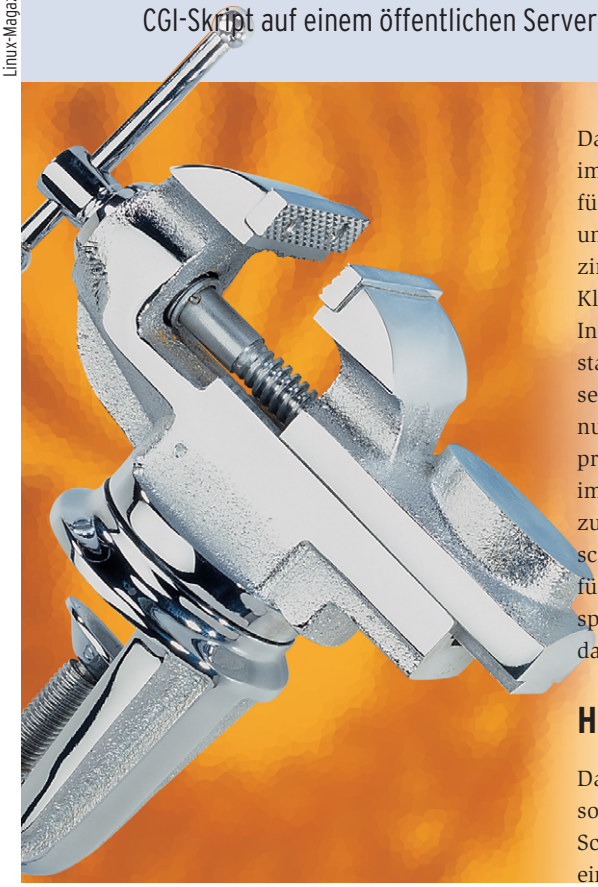
Kurze URLs à la Tinyurl.com selbst gemacht

Gepresste Links

Ellenlange URLs lassen sich nur schwer in Zeitschriftenartikeln drucken oder per E-Mail versenden. Ein CGI-Skript auf einem öffentlichen Server erlaubt drastische Abkürzungen. Michael Schilli



Beim Lesen im neuesten „Cryptogram“, dem monatlichen Rundbrief von Sicherheits-Superheld Bruce Schneier [2], fiel mir auf, dass die sonst eher länglichen URLs zu diversen Literaturverweisen ungewohnt kurz ausfielen: Sie verwiesen alle einfach auf [http://tinyurl.com] und endeten kurz und knapp mit kryptischen Kürzeln.

Alles klar: Sites wie Tinyurl.com oder Makeashorterlink.com bieten einen kostenlosen Service an, der lange URLs speichert und durchnummeriert. Tinyurl führt einen Redirect auf die Ziel-URL aus. So registrierte ich neulich [http://tinyurl.com/kowv] – die Adresse verweist auf den neuesten USA-Rundbrief unter [http://perlmeister.com/rundbrief.archiv/20030801]. Praktisch!

Das Ganze lässt sich auch simpel in Perl implementieren: Ein persistenter Hash führt eindeutige Kürzel als Schlüssel und registrierte URLs als Werte. Statt dezimaler Zahlen werden Gebilde aus Kleinbuchstaben und Zahlen eingesetzt. In einem System zur Basis 36 (26 Buchstaben plus 10 Ziffern) sind zum Beispiel selbst für Zahlen größer als eine Million nur noch vier Zeichen nötig: »4c92« repräsentiert die dezimale Zahl 1000000 im 36er System. Eine vierstellige Zahl zur Basis 36 kann $36^4 = 1679616$ verschiedene Werte abspeichern – das sollte fürs Erste ausreichen. Listing 1 (»u« – sparen wo's geht) zeigt ein CGI-Skript, das »tinyurl.com« nachbildet.

Härten des Skripts

Das ins offene Internet gestellte Skript soll natürlich nicht dem allerersten Schlimmfinger in die Hände fallen, also einige Sicherheitsvorkehrungen:

- Per IP und Tag sind 200 URLs erlaubt – das soll verhindern, dass jemand die Festplatte mit unsinnigen URLs auffüllt. Große Service-Provider wie etwa AOL bedienen viele Kunden gleichzeitig über dieselben IPs. Der eingestellte hohe Wert trägt dem Rechnung.
- Die Maximalgröße für die Datenbankdatei wird auf einen Wert festgelegt (zum Beispiel 20 MByte). Ist diese Größe erreicht, nimmt das Skript keine weiteren URLs mehr an, verzweigt aber weiterhin zu den bereits definierten.
- Eine URL darf bis zu maximal 256 Zeichen messen, längere URLs weist das Skript zurück.
- Das Skript loggt alle Vorkommnisse in eine rotierenden Logdatei mit einstellbarer Maximalgröße.

