Ausgabe 3/2012 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Nr

23

Dist::Zilla
Module verwalten mit Dist::Zilla

Dumbbench
eine intelligente Stoppuhr

EBook::MOBI
Erstellen von E-Books für den Amazon

Yet Another

Perl Conference

EUROPE 2012

$WO =

GOETHE UNI-Frankfurt

$WEB =

http://yapc.eu/2012

$WANN =

20.-22. August 2012

EVOZON · $foo PERL-MAGAZIN · XING · YAPC::Europe · united domains

YAPC::Europe 2012 in Frankfurt

VORWORT

YAPC, Nachwuchs und Urlaub

Die YAPC::Europe 2012 steht vor der Tür: Vom 20. bis 22. August ist es endlich soweit, dass die europäische Veranstaltung endlich wieder in Deutschland gastiert. Die Frankfurt Perlmongers freuen sich auf viele Teilnehmer. Noch sind einige Plätze frei, also nichts wie auf die Webseite unter *http://yapc.eu/2012* und noch angemeldet. Im Wiki sind jetzt auch mehr Informationen wie Hotelempfehlungen und Freizeitprogrammvorschläge zu finden.

Direkt im Anschluss kann man Perl und Urlaub verbinden und nach Norwegen fahren. Dort findet der *Move to Moose*-Hackathon statt. Auf der Veranstaltung soll daran gearbeitet werden, ein Meta-Object-Protocol voranzubringen, damit es mit Perl 5.18.0 ausgeliefert werden kann. Perl 5.18.0 klingt noch so weit entfernt? Ist es nicht, denn in nicht einmal einem dreiviertel Jahr wird es erscheinen. Aber zurück zum Hackathon: Das Ganze findet in Stavanger statt und die Teilnehmer werden auch noch - wenn das Wetter es zulässt und Zeit ist - eine wunderbare Aussicht genießen. Denn der Preikestolen ist nur ein paar Kilometer entfernt.

Wo kann man sonst so schön "Arbeit" und Urlaub verbinden?

Eine interessante Diskussion hat *GwenDragon* auf Perl-Community.de unter *http://www.perl-community.de/bat/poard/thread/17442* angefangen. Sie möchte mehr junge Programmiererinnen zu Perl bringen. Ich hoffe, dass das Bestreben Erfolg hat, denn Nachwuchs ist immer gut und Frauen sind meiner Meinung nach immer noch unterrepräsentiert.

Das Thema Nachwuchs hatte ich schon vor längerer Zeit im Vorwort angesprochen. Daraufhin habe ich mich mit einem Leser unterhalten, der in seiner Region für den Wettbewerb "Jugend forscht" verantwortlich ist bzw. dort hilft. Auf seine Anfrage hin haben wir für den Regionalentscheid einen Sonderpreis bereitgestellt: ein Jahresabo $foo für die beste Arbeit, die mit Perl umgesetzt wurde. Für den Landesentscheid gab es dann einen weiteren Sonderpreis: Die Teilnahme an der YAPC::Europe. Ich freue mich schon, den Gewinner auf der YAPC begrüßen zu dürfen.

Wir sind gerne bereit auch in Zukunft wieder Sonderpreise zu stiften. Wenn es noch weitere Ideen gibt, wie wir Nachwuchs unterstützen können und Anreize bieten können, dann würde ich mich über eine Mail an feedback@perl-magazin.de freuen.

Renée Bäcker

Die Codebeispiele können mit dem Code

1dlvn30

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/issue23

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:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545
Feedback:	feedback@perl-magazin.de

INHALTSVERZEICHNIS

ALLGEMEINES

MODULE

ANWENDUNGEN

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. Er war auch am Aufbau der Wikipedia-Kategorie: "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

Daniel Bruder

Freiberuflicher Programmierer aus München und Liebhaber der Sprache Perl seit Beginn seines Studiums der Computerlinguistik. Auf dem Weg ein "richtiger" Perl-Hacker zu werden.

Boris Däppen

Boris Däppen lernte Perl im Finanz- und Börsenumfeld in Zürich als die inoffizielle helfende Hand im Applikations-Support kennen und schätzen. Er schließt zur Zeit den Master „Technik und Philosophie“ an der TU Darmstadt ab und arbeitet begleitend bei Perl-Services.de. Die Abschlussarbeit thematisiert Entwicklungshilfe, welche er als Informatiker in Kamerun auch vor Ort erlebt hat. Daher dreht sich für ihn (neben Perl) im Moment viel um die Themen Mensch, Kultur, Technik, Entwicklung und Hilfe.

Christian Rost

Christian Rost programmiert, seit ihn ein Casio Taschenrechner vor gut 25 Jahren mit BASIC in seinen Bann gezogen hat. Während seiner Ausbildung zum IT-Kaufmann hat er breites Spektrum an Programmiersprachen, Betriebssystemen und DBMS kennengelernt. Seine Vorliebe gilt inzwischen Perl und MySQL, z.B. in der OTRS Entwicklung. Hier engagiert er sich stark in der OTRS Community. Seinem Job in der IT bei dem Büroartikel-Versandhändler OTTO Office setzt er als Ausgleich das Rudern entgegen.

Herbert Breunung

Dumbbench - eine intelligente Stoppuhr

Ein stets guter Rat: Optimiere nur den Code, der nachgewiesen langsam ist. Für solch einen Nachweis gibt es seit langem das Kernmodul `Benchmark`, welches den Abstand zwischen zwei Zeitpunkten misst. Dem Perl 5-Porter Steffen Müller war es jedoch nicht gut genug und er schuf mit `Dumbbench` eine Alternative, die den Programmieraufwand senken und die Ergebnisse realistischer machen soll - auf Kosten zusätzlicher Rechenzeit. Selbst wenn der Programmierer sich dumm stellt oder die Arbeit scheut, sollten die Ausgaben immer noch nützlich sein. Dies war zumindest der Grundgedanke, welcher zu dem Namen führte.

Warum?

Wie vieles in Perl, so berufen sich auch die Funktionen von Benchmark auf die UNIX-Tradition. Nicht nur die grundlegende Einteilung der Ausgabe in *real*-, *user*- und *sys*-Zeit der UNIX-Kommandos `time` und `times` wurde weitergeführt. Es werden auch hinter dem Vorhang gleichnamige Perlbefehle mit ebenso niedriger Genauigkeit verwendet. Nur mit gesetztem Exporter-Tag `:hireswallclock` gibt es die durch das Modul `Time::HiRes` bereitgestellte, höhere Genauigkeit von mindestens Tausendstel Sekunden. Dumbbench bricht damit. Es stützt sich ausschließlich auf hochauflösende Zeitangaben und zeigt auch nur die *real* vergangene "Wanduhrzeit" an. Das ist jene Zeit, die auch jede Uhr außerhalb des Computers gemessen hätte. Die Überlegung dahinter: Um die wirklich echte Zeit, die ein Stück Code benötigt zu ermitteln, braucht es wesentlich komplexere Überlegungen als nur den Abzug der Zeit, welches das Betriebsystem und andere Prozesse zwischenzeitlich verbraucht haben. *IO*-Geräte wie Festplatten arbeiten bei gleichartigen Zugriffen nicht immer gleich schnell. Aber auch der Prozessverwalter des Betriebsystems (Scheduler) und Prozessoren besitzen ihre Eigendynamik, um Ergebnisse zu verfälschen. Deshalb wählte Müller den Ansatz, den Code genügend oft laufen zu lassen und die Zeiten von Ausreißern, die zu sehr vom Mittelwert (Median, nicht der Durchschnitt) abweichen, auszusortieren. Der Mittelwert der Abweichungen (der übrig gebliebenen Zeiten) vom Mittelwert ist zudem hilfreich um auszurechnen, wie zuverlässig der Endwert in etwa ist. Der Median bezeichnet den Wert an der mittleren Position einer sortierten Folge.

Eine genauere Beschreibung der statistischen Operationen gibt es im zugehörigen Blogbeitrag [1]. Zum Glück ist der Autor ein studierter Physiker und hat Erfahrungen damit, Datenreihen nach Verwertbarem abzugrasen.

Benchmark::Dumb

Praktischerweise fügte er der Distribution zudem das Modul `Benchmark::Dumb` [2] hinzu, dessen Funktionen sehr weit denen von `Benchmark` ähneln. Man erkennt deutlich, dass es als Ersatz gedacht ist. `timeit` dient der stillen Zeitmessung, `timethis` ist die Messung mit Ausgabe. Mehrere Aurufe von Letzterem lassen sich mit `timethese` zusammenfassen. `cmpthese` wird gewählt, wenn es nicht um die Zeiten, sondern um prozentuale Unterschiede geht. Deshalb ist es auch nur konsequent, dass `Dumbbench` angewiesen werden kann, die Anzahl der Wiederholungen selbst zu bestimmen, bis eine gewünschte Messgenauigkeit erreicht wurde. Dazu muss der erste Parameter lediglich Nachkommastellen bekommen. Der ganzzahlige Anteil ist nach wie vor die garantierte Anzahl an Durchläufen, negative Angaben sind nicht gestattet.

```
                          Rate Erestothenes n Erestothenes e
Erestothenes n   14.95+-0.011/s             --           -5.9%
Erestothenes e 15.894+-0.018/s     6.32+-0.14%              --
```

Listing 1

Zum Beispiel wäre 0.005 der Befehl, so viele Wiederholungen einzulegen, bis die Messgenauigkeit bei 0,5 Prozent der Lauflänge des Programmstücks liegt. Nie wieder die Fehlermeldung *warning: too few iterations for a reliable count* und ein Hoch auf die Programmiertugend der intelligenten Faulheit! Durch unsinnige Angaben von zu vielen Wiederholungen oder einer schwer erreichbaren Genauigkeit sind Eigentore aber immer noch nicht ausgeschlossen.

Wie bei `Benchmark` liefert `cmpthese` die Tabelle (siehe Listing 1) der Ergebnisse als Datenstruktur (Array von Arrays) inklusive der geschätzten Genauigkeit.

Die anderen Funktionen liefern ein echtes Objekt. Dessen Methoden geben die Informationen der Messung preis, und können auch die Summe oder Differenz zu einem anderen Messungsobjekt berechnen. Das ist ein weiterer deutlicher Unterschied zu `Benchmark`, das seine Informationen in einem Array lagert, welcher prozedural weiterverarbeitet wird.

Ein dritter eher subtiler Unterschied liegt im Namensbereich, in dem der Beispielcode ausgeführt wird. `Dumbbench` tut es in seinem eigenen. Das hat Auswirkungen, wenn man den Beispielcode nicht als Closure (Block), sondern als später zu evaluierenden String schreiben möchte. Um zu testen wie schnell Perl Zweierpotenzen berechnet, muss man deshalb statt

```
use Benchmark::Dumb qw(:all);

my $i = 3;
timethis(0.02, sub { 2 ** $i++ });
```

folgendes schreiben:

```
our $i = 3;
timethis(0.02, '2 ** $::i++');
```

Dumbbench

Während `Benchmark::Dumb` wie gezeigt Stücke von Perlcode messen kann, ist `Dumbbench` dazu da, ganze Skripte wiederholt laufen zu lassen. Die kleine aber sinnvolle API ist in der Dokumentation gut ausgeschildert und braucht hier nicht wiederholt zu werden. Interesse wecken könnte das mitgelieferte Skript, welches in der Kommandozeile wie `time` eingesetzt wird.

```
dumbbench -p 0.005 -- perl meinskript.pl
```

Durch gut gewählte Defaults (20 Durchläufe, 5% Präzision) reicht meist ein:

```
dumbbench perl report.pl
```

was eine schöne Ausgabe auswirft, die im Unterschied zu `Benchmark::Dumb` nicht nur die Anzahl der Wiederholungen sondern auch die abgelehnten Ausreißer beinhaltet. Die Distribution ist auf jeden Fall einen Probelauf wert. Sie ist mittlerweile zweieinhalb Jahre in Entwicklung, stabil, gut getestet und die Quellen sehen auch solide aus. Die Objekte sind mit `Class::XSAccessor` realisiert, daher sehr schnell und kompakt und die Handvoll eher kleinerer Abhängigkeiten dürfte nicht abschrecken. Im Gegenteil, das ebenfalls von Steffen Müller geschriebene `Statistics::CaseResampling` könnte noch für eigene Datenanalysen nützlich werden.

Links

[1] *http://blogs.perl.org/users/steffen_mueller/2010/09/your-benchmarks-suck.html*
[2] *http://metacpan.org/module/Benchmark::Dumb*

Herbert Breunung

WxPerl-Tutorial - Teil 11: Alternative Gestaltung

Nachdem die Grundlagen gelegt und fast alle wesentlichen Widgets beschrieben wurden, beginnt das große Resteessen. Es sind jedoch genügend Delikatessen für eine vorletzte Folge übrig geblieben. Die meisten der Widgets und Techniken haben sogar ein gemeinsames Thema: Die Anordnung von Elementen ohne die in Folge 3 erläuterten Sizer.

Resteessen ist leider eine passende Umschreibung. Zwar wurde `WxWidgets` 2.9.n gerade frisch zubereitet, aber für die meisten ist das derzeit noch nicht praktisch genießbar. Kaum eine der verbreiteten Distributionen serviert eine solche Version und unter Windows und Mac sieht es noch schlechter aus, denn das neuste `Alien::wxWidgets` 0.59 liefert `WxWidgets` 2.8.12 aus, sofern man die *Build.PL* nicht manuell startet. Die Konstante `&Wx::wxVERSION_STRING` erteilt Auskunft über den eigenen Stand. Außerdem sind die neuen, äußerst interessanten Widgets in ihrer Funktionalität noch im Aufbau.

Das betrifft zum einen die `Wx::RibbonBar`. Die *RibbonBar* wurde in den letzten Jahren vor allem durch *Microsoft Office 2007* bekannter, wo sie die Menü- und Werkzeugleiste ersetzt hat. Es ist im Grunde eine Sammlung von Seiten (`RibbonPage`) mit kleiner Höhe, die durch Reiter anwählbar sind. Jede Seite enthält Sektionen (`RibbonPanel`), meist von Knöpfen (`RibbonToolBar`), welche im Gegensatz zur normalen Werkzeugleiste größer sind und Beschriftungen haben. Während die `Toolbar` eine Sammlung von stets abrufbaren Kommandos darstellt, eignet sich die `RibbonBar` mehr für interaktives Arbeiten in einem Kontext und bietet dem Nutzer wesentlich mehr Führung und auch Vorschaumöglichkeiten.

Zum anderen kommt mit der nächsten *Wx*-Version noch das `Wx::PropertyGrid`. Jene Variante des gleich folgenden `Wx::Grid` eignet sich besonders für Konfigurationsdialoge, da es einfaltbare Gruppen (zweispaltige Tabellen) von Eigenschaftennamen mit einem zugehörigen Auswahl- oder Eingabewidget bietet. Es ähnelt in seiner Funktion grob Hashes, während ein *Grid* ein zweidimensionales Array von Werten verwaltet. Optisch entspricht das `PropertyGrid` der rechten Leiste für Widgeteigenschaften des *WxFormbuilder*, siehe *http://i1-win.softpedia-static.com/screenshots/wxFormBuilder_1.png*. Auch wenn beide Widgets hervorragend zum Thema der Folge passen würden - erst im angekündigten, frei erhältlichen Buch kann die genaue Beschreibung folgen.

Des Weiteren ist auch die Neuentwicklung von `wxWebView` bemerkenswert. Endlich kann man unter *Wx* auf eine *Render-Engine* zugreifen, die *HTML*, *CSS* und *Javascript* im vollen Umfang unterstützt. Dabei wird in guter *Wx*-Tradition auf eine plattformeigene Bibliothek wie *WebKit* oder *Trident* (Microsoft) weitergeleitet.

Wx::Grid

Meist reicht ein Verweis auf *Excel* um klarzustellen, worum es nun geht. Wenn selbst die in der vorvorigen Ausgabe vorgestellte `ListCtrl` im *Report-Modus* nicht ausreicht, lohnt der Einsatz dieser tabellenförmigen Eingabemöglichkeit. Sie ist besonders für große Datensätze optimiert und besitzt eine erstaunliche Reaktionsgeschwindigkeit. Auch gibt es etliche Erweiterungen, um die Eingaben in einzelnen Zellen in genau gewünschte Bahnen zu lenken.

Dieses Superwidget wurde mit Bedacht in Folge 9 ausgelassen, da es meistens viele kleine Textfelder umfasst, die in Folge 10 behandelt wurden. Ein `Grid` besteht jedoch nicht aus `Textctrl` sondern aus `GridCell*` mit wesentlich we-

niger Möglichkeiten. Die verschiedenen Klassen entsprechen meist Textfeldern mit bereits eingebautem `TextValidator` (siehe Folge 10). `GridCellNumberEditor` ignoriert Eingaben von Buchstaben oder Zahlen außerhalb des erwarteten Bereichs und so weiter. Der `GridCellBoolEditor` ist jedoch mit einer `CheckBox` und der `GridCellChoiceEditor` mit einer `Wx::Choice` realisiert. Welchen (Daten-)Typ eine Zelle hat, wird mit der Methode `SetCellEditor` bestimmt, falls es ein anderer als `GridCellEditor` sein soll. Die ersten beiden Attribute sind bei dem Aufruf (wie bei allen weiteren dieser Art) Zeile und Spalte der gemeinten Zelle. Beide Indizes fangen bei Null an. Die Anzahl der Zeilen und Spalten bestimmt man zu Beginn mit `CreateGrid` (funktioniert nur einmal), wobei jederzeit mit `InsertRows`, `InsertCols` und den entsprechenden `Delete*`-Befehlen angepasst werden kann. `SetColPos($alt, $neu)` erlaubt sogar Spalten die Position zu wechseln, auf `SetRowPos` wurde seltsamerweise verzichtet.

Die Zelleditoren erlauben es, Vorder- und Hintergrundfarbe sowie Schriftart zu setzen, jedoch nur für die gesamte Zelle einheitlich (`SetCellTextColour`, `SetCellBackgroundColour`, `SetCellFont`. Oder man tut das für eine gesamte Zeile oder Spalte.

```
my $grid = Wx::Grid->new($panel, -1 );
$grid->CreateGrid(4, 5);
my $attr = Wx::GridCellAttr->new;
$attr->SetTextColour( wxBLUE );
$attr->SetBackgroundColour( wxGREEN );
$grid->SetColAttr( 2, $attr );
```

Das sieht bestimmt nicht gut aus, demonstriert aber die Funktionalität. Sehr schön ist der Umstand, dass `SetColAttr` nur die für `$attr` gesetzten Eigenschaften verändert. Kommt es dennoch zu Konflikten, so gilt die Hackordnung: Zelleigenschaften überschreiben die der Spalte und diese die der Zeile.

Das *Grid* gehört zu den wenigen Widgets, die `Fit` ignorieren und bei ihrer Größe beiben. Die hängt entweder von den durch `SetCellSize` eingestellten Zahlen ab, oder vom Platzbedarf der Inhalte, soweit man dieses Verhalten mit `AutoSize` aktiviert hat, was auch für Zeilen oder Spalten getrennt geht. Tut man es nicht, so lässt sich ein vielleicht unerwarteter Effekt beobachten. Wenn die Zellen rechts daneben leer sind und man einen langen Text eingibt, dann fläzt er sich über den Platz, den er benötigt. `SetCellOverflow($row, $col, 0)` kann das abstellen, aber nur für eine Zelle. Wenn dochеinmal mehr in ein schmales Feld hinein soll, dann hilft:

```
$grid->SetCellRenderer(
    $row, $col,
    Wx::GridCellAutoWrapStringRenderer->new
);
```

Der für die Darstellung verantwortliche *Renderer* bricht den Text bei Bedarf zwischen den Worten (an Leerzeichen) um, was nur dann hilfreich ist, wenn auch vertikal für genügend Platz gesorgt wurde. Der Nutzer kann zwar die Größe aller Zeilen und Spalten verändern (soweit kein `CanDragGridSize(0)` und Konsorten zum Einsatz kam), aber nichts weist ihn darauf hin, dass der Text außerhalb des sichtbaren Bereichs weitergeht. Besser man kennt `SetRowSize($nr, $hoehe)` und seinen Pendants. Bei `SetColLabelValue(0,'Moehren')` passt sich die Breite der Spalte der Beschriftung an.

