Create your own abbreviated, tinyurl.com-style URLs

Compressed Links

Long URLs are hard to print in magazine articles and cause problems in emails if they exceed the 78 characters per line maximum. A CGI script on a public server provides shortcuts.

BY MICHAEL SCHILLI

While I was reading "Cryptogram", the monthly newsletter from security superhero Bruce Schneier [2], I noticed that the typically longish URLs for various cross-references were surprisingly short. All of them pointed to *http://tinyurl.com* and ended in short and cryptic abbreviations. So that's how he did it. Sites like *tinyurl.com* or *makeashorterlink.com* offer a free service that stores long URLs and assigns a letter/digit combination to them. Upon request to the shortcut, Tinyurl redirects browsers to the target URL. This led me to register *http://tinyurl.com/28uo8* recently, pointing to the DVD inlay PDF for this issue.

The whole thing is quite easily implemented in Perl. A persistent hash stores unique abbreviations as keys and registered URLs as values. As shortcuts, instead of decimal serial numbers, it uses groups of small letters and numbers, beaming us into the base 36 system (26 letters and ten numbers). Even numbers of a million or more can be represented with just four characters: for example, *4c92* represents the decimal number 1000000 in base 36. A four digit number in base 36 can store 36_exp_4 = 1679616 different values – that should be enough to keep us going for the time being. Listing 1 (*u* – Save as Save Can) shows a CGI script that imitates *tinyurl.com*.

THE AUTHOR

Michael Schilli works as a Web engineer for AOL/Netscape in Mountain View, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at mschilli@perlmeister.com. His homepage is at http://perlmeister.com.

Hardening the Script

Of course a script that you want to publish on the Internet should not fall victim to the first malevolent hacker that happens to stumble across it. So let's introduce a few security measures: 200 URLs are allowed per IP and day – that should help prevent any evildoer from filling up the hard disk with nonsensical URLs. Major service providers like AOL use the same IP to serve a large number of customers at the same time. The value of 200 should cover us in this situation as well.

- The maximum size of the database file won't exceed a configurable value (such as 10MBytes). When this threshold is reached, the script will not accept any new URLs for storage, although it will still find any URLs that have been previously defined.
- A URL cannot exceed 256 characters in length; the script will refuse to handle longer URLs.
- The script will log any events in a rotating log file with a configurable maximum size. CGI capabilities will be provided by Lincoln Stein's CGI module and its *CGI::Carp* offshoot, with the *fatalsToBrowser* tag enabled to capture any Perl exceptions and display them in the browser to provide debugging information. You might want to uncomment *CGI::Carp* if you put the script into production.

If the script fails to find a *url* parameter from a previously stored request, it will display a URL input form in the browser. A click on the Submit button sends the URL entered by the user back to the script as the *url* parameter. The script then generates a short URL and stores the mapping between the short form and the full URL in its mini-database, if it has not been previously stored. URL abbreviations look like this:

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

The URL's unique ID *xxxx* gets appended to the path, it is sent to the script *u*, where the CGI environment will provide it in the *$ENV{PATH_INFO}* variable. If the script receives a request like this, it will retrieve the appropriate full URL from the database and respond with a *redirect()*, thus sending the browser transparently to the corresponding website.

Rotating Logs

Log::Log4perl qw(:easy) and the *FileRotate* appender from the *Log::Dispatch* collection provide convenient logging with the *DEBUG()*, *INFO()*, and *LOGDIE()* macros. *size = 1000000* sets the maximum logfile size to 1MB. *max = 1* specifies that the *FileRotate* appender will rotate a whole logfile called *shrink.log* to *shrink.log.1* when *shrink.log* exceeds the 1MByte threshold. It will not create any additional backups to avoid using more than 2MBytes of local disk space for the logfiles.

The persistent hash *%URLS* tied to the file */tmp/shrink.dat* using *tie()* will not store a single key-value mapping in listing *u*, but three. This is why the keys each need a prefix:

- *by_shrink/*: abbreviation to URL mapping
- *by_url/*: URL to abbreviation
- *next/*: next abbreviation to assign

There are some circumstances where the script should simply stop – untie the permanent hash and then quit the program. It would be inelegant to do this with *exit()*, as this could cause problems in environments such as *mod_perl*. And *return()* only works if *perl* happens to be performing a subroutine. I finally decided to opt for the good old *goto*, which Real Programmers, as compared to Quiche eaters (see [3]), are not afraid to use. This is why the script uses *goto* to jump to the *END* label, which is defined later.

