

```
# validate from field
$validation->required( 'from' );
if ( $validation->has_error('from') ) {
    $errors{from} = 1;
}
else {
    require Email::Valid;
    my $address = Email::Valid->address(
        $input->{from},
    );
    $errors{from} = 1 unless $address;
}
```

Zuerst wird gesagt, dass das Feld ein Pflichtfeld ist. Wenn keine Eingabe gemacht wurde, wird hier schon gleich intern ein Fehler gesetzt. Das muss dann mit `has_error` geprüft werden. Da es keine eingebaute Prüfmethode für Mailadressen gibt, muss hier noch händisch mit `Email::Valid` gearbeitet werden.

Dann weiter zum nächsten Feld:

```
$validation->required('subject')
           ->size( 3, 255 );
if( $validation->has_error('subject') ){
    $errors{subject} = 1;
}
```

Auch hier wird festgelegt, dass es ein Pflichtfeld ist. Man kann die ganzen Aufrufe der Prüfmethoden direkt aneinanderhängen. Hier wird die Länge der Eingabe geprüft. Bei `size` muss man eine Unter- und eine Obergrenze angeben.

Sollten wir Optionale Felder haben, muss statt `required` einfach `optional` aufgerufen werden. Eins von beiden ist aber Pflicht.

Was man bei diesen beiden Felder schon sehen kann ist, dass die vier Prüfmethoden von Mojolicious selten ausreichen. In so einem Fall kann man mit dem Validator-Objekt (Achtung: Nicht Validation-Objekt!) neue Prüfungen hinzufügen.

```
app->validator->add_check(
    min_length => sub {
        return length $_[2] < $_[3];
    }
);
```

Danach kann man beim Validieren `min_length` verwenden:

```
$validation->required('subject')
           ->min_length( 3 );
if( $validation->has_error('subject') ){
    $errors{subject} = 1;
}
```

Woran man sich gewöhnen muss ist die Tatsache, dass man bei den Checks im Fehlerfall einen wahren Wert zurückliefert und bei gültigen Eingaben einen unwahren Wert. Man kann sich das so merken, dass auf Fehler geprüft nicht, nicht auf Validität.

Mojolicious::Plugin::AdditionalValidationChecks und Mojolicious::Plugin::FormFieldsFromJSON

Zum Abschluss dieses Teils des Mojolicious Tutorials stelle ich noch zwei meiner Module vor, mit denen ich mir die Formularverarbeitung in Mojolicious vereinfache. Mit `Mojolicious::Plugin::FormFieldsFromJSON` kann man die Felder eines Formulars in einer JSON-Datei definieren und kann dabei auch angeben, welche Validierungsmethoden verwendet werden sollen. Für die Validierung selbst werden dann die Mojolicious-Klassen genommen.

Als erstes muss das Plugin geladen und konfiguriert werden:

```
use File::Basename;
app->plugin( FormFieldsFromJSON => {
    dir => dirname __FILE__,
});
```

Damit sagt man dem Plugin, dass die JSON-Dateien mit den Felddefinitionen im gleichen Verzeichnis liegen wie die Anwendung selbst.

Die Felddefinitionen sehen so aus:

```
[
  {
    "name" : "from",
    "type" : "text",
    "validation" : {
      "required" : 1,
      "email"    : 1
    }
  },
  {
    "name" : "subject",
    "type" : "text",
    "validation" : {
      "required" : 1,
      "length"   : 3
    }
  },
  {
    "name" : "comment",
    "type" : "textarea",
    "validation" : {
      "required" : 1,
      "length"   : 3
    }
  }
]
```

In den *validation*-Angaben werden auch Prüfungen genommen, die im Standard-Mojolicious nicht vorhanden sind. Für diesen Fall gibt es das Modul `Mojolicious::Plugin::AdditionalValidationChecks`, in dem folgende zusätzlichen Prüfmethoden definiert:

email
Prüft die Eingabe darauf, ob es eine valide Email-Adresse ist.

int
Ist die Eingabe eine Ganzzahl (z.B. +3, -5, 0, 10)

length
Im Gegensatz zu `size` ist hier eine Obergrenze nicht notwendig

min
Die Zahl muss mindestens dem Wert entsprechen, der `min` übergeben wird

max
Die Zahl darf maximal dem Wert entsprechen, der `max` übergeben wird

not
Darf keinem Wert der übergebenen Liste entsprechen

phone
Der Wert sieht aus wie eine Telefonnummer

url
Der Wert ist eine http oder https URL

color
Der Wert ist eine HTML-Farbangabe

Das Plugin muss auch noch eingebunden werden:

```
app->plugin('AdditionalValidationChecks');
```