Aber zurück zu den *Renderern*. Es lassen sich auch eigene bauen, was in etwa so abläuft wie bei den *Popups* in Folge 7. Man schreibt nur eine Klasse, die von `Wx::PlGridCellRenderer` erbt und die kleine Methode

```
sub Clone { shift->new }
```

hat. Das entscheidende passiert in `Draw`. Sie erhält als Parameter

```
my( $self, $grid, $attr, $dc, $rect, $row,
     $col, $sel ) = @_;
```

Nachdem die Methode `Draw` der Superklasse mit allen Parametern außer `$self` aufgerufen wurde, kann der `DrawContext` (`$dc`) nach Herzenslust gestaltet werden. Details dazu in den Teilen 7 und 12.

Eigene Zelleneditoren können sogar noch nützlicher sein. Dafür erbt man von `Wx::PlGridCellEditor` und hat eine

```
sub new { shift->SUPER::new }
```

Nun kann man ein beliebiges Widget als "Editor" einsetzen (siehe Listing 1).

`Create` wird jedes Mal gerufen, wenn eine Zelle zur Bearbeitung des Wertes aktiviert wird, gefolgt von den ebenfalls überschreibbaren `SetSize` und `Show`-Methoden. Der wirkliche Spaß beginnt jedoch mit `EndEdit`. Hier ein Beispiel, das still die Mehrwertsteuer hinzufügt. `$oldValue` war der Zelleninhalt vor der Eingabe des Nutzers.

```
sub Create {
  my( $self, $parent, $id, $evthandler ) = @_;

  $self->SetControl(
    Wx::TextCtrl->new(
      $parent, $id, 'Default value'
  ) );

  $self->GetControl->PushEventHandler( $evthandler );

}

sub Destroy {
  my $self = shift;

  $self->GetControl->Destroy if $self->GetControl;
  $self->SetControl( undef );
}
```

Listing 1

```
sub EndEdit {
  my( $self, $row, $col, $grid ) = @_;

  my $value = $self->GetControl->GetValue;
  my $oldValue =
     $grid->GetCellValue( $row, $col );

  $grid->SetCellValue(
     $row, $col, $value * 1.19 }

  $self->GetControl->Destroy;
  $self->SetControl( undef );

   return 1;
}
```

Die `1` besagt, dass etwas geändert wurde und der *Renderer* angeworfen werden muss.

Wx::HTML

Dieses Widget wurde bereits in Folge 8 angesprochen. Doch es kann mehr, als lediglich einfaches HTML darzustellen. Durch bestimmte Tags kann jedes Widget auch in HTML eingebettet werden und wird weiterhin normal wie ein Widget behandelt. Das hilft nicht nur, die HTML-Seiten optisch und funktional anzureichern, sondern eröffnet Möglichkeiten, welche fließend ins nächste Kapitel übergehen. Das vollständige Layout einer Oberfläche ließe sich so erzeugen, in einer Datei speichern und wiederverwenden. Es benötigt nur etwas Code, der dem `HtmlParser` beibringt, welches Widget wir mit dem Tag `<piano />` meinen.

```
package Piano::wxHtmlTag::Handler;

use strict;
use base 'Wx::PlHtmlTagHandler';
use Wx::Perl::Piano;

sub new { return shift->SUPER::new}

sub GetSupportedTags { return 'PIANO' }

sub HandleTag {
    my( $this, $htmltag ) = @_;
    my $parser = $this->GetParser;
    my $htmlwin = $parser->GetWindow;
    my $piano =
        Wx::Perl::Piano->new( $htmlwin, -1 );
    my $cell =
        Wx::HtmlWidgetCell->new( $piano );
    my $cnt = $parser->GetContainer;
    $cnt->InsertCell( $cell );

    return 1;
}
```

Dieser *Handler* muss nur noch direkt nach dem

```
my $hw = Wx::HtmlWindow->new($panel, -1);
```

aufgerufen werden.

```
$hw->GetParser->AddTagHandler(
                  Piano::wxHtmlTag::Handler );
```

Jetzt kann das HTML inklusive Klavier mit `SetPage` oder `LoadFile` geladen werden.

Wx::XRC

Nun von HTML zum abstrakteren *XML*. `XRC` steht für *XML Resource Compiler*. Das ist eine Bibliothek die aus XML-Strukturen Widgethierarchien baut (kompiliert), als wären sie von Hand programmiert und dann ausgeführt. Die Abkürzung steht ebenso für das zugehörige Datenformat, das sich an den *XML*-Standard 1.0 hält. Es ist nur dafür gedacht gelesen zu werden. Wenn man den aktuellen Zustand einer Oberfläche speichern möchte, so muss man auf das Marshalling per `Wx::AUI` zurückgreifen, das in Folge 5 gezeigt wurde.

`XRC` wird von Programmen wie dem *WxFormBuilder* erzeugt, die es erlauben, sich seine Oberfläche "zusammenzuklicken". (*WxFormBuilder* ist die ausgereifteste freie Alternative und nicht wesentlich schlechter als die kommerziellen Angebote der *WxWidgets*-Macher: *DialogBlocks* und *WxDesigner*.) Das umfasst nicht den von *VisualStudio* oder *NetBeans* gewohnten Komfort. (Widgets können nicht millimetergenau verschoben werden.) Aber man sieht alle Möglichkeiten und Optionen ausgebreitet und erkennt die Folgen einer Entscheidung sofort, was sicherlich ein guter Weg ist, um *WxWidgets* zu lernen. Weil es vielleicht der einfachste Einstieg in die Programmierung von *Wx* ist, mag sich mancher fragen, warum dieses Tutorial damit nicht begann. Das Ziel dieser Serie ist es, *WxPerl* in seiner Tiefe zu vermitteln, um am Ende beide Wege gehen zu können. Außerdem befreit `XRC` den Programmierer nur teilweise vom Programmieren. Grundlegende Konzepte sollte man trotzdem verstanden haben.

`XRC` erlaubt nicht nur einen spielerischen Einstieg und eine schnellere Erzeugung der Oberfläche. Auch eine Trennung nach *MVC* (Model View Controller) wie in *Catalyst* lässt sich damit einfacher realisieren. Diese Trennung wurde ja ursprünglich unter Smalltalk für die *GUI*-Programmierung "erfunden". Ein wesentlicher Vorteil dieses Ansatzes ist auch die mögliche getrennte Entwicklung von Oberfläche und Logik. Erstere kann dem Kunden oder Chef stets gezeigt werden, selbst wenn das Programm gerade streikt. Jenen Zielpersonen ist es somit auch möglich kleinere Anpassungen selbst vorzunehmen. Auch Übersetzern kann das sehr entgegen kommen, wenn sie anfassen können, was übersetzt werden soll. Nicht zuletzt erleichtert `XRC` auch ein mehrfaches Verwenden von Dialogen, auch über Projekt- und Sprachgrenzen hinweg (*WxPython*).

Leider kamen die Entwickler des *WxFormBuilder* noch nicht auf den Gedanken, dass jemand seine Projekte direkt als *XRC* speichern möchte. Man ist gezwungen, auf der unteren Reiterleiste *XRC* anzuwählen, den Text zu kopieren und in einer Datei zu speichern. Projekte lassen sich per Menü nur als `*.fbp` speichern. Adam Kennedys Modul `FBP` kann die Dateien einlesen und somit `XRC` ersetzen. Nur sind solche Dateien ebenso *XML* und enthalten dazu Einträge für sämtliche nicht benutzten Attribute und Ereignisse, weswegen der Autor *XRC* vorzieht. Auch ist es ein strategisches Risiko, der *Wx*-Entwicklung stets hinterherzulaufen. XRC wird dagegen von den *WxWidgets*-Autoren auf dem Stand gehalten.

Eine `XRC`-Datei kann die Informationen zu einem einzelnen Widget beinhalten oder eine vollständige Oberfläche inklusiver aller Menüs, Dialoge und Satuszeilen. Um sie einzubinden, bedarf es folgender Zeilen:

```
use Wx::XRC;

# Bilder könnten enthalten sein
Wx::InitAllImageHandlers();
my $xr = Wx::XmlResource->new();
$xr->InitAllHandlers();
$xr->Load('ui.xrc');
my $frame = $xr->LoadFrame(undef, 'FrameID');
```

Neben `LoadFrame` gibt es eine Reihe entsprechender Methoden wie `LoadToolBar`, durch welche die angeforderten Teile der XRC-Struktur in einen gebrauchsfertigen Zustand gebracht werden.

```
Wx::XmlResource::GetXRCID('m_textCtrl1');
```

Durch diesen Aufruf bekommt man eine Referenz auf jedes beliebige Widget, sofern es bereits kompiliert wurde. Die dafür nötige XRC-ID ist im *WxFormBuilder* immer die oberste Eigenschaft der Leiste. Sie ist etwas verwirrend *name* genannt, weil es dem Namen des kompilierten Widgets entspricht. Diese ansonsten unbedeutende Eigenschaft eines Widgets (immer der letzte Parameter der `new`-Methode) wurde bis jetzt meist ignoriert. Das `wxID_ANY` der Eigenschaft *id* bezieht sich auf die *Wx*-ID und ist gleichbedeutend dem `-1` der hiesigen Beispiele. Alternativ zu `GetXRCID` kann man auch `FindWindow` verwenden, das alle Widgets unabhängig von `XRC` kennen. Damit sucht man rekursiv alle Kinder eines Widget ab.

```
$frame->FindWindow('m_textCtrl1')
```

Es gibt jedoch sichtbare Elemente, die nicht eigenständig auf den Nutzer reagieren können und die nicht von `Window`

abgeleitet sind, wie etwa Menüeinträge. Um sie mit einem `EVT_MENU` zu verknüpfen, muss `GetXRCID` verwendet werden. Da dieser Befehl sehr schnell Überbreite oder verschachtelte Zeilen im Code erzeugen kann, helfe ich mir gerne mit dem Alias:

```
my $XID = \&Wx::XmlResource::GetXRCID;

EVT_MENU($frame, &$XID('m_menuItem3'),
   sub { ...
```

Benötigt man ein selbstgemachtes oder zu neues Widget (zum Beispiel `Wx::Mein::Beamer`), so ist das kein Grund, auf `XRC` zu verzichten. Im XML könnte an passender Stelle stehen:

```
<object class="Beamer">
  <colour>#ff0000</colour>
  <pos>20, 60</pos>
  <size>200, 50</size>
</object>
```

Damit daraus etwas Benutzbares wird, muss man "nur" einen eigenen `XmlHandler` schreiben, der die Erzeugung des Widgets an adäquater Stelle veranlasst. Dazu erbt unser `Beamer::XmlHandler` von `Wx::PlXmlResourceHandler` und meldet

```
sub CanHandle {
    my( $self, $xmlnode ) = @_;
    return $self->IsOfClass
                ($xmlnode, 'Beamer' );
}

sub DoCreateResource {
    my( $self ) = shift;

    die
    'LoadOnXXX not supported by this handler'
        if $self->GetInstance;

    my $ctrl = Wx::Mein::Beamer->new
      ( $self->GetParentAsWindow,
        $self->GetID,
        $self->GetColour( 'colour' ), ...);

    $self->SetupWindow( $ctrl );
    $self->CreateChildren( $ctrl );

    return $ctrl;
}
```

Um den *Handler* ins `XRC` einzuflechten, fehlt im Hauptprogramm zwischen `Wx::InitAllImageHandlers();` und Laden der Datei nur noch:

```
$xr->AddHandler( Beamer::XmlHandler->new );
```

Weitere Flächen

Zum Schluss sollen noch zwei neuere Widgets nachgereicht werden. Das eine gehört eigentlich in den vierten Teil, in dem es um Flächen ging. Das andere ist auch eine Fläche, allerding die Schwester des randlosen Popupfensters, das im siebenten Teil beschrieben wurde. Dort wurde vorgeführt, wie sich selbstgebaute und bemalte Popups durch Anklicken schließen. Manchen ist das immer noch lästig oder ungewohnt. Sie erwarten, dass Popup-Fenster sich automatisch schließen, sobald sie durch einen Klick woanders hin den Fokus verlieren. Genaus das erhält man, wenn man im Beispiel des Abschnittes "DC" `PopupWindow` mit `PopupTransientWindow` austauscht und `EVT_LEFT_DOWN` ignoriert.

Das `Wx::CollapsiblePane` wird benutzt wie ein normales *Panel*. Es empfängt nur bei der Erzeugung einen dritten Parameter. Die Überschrift links oben im Rahmen, entspricht etwa dem, was man von einer `RadioBox` oder dem `StaticBoxSizer` kennt. Sie ist notwendig, da sie das einzig Sichtbare ist, wenn sich diese Fläche zusammenfaltet (kollabiert). Sie dient auch als Knopf um auf- und zuzufalten. Letzteres ist der anfängliche Zustand, was manchmal ein `$pane->Expand();` oder `$pane->Collapse(0);` notwendig macht. Auf den Moment des Faltens wird per `EVT_COLLAPSIBLEPANE_CHANGED` reagiert. Nur die Bestückung ist etwas hakelig, da das eigentliche Panel nur über die Methode `GetPane` erhältlich ist. Es bekommt den *Sizer* zugewiesen, welcher die Kinder zusammenhält.

Wenn das *CollapsiblePane* das einzige *Panel* eines Fensters ist, wird es sogar noch etwas kniffliger. In den meisten Fällen kann man sich die Laxheit erlauben, das oberste *Panel* einfach nur als Kind des Fensters zu erzeugen, ohne es durch einen *Sizer* einzufassen. Es wird sowieso den ganzen Platz des Fensters einnehmen, auch wenn es nicht die feinste Art ist und die *GUI-Designer* wie *WxFormBuilder* so etwas bewusst nicht tun. Sie spannen immer noch einen zusätzlichen *BoxSizer* um das *Panel*. Ohne den geht es mit *CollapsiblePane* gründlich schief. Wenn man im ausgefahrenen Zustand starten möchte, hat man sogar darauf zu achten, dass der `Fit`-Aufruf nach dem `SetSizer` erfolgt. Um alle Unsicherheiten auszuräumen, folgt der entscheidende Teil eines kleinen Beispielprogramms - siehe Listing 2.

Vorschau

Obwohl `WxPerl` relativ umfangreich ist und dies immer noch nicht alle Widgets waren, kann es nicht jeden speziellen Wunsch erfüllen. Es bietet aber auf mehreren Ebenen Ansatzmöglichkeiten, sich das eigene Idealwidget zu zimmern. Darüber wird im nächsten und letzten Teil geschrieben.

```
my $frame = Wx::Frame->new( undef, -1, __PACKAGE__);
my $pane = Wx::CollapsiblePane->new( $frame, -1, 'Überschrift' );
my $btn = Wx::Button->new($pane->GetPane, -1, 'collapse');
my $sizer = Wx::BoxSizer->new(wxVERTICAL);
my $psizer = Wx::BoxSizer->new(wxVERTICAL);

Wx::Event::EVT_COLLAPSIBLEPANE_CHANGED($frame, $pane, sub {
    $frame->SetTitle('...');
});
Wx::Event::EVT_BUTTON( $btn, $btn, sub {
    $pane->Collapse(1);
});

$psizer->Add($btn, 1,&Wx::wxGROW);
$pane->GetPane->SetSizer($psizer);
$pane->Expand();

$sizer->Add($pane, 1, &Wx::wxGROW);
$frame->SetSizer($sizer);
$frame->Fit();
$frame->Center();
$frame->Show(1);
```

Listing 2

Boris Däppen

Erstellen von E-Books für den Amazon

"Im November 2007 präsentierte Amazon einen neuen E-Book-Reader namens Kindle" [1] und seither hat es die, doch schon in die Jahre gekommene, Vision des elektronischen Buches, tatsächlich in Öffentlichkeit und Wirklichkeit geschafft. Wie bei jeder jungen Technologie herrscht auch hier viel Ungewissheit. Genau so groß wie die Anzahl verschiedener Schreibarten von "E-Book" scheint die Zahl der digitalen Formate zu sein. Für die Rechtschreibung schlägt Duden "E-Book" oder "E-Buch" vor, fürs Englische ist bei Oxford Dictionaries "e-book" zu lesen. Bei den Dateiformaten geht auf lange Sicht die Auswahl - wie in der Rechtschreibung auch - auf wenige "Schreibweisen" zurück. So finden das offene Format EPUB und das proprietäre Mobipocket weltweit Verbreitung. Diese beiden Formate werden wohl in Zukunft vorgeben, wie ein elektronisches Buch "zu schreiben" ist.

Nach wie vor ist in Deutschland das Gerät noch nicht völlig beim Endverbraucher angekommen. Die elektronische Tinte überzeugt nicht jede Leseratte und es bleibt weiterhin unklar in welchem Ausmaß dies gelingen wird. Fakt ist aber, dass der Rubel inzwischen rollt und damit dürfte mit E-Books auch auf lange Sicht zu rechnen sein. Da stellt sich natürlich die Frage, wie man als Perl-Programmierer hier mitmischen kann. Zu guter Letzt ist z.B. auch das $foo-Magazin als Printmedium dazu aufgefordert für die Zukunft gerüstet zu sein.

Wunschmodul

Um Bücher aus einem Perl-Programm heraus generieren zu können wünscht man sich ungefähr so etwas, wie in Listing 1 dargestellt.

```
$buch->titel ('Mein Leben als Programmierer');
$buch->text  ('Ich sehe ein leuchtendes Viereck.');
$buch->render('/home/business/mein_buch.ebook');
```

Listing 1

Zum Glück findet sich auf CPAN bereits einiges in dieser Richtung.

EPUB generieren

Unproblematisch ist der offene Standard EPUB. Zum einen ist er bequem von Hand editierbar, da es sich bei EPUB im Großen und Ganzen nur um gezippte XML-Dateien handelt. Gerade weil der Standard durchschaubar und offen ist, findet sich auf CPAN dafür gute Unterstützung. Ein Beispiel hierfür ist `EBook::EPUB` von Oleksandr Tymoshenko. Das Modul bietet die vorhin beschriebene gewünschte Funktionalität.

Mobipocket generieren

Eigenes Modul EBook::MOBI

Etwas anders sah es bis vor kurzem aus, wenn man mit Perl ein Buch für den populären Kindle von Amazon erstellen wollte. Da Amazon sein Format nicht offen dokumentiert gibt es auch kaum freie Bibliotheken um für diese Plattform zu entwickeln. Zwar gibt es das freie Programm *Calibre* [2], aber aus Entwickler-Sicht hätte man doch lieber pures Perl. So kam es, dass auf Initiative von Renée Bäcker für das $foo-Magazin das Modul `EBook::MOBI` von mir entwickelt und auf CPAN veröffentlicht wurde.

Um es gleich vorweg zu nehmen: Das Modul beruht zu einem großen Teil auch auf der Arbeit anderer, die ihren Code unter Opensource-Lizenzen im Internet veröffentlicht haben.