A file cache with a definable expiry date helps reduce the number of URLs that a user can store per IP address and day. The clever *Cache::Cache* module provides a simple interface that uses *set()* to set new entries and *get()* to retrieve them. The *Cache::FileCache* class derived from this module implements this as a file tree on the disk.

Deliberately Forgetful

For every request, the script increments a per-IP counter. When it reaches the configured maximum value (200), it refuses to handle new URL store requests by this IP, but continues to resolve the abbreviations previously defined. *Cache::FileCache* forgets about an IP after a day of inactivity, effectively resetting the counter to zero.

However, this algorithm isn't entirely accurate: At worst, it could block an IP that keeps requesting new URLs within the daily limits for a day.

The web server's CGI environment uses *$ENV{REMOTE_ADDR}* to provide the client's IP address. When launched on the command line, the script doesn't receive any value in *$ENV{REMOTE_ADDR}*, however. This is why line 148 simply sets it to the *NO_IP* string in this case.

The *default_expires_in* option of the *Cache::FileCache* constructor provides the interval in seconds since the last *set()* after which the cache will simply forget the entry. A true value for *auto_purge_on_get* specifies that the cache should search for expired entries on running any *get()* request, and remove them. This prevents the cache from becoming too bloated with legitimate IP addresses.

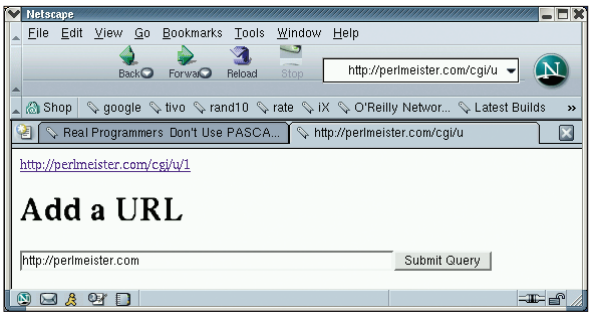

Figure 1: The compressor expects a new URL, stores the URL in its database, and generates an abbreviation.

The Base 36 Universe

The *base36()* function defined in line 123 and following converts decimal numbers to their base 36 equivalents. How does this work? A number in the decimal system is constructed as follows:

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

where a, b, c represent the digits of the number in reverse order of significance. 156 is thus:

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

In contrast to this, base 36 works as follows:

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

The following algorithm converts a decimal number d to base 36: Find the remainder of the division d/36 (that is d modulo 36 or *d % 36*). This results in the last digit of the number in base 36. Then divide *d* by 36, use the integral part of the result, and move on to the next digit (from right to left), repeating the previous instructions.

The *base36()* function first defines all valid characters (that is the numbers 0 through 9 and the lowercase letters a through z) in the array *@chars*. Its length, determined by

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

with *@chars* in scalar context, is unsurprisingly 36. The *for* loop then calculates *$num % $b* with every iteration, resulting in the next (right-to-left) "digit" of the number in base 36. The *for* loop's bump-along instruction – *$num /= $b* – divides *$num* by *$b* neglecting any floating point component, due to the *use integer* pragma set in line 127. The expression

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

extracts the appropriate character from the base 36 character set and adds it to the end of the string in *$result* – the number, or sequence of characters, in the target system is thus constructed from back to front. This is corrected by the following:

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

which flips the string in *$result*. Scalar context is required as *reverse* would simply reverse the order of a list of scalars passed to it in list context, leaving each of them untouched.

Installation

The script requires *Log::Log4perl*, *Log::Dispatch::FileRotate*, and *Cache :: FileCache*, all of which are available on CPAN. The paths to the logfile (line 21) and the database file (line 12) need to be adapted to reflect your local environment, and the script needs to be placed in the *cgi-bin* directory of a web server. If you can run the script from the command line (pay attention to execute and write permissions for the data directories), it should work in a web browser – but watch out for the different user ID (this is typically *nobody*). So go on, be a devil, abbreviate those URLs! ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads*

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

[3] Real programmers vs. quiche-eaters, *http://www-users.cs.york.ac.uk/~susan/joke/quiche.htm* or *http://perlmeister.com/cgi/u/b*