Die CGI-Funktionalität erledigen Lincoln Steins CGI-Modul und dessen Ableger »CGI::Carp«, der mit dem eingestellten »fatalsToBrowser«-Tag jede Perl-Exception abfängt und zu Debugging-Zwecken im Browser darstellt.

Sieht das Skript keinen »url«-Parameter, zeigt es im Browser ein Formular zur Eingabe einer neuen URL an. Der Submit-Button schickt sie als Parameter »url« wieder zum Skript. Das rechnet eine abgekürzte URL aus und speichert die Zuordnung der Abkürzung zur vollen URL in seiner Minidatenbank ab, falls sie dort nicht schon steht. Die abgekürzten URLs haben die Form

```
http://server.com/cgi/u/xxxx
```

und senden die URL-Abkürzung »*xxxx*« angehängt an den Pfad zum Skript »u«, dessen CGI-Umgebung sie in der Variablen »$ENV{PATH_INFO}« zugänglich macht. Sieht das Skript eine derartige

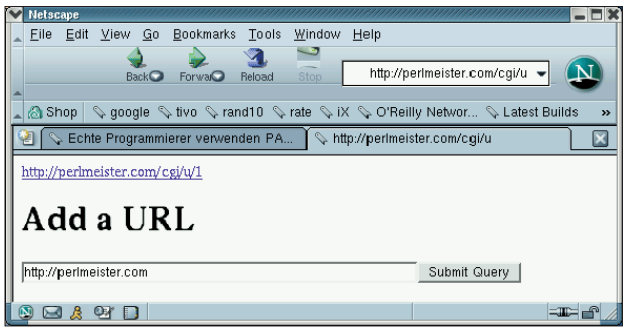


Abbildung 1: Der Kompressor nimmt eine neue URL entgegen, speichert sie in seiner Datenbank und gibt ihre Kurzform aus.

Anfrage, holt es die zugehörige lange URL aus der Datenbank und sendet einen »redirect()«, was den Browser nahtlos zu der entsprechenden Website verzweigen lässt.

Loggen mit Abroll-Komfort

Komfortables Logging mit den Makros »DEBUG()«, »INFO()« und »LOGDIE()« besorgt wieder einmal »Log::Log4perl qw(:easy)« zusammen mit dem »FileRotate«-Appender aus dem »Log::Dispatch«-Fundus. »size = 1000000« legt dabei 1 MByte als maximale Größe fest. »max = 1« bestimmt, dass der »FileRotate«-Appender ein volles Logfile »shrink.log« nach »shrink.log.1« abrollt, wenn »shrink.log« 1 MByte überschreitet und keine weiteren Backups vornimmt, damit immer nur höchstens 2 MByte an Logdateien auf der Platte sind.

Der mit Hilfe von »tie()« an die Datei »/tmp/shrink.dat« gebundene persistente Hash »%URLS« speichert in »u« nicht wie üblich eine einzelne Key-Value-Zuordnung, sondern gleich drei. Daher erhalten die Schlüssel jeweils ein Präfix:

- »by_shrink/«: Abkürzung zur URL
- »by_url/«: URL zur Abkürzung
- »next/«: Nächste zu vergebende Abkürzung ▶