Zwar hatte auch ich noch einige Kniffe zu lösen, aber ohne diese Vorarbeit wäre `EBook::MOBI` wohl nicht möglich geworden. Daher nenne ich die zentralen Anlaufstellen gerne:

• Das auf CPAN auffindbare Modul `Palm` ist essentiell. Denn Mobipocket ist lediglich eine Erweiterung des älteren Palm-DOC Standards und kann darum auch mit diesen Palm-Bibliotheken gepackt werden. Es wird eine lokale Kopie unter dem Namensraum `EBook::MOBI::MobiPerl::Palm` benutzt, da es leichte Abänderungen zum offiziellen `Palm` auf CPAN gibt.

• Mit Hilfe von `Palm` hat Tompe das Projekt *MobiPerl* [3] umgesetzt. Es handelt sich hier um ein Tool zum Konvertieren von HTML. `EBook::MOBI` ist im Prinzip ein API-Wrapper um MobiPerl, was dank der GPL auch rechtlich kein Problem ist. Die von mir benötigten Dateien von MobiPerl liegen unter `EBook::MOBI::MobiPerl`.

• Außerdem war auch die Community [4] mit ihren Dokumentationen und Foren sehr hilfreich. Da Mobipocket nicht offen ist, beruht das meiste Wissen auf Reverse Engineering, welches durch verschiedene Leute und Projekte zusammengetragen wird.

Erstellen eines E-Books aus POD

Die Stärke von `EBook::MOBI` liegt in der Fähigkeit E-Books direkt aus Perls eigenem Dokumentationsformat POD zu generieren. Zu diesem Zweck wird das POD intern in Mobipocket-verträgliches HTML umgewandelt. Anhand eines Beispiels will ich den Code näher erläutern.

Zuallererst aber noch ein Hinweis wegen Encoding-Problemen. Soweit ich weiß werden in Mobipocket nur zwei Encodings unterstützt: CP1252 (WinLatin1) und UTF-8 [5]. Wenn Inhalt direkt im Perl-Skript steht sollte daher

```
use utf8;
```

verwendet werden, damit alle Strings Perl-intern im richtigen Format sind, sonst werden Sonderzeichen nicht richtig dargestellt. Es muss also auf jeden Fall auf die Kodierung geachtet werden!

Das Modul wird über ein OO-Interface angesprochen.

```
use EBook::MOBI;
my $book = EBook::MOBI->new();
```

Übliche Meta-Angaben können vorgenommen werden, man kann sie aber auch weglassen. Bei Titel und Autor handelt es sich nicht um den Inhalt des Buches, dies sind lediglich Meta-Angaben die im Reader dann bei der Buchauswahl angezeigt werden.

```
$book->set_filename('./foo-Beispiel.mobi');
$book->set_title   ('foo-Magazin');
$book->set_author  ('Boris Daeppen');
```

Der Inhalt des Buches kann nun in der benötigten Reihenfolge hinzugefügt werden. Als erstes kommt meistens eine Titelseite. Hierfür verwenden wir die MHTML-Methode. Das "m" steht hier für "mobi" und meint HTML welches für Mobipocket geeignet ist. (Diese Terminologie stammt aus eigener Feder, ist also zur Recherche nicht geeignet.)

```
$book->add_mhtml_content(
    " <h1>foo-Magazin</h1>
     <p>Ein Beispiel für Ausgabe 23.</p>\n"
);
```

Der aufmerksame Leser hat bemerkt, dass vor dem `h1`-Tag ein Leerzeichen ist. Das verhindert, dass dieser Titel ins Inhaltsverzeichnis aufgenommen wird, was auch der Dokumentation zu entnehmen ist.

Wir könnten nun eine neue Seite beginnen.

```
$book->add_pagebreak();
```

Ich werde dies aber für das Beispiel nicht tun, da es am Schluss ein schönes Foto geben soll, also alles auf eine Seite!

Nach der Titelseite folgt üblicherweise das Inhaltsverzeichnis. Also fügen wir dies ein.

```
$book->add_toc_once();
```

Die Methode hat "once" im Namen, weil sie nur einmal sinnvoll aufgerufen werden kann. Jeder weitere Aufruf wird einfach ignoriert und hat keine Folgen. Da ich selbst keine andere Möglichkeit gefunden habe, wird das Inhaltsverzeichnis als normale Buchseite gerendert. Der Aufruf der Methode definiert lediglich die Position des Inhaltsverzeichnisses in den Seiten, berechnet wird das Inhaltsverzeichnis erst später mit `make()` (siehe weiter unten). Bei EPUB wird das Inhaltsverzeichnis jeweils über ein eigenes Menu angesteuert - hier wird es wie jede andere Seite behandelt (also z.B. als die Seite nach der Titelseite). Auch hier verzichten wir um des Fotos willen auf einen Seitenumbruch.

Nun ist es an der Zeit, dem Buch den eigentlichen Inhalt zu spendieren. Wir könnten dies wieder über die MHTML-Methode tun, wie wir es bei der Titelseite gemacht haben. Viel spaßiger ist es aber, direkt POD einzufügen!

```
my $pod = <<END;
=head1 Wunsch Modul
[...]
END

$book->add_pod_content($pod);
```

Auf diese Art können nun weitere Kapitel angehängt werden. Das Hinzufügen von POD kann auf verschiedene Arten beeinflusst werden. So erlaubt `head0_mode` das Verwenden von `=head0` im POD als oberste Kapitel, was unter Umständen beim Umgang mit Perl-Dokumentation nützlich ist. Mit `pagemode` kann bewirkt werden, dass vor jedes neue Kapitel automatisch ein Seitenumbruch gesetzt wird. Diese Möglichkeiten möchte ich nur kurz erwähnen, sie sind nicht notwendig und können bei Interesse in der Dokumentation des Moduls nachgeschlagen werden. Ins Inhaltsverzeichnis werden später nur die obersten Kapitel übernommen, also alles was als MHTML dann als `h1`-Tag geschrieben wird. Je nach Modus betrifft dies dann `=head1` oder `=head0`.

Irgendwann ist man mit dem Hinzufügen fertig, nun muss das Buch "gemacht" werden. Das Umwandeln des POD in MHTML geschieht bereits beim Hinzufügen. Was aber noch fehlt, ist die Berechnung der Links für das Inhaltsverzeichnis.

```
$book->make();
```

Wer sich das Ergebnis als komplettes MHTML anschauen möchte kann dies tun.

```
$book->print_mhtml();
```

Nun muss das Ganze nur noch in das Mobipocket-Format gepackt und als Datei gespeichert werden. Hier werden auch allfällige Bilder ins Format gepackt (siehe nächstes Kapitel).

```
$book->save();
```

Sonderfall: Bilder im POD

Um POD für Bücher zu gebrauchen wäre die Fähigkeit mit Bildern umzugehen ganz nett. `EBook::MOBI` wertet hierfür ein inoffizielles "image" aus.

```
=image /pfad/zum/bild.jpg Abb. 1: Ein Kamel.
```

Alles was nach dem Pfad kommt wird als Bildunterschrift interpretiert. Bilder sollten besser über die POD-Methode hinzugefügt werden. Über die MHTML-Methode kann man ebenfalls Bilder hinzufügen, man muss jedoch sehr auf die Syntax achten. Dieser Regex hier (aus `EBook::MOBI::Mhtml2Mobi`) muss matchen:

```
m/.*<img.*\ssrc=["']([\S*)["']\s.*>/g
```

Als Beispiel würde, der in Listing 2 gezeigte Code, gehen.

Die Details zu `recindex` sind weiter unten beschrieben. Ich empfehle aber generell, bei Bildern über `add_pod_content()` zu gehen, da dann alles automatisch berechnet wird.

Technische Details

Besonderheiten von Mobipocket

Nach meinen eigenen Recherchen besitzt Mobipocket einige Besonderheiten. Überraschend war zum Beispiel, dass auch in Mobipocket letzten Endes nur komprimiertes HTML steckt (wie es ja auch bei EPUB der Fall ist). Leider gibt es aber einige spezifische Anpassungen, ganz so einfach ist es dann also doch nicht. Die folgenden Erkenntnisse möchte ich daher niemandem vorenthalten:

• Das `a`-Tag für Links geht zwar für Hyperlinks, dokumentintern funktionieren sie jedoch nach meiner Erfahrung nicht. Durch Reverse Engineering fertiger E-Books wurde ersichtlich, dass man ein spezielles Attribut benutzen muss, welchem das Ziel des Links als Zeichenposition mitgegeben wird! Das sieht dann z.B. so aus, wie in Listing 3 dargestellt.

```
$book->add_mhtml_content(
      "<img src="/pfad/zum/bild.jpg" recindex="1" >\n"
    . "<p>Bild 1: Ein Kamel</p>\n"
);
```

Listing 2

```
<!-- Hinweis: Kapitel "Kamel" fängt beim 458zigsten Zeichen an -->
<a filepos="00000458">Kamel</a>
```

Listing 3

Dies ist leider viel umständlicher als symbolische Links, da sich bei jedem eingefügten Zeichen alle Positionen dahinter verschieben.

• Mobipocket besitzt ein eigenes Tag um das Ende von "Buchseiten" zu markieren. Dies ist zum Beispiel nach der Titelseite oder vor jedem Kapitel angebracht. Das Tag sieht so aus:

```
<mbp:pagebreak />
```

• Die Syntax für Bilder ist ebenfalls abgewandelt. Da die Bilder nicht auf einem Dateisystem, sondern im Mobipocket-Container liegen, ändert sich die Adressierung. Mit `recindex` wird auf die internen Container von Mobipocket verwiesen:

```
<img recindex="0001">
```

Diese Erkenntnisse wirken im Nachhinein etwas trivial, es hat jedoch einiges an Spitzfindigkeit gekostet an sie zu gelangen, zumal ich im Internet hierzu nicht immer fündig wurde. Weitere Informationen und Ausführungen sind in der Dokumentation zu `EBook::MOBI::Mhtml2Mobi` zu finden.

Umwandeln von POD in Mobi-HTML

Mit dem Aufruf der Methode add_pod_content() geschieht eine Umwandlung des POD in Mobi-spezifisches HTML, welches ich MHTML nenne. Der ganze Code dazu ist ins Modul `EBook::MOBI::Pod2Mhtml` ausgelagert. Dieses Modul kann bei Bedarf unabhängig verwendet werden, die Dokumentation liegt dem Code bei.

Da einige Features von MHTML sehr speziell sind, wurde die volle Kontrolle über den Parser beim Umwandeln benötigt. Die Wahl fiel auf `Pod::Parser`.

Der Grund hierfür war die steile Lernkurve und große Flexibilität bei der Implementation. Von `Pod::Parser` erbt man lediglich die Grundfunktionalität:

```
use Pod::Parser;
our @ISA = qw(Pod::Parser);
```

Nun werden die Methoden von `Pod::Parser` überschrieben. Die folgenden Methoden wurden von mir benutzt:

```
# Anfang parsen
begin_input()
# Ende parsen
end_input()
# ein Kommando (mit = eingeleitet)
command()
# Code (mit Leerzeichen eingeleitet)
verbatim()
# Normaler Text
textblock()
# Inline POD für Markup
interior_sequence()
```

Jedes Mal, wenn `Pod::Parser` später im POD auf einen Abschnitt stößt, ruft er die entsprechende Methode auf und übergibt das Gefundene. Speziell ist hierbei die Methode `interior_sequence()` (welche für die Übersetzung des inline-POD zuständig ist), da sie nicht vom Parser aufgerufen wird. Diese Methode muss bei Bedarf selbst durch `interpolate()` aufgerufen werden, was zum Beispiel innerhalb von `textblock()` Sinn machen kann:

```
my $inlinePOD_processed =
 $parser->interpolate($paragraph, $line_num);
```

Alle anderen Methoden werden aber vom Parser bei der Abarbeitung selbst aufgerufen. Stößt er z.B. auf ein Kommando ruft er `command()` auf und übergibt den aktuellen Kontext. So sieht der Kopf der Methode aus, der überschrieben werden muss:

```
sub command {
  my ($parser, $command, $paragraph,
    $line_num) = @_;
```

Auf diese Weise kann ich sehr einfach eigene Kommandos parsen. Da POD keine Bilder unterstützt, dies aber bei E-Books wünschenswert sein kann, definiere ich einfach ein Kommando `=image` und behandle den Fall in der Methode entsprechend:

```
if ($command eq 'image') {
```

Mit einer kurzen Befehlskette auf der Shell lässt sich eine Übersicht aller unterstützten Kommandos erstellen:

```
$ grep 'if (
  $command ' lib/EBook/MOBI/Pod2Mhtml.pm | \
  > cut -d \' -f2 | pr -T -2
image                           head1
over                            head2
back                            head3
cut                             head4
head0                           item
```

Hier sieht man neben `=image` auch die Implementation von `=head0`, was ja ebenfalls nicht dem POD-Standard entspricht, aber nützlich sein kann um mehrere `=head1` zu strukturieren!

Das Übersetzen von Listen hat sich als nicht ganz simpel herausgestellt, POD lässt dem Autor hier sehr viel Freiheiten die alle abgefangen werden müssen. So kann der Text eines Items auch nach zwei Zeilenumbrüchen folgen, statt direkt auf der Zeile zu stehen (beides Zusammen geht aber auch). Eine Eintrag mit dem Inhalt "5" in einer unsortierten Liste sieht üblicherweise so aus:

```
=item * 5
```

Die Formatierung darf aber auch anders sein (was vor allem bei viel Text Sinn macht):

```
=item *

5
```

`POD::Parser` arbeitet hier leider nicht ganz sauber und ruft im zweiten Fall für die "5" statt der Methode `item()` die Methode `textblock()` auf. Man muss dann selbst mit internen Variablen feststellen ob dies nun ein Textblock im Kontext einer Liste ist oder nicht. Bei mehrfach verschachtelten Listen wird dies dann etwas unübersichtlich. Hier würde ich mir eine besser Unterstützung seitens des Parsers wünschen.

Nachdem alle benötigten Methoden überschrieben worden sind, lässt sich der Parser starten:

```
my $parser = EBook::MOBI::Pod2Mhtml->new();
$parser->parse_from_filehandle(
  $pod_in, $html_out);
```

Die Schnittstelle zu MobiPerl

Nachdem man das E-Book als MHTML hat, muss es in Mobipocket gepackt werden. Dies ist genau der Job den MobiPerl erledigen kann. Die Schnittstelle zu MobiPerl habe ich unter `EBook::MOBI::Mhtml2Mobi` implementiert, wobei dieses Modul bereits aus abgeändertem MobiPerl-Code besteht. Hier werden das MHTML und die Meta-Angaben ins Format geschrieben. Außerdem werden die benötigten Bilder aus dem MHTML rausgesucht, vom Dateisystem gelesen und ebenfalls ins Format gepackt. Noch zu erwähnen ist, dass die Bilder mittels `EBook::MOBI::Picture`, falls nötig auf 520 x 622 Pixel verkleinert werden. Dies scheint für den Kindle ein guter Wert zu sein [6] und spart Speicherplatz.

Fazit

In Perl lässt sich einiges mit E-Books machen. Zwar fehlt ein mächtiges Tool für Mobipocket, `EBook::MOBI` füllt diese Lücke aber durch die pragmatische Möglichkeit, solche Bücher zu erstellen. Die Flexibilität in der Gestaltung und Platzierung der Seiten ist beschränkt, dürfte aber für so einige Szenarien ausreichen. Wer selber tricksen will kann sich an der `add_mhtml_content()` Methode versuchen.

Die Generierung von E-Books für den proprietären Kindle stellt nun ein Kinderspiel dar. Für das offene EPUB gilt dies sowieso.

Da das Modul für einen konkreten Anwendungsfall entstanden ist und diesem vorerst auch genügt, wird es von selbst nicht weiterentwickelt werden. Bei fehlenden Features oder Verbesserungsvorschlägen ist Mitarbeit daher herzlich willkommen. Das Projekt lädt auf Github [7] zum *social coding* ein.

Links

[1] *http://de.wikipedia.org/wiki/Ebook* am 03.05.2012
[2] *http://calibre-ebook.com/*
[3] *https://dev.mobileread.com/trac/mobiperl*
[4] *http://www.mobileread.com/*
[5] *http://www.mobipocket.com/dev/article.asp?BaseFolder=prcgen&File=tagref_opfxmetadata.xml*
[6] *http://kindleformatting.com/formatting.php*
[7] *https://github.com/borisdaeppen/EBook--MOBI*

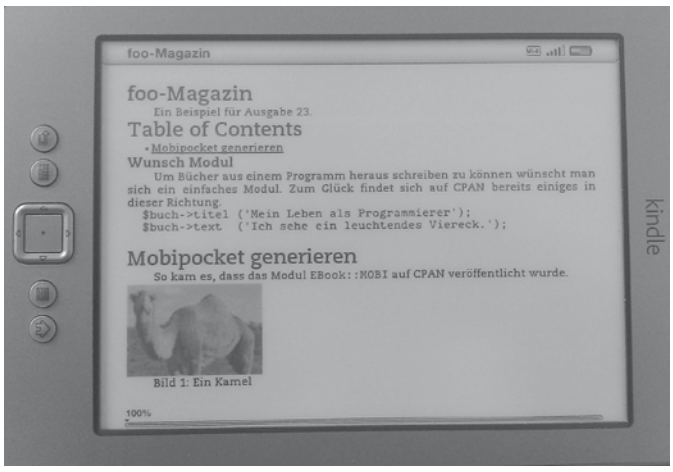


Abb. 1: Das E-Book aus den Beispielen im Artikel

Daniel Bruder & Renée Bäcker

Module verwalten mit Dist::Zilla

Zum guten Ton in der Perl-Programmierung gehört es, modular zu programmieren. Das Aufteilen des Programmcodes in verschiedene Module bietet viele Vorteile. So wird die Wartbarkeit der Anwendung erhöht, da immer nur einzelne Module verbessert werden müssen und nicht die ganze Anwendung.

Auch kann man einzelne Teile leichter austauschen ohne dass ein Update der gesamten Anwendung notwendig wird.

Man kann sich dann auch leicht mit Pinto oder etwas ähnlichem eine eigene CPAN-ähnliche Infrastruktur aufbauen (siehe auch $foo Ausgabe 2/2012). Dann können beim Kunden sehr einfach die Module installiert werden.

Einen kleinen Nachteil haben einzelne Module bzw. Distributionen aber doch: Man muss viel mehr Infrastruktur schaffen und warten. So sind in jeder Distribution Dateien wie MANIFEST, Changes, README etc. enthalten. Man muss bei jedem Release darauf achten, dass man diese Infrastruktur aktualisiert.

Wenn dann noch viele Module auf dem CPAN liegen und man Patches von fremden Personen bekommt, kann die Pflege der eigenen Module viel Zeit beanspruchen.

An dieser Stelle kommt Dist::Zilla ins Spiel. Mit Dist::Zilla wird die Pflege und das Releasen von Distributionen stark vereinfacht. In diesem Artikel wird Dist::Zilla vorgestellt und anhand eines Beispiels gezeigt, was man noch an nützlichen Erweiterungen nutzen kann. Auch werden wir zeigen, wie man eigene Erweiterungen erstellt.

Als Versionskontrollsystem wird *git* zum Einsatz kommen. Zum einen, weil *git* immer häufiger bei Perl-Programmierern zum Einsatz kommt und zum anderen gibt es einige nützliche Dist::Zilla-Plugins, die *git* voraussetzen.

Einführung in Dist::Zilla

Das Programmieren kann Dist::Zilla einem nicht abnehmen, aber das Verwalten der Distribution kann weitgehend automatisiert werden.