Die `_validate`-Methode schrumpft damit auf folgenden Code:

```
sub _validate {
  my $c     = shift;
  my $input = shift;

  my %errors = $c->validate_form_fields(
      'contact'
  );

  return %errors;
}
```

Auf den ersten Blick ist das - durch die JSON-Datei - wesentlich mehr Schreibarbeit, aber wir können ein weiteres Feature von *FormFieldsFromJSON* nutzen: das Erzeugen der Formularfelder. Dafür erweitern wir die Konfiguration und bestimmen für jedes Feld noch ein Label.

```
{
  "name" : "comment",
  "type" : "textarea",
  "label" : "Nachricht",
  "attributes" : {
    "rows" : 5,
    "cols" : 40
  },
  "validation" : {
    "required" : 1,
    "length"   : 3
  }
}
```

Und beim Einbinden des Plugins geben wir noch ein *template* an:

```
use File::Basename;
app->plugin( FormFieldsFromJSON => {
    template =>
        '<%= $label %>: <%= $field %>
         <br />',
    dir => dirname __FILE__,
});
```

Danach können wir das Template des Formulars anpassen:

```
@@form.html.ep
%layout 'default';
<h1>Kontakt</h1>
<form action="/send" method="post">
  <%= form_fields('contact') %>
  <button type="submit">Absenden
    </button>
</form>
```

Kommen jetzt noch weitere Felder hinzu, muss nur noch die Konfiguration geändert werden und schon wird das neue Feld angezeigt und die Validierung wird auch automatisch

http://perl-academy.de

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

Promotion-Code

fookurs14

15% Rabatt

CPAN News XXXII

Devel::Cycle

Zirkuläre Referenzen sorgen für so manche Kopfschmerzen. Gerade wenn man Datenstrukturen an mehrere Subroutinen und Methoden übergeben werden und dort Änderungen vorgenommen werden. Zirkuläre Referenzen sorgen für Speicherleaks, die gerade bei persistenten Anwendungen wie mit mod_perl zu einem großen Problem werden können. Mit `Devel::Cycle` können solche Problemstellen gefunden werden.

```
#!/usr/bin/perl
 use Devel::Cycle;
 my $test = {fred   => [qw(a b c d e)],
         ethel  => [qw(1 2 3 4 5)],
         george => {martha => 23,
                    agnes  => 19}
        };
 $test->{george}{phyllis} = $test;
 $test->{fred}[3]       = $test->{george};
 $test->{george}{mary} = $test->{fred};
 find_cycle($test);
 exit 0;

# output:

 Cycle (1):
                     $A->{'george'} => \%B
                    $B->{'phyllis'} => \%A

 Cycle (2):
                     $A->{'george'} => \%B
                       $B->{'mary'} => \@A
                            $A->[3] => \%B

 Cycle (3):
                       $A->{'fred'} => \@A
                            $A->[3] => \%B
                    $B->{'phyllis'} => \%A

 Cycle (4):
                       $A->{'fred'} => \@A
                            $A->[3] => \%B
                       $B->{'mary'} => \@A
```

Convert::Color::Library

Welche Farbwerte hat "rot"? Ganz so einfach ist die Frage gar nicht zu beantworten. Nicht nur dass es viele verschiedene Rottöne gibt, je nach "Bibliothek" hat "rot" eine andere Bedeutung

```
use Convert::Color::Library;
use feature 'say';

my $red = Convert::Color::Library->new(
    'html/red',
);
my $red2 = Convert::Color::Library->new(
    'xkcd/red',
);

for ( $red, $red2 ) {
    say sprintf "%s in %s
====================
RGB: %s/%s/%s
Hex: %s
", $_->name, $_->dict,
   $_->red, $_->green, $_->blue,
   $_->hex;
}

$ perl color.pl
red in html
====================
RGB: 255/0/0
Hex: ff0000

red in xkcd
====================
RGB: 229/0/0
Hex: e50000
```

Data::Coloured

Nochmal um Farbe geht es auch bei `Data::Coloured`. Ist man auf Fehler gestoßen und möchte einen (ASCII-)Datenstrom debuggen, stößt man häufig nicht auf das Problem weil Kontrollzeichen die Ursache für das Problem. Um diese sichtbar zu machen, kann man `Data::Coloured` verwenden.