Listing 1: »u«

```
001 #!/usr/bin/perl
002 ##########################################
003 # Mike Schilli, 2003 (m@perlmeister.com)
004 ##########################################
005 use warnings;
006 use strict;
007 use Log::Log4perl qw(:easy);
008 use Cache::FileCache;
009
010 my $DB_FILE     = "/tmp/shrinky.dat";
011 my $DB_MAX_SIZE = 10_000_000;
012 my $MAX_URL_LEN = 256;
013 my $REQS_PER_IP = 200;
014
015 Log::Log4perl->init(\ <<"EOT");
016 log4perl.logger = DEBUG, Rot
017 log4perl.appender.Rot=\\
018   Log::Dispatch::FileRotate
019 log4perl.appender.Rot.filename=\\
020   /tmp/shrink.log
021 log4perl.appender.Rot.layout=\\
022   PatternLayout
023 log4perl.appender.Rot.layout.\\
024 ConversionPattern=%d %m%n
025 log4perl.appender.Rot.mode=append
026 log4perl.appender.Rot.size=1000000
027 log4perl.appender.Rot.max=1
028 EOT
029
030 use CGI qw(:all);
031 use CGI::Carp qw(fatalsToBrowser);
032 use DB_File;
033
034 tie my %URLS, 'DB_File', $DB_FILE,
035     O_RDWR|O_CREAT, 0755 or
036         LOGDIE "tie failed: $!";
037
038     # First time initialization
039 $URLS{"next/"} ||= 1;
040
041 my $redir = "";
042
043 if(exists $ENV{PATH_INFO}) {
044         # Redirect requested
045     my $num = substr($ENV{PATH_INFO}, 1);
046     $redir = $URLS{"by_shrink/$num"} if
047         $num ne "_"
048         and exists $URLS{"by_shrink/$num"};
049 }
050
051 if($redir) {
052     print redirect($redir);
053     goto END;
054 }
055
056 print header();
057
058 if(my $url = param('url')) {
059
060     if(length $url > $MAX_URL_LEN) {
061       print "Sorry, URL too long.\n";
062       goto END;
063     }
064
065     my $surl;
066
067     # Does it already exist?
068     if(exists $URLS{"by_url/$url"}) {
069       DEBUG "$url exists already";
070       $surl = $URLS{"by_url/$url"};
071
072     } else {
073       if(-s $DB_FILE > $DB_MAX_SIZE) {
074         DEBUG "DB File maxed out " .
075               (-s $DB_FILE) . " > $DB_FILE";
076         print "Sorry, no more URLs.\n";
077         goto END;
078       }
079
080       if(rate_limit($ENV{REMOTE_ADDR})) {
081         print "Sorry, too many requests " .
082               "from this IP\n";
083         goto END;
084       }
085
086       # Register new URL
087       my $n = base36($URLS{"next/"}++);
088       INFO "$url: New shortcut: $n";
089       $surl = url() . "/$n";
090         $URLS{"by_shrink/$n"} = $url;
091         $URLS{"by_url/$url"}  = $surl;
092     }
093     print a({href => $surl}, $surl);
094   }
095
096       # Accept user input
097   print h1("Add a URL"),
098         start_form(),
099         textfield(-size    => 60,
100                   -name    => "url",
101                   -default => "http://"),
102         submit(), end_form();
103
104 END:
105
106 untie %URLS;
107
108 ##########################################
109 sub base36 {
110 ##########################################
111     my ($num) = @_;
112
113     use integer;
114
115     my @chars  = ('0'..'9', 'a'..'z');
116     my $result = "";
117
118     for(my $b=@chars; $num; $num/=$b) {
119         $result .= $chars[$num % $b];
120     }
121
122     return scalar reverse $result;
123 }
124
125 ##########################################
126 sub rate_limit {
127 ##########################################
128     my ($ip) = @_;
129
130     $ip = 'NO_IP' unless defined $ip;
131
132     INFO "Request from IP $ip";
133
134     my $cache = Cache::FileCache->new(
135         { default_expires_in  => 3600*24,
136           auto_purge_on_get   => 1,
137         }
138     );
139
140     my $count = $cache->get($ip);
141
142     if(defined $count and
143        $count >= $REQS_PER_IP) {
144         INFO "Rate-limiting IP $ip";
145         return 1;
146     }
147
148     $cache->set($ip, ++$count);
149
150     return 0;
151 }
```

An einigen Stellen soll das Skript einfach aufhören – schnell noch vom persistenten Hash abkoppeln und dann das Programm beenden. Mit »exit()« aussteigen wäre schlechter Stil, da dies in Umgebungen wie »mod_perl« Probleme verursacht. »return()« geht auch nur, wenn »perl« gerade eine Subroutine ausführt. Schließlich kam das gute alte »goto« zum Einsatz, das echte Programmierer im Gegensatz zu so genannten Quiche-Fressern laut [3] ja nicht fürchten. An den paar Abbruchstellen im Skript springt »goto« einfach zum weiter unten definierten Label »END«.

Ein File-Cache mit einstellbarem Verfallsdatum dient dazu, die Anzahl der URLs zu limitieren, die Benutzer pro IP-Adresse und Tag speichern dürfen. Das schlaue Modul »Cache::Cache« bietet eine Schnittstelle, die mit »set()« neue Einträge setzt und sie mit »get()« wieder holt. Die abgeleitete Klasse »Cache::FileCache« implementiert dies als Dateihierarchie auf der Platte.

Geplante Vergesslichkeit

Das Skript speichert pro IP einen Zähler und erhöht ihn mit jeder von dieser Internetadresse angeforderten URL. Ist der Höchststand erreicht, blockiert es weitere Anträge zu neuen URLs, liefert aber weiterhin die Kürzel zu bereits definierten aus. Nach Ablauf eines Tages vergisst »Cache::FileCache« die vorgenommenen Einträge – der Zählvorgang beginnt von vorn. Das simple Verfahren blockiert schlimmstenfalls für einen Tag jene IPs, die stetig, aber innerhalb der Tagesgrenze, neue URLs anfordern.

Die IP-Adresse des Clients stellt die CGI-Umgebung des Webservers in der Variablen »$ENV{REMOTE_ADDR}« bereit. Von der Kommandozeile aus aufgerufen setzt das Skript die Variable »$ENV{REMOTE_ADDR}« jedoch nicht. Darum setzt Zeile 130 anstelle der IP-Adresse einfach den String »NO_IP«.