Auf *dzil.org* heißt es

Dist::Zilla is meant to make it easy to release your free software to the CPAN. Like ExtUtils::MakeMaker or Module::Build, it takes your source code and builds a tarball that can be installed by other users. Unlike those tools, though, Dist::Zilla takes care of all kinds of boring and mindless tasks so that you, the programmer, are free to just write your code.

Das beschreibt den Sinn und Zweck von Dist::Zilla ganz gut. Nur muss man nicht unbedingt CPAN als Ziel der Veröffentlichung benutzen, wie oben schon angedeutet. Mit ExtUtils::MakeMaker können auch viele Aufgaben automatisiert werden, aber vieles ist nicht so bequem wie bei Dist::Zilla.

In Abbildung 1 ist der Lebenszyklus eines Moduls beispielhaft dargestellt.

Abb 1: Lebenszyklus eines Moduls

Bei der Dokumentation kann Dist::Zilla zumindest teilweise helfen. Dazu greift es auf Pod::Weaver zurück, das in einem der folgenden Abschnitte genauer vorgestellt wird. Die Schritte 2 bis 6 kann Dist::Zilla selbstständig übernehmen.

Vor dem ersten Modul

Bevor es jetzt richtig losgeht, muss natürlich Dist::Zilla installiert werden. Mit `cpan Dist::Zilla` kommt ein ganzer Haufen an CPAN-Modulen auf den eigenen Rechner. Die Liste der Abhängigkeiten (und deren Abhängigkeiten) ist ziemlich lang, man sollte es also in einer ruhigen Minute installieren.

Nach der Installation steht ein neues Tool zur Verfügung: `dzil`

Als erstes kann man sich einmal den Output von `dzil` anschauen - siehe Listing 1.

Die wichtigsten Kommandos während der Entwicklung einer Distribution werden zunächst folgende sein:

- `setup` zur Vornahme der Erstkonfiguration von Dist::Zilla
- `new` zum Anlegen einer neuen Distribution
- `build` zum Zusammenbauen einer geschriebenen Distribution
- `test` zum Testen der Distribution vor deren Veröffentlichung
- `release` zum Veröffentlichen der Distribution
- `clean` zum Entfernen des letzten Build

Als weitere interessante Tools sind unbedingt noch `listdeps` und `authordeps` zu erwähnen:

- `listdeps` listet die von der Distribution benötigten Abhängigkeiten auf. Diese sind in dem Sinne von den Abhängigkeiten, die `authordeps` auflistet, zu unterscheiden, als `listdeps` die von der Distribution *selbst* benötigten Abhängigkeiten auflistet. Währenddessen listet `authordeps` die Abhängigkeiten auf, die benötigt werden, um die Distribution bei sich selbst zusammenzubauen.

Vor dem Erstellen des ersten Moduls muss Dist::Zilla zumindest rudimentär konfiguriert werden. Dazu ruft man `dzil setup` auf (siehe Listing 2).

Wie man sieht, sind hier noch keine Angaben zu irgendwelchen Modulen gemacht, sondern nur ganz allgemeine Angaben wie Autorenname, verwendete Lizenz und PAUSE-Account.

```
$ dzil
   Available commands:

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

            add: add modules to an existing dist
     authordeps: list your distribution's author dependencies
          build: build your dist
        install: install your dist
       listdeps: print your distribution's prerequisites
            new: mint a new dist
            nop: do nothing: initialize dzil, then exit
        release: release your dist
            run: run stuff in a dir where your dist is built
          setup: set up a basic global config file
          smoke: smoke your dist
           test: test your dist
```

Listing 1

```
$ dzil setup
  What's your name? $foo Perl-Magazin
  What's your email address? info@perl-magazin.de
  Who, by default, holds the copyright on your code?  [$foo Perl-Magazin]:
  What license will you use by default (Perl_5, BSD, etc.)?  [Perl_5]: Artistic_2_0
  Do you want to enter your PAUSE account details?  [y/N]: y
  What is your PAUSE id? foo
  What is your PAUSE password? foo
  config.ini file created!
  $
```

Listing 2

PAUSE ist der *Perl Author Upload Server*. Über diesen Upload Server kommen die ganzen Distributionen auf CPAN.

Die erstellte *config.ini* sieht damit folgendermaßen aus:

```
[%User]
name  = $foo Perl-Magazin
email = info@perl-magazin.de

[%Rights]
license_class    = Artistic_2_0
copyright_holder = $foo Perl-Magazin

[%PAUSE]
username = foo
password = foo
```

Wie Dist::Zilla arbeitet

Im `setup`-Schritt hat Dist::Zilla in dem Verzeichnis *~/.dzil* eine Datei *config.ini* erstellt. Diese wird in allen weiteren Entwicklungsschritten ausgelesen und benutzt, um die Distribution zusammenzubauen.

Man kann sagen, Dist::Zilla sucht sich seine Informationen kaskadenartig zusammen: *config.ini* dient dabei als allgemeine Basis und wird ergänzt durch die projektspezifische *dist.ini* (und optional noch die projektspezifische *weaver.ini*, um den Zusammenbau der Dokumentation noch feiner zu steuern).

Minting

Ein neues Modul wird geboren

Nun aber zum Erstellen einer neuen Distribution:

```
$ dzil new App::Test
[DZ] making target dir /Users/dbruder/
                                      App-Test
[DZ] writing files to /Users/dbruder/App-Test
[DZ] dist minted in ./App-Test
```

Dist::Zilla legt dabei folgende Struktur an:

```
$ tree App-Test
   App-Test
   |-- dist.ini
   `-- lib
    `-- App
           `-- Test.pm
```

Die Datei *dist.ini* sieht in diesem Fall so aus:

```
$ cat App-Test/dist.ini
   name     = App-Test
   author   = $foo Perl-Magazin
   license  = Artistic_2_0
   copyright_holder = $foo Perl-Magazin
   copyright_year   = 2012

   version = 0.001

   [@Basic]

   ; Kommentare werden mit einem
   ;Semikolon eingeleitet
```

Die meisten dieser Informationen werden aus der Datei *config.ini* aus dem Schritt vorher zusammengetragen und beim Bau der Distribution mit `dzil build` ausgelesen und verwendet.

Zur Hintergrundinformation ist zu sagen, dass sich Dist::Zilla beim Anlegen einer neuen Distribution mit `dzil new` eines sogenannten *Minting Process* bedient. Dieser ist zunächst vorkonfiguriert, kann aber ebenfalls (wie die meisten Schritte) angepasst und auf die eigenen Bedürfnisse angepasst werden. Zum Anpassen des Minting Prozesses später mehr.

Die Syntax von .ini-Dateien

Dist::Zilla funktioniert über weite Strecken durch die Verwendung von Plugins. Die Verwendung dieser Plugins wird über Konfigurationsdateien im INI-Format geregelt.

Um den Einsatz vieler Plugins zu vereinfachen, kann man diese Bündeln. In der Beispiel-*dist.ini* ist der Befehl `[@Basic]` zu finden.

`[@Basic]` bindet das PluginBundle *Basic* ein. Das dazugehörige Modul heißt Dist::Zilla::PluginBundle::Basic und beinhaltet bereits die folgenden Plugins:

- Dist::Zilla::Plugin::GatherDir
- Dist::Zilla::Plugin::PruneCruft
- Dist::Zilla::Plugin::ManifestSkip
- Dist::Zilla::Plugin::MetaYAML
- Dist::Zilla::Plugin::License
- Dist::Zilla::Plugin::Readme
- Dist::Zilla::Plugin::ExtraTests
- Dist::Zilla::Plugin::ExecDir
- Dist::Zilla::Plugin::ShareDir
- Dist::Zilla::Plugin::MakeMaker

- Dist::Zilla::Plugin::Manifest
- Dist::Zilla::Plugin::TestRelease
- Dist::Zilla::Plugin::ConfirmRelease
- Dist::Zilla::Plugin::UploadToCPAN

Aber nicht immer werden alle Plugins eines Bundles benötigt. Einzelne Plugins werden durch die Angabe

```
[Pluginname]
```

eingebunden. Erwarten die Plugins weitere Konfigurationsparameter, so können diese wie folgt festgelegt werden:

```
[Pluginname]
   Option1 = Wert1
   Option2 = Wert2
```

Die vielen Plugins bedeuten auch, dass man sich anschauen muss, wie der eigene Workflow ist und welche Plugins diesen unterstützen. Das bedeutet, dass man sich eine Zeit lang auf CPAN aufhalten und einiges an Dokumentation lesen muss.

Phasen

Dist::Zilla ist nicht nur durch Plugins extrem erweiterbar und auf die persönlichen Bedürfnisse anpassbar, sondern arbeitet zusätzlich in verschiedenen Build-Phasen. Welche Phasen es gibt und welche Plugins zum Einsatz kommen, kann mit dem Modul Dist::Zilla::App::Command::dumpphases (siehe Listing 3) ausgegeben werden (Ausgabe ist gekürzt).

Das Modul wird flügge

Nachdem die Basis für das Modul gelegt wurde, geht es jetzt darum, das Modul auf die Öffentlichkeit loszulassen. Der Schritt 2 aus dem Lebenszyklus wird später noch angesprochen. Der Schritt in die Öffentlichkeit beginnt hier mit dem Schritt 3, dem Testen.

Um die Testdateien muss sich der Programmierer selbst kümmern. Aber ähnlich wie der aus der Installation von Modulen bekannte Befehl `make test` gibt es bei Dist::Zilla `dzil test`. Dieser Befehl führt die Testskripte aus.

Der vierte Schritt ist der Schritt, bei dem Dist::Zilla seine ganze Stärke ausspielen kann. Bei der Erstellung der Distribution kommen die meisten Plugins von Dist::Zilla zum Zuge. Die Dateien wie *MANIFEST*, *README* und *META.yml* werden erstellt, Extra-Tests werden erzeugt, und das Archiv wird gepackt. Der Befehl für das Erstellen der Distribution ist `dzil build`.

Der abschließende Schritt wird mit `dzil release` gestartet. Das Release beinhaltet auch den `build`-Schritt.

Interaktion mit Git

Da *git* bei Perl-Programmierern immer beliebter wird, gibt es hier auch besonders viele Plugins. Einige werden im PluginBundle *Git* zusammengefasst. Zu diesen Plugins gehört `Git::Check`, mit dem z.B. überprüft wird, dass keine Dateien in den Verzeichnissen existieren, die nicht im Repository enthalten sind. Mit `Git::Commit` werden geänderte Dateien automatisch committed.

```
$ dzil dumpphases
   Phase: ExecFiles
    - role: -ExecFiles
    * @Basic/ExecDir => Dist::Zilla::Plugin::ExecDir

   Phase: ShareDir
    - role: -ShareDir
    * @Basic/ShareDir => Dist::Zilla::Plugin::ShareDir

   Phase: Gather Files
    - role: -FileGatherer
    * @Basic/GatherDir => Dist::Zilla::Plugin::GatherDir

   Phase: Releaser
    - role: -Releaser
    * @Basic/UploadToCPAN => Dist::Zilla::Plugin::UploadToCPAN

   Phase: Before Release
    - role: -BeforeRelease
    * @Basic/TestRelease => Dist::Zilla::Plugin::TestRelease
    * @Basic/ConfirmRelease => Dist::Zilla::Plugin::ConfirmRelease
    * @Basic/UploadToCPAN => Dist::Zilla::Plugin::UploadToCPAN
```

Listing 3

Auch für die Interaktion mit Github gibt es entsprechende Plugins.

Dank dieser Plugins sind einige Hausmeisterarbeiten nicht mehr vom Programmierer zu erledigen, sondern werden von Dist::Zilla durchgeführt.

Immer wieder etwas Neues

Zu diesem Zeitpunkt ist das Modul veröffentlicht. Hoffentlich kommt dann Feedback von den Nutzern. Nachdem die Patches eingespielt wurden, reicht in der Regel:

```
dzil test
dzil release
```

Eine bestehende Distribution umwandeln

Eine bereits existierende Distribution so umzuwandeln, dass man Dist::Zilla nutzen kann, ist sehr einfach. Für den Einstieg reicht es, die *README*, *META.(json|yml)* und *MANIFEST* zu löschen und dafür eine *dist.ini* anzulegen. In dieser stehen erst einmal nur die grundlegenden Dinge, vergleiche dazu die *dist.ini* oben.

Auf CPAN gibt es zusätzlich `Dist::Zooky`. Das Tool kann mit `Module::Build`, `ExtUtils::MakeMaker` und `Module::Install` umgehen und liest Metadaten aus. Mit Hilfe dieser Informationen wird anschließend eine *dist.ini* erstellt.

Alle weiteren Einstellungen können dann nach und nach vorgenommen werden.

POD erstellen lassen mit Pod::Weaver

Das Plugin Pod::Weaver übernimmt die oft mühselige Aufgabe, das Modul bzw. die Distribution mit den notwendigen zusätzlichen Informationen, wie Autor, Versionsnummer, Copyright-Informationen u.ä. zu versehen. Im Hinblick auf die Gleichförmigkeit der Dokumentation leistet es hierdurch hervorragende Dienste.

Pod::Weaver geht dabei folgendermaßen vor:

Es gibt ein generelles *Outline*, wie die Dokumentation aussehen soll. Für dieses sucht Pod::Weaver selbstständig die dazu notwendigen Teile zusammen. Zusätzlich können dem Outline weitere Kapitel hinzugefügt werden (etwa ein besonderer Disclaimer oder ähnliches). Desweiteren lassen sich Kapitel definieren, die beim Bau der Dokumentation "übriggebliebene" Teile sammeln und an der definierten Stelle ausgeben. Letzteres kann sehr praktisch sein, um beispielsweise alle Methoden in einem Block in der Dokumentation zusammenzufassen. Die Standarddefinition der Pod::Weaver-"Abteilungen" sieht folgendermaßen aus:

```
[@CorePrep]

[Name]
[Version]

[Region  / prelude]

[Generic / SYNOPSIS]
[Generic / DESCRIPTION]
[Generic / OVERVIEW]

[Collect / ATTRIBUTES]
command = attr

[Collect / METHODS]
command = method

[Leftovers]

[Region  / postlude]

[Authors]
[Legal]
```

Aus der Dokumentation auf *dzil.org*:

To get most of the benefit of Pod::Weaver, just add this line to your dist.ini:

[PodWeaver]

This will add sections for each module's name, abstract, version, authors, license, and a bunch of other bits in the middle like the synopsis and methods. It will let you use the commands =method, =attr, and =func to set up self-organizing sections for methods, attributes, and functions.

Insgesamt erleichtert Pod::Weaver die Arbeit mit dem oft mühseligen POD-Format erheblich. Darüber hinaus empfiehlt sich auch ein Blick auf `Pod::Weaver::Plugin::Wikidoc`, wenn man dem POD-Format noch weiter aus dem Weg gehen will.

Pod::Weaver::Plugin::Wikidoc würde folgendermaßen in *dist.ini* eingebunden werden:

```
[PodWeaver]
```

und danach erstellt man eine Datei *weaver.ini* (siehe Listing 4) mit gezeigten Inhalt:

```
[@Default]  ; Pod::Weaver::PluginBundle::Default
[-Encoding] ; auch sehr praktisch: gibt dem POD die Zeile '=encoding utf-8' hinzu...
[-WikiDoc]  ; Plugins werden bei Pod::Weaver mit '-Plugin' eingebunden...
    comment_blocks = 1
```

Listing 4

```
name            = Some-Cool-Module
author          = PAUSEID <PAUSEID@cpan.org>
license          = Perl_5
copyright_holder = PAUSEID
copyright_year   = 2012

version          = 1.000

[@Filter]
  -bundle = @Classic
  -remove = ExtraTests
  -remove = PodVersion

[Authority]
    authority = cpan:PAUSEID

[ConfirmRelease]
; und viele weitere Plugins
[PodWeaver]

[Test::Portability]
  test_vms_length  = 0
  test_ansi_chars  = 0
  test_one_dot     = 0

[PruneCruft]
  except = '.gitignore'

[ExecDir]
  dir= bin
```

Listing 5

Dist::Zilla anpassen: PluginBundles, Plugins und MintingProfiles schreiben

Der Vorteil von PluginBundles

Selbstverständlich bietet sich auch die Möglichkeit, sowohl *Plugins*, *PluginBundles* oder *MintingProfiles* zu schreiben, und die Arbeit mit Dist::Zilla damit noch weiter zu vereinfachen und das Verhalten von Dist::Zilla noch weiter an seine Bedürfnisse anzupassen.

Ein PluginBundle erlaubt es einem, eine lange, komplexe *dist.ini*-Datei durch eine einzige Zeile (oder zumindest eine Datei mit drastisch weniger Zeilen) zu ersetzen und auf diese Art und Weise auch über alle eigenen Distributionen und Module hinweg komplett einheitlich zu halten.

Mit der Zeit wird man nämlich sein eigenes Profil mehr und mehr zurechtschneiden und am Ende eine lange *dist.ini*-Datei entwickelt haben, welche man standardmäßig in jedes Projekt kopiert.

Beispielsweise würde durch ein eigenes PluginBundle mit dem Namen `Dist::Zilla::PluginBundle::Author::PAUSEID` aus einer komplexen, langen *dist.ini* wie in Listing 5 gezeigt wird, eine *dist.ini*-Datei werden wie diese:

```
name            = Some-Cool-Module
author          = PAUSEID <PAUSEID@cpan.org>
license          = Perl_5
copyright_holder = PAUSEID
copyright_year   = 2012

version          = 1.000

[@Author::PAUSEID]
```

Abgesehen von der einfacheren Handhabung, Übersichtlichkeit und Portablität, besteht der größte Vorteil eines PluginBundles in seiner Zentralisierung der umfangreichen *dist.ini*: so wird durch die Form des PluginBundles bei jeder Weiterentwicklung meiner persönlichen *dist.ini* jedes meiner Projekte automatisch seitens Dist::Zilla auf den neuesten Stand gebracht, heureka!

Für die Erstellung eines eigenen PluginBundles empfiehlt sich unbedingt ein Blick auf `Dist::Zilla::Role::PluginBundle::Easy`.

MintingProfiles

Durch ein `Dist::Zilla::MintingProfile::Author::PAUSEID` lässt sich zusätzlich noch das Verhalten von Dist::Zilla beim Erstellen einer neuen Distribution mit `dzil new` weiter an die eigenen Bedürfnisse anpassen. So lässt sich zum Beispiel

- die grundlegende Struktur eines neuen Moduls,
- der Inhalt von *dist.ini*
- der Inhalt von anderen Dateien, wie *weaver.ini*, etc.

anpassen.

Desweiteren kann man nicht nur ein Default-Verhalten erstellen sondern weitere "Untergeschmäcker" festlegen, welche steuern, wie das neue Modul angelegt wird. So kann ich also ein MintingProfile nach meinem eigenen Geschmack zu meinem persönlichem Einsatz anlegen und dazu noch ein weiteres "Unterprofil", welches ich für den professionellen Einsatz verwende.

Um ein Modul mit dem eigenen MintingProfile anzulegen, gibt man `dzil` den Parameter `-P Author::PAUSEID` mit, um das bestimmte MintingProfile `Dist::Zilla::MintingProfile::Author::PAUSEID` zu verwenden, und kann dabei noch eine spezielle Unterkonfiguration mit `dzil new -P Author::PAUSEID -p professional` anfordern.

Dist::Zilla::PluginBundle::PAUSEID vs. Dist::Zilla::PluginBundle::Author::PAUSEID?