Ein ganz einfaches Programm ist hier zu sehen

```
while ( my $line = <> ) {
  chomp $line;
  if ( $line eq 'test' ) {
      print "IT'S A TEST";
      last;
  }
}
```

Wenn wir jetzt eine Testdatei übergeben, passiert - nichts. Also schnell ein `print $line;` eingefügt und sieht da "test" stehen.

```
$ cat test | perl data.pl
ein
test
mit
```

Komisch, da sollte doch die Zusatzausgabe gemacht werden. Bevor man sich jetzt den Kopf zerbricht, kann man `use Data::Coloured qw(poloured);` nutzen.

```
use Data::Coloured qw(poloured);

while ( my $line = <> ) {
  poloured $line;
  chomp $line;
  if ( $line eq 'test' ) {
      print "IT'S A TEST";
      last;
  }
}
```

Anschließend findet man den Fehler schnell Innerhalb von "test" gibt es ein Kontrollzeichen.

```
$ cat test | perl data.pl
ein[LF]te[NUL]st[LF]mit
```

Text::Unidecode

Wer kennt nicht das Problem: Es liegen Daten in Unicode vor, kann diese aber nicht nutzen. Z.B. weil die Anwendung kein Unicode unterstützt oder die verwendete Schriftart kennt die Zeichen nicht. Dann sieht man nur Datenmüll. In so einem Fall ist es wünschenswert, eine andere Darstellung zu bekommen. Genau dafür ist `Text::Unidecode` gemacht.

```
use Text::Unidecode;
print unidecode(
  "\x{5317}\x{4EB0}"
);

# That prints: Bei Jing
```

Mojo::SNMP

In einem großen Netzwerk alle Devices per SNMP abzufragen kann lange dauern. Um die Wartezeit zu verkürzen kann man `Mojo::SNMP` verwenden. Auch wenn `Mojo` im Namen zu finden ist, hat das Modul primär nichts mit Webentwicklung zu tun. Es nutzt vielmehr die IOLoop von Mojo.

```
use Mojo::SNMP;
my $snmp = Mojo::SNMP->new;
my @response;

$snmp->on(response => sub {
  my($snmp, $session, $args) = @_;
  warn "Got response from $args->{hostname}
       on $args->{method}
          (@{$args->{request}})...\n";
  push @response, $session->var_bind_list;
});

$snmp->defaults({
  community => 'public', # v1, v2c
  username => 'foo', # v3
  version => 'v2c', # v1, v2c or v3
});

$snmp->prepare('127.0.0.1',
    get_next => ['1.3.6.1.2.1.1.3.0']);
$snmp->prepare('localhost',
    { version => 'v3' },
    get => ['1.3.6.1.2.1.1.3.0']);

# start the IOLoop unless it is
# already running
$snmp->wait
   unless $snmp->ioloop->is_running;
```

Map::Metro

Welche Linie muss ich nehmen wenn ich mit Berlins U-Bahn vom Rathaus Spandau nach Rohrdamm fahren will? Diese Frage kann `Map::Metro` im Zusammenspiel mit dem Plugin `ap::Metro::Plugin::Map::Berlin` beantworten.

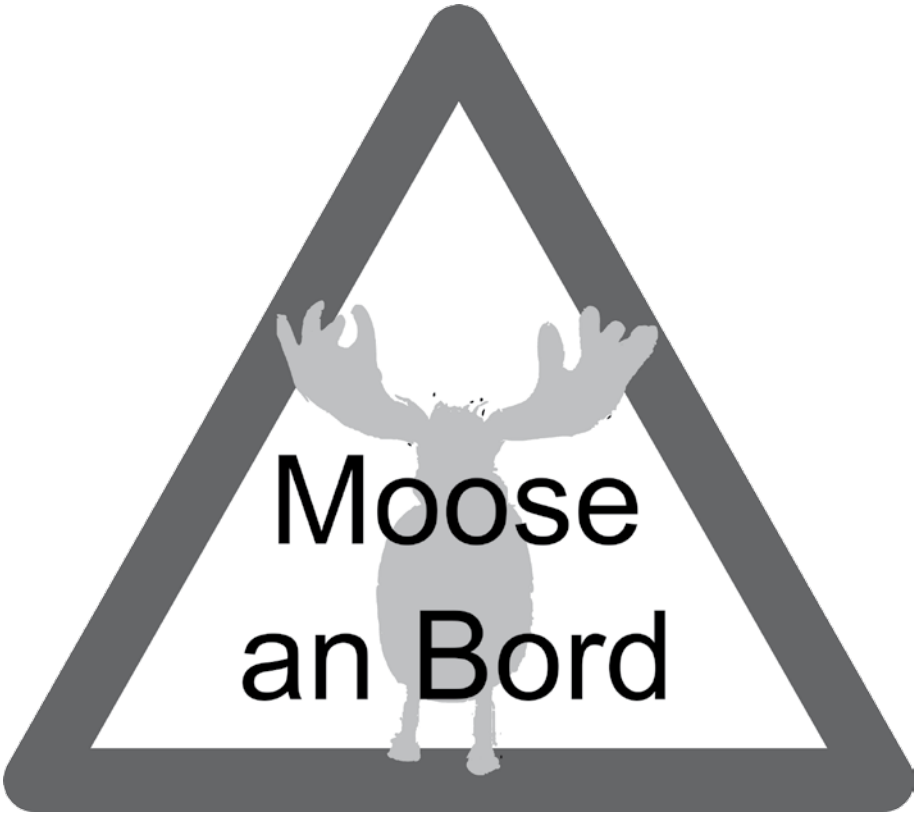
```
my $graph = Map::Metro->new(
    'Berlin',
    hooks => ['PrettyPrinter'],
)->parse;
my $routing = $graph->routing_for(
    'Rathaus Spandau',
    'Rohrdamm',
);
```