Die Option »default_expires_in« des »Cache::FileCache«-Konstruktors gibt die Zeitspanne in Sekunden nach dem letzten »set()« an, ab der der Cache sich einfach nicht mehr an den Eintrag erinnern kann. »auto_purge_on_get« gibt weiter an, dass der Cache automatisch bei jedem »get()«-Aufruf bereits verfallene Einträge aufspürt und von der Platte putzt – das stellt sicher, dass der Cache nicht unendlich mit den legitim operierenden IP-Adressen anwächst.

Das 36er Universum

Die ab Zeile 109 definierte Funktion »base36()« wandelt Dezimalzahlen platzsparend in solche zur Basis 36 um. Wie geht das? Eine Zahl im Zehnersystem baut sich wie folgt auf:

```
a*1 + b*10 + c*10*10 + ...
```

Wobei a, b, c die Ziffern der Zahl nach ihrem Stellenwert angeben. 156 ist also:

```
6*1 + 5*10 + 1*10*10
```

Im 36er System hingegen gilt:

```
a*1 + b*36 + b*36*36 + ...
```

Folgender Algorithmus wandelt eine Dezimalzahl d dahin um: Bestimme den Rest der Division d : 36 (also d modulo 36 oder »d % 36«). Dies ergibt das letzte Zeichen der Zahl zur Basis 36. Dann teile die umzuwandelnde Zahl durch 36, nimm den ganzzahligen Anteil des Ergebnisses und fahre fort, um die nächsten Zeichen (von rechts nach links) der Zielzahl zu ermitteln.

In die Funktion »base36()« kommen im Array »@chars« zunächst alle gültigen Zeichen, also die Ziffern von 0 bis 9 und die Kleinbuchstaben von a bis z. Die danach folgende For-Schleife ermittelt zunächst mit

```
my $b = @chars;
```

und wegen des in skalaren Kontext gestellten Arrays »@chars« dessen Länge in »$b« – das ist die Anzahl der gültigen Zeichen, 36 im vorliegenden Fall.

Mit »$num % $b« wird ermittelt, wie viel als Rest übrig bleibt, wenn man die umzuwandelnde Zahl »$num« durch die Zahl der verfügbaren Zeichen im 36er System teilt – also genau die letzte „Ziffer“ der umgewandelten Zahl im Zielsystem. Die Fortsetzung der »for«-Schleife – »$num /= $b« – führt wegen der vorher in »base36()« angeforderten Direktive »use integer« mit »$num« und »$b« eine Division ohne Fließkomma-Anteil durch. Der Ausdruck

```
$result .= $chars[$num % $b];
```

schnappt sich das richtige Zeichen aus dem 36er Zeichensatz und fügt es ans Ende des Strings in »$result« an – die Zahl beziehungsweise Zeichenfolge im Zielsystem wird also verkehrt herum aufgebaut. Das wiederum korrigiert der Ausdruck

```
return scalar reverse $result;
```

indem er den String in »$result« umdreht. Skalarer Kontext ist notwendig, da »reverse« im List-Kontext einfach eine ihr übergebene Liste umdreht und nicht die Zeichen eines Einzelwerts.

Installation

Das Skript benötigt »Log::Log4perl«, »Log::Dispach::FileRotate« und »Cache ::FileCache«, alle sind vom CPAN erhältlich. Die Pfade zur Logdatei (Zeile 20) und zur Datenbankdatei (Zeile 10) sind an die lokalen Verhältnisse anzupassen und das Skript ist ins »cgi-bin«-Verzeichnis eines Webservers zu verfrachten. Wenn der Aufruf von der Kommandozeile funktioniert (auf Ausführungsrechte und Schreibrechte für die Datenverzeichnisse achten), sollte es auch im Webbrowser klappen – aber bitte auf die unterschiedliche User-ID (meist ist sie »nobody«) achten. Staucht also eure überlangen URLs zusammen, kürzt gnadenlos! *(uwo)* ■

Infos

[1] Listings zu diesem Artikel: [ftp://www.linux-magazin.de/pub/listings/magazin/2004/02/Perl] oder [http://perlmeister.com/cgi/u/2]

[2] Cryptogram: [http://www.counterpane.com/crypto-gram.html] oder [http://perlmeister.com/cgi/u/3]

[3] Echte Programmierer vs. Quiche-Fresser, [http://www.rzuser.uni-heidelberg.de/~mhermann/realpro.html] oder [http://perlmeister.com/cgi/u/4]

Der Autor

Michael Schilli arbeitet als Web-Engineer für AOL/Netscape in Mountain View, Kalifornien. Er hat die Bücher „Goto Perl 5“ und „Perl Power“ geschrieben und ist unter [mschilli@perlmeister.com] zu erreichen.