Kurz gesagt erleichtert die Konvention, PluginBundles und MintingProfiles, die vorrangig dazu dienen, persönliche Einstellungen vorzunehmen, hinter einem Präfix `Author::` zu indexieren, sowohl auf Tester- als auch auf User-Ebene auf den ersten Blick zu ersehen, dass an dieser Stelle keine "offizielle" Distribution steht, sondern diese *persönliche Vorlieben* zusammenfassen und nicht im direkten Sinne zu Dist::Zilla's Kern zu zählen sind. Auch können Tester auf diese Art und Weise all jene PluginBundles einfacher herausfiltern und die Unterscheidung zwischen einem "tatsächlichen" Dist::Zilla::PluginBundle::Default und dem PluginBundle mit einer (erfundenen PAUSEID) "Default" als Dist::Zilla::PluginBundle::Author::Default wird dadurch deutlich einfacher...

Plugins

Sollte in dem Meer von Plugins doch noch etwas fehlen, kann man eigene Plugins schreiben. Der erste Schritt dabei ist die Überlegung, in welcher Phase (vgl. Ausgabe von `dumpphases`) das Plugin ausgeführt werden soll. Für jede Phase gibt es eine eigene Rolle, die eingebunden werden muss.

Wenn Dateien für die Distribution zusammengesammelt werden sollen, werden FileGatherer-Plugins verwendet. Wenn es einen finalen Check vor dem Release geben soll, werden BeforeRelease-Plugins aufgerufen. Wenn man sich unsicher ist, was möglich ist, sollte man die bestehenden Plugins anschauen.

Hier soll ein Plugin entwickelt werden, das nach dem Release eine neue Queue in einer OTRS-Instanz für die neue Version anlegt. Mit dem "nach dem Release" ist schon klar, dass die Rolle "AfterRelease" verwendet werden muss. Damit sieht der Anfang des Moduls wie folgt aus:

```
package Dist::Zilla::Plugin::CreateOTRSQueue;

use Moose;
with 'Dist::Zilla::Role::AfterRelease';
```

Die Rollen in `Dist::Zilla` sind als Interfaces aufgebaut (vgl. Moose-Tutorial in $foo Ausgabe 18) und die Rolle `AfterRelease` hat nur einen wichtigen Befehl:

```
require 'after_release';
```

und genau diese Methode muss das Plugin implementieren.

Die OTRS-Instanz wird mittels SOAP angesprochen. Dafür wird ein Benutzername und ein Passwort benötigt. Damit wird deutlich, dass das Plugin konfigurierbar sein muss. Die Einbindung über die *dist.ini* soll folglich so aussehen:

```
[CreateOTRSQueue]
user = SOAPUser
password = SOAPPassword
; hier folgen noch weitere Optionen
```

Um Konfigurationsparameter für das Plugin zu haben, müssen einfach nur Moose-Attribute definiert werden:

```
has user => (
    isa     => 'Str',
    is      => 'ro',
    default => 'root@localhost',
);

has password => (
    isa     => 'Str',
    is      => 'ro',
    default => 'root',
);
```

Das war's.

Soll es ein Konfigurationsparameter sein, der mehrere Werte akzeptiert, muss dieser als *multivalue* markiert werden:

```
has users => (
    isa     => 'ArrayRef[Str]',
    is      => 'ro',
    default => sub{[]},
);

sub mvp_multivalue_args {
  return qw(users);
}
```

Für die SOAP-Kommunikation wird `SOAP::Lite` verwendet:

```
use SOAP::Lite (
  autodispatch =>
    proxy => 'http://localhost/otrs/rpc.pl',
);
```

Der `after_release`-Methode wird der Dateiname des erstellten Archivs übergeben. Daraus kann man den Namen und die Version der Distribution herausziehen (siehe Listing 6).

Zusammenfassung

Die zuletzt gezeigten Schritte führen über die grundlegende Verwendung von Dist::Zilla für den Anfänger schon hinaus. Sie sollen aber aufzeigen, wie vielseitig sich Dist::Zilla steuern und anpassen lässt und sie sollen erste Schritte aufzeigen, für diejenigen, die mehr wollen. Zunächst aber wird es empfehlenswert sein, sich eine grundlegende Dist::Zilla-Konfiguration anzulegen, das heißt, sich auf CPAN im Dist::Zilla::Plugin::*-Namensraum umzusehen, und Stück für Stück die persönliche *dist.ini* zusammenzustellen. Wenn dies zuweilen auch ein etwas mühevoller Prozess sein kann, so macht er trotz allem viel Spaß, und, nicht zuletzt, erspart es einem in Zukunft eine Menge mühevoller Kleinarbeit. Die Autoren wünschen viel Erfolg beim Zusammenstellen des eigenen Toolkits!

```
sub after_release {
    my ($tgz) = @;

    my ($name,$version) = $tgz =~ m/(.*?)-(\d+\.\d+(?:\.\d+)/;

    my %options = (
        Name            => $name . '::' . $version,
        GroupID         => $self->group_id,
        ValidID         => 1,
        SystemAddressID => $self->system_address,
        SalutationID    => $self->salutation_id,
        SignatureID     => $self->signature_id,
        Comment         => 'Queue for version ' . $version,
        UserID          => 1,
    );

    my $rpc      = Core->new;
    my $queue_id = $rpc->Dispatch(
        $self->user   => $self->password,
        'QueueObject' => 'QueueAdd',
        %options,
    );

    print "New queue: ", $queue_id, "\n";
}
```

Listing 6

Herbert Breunung

Rezension - Das Kamel

Tom Christiansen, brian d foy & Larry Wall,
with Jon Orwant
Programming Perl
4. Edition, 1174 Seiten,
O'Reilly, März 2012
ISBN 978-0-596-00492-7
€45,00

Deutsche Übersetzung von Peter Klicman
Programmieren mit Perl
2. Auflage, 1128 Seiten broschiert
ISBN 978-3-89721-144-5
€56,00

Gilbert Steger
Zukunft.pl
18. Februar 2012
ASIN: B007B15Q3C
€4,11

Dustin Boswell, Trevor Foucher
The Art of Readable Code
202 Seiten
O'Reilly, 2011
ISBN 978-0-596-80229-5
€29,00

Endlich ist die Karawane mit dem Wüstenschiff eingetroffen. Jenes gutmütige aber eigenwillige Tier, das vielerorts nach wie vor das Symbol für Perl ist, zierte bereits 1991 den Buchdeckel der ersten Auflage von "Programming Perl". Larry Wall änderte sogar die Versionsnummer auf 4.0, um deutlich zu machen, welchen Stand der Sprache das Buch beschreibt. Die Karriere des Verlages O'Reilly nahm damals mit den Tierbüchern wie dem Kamel richtig Fahrt auf und Tim O'Reilly hat quasi die Entwicklung der neuen Generation (Perl 5) gesponsort, indem er Larry anstellte.

Mit dieser Geschichte und dem Erfinder als Koautor ist es nicht zu hoch gegriffen hier von der Perl-Bibel zu sprechen, was auf deutsch ja auch nur "das Buch" bedeutet. Doch bevor der Foliant aufgeschlagen wird - noch zwei andere Hinweise:

Passend zum Schwerpunkt dieser Ausgabe soll das erste in Eigenregie auf Amazon erschienene deutsche eBook über Perl kurz betrachtet werden, sowie ein schmales Buch über sauberes Progammieren. Einzige Meldung vom Ticker ist für dieses Mal: Die sechste Auflage des englischen *Learning Perl* von brian d foy erschien im Juni, die fünfte wurde rezensiert in Ausgabe 3/2011.

The Art of Readable Code

Dieses Buch ließ ebenfalls O'Reilly drucken und vervollständigt was in der $foo 2/2011 über das Standardwerk *Clean Code* und in der nächsten Ausgabe über *Perl Best Practices* geschrieben wurde. Es versprüht nicht Damians souveränen Humor und dreht sich nicht um Perl, sondern um `C++`, `Python`, `Java` und `Javascript`, ja ist nicht einmal auf deutsch geschrieben (und wird es auch nicht so schnell). Aber es behandelt selbstsicher, locker und auf den Punkt was Anfänger gleich zu Beginn in punkto Sauberkeit und Umsicht lernen sollten. Und es ist so aufbereitet, dass Anfänger es wirklich schnell verstehen können.

Der Text ist sehr knapp gehalten. Kein Absatz ist länger als drei Zeilen und Comics, Diagramme, Listen und Tabellen lockern das Erscheinungsbild auf beinah jeder Seite auf. Der Inhalt wird jedoch von den zahlreichen und gut gewählten Beispielen getragen. Deren Komplexität wächst kaum über die Anordnung der Zeilen in einer Routine. Es ist, wie bereits erwähnt, ein reines Anfängerbuch, aber dafür ein gutes.

Zukunft.pl

Das deutsche, elektronische Buch wendet sich ebenfalls an Anfänger und begleitet die Schritte während der allerersten kleinen Skripte, was der Untertitel auch ankündigt: "Ein verständlicher und unterhaltsamer Einstieg in die prozedurale Programmierung mit Perl." Skalare, Arrays und Hashes werden vorgestellt, die Pragmas `strict` und `warnings`, sowie zwei Handvoll Perl-Befehle. Dazu finden einige wenige Funktionen aus `GetOpt::Long`, `Time::Local`, `Date::Calc` und `POSIX` ihren Einsatz.

Obwohl der Titel "Zukunft" enthält, wird lediglich Perl 5.8 (2002) gelehrt. Gilbert Steger (*http://herr-steger.de/*), der sich als "Softwarearchitekt in der Bankenbranche" vorstellt, wollte wahrscheinlich in seiner Begeisterung für Perl klarstellen, dass die Programmiersprache eine glänzende Zukunft hat. Mit viel Fleiß und Enthusiasmus erstellte er viele Screenshots, Diagramme und Mindmaps, welche die Gedankengänge in kleinen Schritten gut illustrieren. Dabei geht er auf viele Probleme ein, die man mit mehr Erfahrung nicht mehr sieht, wie etwa die Installation von Perl, Planung von Programmen oder was überhaupt Schleifen sind. Zusammen mit der unverblümten Wortwahl in kurzen Sätzen kommt das vielen Anfängern entgegen. Doch es wäre noch ein strengeres Lektorat notwendig gewesen um von mir fünf Sterne zu bekommen.

Unlesbar kleine Schrift in zwei, drei Screenshots und `&quot;` statt Leerzeichen in wenigen Kapitelüberschriften schmälern etwas die Freude am Lesen, genau wie der fehlende Index und das eigenwillige Syntaxhighlighting, das nur drei Farben kennt und die Sigills anders markiert als die Variablennamen. Die Beispiele sind gut, doch die Beschränkung auf Windows, ActivePerl sowie das augenscheinlich eingestellte Eclipse-Plugin EPIC gefällt sicher nicht jedem. Wen dies und einige inhaltliche Ungenauigkeiten jedoch nicht stört, findet hier einen preiswerten Schnupperkurs dessen Stärke es ist, auf die Mentalität vieler Anfänger einzugehen und sie über die ersten Unsicherheiten zu führen.

Programming Perl

Doch nun zum Hauptakt, der zweisprachig aufgeführt wird. Denn einerseits ist das Fohlen aus Amerika so jung, dass es im Mutterleib noch eine Ahnung von 5.16 bekam. Sein elfjähriger, deutscher Cousin spricht dagegen nur 5.6. (Es ist die zweite deutsche Ausgabe, die der dritten englischen entspricht.) Da O'Reilly nicht plant das zu ändern, werden beide Auflagen vergleichend in Augenschein genommen.

Das kann unmöglich umfassend geschehen, da allein der 5,1 cm dicke Buchrücken Erklärung genug ist für die wiederholten Verzögerungen und die Zurückhaltung in Sachen Übersetzung. Ein Blick auf *http://yfrog.com/oc5tncdpj* liefert einen schönen optischen Vergleich über das Wachstum von Ausgabe zu Ausgabe. Ansonst fällt äußerlich noch der neue Untertitel auf: "Unmatched power for text processing and scripting".

Die Struktur ist im Wesentlichen gleich geblieben: Übersicht (50 Seiten), Details (460 S.), Technische Details (130 S.), Perlkultur (115 S.) und Referenz (280 S.). Darauf folgt ein 37-seitiges, lesenswertes Glossar und ein doppelter Index. Der erste referenziert alle verwendeten Module, der zweite sämtliche Begriffe. Da die Seiten des Index schwarz markiert sind, lässt er sich bereits gezielt öffnen.

Im Referenzteil wurden lediglich die schnell alternden Informationen über Fehlermeldungen und Kernmodule entfernt, insgesamt knapp 120 Seiten. Dass der Umfang dennoch um 80 Seiten zunahm, liegt schlicht daran, dass sich viel in der Perl-Welt tat. Damit sind keine Module gemeint. Der Elch wird nur zwei Seiten lang beschnuppert. Die Geschichte des Interpreters (oder Compilers) ist einfach zwölf Jahre länger geworden, weswegen circa ein Drittel umgeschrieben werden musste. In diesem Buch findet sich fast jede Einzelheit dazu, sogar dass der deutsche Perl-Workshop die erste Konferenz nur zu Perl überhaupt war. Selbst der einleitende Abschnitt, der darauf ausgerichtet ist, den Leser schnell auf seine ersten Skripte vorzubereiten, vermittelt ihm dabei ein tieferes Verständnis für die Philosophie und Aufbau von Syn-

tax und Semantik. Es wird auf Vollständigkeit geachtet wie in kaum einem zweiten Perlschmöker. Zum Beispiel wird beim obligatorischen Verweis auf `strict` und `warnings` auch `diagnostics` und `sigtrap` genannt.

Der zweite Teil widmet sich nicht unerwartet allen Feinheiten der Sprache. Alles was sich außerhalb des heimischen Prozesses abspielt wie Dateien, Signale, Prozesse, XS, Debugger und CPAN sucht man besser im dritten Teil. Der vierte hat wesentlich mehr im Visier als nur die Konferenzen und Wettbewerbe für das poetischste oder unleserlichste Programm. Zur Perlkultur zählen hier auch die Sicherheit und Portabilität von Programmen, Fallen für Anfänger und Umsteiger von anderen Sprachen sowie Hinweise wie in welche Richtung optimiert werden kann. Das Kapitel enthält überdies Empfehlungen für idiomatisches Perl, Larry Walls Vorstellungen zum Thema *Best Practices*. Der fünfte Teil ist ein alphabetisch sortiertes Nachschlagewerk zu allen Befehlen, Variablen und anderen benannten Konstrukten. Es ist etwas kompakter als die `perldoc`, dafür im Durchschnitt aussagekräftiger.

Auch wenn im ganzen Buch der Text stark überwiegt, so gibt es dennoch keine Bleiwüsten. Es sind genügend Beispiele enthalten mit gelegentlichen Tabellen und Graphiken. Die Sprache ist sehr gediegen und macht nirgends Umwege. Behutsam, klar und manchmal witzig erzählt sie technische Details wie eine Geschichte. Dabei kommt der Humor manchmal als schnoddriges Wortspiel daher, wird aber an anderer Stelle so still und selbstverständlich serviert, dass erst beim zweiten oder dritten Lesen offenbar wird, wie dreist dies geschieht. Ohne jegliche Auflösung hinterher, beschreibt das Glossar zum Beispiel BSD als psychoaktive Substanz. Schon allein, weil dies nur in Teilen übersetzbar ist, sollte man die englische Fassung deutlich vorziehen.

Dass ein zwölf Jahre altes Computerbuch nicht immer hilfreich ist, versteht sich sowieso. Doch Perl hat die für seine Entwickler unbequeme Tradition, nur sehr selten etwas zu entsorgen und die Kompatibilität unter allen Umständen möglichst aufrecht zu halten. Deshalb ist es insgesamt begrüßenswert, dass O'Reilly die deutsche Ausgabe nicht vom Markt nimmt. Fast alles was dort steht, gilt heute auch noch, selbst wenn es manchmal nicht mehr der empfohlene Ansatz ist. Die vierte Auflage wirkt zudem runder, weil die Schreiber Erfahrung gewannen und inhaltliche Lücken schlossen, die vorher unbemerkt blieben.

Der Schreibstil macht für ein Computerbuch erstaunlich viel Freude. Man wird verleitet es nur zum Vergnügen zu lesen und lernt Dank der inhaltlichen Tiefe ständig dazu. Allerdings ist die Wortwahl sehr anspruchsvoll. Nicht-Muttersprachlern könnte der Aufwand zu hoch sein, öfters das Online-Wörterbuch besuchen zu müssen. Ihnen bleibt dann wirklich nur die einfacher formulierte Übersetzung.

Dies ist eindeutig ein Wälzer für jene ruhigen Winterabende in denen man endlich einmal verstehen möchte wie man Unicodestrings akzenttolerant matcht, oder was sich Larry bei diesem oder jenem genau gedacht hat. (Auch wenn der Hauptautor nach wie vor Tom Christiansen lautet.) Dieser Foliant beeindruckt nicht nur nur durch seinen breiten Rücken im Regal, sondern kann seinem Besitzer das gute Gefühl geben, jede Frage, die er jemals zu Perl 5 haben wird, höchstwahrscheinlich beantwortet zu bekommen.

Ausblick

Für die nächste Bücherecke sind folgende Rezensionen geplant:

- *Perl One-Liners Explained* von Peteris Krumins
- *Beginning Perl* von Curtis “Ovid” Poe
- *IT-Projektmanagement* von Matthias Geirhos.

Christian Rost

Rezension - High Performance MySQL, 3rd Edition

High Performance MySQL
3. Ausgabe, März 2012
Baron Schwartz, Peter Zaitsev, Vadim Tkachenko
O'Reilly Media
828 Seiten
ISBN 978-1449314286

Das Buch 'High Performance MySQL' wurde, wie auch der Vorgänger, von den Autoren Baron Schwartz, Peter Zaitsev und Vadim Tkachenko geschrieben und von O'Reilly herausgegeben. Vielleicht nach dem Motto: „Zu viele Köche verderben den Brei“ stehen Zawodny, Lentz und Balling aus der zweiten Ausgabe jedenfalls nicht mehr explizit in der Autorenliste. Die Autoren sind definitiv Koryphäen auf ihrem Gebiet und verstehen sich gleichzeitig darauf, ihr Wissen auch dem Anfänger näher zu bringen. Die dritte Ausgabe ist wesentlich mehr als nur ein Update in Form von neuen Kapiteln oder eine Aktualisierung an die neueren MySQL-Versionen. Viele Bereiche wurden komplett neu verfasst oder umgeschrieben.

Für wen ist das Buch also geeignet? Besser wäre die Frage: Für wen ist es das nicht? Nicht geeignet ist das Buch für absolute Anfänger in Fragen DBMS. Begriffe wie ACID, Indexing und Clustering sollten nicht direkt in eine Googlesuche münden. Alle anderen, die der DBMS-Grundbegriffe mächtig sind, erste MySQL-Datenbanken am Laufen haben und sie administrieren dürfen, werden mit dem Verständnis definitiv keine Probleme haben. Die Autoren holen den Leser an einem niedrigen Level, allerdings nicht ohne Grundniveau, ab. Dafür sorgt unter anderem der Aufbau der Kapitel, die sich von der MySQL-Architektur über die spezifischen Datentypen und performante Indizierung direkt in Richtung Performanceoptimierung wenden.

Vor allem die (eigentlich) logischen Kleinigkeiten bringen einen weit nach vorne. So wird zum Thema Indexing auf Seite 152 darauf hingewiesen, dass aus einem B-Tree-Index sämtliche Spalten rechts von einer Bereichsbedingung nicht mehr vom Query Optimizer profitieren. Wer also ein ... *WHERE fname='John' AND lname LIKE 'Do%' AND bdate='1970-02-02'* ausführt und einen Index über alle drei Spalten hat, muss deshalb feststellen, dass das Geburtsdatum nicht mehr aus dem Index gesucht wird. Das macht also nicht nur die Auswahl der Spalten wichtig, sondern auch die richtige Reihenfolge der Spalten im Index.