Das erzeugt die Ausgabe

```
From Rathaus Spandau to Rohrdamm
================================

-- Route 1 (cost 6) ----------
[   U7 ] Rathaus Spandau
[   U7 ] Altstadt Spandau
[   U7 ] Zitadelle
[   U7 ] Haselhorst
[   U7 ] Paulsternstr.
[   U7 ] Rohrdamm

U7

*: Transfer to other line
+: Transfer to other station
```

Das gleiche Ergebnis bekommt man auch mit dem Skript *map-metro.pl*:

```
$ map-metro.pl route Berlin \
  "Rathaus Spandau" "Rohrdamm"
```



Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de

Termine

Januar 2015

01. Treffen Dresden.pm
06. Treffen Frankfurt.pm
Treffen Hannover.pm
15. Treffen Erlangen.pm
20. Treffen Hannover.pm
28. Treffen Berlin.pm
31. FOSDEM

Februar 2015

01. FOSDEM
03. Treffen Hannover.pm
Treffen Frankfurt.pm
05. Treffen Dresden.pm
17. Treffen Hannover.pm
19. Treffen Erlangen.pm
25. Treffen Berlin.pm

März 2015

03. Treffen Frankfurt.pm
Treffen Hannover.pm
05. Treffen Dresden.pm
17. Treffen Hannover.pm
19. Treffen Erlangen.pm
25. 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

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.


$foo
PERL MAGAZIN
JMX4Perl
WebSockets
Perl und SVG
Nr 17


$foo
PERL MAGAZIN
CPAN-Distributionen
Tapper
Hack your applications!
Nr 18


$foo
PERL MAGAZIN
OpenGL
Rollen für Nicht-Moose-Anwendungen
OTRS "LinkedObjects"
Nr 19


$foo
PERL MAGAZIN
SNMP::Info
Perl in Multithreaded Programmen einbetten
$foo als ebook
Nr 20


$foo
PERL MAGAZIN
Rex
Jenkins
VMware
Nr 21


$foo
PERL MAGAZIN
XML und Perl
Perl 5.16
App-Entwicklung
Nr 22


$foo
PERL MAGAZIN
Nr 23
Dist::Zilla
Dumbbench
EBook::MOBI


$foo
PERL MAGAZIN
Compile- vs. Runtime
XML::Compile
Mojolicious Tutorial
Nr 24


$foo
PERL MAGAZIN
Synchrone Operationen
Mojolicious Tutorial
Nr 25


$foo
PERL MAGAZIN
Perl 6 Exceptions
LOG::REPORT
Mosaik-Bilder
Nr 26


$foo
PERL MAGAZIN
Hannover.pm
Debugging
VM ansteuern
Nr 27


$foo
PERL MAGAZIN
plenv
ZeroMQ
Mensch oder nicht Mensch
Nr 28


$foo
PERL MAGAZIN
Email::Stuffer
LWP::UserAgent, SSL und Proxy
Locale::TextDomain
Nr 29


$foo
PERL MAGAZIN
Was ist neu in Perl 5.20?
App::Fatpack
Nr 30
Subroutinen-Signaturen


$foo
PERL MAGAZIN
Messaging
SSL mit Perl
Lisp in Perl
Nr 31


$foo
PERL MAGAZIN
Websockets mit Mojolicious
PowerShell
WWW::Mechanize::PhantomJS
Nr 32

BOOKING.COM

online hotel reservations

NOW HIRING!

SysAdmins

MySQL DBAs

Perl Devs

Software Devs

Web Designers

Front End Devs ...

Interested? Booking.com/jobs

Booking.com B.V., part of Priceline.com (Nasdaq:PCLN), owns and operates Booking.com (TM), one of the world's leading online hotel reservations agencies by room nights sold, attracting over 30 million unique visitors each month via the Internet from both leisure and business markets worldwide.

We use Perl, puppet, Apache, MySQL, Memcache, Git, Linux …and many more!

Established in 1996, Booking.com B.V. guarantees the best prices for any type of property, ranging from small independent hotels to a five star luxury through Booking.com. The Booking.com website is available in 41 languages and offers 120,000+ hotels in 99 countries.

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