Listing 1: *u* – Save as Save Can

```
001 #!/usr/bin/perl
002 ##############################
003 # Mike Schilli, 2003
004 # (m@perlmeister.com)
005 ##############################
006 use warnings;
007 use strict;
008 use Log::Log4perl qw(:easy);
009 use Cache::FileCache;
010
011 my $DB_FILE =
012           "/tmp/shrinky.dat";
013 my $DB_MAX_SIZE = 10_000_000;
014 my $MAX_URL_LEN = 256;
015 my $REQS_PER_IP = 200;
016
017 Log::Log4perl->init(\ <<EOT);
018 log4perl.logger = DEBUG, Rot
019 log4perl.appender.Rot=\\
020   Log::Dispatch::FileRotate
021 log4perl.appender.Rot.
    filename=/tmp/shrink.log
022 log4perl.appender.Rot.
    layout=PatternLayout
023 log4perl.appender.Rot.layout.
    ConversionPattern=%d %m%n
024 log4perl.appender.Rot.mode=
    append
025 log4perl.appender.Rot.size=
    1000000
026 log4perl.appender.Rot.max=1
027 EOT
028
029 use CGI qw(:all);
030 use CGI::Carp
031          qw(fatalsToBrowser);
032 use DB_File;
033
034 tie my %URLS, 'DB_File',
035   $DB_FILE, O_RDWR|O_CREAT,
036   0755 or LOGDIE "tie: $!";
037
038     # First time init
039 $URLS{"next/"} ||= 1;
040
041 my $redir = "";
042
043 if(exists $ENV{PATH_INFO}) {
044       # Redirect requested
045   my $num = substr(
046          $ENV{PATH_INFO}, 1);
047   $redir =
048   $URLS{"by_shrink/$num"} if
049     exists
050     $URLS{"by_shrink/$num"};
051 }
052
053 if($redir) {
054     print redirect($redir);
055     goto END;
056 }
057
058 print header();
059
060 if(my $url = param('url')) {
061
062   if(length $url >
063      $MAX_URL_LEN) {
064     print "URL too long.\n";
065     goto END;
066   }
067
068   my $surl;
069
070   # Does it already exist?
071   if(exists
072       $URLS{"by_url/$url"}) {
073     DEBUG "$url exists";
074     $surl =
075         $URLS{"by_url/$url"};
076
077   } else {
078     if(-s $DB_FILE >
079               $DB_MAX_SIZE) {
080       DEBUG "DB File full " .
081             (-s $DB_FILE) .
082             " > $DB_FILE";
083       print "We're full.\n";
084       goto END;
085     }
086
087     if(rate_limit(
088         $ENV{REMOTE_ADDR})) {
089       print "To many URLs " .
090             "from this IP.";
091       goto END;
092     }
093
094     # Register new URL
095     my $n = base36(
096            $URLS{"next/"}++);
097     INFO "$url: New: $n";
098     $surl = url() . "/$n";
099     $URLS{"by_shrink/$n"} =
100                          $url;
101     $URLS{"by_url/$url"} =
102                         $surl;
103   }
104
105   print a({href => $surl},
106           $surl);
107 }
108
109     # Accept user input
110 print h1("Add a URL"),
111       start_form(),
112       textfield(
113       -size    => 60,
114       -name    => "url",
115       -default => "http://"),
116       submit(), end_form();
117
118 END:
119
120 untie %URLS;
121
122 ##############################
123 sub base36 {
124 ##############################
125   my ($num) = @_;
126
127   use integer;
128
129   my @chars  = ('0'..'9',
130                 'a'..'z');
131   my $result = "";
132
133   for(my $b=@chars; $num;
134       $num/=$b) {
135     $result .=
136       $chars[$num % $b];
137   }
138
139   return scalar
140          reverse $result;
141 }
142
143 ##############################
144 sub rate_limit {
145 ##############################
146   my ($ip) = @_;
147
148   $ip = 'NO_IP'
149     unless defined $ip;
150
151   INFO "Request from IP $ip";
152
153   my $cache =
154     Cache::FileCache->new({
155       default_expires_in =>
156         3600*24,
157       auto_purge_on_get  =>
158         1,
159       }
160   );
161
162   my $count =
163             $cache->get($ip);
164
165   if(defined $count and
166     $count >= $REQS_PER_IP) {
167     INFO "Rate-limit: $ip";
168     return 1;
169   }
170
171   $cache->set($ip, ++$count);
172
173   return 0;
174 }
```