Besonders herausgearbeitet wird, wie die Performance des Systems bewertet werden kann und wie man zu aussagekräftigen Messwerten kommt. Eindrücklich wird darauf hingewiesen, dass das System selbstverständlich nicht nur mit dem DBMS, sondern auch auf Application Level betrachtet werden muss. Was auf den ersten Blick als logisch erscheint, ist es hier im Besonderen. So betrachtet das Buch nicht nur die Performance auf DBMS-Ebene, sondern alles, vom Dateisystem, über die Netzwerkanbindung, den zugehörigen Application Server und noch viel mehr.

Sehr gut an diesem Buch ist, dass nicht nur auf die Performance zur Laufzeit eingegangen wird. Die Themen Replikation, Hochverfügbarkeit, Skalierung und vor allem Backup werden in umfangreichen Kapiteln behandelt und stellen eine wertvolle Ergänzung dar.

Angenehm fällt der Schreibstil auf, der sich direkt an Autodidakten zu richten scheint. Noch viel mehr als um das *wie* geht es um das genaue Verständnis, warum etwas auf eben diese Art und Weise implementiert wurde und nicht anders. Was mich besonders beeindruckt, ist der Umstand, dass das Buch dabei so interessant und in einem gut verständlichen

Englisch geschrieben ist, dass ich es direkt als Urlaubslektüre empfehlen würde. Wäre da nicht das Gewicht der fast 800 Seiten. Für Flugreisende ist der Schinken damit eher ungeeignet. Wer jedoch mit der Zeit geht, darf sich für diese Zwecke bei O'Reilly das Ganze als leichtes eBook einpacken.

Fazit

Es fällt mir schwer ein Fazit zu schreiben, welches fast ohne negative Kritik auskommen muss. Irgendetwas stört doch immer. Gut, praktische Beispiele zum Thema Storage Engines fehlen hier im gewohnten Umfang des Buches, mit Ausnahme von *MyISAM* und *InnoDB*. Davon abgesehen hat es mich genau an meinem Wissensstand abgeholt und dorthin gebracht, wo ich hin wollte. Der Schreibstil ist wirklich hervorragend und die Autoren täten gut daran, nicht nur Beratung in Sachen MySQL, sondern auch im Bereich der Fachbuchautoren anzubieten. Das Buch liest sich fast wie ein Roman. Man legt es nicht mehr aus der Hand, denn die einzelnen Kapitel erzeugen steile Lern- und Spannungskurven. Ich wünschte, die Schulbücher in meiner Kindheit wären in demselben Stil verfasst worden.

Renée Bäcker

Ein Writeboard-Ersatz mit Mojolicious

Ich habe immer sehr gerne writeboard.com benutzt, um Artikel oder andere Texte zusammen mit anderen zu schreiben. Es ging nicht darum, gleichzeitig in dem Text zu arbeiten, aber "abwechselnd". Das Gegenüber war meistens auf einem anderen Erdteil unterwegs, so dass man nicht direkt zusammenarbeiten konnte. Bei Writeboard konnte man sich also so ein Notizboard einrichten und an dem Text arbeiten. Die Änderungen konnte man auch mit einfachen Mitteln verfolgen.

Leider wurde der Service geschlossen bzw. ist nur noch Bestandteil eines kostenpflichtigen Dienstes. Also habe ich mich entschlossen, schnell etwas selbst zu programmieren und so ist notepad.perl-services.de entstanden. Dort kann jeder Texte ablegen und bearbeiten. Der grundlegende Aufbau der Anwendung ist in Abbildung 1 ersichtlich.

Für die Versionskontrolle wird *git* eingesetzt, da hier schon alles vorhanden ist. Außerdem bietet es viele nützliche Befehle wie einen farbigen wortbasierten Diff, doch dazu später mehr. Da keine große Datenbank benötigt wird, kommt hier *SQLite* zum Einsatz. Dank `DBI` und `DBD::SQLite` ist der Zugriff auf die Datenbank genauso wie bei anderen Systemen. Und falls die Datenbank irgendwann doch zu groß werden sollte, kann man leicht auf ein anderes System umstellen.

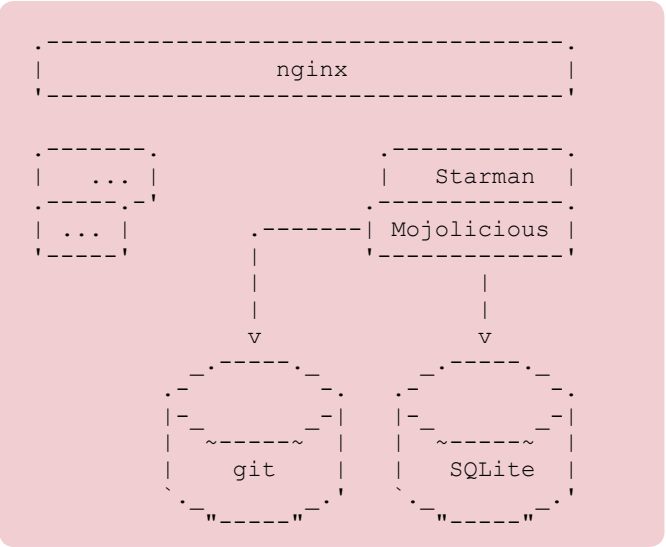


Abb. 1: Der grundlegende Aufbau von Notepad

Für die Weboberfläche wird `Mojolicious` eingesetzt und als Template-Engine wird `Template::Toolkit` genommen.

Die Anwendung ist noch am Anfang der Entwicklung und auf das Layout wurde erstmal kein Wert gelegt. Da der `Dist::Zilla`-Artikel geschrieben werden musste, ging es erstmal darum, funktionsfähigen Code zu bekommen.

Nach einer Einführung in die verwendeten Module gibt es noch eine genauere Einführung in die Anwendung.

Mojolicious

`Mojolicious` ist ein schlankes Webframework für Perl, das außer Perl >= 5.10.1 keine weiteren Abhängigkeiten hat. Nur wer zusätzliche Features oder Plugins benutzen möchte, kommt um die Installation weiterer Module nicht herum. Durch die wenigen Abhängigkeiten ist es auch kein Problem, Mojolicious-Anwendungen auf einem Shared-Webhosting-Account laufen zu lassen (wenn die Perl-Abhängigkeit erfüllt ist).

Mojolicious erlaubt zwei Arten, wie man Webanwendungen erstellt: Mit `Mojolicious::Lite` ist die Anwendung in einer einzigen Datei und mit `Mojolicious` kann man leicht auch größere Anwendungen umsetzen. Da hier nur wenige Aktionen möglich sind, ist keine ausgewachsene Mojolicious-Anwendung nötig.

In den nächsten Abschnitten wird eine kleine Einführung in Mojolicious gegeben, die aber nicht alles abdecken kann. In den kommenden Ausgaben werden immer wieder einzelne Artikel über Aspekte von Mojolicious erscheinen.

get/post/any

Sobald `Mojolicious::Lite` eingebunden wurde, stehen die Methoden `get`, `post` und `any` zur Verfügung. Damit kann man bestimmen, für welche HTTP-Methode eine URL aufrufbar sein soll.

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

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

    my $id = create_notepad( $self );
    return $self->redirect_to(
        '/' . $id . '/edit'
    );
};

any '/about' => {
    shift->render( 'about' );
};
```

Setzt man einen `GET`-Request auf `/` ab, so landet man in der ersten Subroutine, bei einem `POST`-Request in der zweiten. In die letzte Subroutine kommt man entweder mit einem `GET` *oder* einem `POST`-Request.

Der erste Parameter ist die Route (der Teil nach der Domain) und der zweite Parameter eine anonyme Subroutine. Dieser Subroutine wird nur ein `Mojolicious::Controller`-Objekt übergeben. Über dieses Objekt kann man alle Informationen zu einem Request bekommen oder für eine Antwort setzen.

Routen

Eben wurden schon die Routen angesprochen. In den Beispielen oben sind es einfach feste Routen. Aber bei Webanwendungen gibt es selten nur feste Routen. Mindestens ein dynamischer Teil ist meistens drin. Diese dynamischen Teile kann man auch verwenden.

Angenommen die möglichen Routen sollen so aussehen:

```
/12345/edit
/35681/edit
/03813/edit
```

Dann bezeichnet die Zahl die ID des Dokuments, das bearbeitet werden soll. Solche variablen Teile kann man über Platzhalter realisieren. Bei Mojolicious gibt es drei verschiedene Arten von Platzhaltern:

- Generische Platzhalter (siehe Listing 1): Generische Platzhalter sind die einfachste Form von Platzhaltern. Sie matchen alles außer `.` und `/`. Die Platzhalter sind an dem führenden Doppelpunkt zu erkennen.

- Relaxed Platzhalter (siehe Listing 2): Relaxed Platzhalter matchen zusätzlich zu den Zeichen der generischen Platzhalter auch noch den Punkt. Relaxed Platzhalter sind an der führenden Raute zu erkennen.

- Wildcard Platzhalter (siehe Listing 3): Und die folgenden Platzhalter matchen wirklich alles, auch `.` und `/`.

```
/hello              -> /:name/hello -> undef
/sebastian/23/hello -> /:name/hello -> undef
/sebastian.23/hello -> /:name/hello -> undef
/sebastian/hello    -> /:name/hello -> {name => 'sebastian'}
/sebastian23/hello  -> /:name/hello -> {name => 'sebastian23'}
/sebastian 23/hello -> /:name/hello -> {name => 'sebastian 23'}
```

Listing 1

```
/hello              -> /#name/hello -> undef
/sebastian/23/hello -> /#name/hello -> undef
/sebastian.23/hello -> /#name/hello -> {name => 'sebastian.23'}
/sebastian/hello    -> /#name/hello -> {name => 'sebastian'}
/sebastian23/hello  -> /#name/hello -> {name => 'sebastian23'}
/sebastian 23/hello -> /#name/hello -> {name => 'sebastian 23'}
```

Listing 2

```
/hello              -> /*name/hello -> undef
/sebastian/23/hello -> /*name/hello -> {name => 'sebastian/23'}
/sebastian.23/hello -> /*name/hello -> {name => 'sebastian.23'}
/sebastian/hello    -> /*name/hello -> {name => 'sebastian'}
/sebastian23/hello  -> /*name/hello -> {name => 'sebastian23'}
/sebastian 23/hello -> /*name/hello -> {name => 'sebastian 23'}
```

Listing 3

Die Werte der Platzhalter kann man über die `param`-Methode des Controller-Objekts bekommen:

```
get '/:id' => sub {
        my $app = shift;
        my $id  = $app->param('id');

        $app->render( text =>  "ID: $id" );
};
```

Grundsätzlich es es so, dass man Formularfelder bzw. GET-Parameter auch über diese Methode geholt werden können. Man muss nur aufpassen, wenn es Platzhalter und Formularfelder mit den gleichen Namen gibt:

```
get '/:id' => sub {
    my $app = shift;

    my $id  = $app->param('id');
    my $id2 = $app->req->param( 'id' );

    my $t   = $app->param('t');
    my $t2  = $app->req->param('t');

    print "$id // $id2 .. $t // $t2";
};

# http://localhost/1234?id=hallo;t=5
# -> 1234 // hallo .. 5 // 5
```

under

Wer sich den Code von Notepad anschaut (auf GitHub verfügbar), wird entdecken, dass es nach den beiden oben gezeigten Routen noch weitere gibt. Diese stehen aber unter dem Aufruf von `under`. Eine Subroutine unterhalb von `under` wird nur dann ausgeführt, wenn die Subroutine bei `under` *wahr* zurückliefert.

```
under sub {
    my $self = shift;

    return 1 if is_authenticated( $self );

    my $id = $self->url_for;
    $id =~ s/[^a-zA-Z0-9]//g;

    $self->stash( id => $id );
    $self->render( 'login' );
    return;
};

get '/geheim' => sub {};
```

Inline-Templates

Mojolicious hat eine eigene Template-Engine. Eine weitere Nettigkeit von Mojolicious ist es, dass man die Templates direkt in den `__DATA__`-Bereich schreiben kann.

```
get '/bar' => sub {
  my $self = shift;
  $self->stash(one => 23);
  $self->render('baz', two => 24);
};

app->start;
__DATA__

@@ baz.html.ep
The magic numbers are <%= $one %>
   and <%= $two %>.
```

Server während der Entwicklung

Mojolicious liefert auch einen Server mit, den man gut während der Entwicklung nutzen kann. Dieser Server heißt *morbo* und eignet sich besonders für die Entwicklungszeit, weil er bei jeder Änderung im Projekt neu startet. Das ist aber auch genau der Grund, warum er im Live-Betrieb besser nicht genutzt werden sollte.

Starman hinter nginx

Im Live-Betrieb läuft ein Starman-Server hinter einem *nginx*-Reverse-Proxy. Dieses Setup wurde gewählt, weil auf dem Server verschiedene Anwendungen mit unterschiedlichen Technologien laufen. Mit dem *nginx*-Reverse-Proxy sind alle Anwendungen gut getrennt zu betreuen.

Die Anwendung wird mit

```
nohup starman --listen :5001 app.psgi &
```

gestartet.

Die *nginx*-Konfiguration sieht wie folgt aus:

```
upstream pod_book {
  server 127.0.0.1:5001;
}
server {
  listen 80;
  server_name notepad.perl-services.de;
  location / {
    proxy_read_timeout 300;
    proxy_pass http://pod_book;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For \
        $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-HTTPS 0;
  }
}
```

Dieses Setup hat auch Auswirkungen auf die Mojolicious-Anwendung. Man muss ihr mitteilen, dass sie hinter einem Reverse-Proxy sitzt, damit man an die IP-Adresse des Besuchers kommt. Sonst bekommt man nur die IP des nginx.

Sitzt die Mojolicious-Anwendung hinter einem Reverse-Proxy, muss die Umgebungsvariable `MOJO_REVERSE_PROXY` auf "1" gesetzt werden.

Git

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

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

Einen Vergleich zwischen *git* und `Mercurial` wird Herbert Breunung in der nächsten Ausgabe vorstellen.

Bei dieser Webanwendung wird ebenfalls *git* eingesetzt und für die Interaktion zwischen dem VCS und der Anwendung kommt `Git::Repository` zum Einsatz.`Git::Repository` erlaubt es, komplett neue Repositories zu initialisieren oder mit einem bestehenden Repository zu arbeiten:

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

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

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

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

Sobald ein Notepad angelegt wird, wird ein neues Dokument im Repository hinzugefügt. Um eindeutige Namen zu bekommen, wird für jedes Dokument eine eindeutige ID mit Hilfe von `Data::UUID` erzeugt:

```
my $uuid         = $uuid_obj->create_str;
my $notepad_path = _get_notepad_path(
    $uuid,
);

open my $fh, '>', $notepad_path;
close $fh;

chdir $git_dir;

$notepad_path =~ s/\A \Q$git_dir\E \///x;

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

my $commit_output = Git::Repository->run(
    commit =>
    '-m'       => 'created notepad',
    '--author' =>
        ( $owner || 'tester' ) .
        ' <notepad@perl-services.de>',
    $notepad_path
);

my ($id) = $commit_output =~
    m! \[ master \s+ ([a-zA-Z0-9]+) \] !x;

return $id;
```

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

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

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

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

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

```
my $commit_output = Git::Repository->run(
    commit =>
    '-m'   => 'created notepad',
    '-a',
);
```

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

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

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

$cmd->close;
```

Notepad

Um ein solches Notepad benutzen zu können, muss es erst erzeugt werden. Dazu muss man auf der Startseite nur einen Namen für das Notepad und eine E-Mail-Adresse angeben. Wenn es ein nicht-öffentliches Notepad werden soll, kann man auch ein Passwort vergeben.

Nachdem das Notepad erstellt wurde, bekommt man einen Editor zu sehen, in dem das Dokument erstellt wird. Als "Sprache" für die Dokumente wurde Markdown gewählt, weil es dafür Parser auf CPAN gibt und es weitverbreitet ist, auch außerhalb der Perl-Community. Der Editor ist von http://markitup.jaysalvat.com/.

Speichern der Dokumente

Nach dem Bearbeiten des Dokuments wird es gespeichert. Hier kann ein Autorenname angegeben werden. Dieser wird auch direkt an *git* übergeben, sodass der Autor immer nachvollziehbar ist.

```
my $commit_output = Git::Repository->run(
    commit =>
    '-m'       => $comment || '<nothing>',
    '--author' => ($author || 'anonymous')
        . ' <notepad@perl-services.de>',
    $notepad_path,
);
```

Anzeige des Dokuments

Das Dokument wird als Markdown-Datei gespeichert, für die Umsetzung von Markdown nach HTML wird das Modul `Text::Markdown` verwendet. Der Methode `markdown` muss man nur den Text in Markdown-Syntax übergeben und zurück bekommt man das fertige HTML. Leider kann man hier kein eigenes CSS für die einzelnen Elemente übergeben. Das muss dann über ein globales CSS gemacht werden.

```
$self->stash(
    %metadata,
    history => $history,
    body    => markdown( $content ),
);
$self->render( 'show' );
```

Diffs

Über die Historie lassen sich die einzelnen Änderungen anschauen. Da gerade nach dem Korrekturlesen sich die Änderungen in Grenzen halten und selten ganze Sätze ausgetauscht werden, ist ein wortbasiertes `diff` sehr praktisch. Und wenn die Diffs dann noch farbig sind, muss man nach den Änderungen nicht lange suchen. *git* kennt dafür die Option `--color-words` für den Befehl `diff`:

```
my $diff    = $git->run(
    diff =>
    '--color-words',
    $commit . '^',
    $commit,
    $path_to_file,
);
```

Der Befehl erzeugt aber eine farbige Ausgabe für das Terminal. Die Steuerzeichen müssen für die Ausgabe im Browser umgewandelt werden. Dazu eignet sich das Modul `HTML::FromANSI::Tiny` sehr gut. Das Modul liefert dass CSS mit einem entsprechenden Tag über die Methode `style_tag`.

```
my $htmler = HTML::FromANSI::Tiny->new(
    background => 'white',
    foreground => 'black',
);

my $diff_text =
    $htmler->style_tag .
    $htmler->html( $diff );
```

Viel Spaß beim Ausprobieren!

Boris Däppen

CPAN-Dokumentation als E-Book

Dass Dokumentation ein wichtiger Aspekt der Programmierung ist, wird immer wieder betont. Weitgehend bekannt ist aber auch, das diese Fleißarbeit nicht jedes Programmierers Sache ist. Um so erfreulicher ist daher der Fakt, dass es durchaus Open-Source-Projekte mit umfangreicher Dokumentation ans Licht der Welt schaffen. Es ist halt doch so, dass mit einer richtig dosierten und gut gewarteten Dokumentation - auf lange Sicht - Arbeit gespart wird.

Für die meisten Programme und Module in Perl bieten *cpan.org* (bzw. *metacpan.org*) die Anlaufstelle um sich solche Dokumentationen anzuschauen. Hier bekommt man das hauseigene Dokumentationsformat POD als HTML mit schön buntem Markup gut leserlich im Browser angezeigt.

Da einige große Distributionen wie z.B. `Moose` oder `Mojolicious` sehr viel Dokumentation mitliefern, besteht bei manchem das Bedürfnis diese Dokumentation auch mal offline zum Durchlesen zur Verfügung zu haben. So soll es sogar auch Informatiker geben, die gerne mal Abseits von Internet und Monitor etwas Gutes lesen. Da käme doch ein Buch mit der Dokumentation des neuesten Moduls in das man sich einarbeiten muss (oder will) gerade recht. Fern aller Mails und Tweets, als kurzweilige Abwechslung auf Reisen, schnell Griffbereit in der Besprechung oder gar zur Bettlektüre.

perlybook.org

Zu diesem Zweck wurde die Webseite *perlybook.org* [1] ins Leben gerufen. Der Name spielt auf "Perl-*E*-Book" an, bei der Aussprache ist also Kreativität erlaubt. Die Webseite bietet ein Eingabefeld für den Modulnamen der gewünschten Dokumentation und zwei Buttons um das Format des E-Books zu bestimmen. Zur Verfügung stehen im Moment die Formate EPUB und MOBI (siehe hierzu auch den Beitrag in dieser Ausgabe). Standardmäßig wird nicht nur das eingegebene Modul als E-Book gerendert, sondern das ganze Release. Sucht man z.B. das E-Book zu `Moose::Role` so bekommt man ein E-Book mit der kompletten aktuellen Dokumentation von Moose geliefert, wobei die einzelnen Module dann im Inhaltsverzeichnis gelistet sind. `Moose::Test` wäre dann z.B. auch im Buch und über das Inhaltsverzeichnis zu finden.

Wenn dies nicht gewünscht ist, kann die Checkbox "fetch complete release" abgehakt werden, dann kommt lediglich das einzelne Modul als Buch. Dies ist z.B. sinnvoll wenn man sich das Perl Unicode Tutorial (perlunitut und perlunifaq) als Buch gönnen will. Da diese Dokumentationen im Release "perl" auf CPAN eingestellt sind, würde ein riesiges E-Book mit viel unerwünschtem Inhalt zurückgegeben. Solchen, wegen der Komplexität von CPAN, schlecht vorhersehbaren Problemen kann der Nutzer somit selbst begegnen und die entsprechende Option bestimmen.

Die Technologie dahinter

Der Code von perlybook.org ist auf github [2] einsehbar. Die Applikation ist komplett in Perl geschrieben.

Mojolicious

Die Web-Applikation wurde mit Hilfe von `Mojolicious` umgesetzt. Mojolicious ist ein schlankes Webframework, mit dem solche Webanwendungen wie perlybook.org schnell und einfach Realität werden können. Eine kleine Einführung in Mojolicious ist im Notepad-Artikel in dieser Ausgabe zu finden. Ein ausführlicheres Tutorial wird in den nächsten Ausgaben folgen.

Der Vorteil von Mojolicious liegt darin, dass es keine Abhängigkeiten hat, nur ein Perl > 5.10 muss vorhanden sein. Das ist zwar bei vielen Shared Hostern ein Problem, aber mit ein wenig Recherche oder einem eigenen Virtuellen Server stellt es keine Hürde mehr dar.

Mit `Mojolicious::Lite` kann man kleinere Webanwendungen umsetzen und das wäre hierfür auch vollkommen ausreichend gewesen. Bei einer Lite-Applikation steht der Code in einem Skript und es stehen zusätzliche Hilfsfunktionen zur Verfügung, die viel Arbeit abnehmen. Aber um für die Zukunft gewappnet zu sein, wurde darauf verzichtet, eine Lite-App zu entwickeln und es wurde gleich auf eine vollwertige Mojolicious-Anwendung gesetzt.

jQuery

Um den Benutzer bei der Eingabe zu unterstützen wurde eine Autovervollständigung eingebaut. Schaut man sich auf den aktuellen großen Webseiten um, scheint man ja nicht mehr darum herum zu kommen. Die Tipphilfe ist meist auch sehr komfortabel und selbsterklärend. Mit Hilfe von jQuery wurde der Java-Script Anteil umgesetzt. Das einzige was wirklich selbst gemacht werden muss ist die Übergabe der Tippvorschläge an jQuery. Glücklicherweise bietet Metacpan eine Schnittstelle mit entsprechender Funktionalität. Der Suchbegriff kann zusammen mit der gewünschten Antwortgröße an den Server von Metacpan geschickt werden. In Javascript sieht das dann etwa so aus:

```
$.getJSON(
  "http://api.metacpan.org/v0/search/
                              autocomplete?",
    { q: req.term, size: listsize },//req,
```

Im Prinzip wird hier einfach eine URL zusammengebaut die dann z.B. so aussehen kann:

```
.../search/autocomplete?q=ebook&size=10
```

Die Daten der Antwort werden mittels JSON kodiert, leider passen hier die Formate nicht zusammen. jQuery möchte gerne ein einfaches Array mit den Namen, Metacpan liefert eine komplexe verschachtelte Struktur mit allen möglichen Zusatzinformationen. An dieser Stelle muss also mit Hilfe von etwas Javascript-Knowhow eine Anpassung erfolgen.

Als Beispiel für das Aussehen des Datenformats rufen wir folgende URL auf:

```
http://api.metacpan.org/v0/search/
        autocomplete?q=moose&size=2
```

Das gibt folgendes zurück:

```
{
 "timed_out" : false,
 "hits" : {
 "hits" : [
    {
       "_score" : 1.3642136,
       "fields" : {
          "documentation" : "Moose",
          "release" : "Moose-2.0602",
          "author" : "DOY",
          "distribution" : "Moose"
       },
       "_index" : "cpan_v1",
       "_id" : "VcvGry10CANH1XmvPapRL3V19uA",
       "_type" : "file",
       "_version" : 3
    },
    {
       "_score" : 1.3242135,
       "fields" : {
          "documentation" : "IO::Moose",
          "release" : "IO-Moose-0.1004",
          "author" : "DEXTER",
          "distribution" : "IO-Moose"
       },
       "_index" : "cpan_v1",
       "_id" : "DM8N_WaRPxb9XNFvp5yGt4_CGzQ",
       "_type" : "file",
       "_version" : 3
    }
  ],
  "max_score" : 1.3642136,
  "total" : 921
 },
 "_shards" : {
   "failed" : 0,
   "successful" : 5,
   "total" : 5
 },
 "took" : 10
}
```

Da Javascript nicht jedem Perl-Entwickler sofort eingängig sein dürfte, empfehle ich bei Interesse einfach direkt einen Blick in den HTML-Code von perlybook.org. Das Javascript steht dort direkt im head-Teil.

MetaCPAN::API

Bisher ging es aber nur um den Zucker, also darum die Eingabe angenehm zu gestalten. Die Hauptarbeit ist noch nicht getan. Zum Glück kann die fast gänzlich dem Modul `MetaCPAN::API` überlassen werden.

```
my $mcpan  = MetaCPAN::API->new;
```

Das POD aus einem einzelnen Modul zu extrahieren stellt eigentlich keine Schwierigkeit dar:

```
my $pod = $mcpan->pod(
    module         => 'Mojolicious::Lite',
    'content-type' => 'text/x-pod',
);
```

CPAN ist aber nicht nur in Modulen organisiert. Einzelne Module sind unter Umständen zu Distributionen zusammengefasst. Diese Distributionen wiederum werden dann mit entsprechenden Versionsnummern zu einem Release gepackt. Ein Beispiel:

```
Modul:        EBook::MOBI::Pod2Mhtml
Distribution: EBook-MOBI
Release:      EBook-MOBI-0.43
```

Oder etwas bekannter:

```
Modul:        Moose::Role
Distribution: Moose
Release:      Moose-2.0602
```

Aufgepasst: Meistens gibt es in jeder Distribution ein gleichnamiges Modul! So kann Moose für die Distribution stehen, es kann aber auch das Modul gemeint sein. Aus diesem Grund muss man bei Perlybook mittels Checkbox explizit angeben wenn man lediglich ein Modul haben will.

Für ein E-Book macht es Sinn, ein solches Release zusammenzufassen. Auf diese Weise hat man sämtliche Module griffbereit, die ja meist miteinander verzahnt sind. Dafür ist jedoch etwas Vorarbeit nötig.

Zuerst müssen wir wissen, zu welchem Release dieses Modul gehört. Hier gibt es zwei Ansätze. Zum Beispiel kann diese Information mit der Funktionalität der automatischen Vervollständigung gewonnen werden - siehe Listing 1.

Dies ist die Art und Weise wie es in Version 0.1 von Perlybook gemacht wird. Dies kann jedoch zu Problemen führen, da die Autocomlete-Funktion nicht immer wie erwartet antwortet. Die Idee dahinter war, dass der beste Treffer (size=1) immer der richtige Treffer ist, dies ist jedoch nicht der Fall. Das führt dann dazu, dass ein falsches Release ausgewählt wird. Ich habe hierzu ein Issue eröffnet [4], aber es ist wohl nicht so einfach ein perfektes Autocomplete zu implementieren. Deswegen wird Perlybook im nächsten Release eine andere Lösung einsetzen. Folgender Aufruf bietet die selben Informationen:

```
http://api.metacpan.org/v0/module/
   Mojolicious::Lite
```

Mit `MetaCPAN` würde dies so aussehen:

```
my $info = $mcpan->fetch(
                  'module/' . $module_name );
```

Der Code ist noch nicht in Perlybook implementiert. Bis dieser Text erschienen ist könnte das jedoch der Fall sein.

Ist die Distribution bekannt, so kann auf alle Informationen zugegriffen werden. `MetaCPAN::API` baut die Queries zusammen und setzt die Requests ab. Als Rückgabe bekommt man dann jeweils das geparste JSON. So muss man sich selbst nicht mit der Datenstruktur auseinandersetzen.

```
my $result = $mcpan->fetch(
                            'release/Moose' );
```

Der Request der intern abgesetzt wird sieht dann so aus:

```
http://api.metacpan.org/v0/release/Moose
```

Das Ergebnis in JSON (gekürzt) ist in Listing 2 dargestellt.

Dank diesem Request haben wir nun die nötigen Informationen zum Abrufen von einzelnen Files. Hierzu werden nämlich jeweils neben Distribution auch Autor und Pfad benötigt. Auf diese Weise kann nun das *MANIFEST* geholt werden. Da das Manifest immer am selben Ort liegt ist der Pfad bekannt. Autor und Release entnehmen wir dem Resultat des obigen Aufrufs.

```
my $ua = Mojo::UserAgent->new;
my $url = 'http://api.metacpan.org/v0/search/autocomplete?q='
          . $module_name
          . '&size=1';

my $res = $ua->get($url)->res;

my $fields = $res->json->{hits}->{hits}->[0]->{fields};
my $complete_release_name = $fields->{release};
my $distribution          = $fields->{distribution};
```

Listing 1

```
{
   "resources" : {
      "repository" : {
         "web" : "http://git.shadowcat.co.uk/...",
         "url" : "git://git.moose.perl.org/Moose.git",
         "type" : "git"
      },
      "bugtracker" : {
         "web" : "http://rt.cpan.org/...",
           "mailto" : "bug-moose@rt.cpan.org"
        }
   },
   "status" : "latest",
   "date" : "2012-05-07T20:02:37",
   "author" : "DOY",
   "maturity" : "released",
   "dependency" : [
      {
         "relationship" : "requires",
         "phase" : "configure",
         "version" : "0.02",
         "version_numified" : 0.02,
         "module" : "Dist::CheckConflicts"
      }
   ],
   "first" : false,
   "archive" : "Moose-2.0602.tar.gz",
   "version" : "2.0602",
   "name" : "Moose-2.0602",
   "version_numified" : 2.0602,
   "license" : [
      "perl_5"
  ],
   "distribution" : "Moose",
   "tests" : {
      "pass" : 377,
      "fail" : 2,
      "unknown" : 1,
      "na" : 5
   },
   "abstract" : "A postmodern object system for Perl 5"
}
```

Listing 2

```
my $manifest = $mcpan->source(
    author  => $result->{author},
    release => $result->{name},
    path    => 'MANIFEST',
);
```

Im Manifest sind sämtliche Dateien aufgelistet (wenn der Autor alles richtig gemacht hat). Durch Analyse dieser Datei können wir somit den Pfad von allen Dateien die POD enthalten könnten herausfinden. Perlybook holt jede Datei die auf .pod oder .pm endet, ignoriert aber alles was in `t/` oder `example/` liegt.

Das Ganze sollte in einen `eval`-Block gepackt werden, da das Skript stirbt wenn ein File kein POD hat. Dies passiert z.B. dann, wenn eine .pm-Datei keine Dokumentation beinhaltet. Dies sollte unsere Webanwendung nicht gleich ins Stolpern bringen.

```
eval {
    $pod = $mcpan->pod(
        author         => $result->{author},
        release        => $result->{name},
        path           => $file,
        'content-type' => 'text/x-pod',
    );

    1;
};
```

Die API von MetaCPAN kann das Pod in verschiedenen Formaten bereitstellen:

- Text (text/plain)
- HTML (text/html)
- Pod (text/x-pod)
- Markdown (text/x-markdown)

Welches Format geliefert wird, wird mittels des *content-type*-Parameters festgelegt.

Die Möglichkeiten der MetaCPAN-API kann über einen eigenen Explorer [5] erkundet und getestet werden. Das ist beim Debugging sehr hilfreich. Wenn man nicht weiß, wie die Datenstruktur aussieht, kann man hier einen Blick darauf werfen.

Die Code-Beispiele beziehen sich zwar jeweils auf Perlybook, sind jedoch zum großen Teil im Modul `EPublisher` implementiert. Über die Möglichkeit von Plugins kann mit EPublisher POD in (fast) beliebige Formate umgewandelt werden. In dem konkreten Fall übenimmt das Sammeln der Daten das Plugin `EPublisher::Source::Plugin::MetaCPAN`. Für die Ausgabe sind dann die Plugins `EPublisher::Target::Plugin::Epub` und `EPublisher::Target::Plugin::Mobi` zuständig. Hinter diesen Plugins steht das Modul `EBook::EPUB` bzw. `EBook::MOBI` (siehe hierzu Artikel in dieser Ausgabe). In der Ausgabe 20 wurde EPublisher schonmal näher vorgestellt.

CHI

Ist ein E-Book erst mal erstellt wird es auf dem Server gespeichert um bei erneuter Anfrage sofort zur Stelle zu sein. Hierzu wird das Modul `CHI` verwendet. Die Schnittstelle ist denkbar einfach:

```
my $cache = CHI->new(
    driver    => 'File',
    root_dir  => $cache_dir,
    namespace => $cache_namespace,
);
```

Hiermit wird ein Cache auf Dateibasis erzeugt. CHI stellt auch Implementationen für da Management des Caches direkt im Memory oder in verschiedenen Datenbanken bereit. Grundsätzlich können aber auch eigene Treiber geschrieben werden. Der Cache wird unter `root_dir` in einem eigenen Unterverzeichnis `namespace` gespeichert. Existiert der Cache bereits, wird er im weiteren Verlauf verwendet, andernfalls wird er neu angelegt.

Nun können Daten unter einem Schlüssel (*key*) abgelegt werden. Im Falle von Perlybook wird das E-Book unter dem Distributions-Namen und der Versionsnummer abgelegt. Hierdurch werden Doppelläufigkeiten mit einem neuen Release vermieden, denn eine neue Versionsnummer ergibt einen neuen Schlüssel.

```
$cache->set(
    $cache_key,
    $book,
    $expires_in,
);
```

Als letztes Argument kann eine Zeit übergeben werden, nach der der Eintrag nicht mehr im Cache gehalten werden soll. Der Eintrag wird bei Ablauf der Zeit aber nicht automatisch gelöscht. Erst ein Aufruf von `clear()` löscht alle veralteten Daten:

```
$cache->clear();
```

In Perlybook wird dies mit einem Cronjob erledigt, durch den der Cache ab und an von alten Einträgen befreit wird. Um die gespeicherten Daten auszugeben genügt folgender Aufruf:

```
$book = $cache->get($cache_key);
```

Bei Bedarf kann vorher geprüft werden ob unter einem Schlüssel bereits Daten liegen:

```
$cache->is_valid($cache_key);
```

Problemmeldungen

Bei Problemen ist das Projekt dankbar für Rückmeldungen. Diese können in Github [3] eingereicht werden. Für allgemeines Feedback oder falls kein Account auf Github vorhanden ist kann auch gerne eine Mail an perlybook@perl-services.de geschrieben werden.

Zwei bekannte Unschönheiten für die Version 0.1 können vorab genannt werden. Zum einen werden interne Links im POD nicht unterstützt. Die Verweise zeigen unter Umständen ins Internet oder ins Leere. Es muss also jeweils das Inhaltsverzeichnis in Anspruch genommen werden um zu Modulen zu springen. Zum anderen führt der bei E-Books übliche automatische Zeilenumbruch bei Code-Beispielen oft zu unschöner Darstellung. Hier kann der Ersteller der Dokumentation mit kurzen Zeilen abhelfen, der Leser kann, sofern das Gerät dies unterstützt, im Modus für Breitbild lesen, also den Reader quer halten.

Fazit

Mit dem Angebot, CPAN-Dokumentation als E-Book zu rendern, wird eine kleine aber feine Möglichkeit geschaffen in Zukunft noch flexibler und fitter mit POD unterwegs zu sein. *Perlybook.org* bietet dies allen Perl-Programmierern und solchen die es werden wollen an. Und lädt somit auch dazu ein, sich wieder mal an einer Dokumentation zu beteiligen oder die des eigenen Moduls aufzupolieren.

Links

[1] *http://perlybook.org/*
[2] *https://github.com/reneeb/cpan2ebook*
[3] *https://github.com/reneeb/cpan2ebook/issues*
[4] *https://github.com/CPAN-API/cpan-api/issues/203*
[5] *http://explorer.metacpan.org*

TPF News

Fixing Perl 5 Core Bugs

Dave Mitchell hat in den letzten Monaten fast ausschließlich daran gearbeitet, das Ausführen von Code-Blöcken in Regulären Ausdrücken zu verbessern. Dazu gehört, das Feature stabiler und wartbarer zu machen. So hat Mitchell schon existierende Funktionen in Perl verwendet, um auch in Zukunft einfacher von Bugfixes profitieren zu können. Jetzt führen auch Befehle wie `die`, `next`, etc. nicht zu undefiniertem Verhalten oder Segmentation Faults.

Mit `(?{})` kann man beliebigen Perl-Code innerhalb von Regulären Ausdrücken ausführen:

```
# perl -E 'say "aaa" =~ /(aa)(?{say $1})/'
aa
aa
# perl -E 'say "aba" =~ /(aa)(?{say $1})/'
```

Bis Ende Mai hat Dave Mitchell insgesamt über 1.200 Stunden an der Beseitigung von Perl5 Core Bugs gearbeitet.

Improving Perl 5

Das Gebiet, das Nicholas Clark im Perl5 Core bearbeitet, ist sehr breit gefächert. Dazu zählt, dass er den Build-Prozess aufgeräumt hat. Dazu hat Clark den Prozess zur Erstellung der Makefiles analysiert und problematische Befehle entschärft. Darunter war auch ein Befehl, der eine fork-Bombe auslösen konnte.

Auch Pod::Html wurde von Nick verändert. Tests schlugen fehl, weil in den Tests die Liste der erwarteten Ergebnisse hardcodiert war, während die tatsächlichen Pfade mit File::Spec zusammengesetzt wurden. Dadurch kam es zu Unterschieden, weil unter Win32 die Groß-/Kleinschreibung keine Rolle spielt.

Im Perlfoundation Blog sind die sehr ausführlichen Berichte von Nick zu finden. Diese sind sehr interessant, wenn man den Kern von Perl 5 besser verstehen möchte.

Alien::Base

Joel Berger hat hauptsächlich Bugfixes zu Alien::Base hinzugefügt. Er bekommt mehr Fehlermeldungen von Solaris und BSD als erwartet. Die Unterstützung für Darwin und Win32 wird dagegen immer besser.

Beim dynamischen Laden verlässt sich Berger jetzt auf die Fähigkeiten von DynaLoader anstatt selbst mit Umgebungsvariablen zu hantieren. Dadurch wird der Code plattformunabhängiger.

Grant akzeptiert: Improving Cross-compilation of Perl

Perl 5 bietet die Möglichkeit, Perl für andere Systeme zu kompilieren. Das funktioniert aber nicht immer einwandfrei. Jess Robinson will daran arbeiten, Perl für Android zu kompilieren. Dazu wird sie die gesamte Infrastruktur für Cross-Compilation anfassen und ändern.

Grants im zweiten Quartal

Insgesamt standen drei Grant-Anträge zur Abstimmung. Zwei davon wurden abgelehnt. Angenommen wurde der Antrag von Enrique Nell und Joaquin Ferrero, die an der spanischen Übersetzung der Perl Core Dokumentation arbeiten wollen.

Perl 6 Tablets

Nach über einem Jahr Ruhe im Blog der Perl Foundation hat Tom Hukins berichtet, dass Herbert Breunung seine Arbeit an dem Grant fortsetzte, aber noch nicht abgeschlossen hat. Die Dokumentation wurde ins Markdown-Format konvertiert und ist in ihrer HTML-Version nun unter der Adresse *http://tablets.perl6.org/* zu finden. Vor allem Index A und B wurden aktualisiert, erweitert und stärker verlinkt, sodass sie bereits als Kurzreferenz für Perl 6 benutzbar sind. Breunung dokumentiert seine Fortschritte unter *http://blogs.perl.org/users/lichtkind/*.

Devel::Cover

Ein weiterer Grant wird aus dem Spendentopf für Perl 5 bezahlt: Verbesserungen an Devel::Cover. Mittlerweile ist Devel::Cover kaum noch aus dem Fundus von Perl-Programmierern wegzudenken. Vor dem Grant hatte Paul Johnson immer weniger (Frei-)Zeit, um an dem Modul zu arbeiten. Durch den Grant ist er in der Lage, ältere Bugs zu fixen und Verbesserungen hinzuzufügen.

In den vergangenen Monaten hat Johnson einige neue Versionen von Devel::Cover veröffentlicht. Etliche der gefixten Bugs betrafen Devel::Cover im Zusammenspiel mit Moose, was nicht immer reibungslos funktionierte.

Paul Johnson hat cpancover.com gestartet, auf dem man die Code-Abdeckung von etlichen CPAN-Modulen nachschauen kann. Wem ein Modul in der Liste fehlt, kann das gewünschte Modul unter `utils/install_modules` im Github-Repository von Devel::Cover hinzufügen und Johnson einen Pull-Request senden.

White Camel Awards 2012

Auf der YAPC::NA wurden die White Camel Awards 2012 vergeben. Die Gewinner sind:

- Renée Bäcker
- Breno
- James Keenan

Cooking Perl with Chef

Mit Chef kann man automatisiert Software ausrollen. David Golden arbeitet an "Kochrezepten" für die Perl-Welt. Im Rahmen seines Grants sind die folgenden Rezepte schon fertig:

1. Publish a Chef cookbook for Perl interpreter deployment
2. Publish a Chef cookbook for CPAN module deployment
3. Publish a Chef cookbook for Plack application deployment

Diese Kochrezepte sind auf der Webseite der Opscode Community unter *http://community.opscode.com/cookbooks* zu finden. Für die ersten beiden Punkte gibt es das `perlbrew`-Rezept, für den dritten Punkt das `carton`-Rezept.

Weiterhin hat David Golden *Pantry* veröffentlicht. Mit diesem Skript können Nodes und Rollen für Chef Solo erstellt werden.

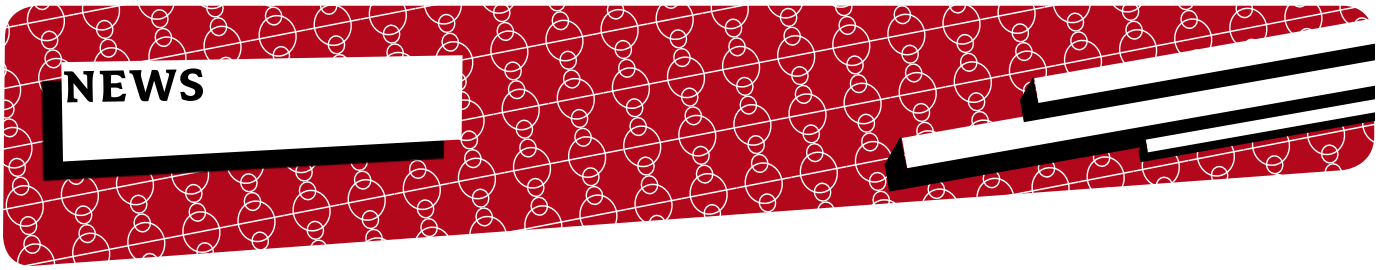
Hague Grant Report: Structured Error Messages

Moritz Lenz hat in den vergangenen Monaten an strukturierten Fehlermeldungen in Rakudo gearbeitet. Zu dem Grant gehörte es, das System für Fehlermeldungen zu entwicklen und in Rakudo zu implementieren.

Unter S32::Exception ist die Spezikation zu finden, in die Arbeit von Lenz eingeflossen ist.
FRAGE: Was ist die Aussage dieses Satzes? Die Arbeit von Lenz ist in die Spezifikation eingeflossen? Oder umgekehrt? Außerdem konnte ich S32::Exception nicht auf CPAN finden.

Darüber hinaus ist dort ein Fehlerkatalog zu finden.

Neben dem Grant hat Moritz Lenz an weiteren Themen gearbeitet, die mit der Fehlerbehandlung zu tun haben.



CPAN News XXIII

Net::Jenkins

In Ausgabe 21 wurde gezeigt, wie man auch für Perl-Projekte das Continuous Integration Tool *Jenkins* verwenden kann. In diesen CPAN-News wird ein Modul vorgestellt, mit dem man Jenkins weitestgehend automatisieren kann.

Informationen über Jenkins

Zuerst werden ein paar Informationen über die Jenkins-Installation geholt. Dabei handelt es sich aber nur um grundlegende Informationen wie den Status oder eine Liste von Jobs:

```
my $jenkins = Net::Jenkins->new;
my $summary = $jenkins->summary;
print Dumper $summary;
```

Einige dieser Informationen kann man auch direkt mit Methoden abrufen, z.B. gibt

```
$jenkins->mode;
```

den aktuellen Status der Jenkins-Installation aus.

Jobs klonen, starten und noch mehr

Der wichtigste Bestandteil von Jenkins sind jedoch die Jobs. Auch darauf kann man mit `Net::Jenkins` zugreifen. In den folgenden Abschnitten werden Jobs erzeugt, geklont, gestartet und gelöscht.

Zuerst wird also ein Job erzeugt. Jenkins wird mit XML-Dateien konfiguriert - und damit auch die Jobs. Am besten schaut man sich so eine XML-Datei mal bei einer Jenkins-Installation an. Im Download-Paket auf der Webseite des Perl-Pakets ist eine komplette Beispieldatei enthalten.

In der Konfigurationsdatei sind Informationen darüber zu finden, welches Versionskontrollsystem verwendet wird und welche Schritte beim Build-Vorgang ausgeführt werden sollen.

Diese XML-Konfiguration muss eingelesen werden und einfach an die Methode `create_job` übergeben werden. Der erste Parameter ist der Name des Jobs.

```
my $xml = do{
    local (@ARGV,$/) = 'config.xml'; <>
};

my $success =
    $jenkins->create_job('foo_23', $xml);

if( $success ) {
    print "created job for foo_23\n";
}
```

Damit ist der Job erstellt. Ob es tatsächlich in Jenkins angekommen ist, kann man testen, indem man alle Jobs aus Jenkins holt.

```
for my $job ( $jenkins->jobs ) {
    print Dumper $job->details;
}
```

Soll die gleiche Konfiguration für weitere Jobs verwendet werden, kann man Jobs klonen bzw. kopieren.

```
$jenkins->copy_job( 'test2' , 'Phifty' );
```

Der erste Parameter ist der Name des Zieljobs und der zweite Parameter der Quelljob. Was an dieser Stelle fehlt, ist das Aktualisieren einzelner Einstellungen des Jobs. Ein Update ist nur mit einer neuen Konfigurationsdatei möglich:

```
$job->update( $xml );
```

Ist der Job vorhanden, kann man Jenkins anweisen den Job auszuführen und so einen Build-Vorgang zu starten.

```
$job->build;
```

Warten bis der Build-Vorgang durchgelaufen ist, kann man mit

```
sleep 1 while $job->in_queue;
```

Alle Details des letzten und der vorangegangenen Build-Vorgänge sind ebenfalls abrufbar.

Wozu dieses Modul?

Die Aufgaben des Moduls kann man auch selbst über HTTP-Requests und die JSON-Antworten erledigen. Das Modul nimmt einem aber einiges an Arbeit ab und man hat nur Perl-Objekte vor sich.

Ein Einsatzgebiet könnte sein, dass man die Funktionalitäten in andere Tools wie ein Intranet etc. integriert ohne dass die Mitarbeiter direkten Zugriff auf die Jenkins-Installation benötigen. Mit diesem Modul wäre es auch möglich, ein `Dist::Zilla`-Plugin zu schreiben (siehe Dist::Zilla-Artikel), das vor einem Release einen entsprechenden Job anlegt und erstmal laufen lässt.

So gibt es viele mögliche Einsatzszenarien für dieses Modul. Leider ist die Dokumentation noch nicht ganz so ausgereift, aber das kann ja noch werden.

Text::SimSearch

`Text::SimSearch` ist ein Modul, mit dem eine einfache Suchmaschine umgesetzt werden kann. Damit Suchmaschinen überhaupt etwas finden können, benötigen sie einen Index, in dem die Zuordnung von Dokument zu Wort bzw. umgekehrt zu finden ist. `Text::SimSearch` erstellt einen invertierten Index, also die Zuordnung von Wort zu Dokument und kann so Suchbegriffe schnell finden.

Einen Index aufbauen

Der Index enthält alle Informationen aus den Dokumenten, also müssen diese erst analysiert und dann indiziert werden.

Texte zerlegen

Wie man einen Text zerlegt um am Ende die wichtigen Worte herauszubekommen, ist gar nicht so einfach. Welche Worte sollen gar nicht beachtet werden? Welches Wort ist wie wichtig? Für diesen Artikel werden aus einem Textkörper einfach alle Stoppwörter (Artikel, Präpositionen, ...) rausgeschmissen. Danach werden die gesamten Worte gezählt und die Häufigkeit der Vorkommen von einzelnen Wörtern betrachtet:

```
my @files     = ('sample.txt');
my $stopwords = getStopWords('de');

my %index;

for my $file ( @files ) {
   my $text = slurp $file;

    my %words_count;
    for my $word ( split /\s/, $text ) {
        $word = lc $word;
        next if $stopwords->{$word};
        $words_count{$word}++;
    }

    my $sum = 0;
    $sum += $_ for values %words_count;

    $words_count{$_} =
        ( $words_count{$_} / $sum )
        for keys %words_count;

    $index{$file} = \%words_count;
}
```

Diese Informationen landen in einem Hash:

```
{
    'sample.txt' => {
        Artikel => 0.05,
        Magazin => 0.1,
        Autor   => 0.02,
    },
}
```

Den Text dem Index hinzufügen

Nachdem der Hash mit den ganzen Informationen zusammengebaut ist, müssen die Informationen noch in den Index eingebaut werden. `Text::SimSearch` ist hier nicht besonders komfortabel, da es nur die Möglichkeit bietet, mit `add_item_from_file` zu arbeiten. Hierzu muss eine Datei vorliegen, die folgendermaßen aufgebaut ist:

```
Label \t Wort \t Gewichtung \t Wort
  \t Gewichtung ...
```

zum Beispiel:

```
sample.txt  Artikel 0.05
    Magazin 0.1  Autor 0.02
```

Die Gewichtung ist wichtig, um am Ende die relevanten Dokumente herauszufinden. Der Hash mit den Informationen für die Suchmaschine wird in eine Datei gespeichert:

```
my $index_dat = 'index.dat';
if ( open my $fh, '>', $index_dat ) {
    for my $label ( keys %index ) {
        my $data = $index{$label};
        print $fh $label, "\t",
            join "\t",
                map{
                    sprintf "%s\t%s",
                        $_, $data->{$_}
                }keys %{$data};
        print $fh "\n";
    }
}
```

und dann mit der Methode aus `Text::SimSearch` wieder eingelesen. Anschließend wird der Index gespeichert.

```
my $indexer = Text::SimSearch->new;
$indexer->add_item_from_file($index_dat);

$indexer->save( 'save.bin' );
```

Sollte später ein neuer Text in den Index aufgenommen werden, müssen erst die alten Informationen ausgelesen und die neuen hinzugefügt werden. Dann kann der Index neu erstellt werden.

Die kleine Suchmaschine

Der Index ist jetzt erstellt und die kleine Suchmaschine kann losgehen:

```
my $indexer = Text::SimSearch->new;
$indexer->load( 'save.bin' );
```

Nachdem der Index eingelesen wurde, kann man den Benutzer nach Suchbegriffen fragen und die Ergebnisse ausspucken:

```
while ( 1 ) {
    print "Wort: ";
    chomp( my $in = <STDIN> );

    my $result = $indexer->search(
         { $in => 1 },
         5,
    );

    print Dumper $result;
}
```

Als ersten Parameter erwartet die Methode `search` eine Hashreferenz mit den Suchbegriffen und deren Gewichtung. Da man bei einer Standardsuche vorher nicht wissen kann, wie das Suchwort gewichtet wird, wird in dem Beispiel immer eine "1" genommen. Der zweite Parameter ist die Anzahl von Treffern, die zurückgeliefert werden soll.

Als Ergebnis bekommt man eine Hashreferenz, die ungefähr so aussieht:

```
  $VAR1 = {
    'return_num' => 2,
    'retrieved_list' => [
      {
        'similarity' => '0.179605',
        'label' => 'sample.txt'
      },
    ],
    'elapsed' => '0.000371'
  };
```

App::AltSQL

Die meisten Datenbanksysteme liefern schon ein Kommandozeilentool mit. Von der Darstellung sind diese aber oft nicht so gut, wie man es sich wünschen würde. Eric Waters hat mit `App::AltSQL` eine Anwendung geschrieben, mit der man (theoretisch) alle Datenbanken ansteuern kann. Zur Zeit ist jedoch nur die Anbindung an MySQL umgesetzt.

Das System arbeitet mit Plugins, so dass man sich die Ausgabe so gestalten kann wie man möchte. In der aktuellen Distribution sind auch Plugins enthalten, mit denen man die Ergebnisse in verschiedene Formate wie JSON, YAML oder XML umwandeln kann.

Nach der Installation der Distribution steht die Anwendung `altsql` zur Verfügung.

```
./altsql -h <host> -u <username> -D <database> -p<password>

altsql> select * from film limit 4;
+---------+-------------------+-------------------------------------+
| film_id | title             | description                         |
+---------+-------------------+-------------------------------------+
|       1 | ACADEMY DINOSAUR  | A Epic Drama of a Feminist          |
|       2 | ACE GOLDFINGER    | A Astounding Epistle of a D         |
|       3 | ADAPTATION HOLES  | A Astounding Reflection of          |
|       4 | AFFAIR PREJUDICE  | A Fanciful Documentary of a         |
+---------+-------------------+-------------------------------------+
4 rows in set (0.00 sec)
```

Data::OpeningHours

Hat mein Lieblingsladen um die Ecke jetzt noch geöffnet? Diese Frage lässt sich mit `Data::OpeningHours` leicht beantworten. Meistens weiß man so etwas zwar sowieso, aber vielleicht lässt sich damit eine nützliche Anwendung erstellen.

```
use DateTime;
use Data::OpeningHours 'is_open';
use Data::OpeningHours::Calendar;

my $cal = Data::OpeningHours::Calendar->new();
$cal->set_week_day(1, [['13:00','18:00']]); # monday
$cal->set_week_day(2, [['09:00','18:00']]);
$cal->set_week_day(3, [['09:00','18:00']]);
$cal->set_week_day(4, [['09:00','18:00']]);
$cal->set_week_day(5, [['09:00','21:00']]);
$cal->set_week_day(6, [['09:00','17:00']]);
$cal->set_week_day(7, []);
$cal->set_special_day('2012-01-01', []);
is_open($cal, DateTime->now());
```

Mojolicious::Plugin::RenderFile

Ein wirklich nützliches Modul für alle, die mit `Mojolicious` arbeiten und Dateien zum Download anbieten, ist `Mojolicious::Plugin::RenderFile`. Es übernimmt das Setzen von Header-Angaben und das Streamen der Dateien.

```
# Mojolicious
$self->plugin('RenderFile');

# Mojolicious::Lite
plugin 'RenderFile';

# In controller
$self->render_file(
        filepath => '/tmp/files/file.pdf');
```

Try::Tiny::SmartCatch

Es gibt einige Module, die die Fehlerbehandlung vereinfachen. Die bekanntesten dürften `Try::Tiny` und `TryCatch` sein. Mit `Try::Tiny::SmartCatch` ist ein Modul von Marcin Sztolcman veröffentlich worden, das eine Mischung aus den beiden darstellt. Es ist nicht so schwergewichtig implementiert wie `TryCatch`, aber es hat mehr Möglichkeiten als `Try::Tiny`. `catch_when` benutzt den Smart-Match-Operator, um Fehlerbehandlungen für einzelne Fehler zu ermöglichen.

```
use Try::Tiny::SmartCatch;

# call some code with expanded error
# handling (throw exceptions as object)
try sub {
    die (Exception1->new ('some error'));
},
catch_when 'Exception1' => sub {
    # handle Exception1 exception
},
catch_when ['Exception2', 'Exception3'] =>
sub {
    # handle Exception2 or Exception3
    # exception
    },
catch_default sub {
    # handle all other exceptions
},
finally sub {
    # and finally run some other code
};
```

Termine

August 2012

02. Treffen Dresden.pm
07. Treffen Stuttgart.pm
Treffen Frankfurt.pm
13. Treffen Ruhr.pm
15. Treffen Darmstadt.pm
20.-22. YAPC::Europe 2012
20. Treffen Erlangen.pm
25.-26. FrOSCon
26.-30. Moving To Moose Hackathon
28. Treffen Bielefeld.pm
29. Treffen Berlin.pm

September 2012

04. Treffen Stuttgart.pm
Treffen Frankfurt.pm
10. Treffen Ruhr.pm
17. Treffen Erlangen.pm
19. Treffen Darmstadt.pm
25. Treffen Bielefeld.pm
26. Treffen Berlin.pm
27.-29. YAPC::Asia

Oktober 2012

04. Treffen Frankfurt.pm
Treffen Stuttgart.pm
08. Treffen Ruhr.pm
17. Treffen Darmstadt.pm
19.-20- YAPC::Brasil
22. Trefen Erlangen.pm
30. Treffen Bielefeld.pm
31. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

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

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perl-foundation.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.

FrOSCon

Free and Open Source Software Conference

Hochschule Bonn-Rhein-Sieg, Grantham Allee 20, 53757 Sankt Augustin

25. + 26. August 2012

Über 60 Aussteller und Projekte
Über 100 Vorträge
LPI-Prüfungen
Kinder-und Jugendtrack
Social Event
Hüpfburg und Bällebad

www.froscon.de

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