

Easy-to-use Web Jukebox with Perl and Apache::MP3

Music on the Move

The Apache::MP3 module builds a convenient Web jukebox to organize jumbled MP3 collections. You can even use a Perl script to introduce a simple hierarchy.

Rescuing your ever growing music collection from the chaos of an unordered world is a tedious task, but one that can be accomplished.

BY MICHAEL SCHILLI



In the course of the last few months I have been busy ripping all my CDs to create MP3s, which are now stored on an enormous (120 GBytes!) hard disk. Having finished that step, I then connected a lead up to my sound card, plugged it into my stereo and happily zapped around in a collection of thousands of tracks using only a normal household browser. I was amazed at the tracks I had obviously bought at some time, but completely forgotten! After the dust, and the initial excitement, had settled, I started working on some Perl scripts to add some order to this chaos and make my MP3 collection easier to use.

Okay, I know it was a lot of work ripping all those CDs, but for those of you wary of all this effort, let me just ask if you have ever wished for any of the following:

- That you could launch a track that has been at the back of your mind within a matter of seconds
- That your music collection could be searched by keyword

- That you could create, manage and run playlists of music depending on your current whims
- To be able to run a music server on your private Ethernet to serve up tracks to multiple servers/stereos all over your home

- To categorize songs by their smooth factor and choose a dozen or so to suit the current mood.

Best thing of all, this is entirely legal. Once you have become used to leaving those silver disks in the cupboard, and have come to appreciate the finer points of those extended search and sort facilities, you will hardly be able to imagine the lengths you had to go to, to listen to music way back in the Stone Age of CD technology.

The Ripper is at it again

I used a Perl script called *crip*, which is available for free at [1] to rip my CD collection. Simply place a CD in the drive and *crip* runs off to check for information on the artist, album and tracks in the CDDB database. This information is then stored along with the tracks themselves in an MP3 file. I originally intended to migrate to the Ogg Vorbis format, which is the only one supported by *crip* today (MP3 support is only available up to version 1.0), but I had to change my plans, as my wife's MP3



Figure 1: The jukebox offering to stream the tracks on Green Day's CD "Shenanigans"

player does not support the Ogg Vorbis format. C'est la vie!

MP3 Partitions

To allow for more convenient partitioning of the enormous mass of data (25 GBytes), I decided to store the MP3 files in so-called “pods”; these are subdirectories with three-digit serial numbers (001, 002, ...) of 700 MB each.

Why did I choose that value? Well, 700 MB will easily fit on a CD-Rom, allowing me to create back ups of all that hard work. Let's assume I want to back up pod 027; I simply insert a writable disk and type:

```
cdr 027
```

to start burning the CD. Of course *cdr* is a simple one-liner shell script with the following content:

```
mkisofs -R $* | cdrecord -v >
speed=4 dev=0,0 -
```

If you are not happy with that, refer to [5] for a few useful tips on using typical household CD writers on Linux.

A pod can store somewhere between 150 and 200 MP3 files, and I created 33 pod directories, numbered 001 through 033.



Figure 2: The Jukebox displaying my collection of “Green Day” CDs

Pods can be stored on various partitions on a single or multiple hard disks, where symbolic links in a central file structure are used to reference the MP3 files themselves. This allows you to create multiple views of your CD collection – sorted by album, artist, genre, smooch factor, and so on, without needing to copy the heavyweight MP3 files. They simply stay in their pods.

A script aptly named *topod* picks up the MP3 files from a temporary directory that *crip* uses for ripping CDs, and stores them in the next available pod. The script uses the module *Algorithm::Bucketizer* from CPAN to fill the “buckets” in the pod chain up to the 700 MByte limit and, if necessary, add a new bucket. It also uses a *%seen* hash to locate dupli-

cates in the collection, and prevent new duplicates from occurring. Line 18 in *topod* initializes an *Algorithm::Bucketizer* type object with a 700,000,000 Byte bucket size. It uses the simple algorithm; that is it simply fills the last bucket, before starting on a new one. This ensures that only the latest pod will be subject to change, thus allowing you to create CD ROM back ups of all the other pods.

If the script notices that you already have a few pods on your disk, it first has to convert them to virtual buckets and use the *prefill_bucket* method to pass them to the *Algorithm::Bucketizer* object.

This setup allows the *Algorithm::Bucketizer* to use a while loop starting in line 38 with the *add_item* method to accept

Listing 1: topod

```
01 #!/usr/bin/perl                               $_\n";
02 #####                                         31 }
03 # topod                                       32 $seen{$base}++;
04 # Mike Schilli, 2003                          33 $b->prefill_bucket($idx - 1,
05         (m@perlmeister.com)                  34         $_, -s $_);
06 #####                                         35 }
07 use warnings;                                36 }
08 use strict;                                   37
09                                               38 while(<*.mp3>) {
10 my $POD_DIR =                                  39     if(exists $seen{$_}) {
11     "/ms1/SONGS/pods";                        40         print "Not adding dupe:
12                                               $_\n";
13 use File::Basename;                           41     next;
14 use Algorithm::Bucketizer;                    42 }
15 use File::Copy;                               43
16                                               44 $seen{$_}++;
17 my %seen = ();                                45
18                                               46 my $bucket = $b->add_item
19 # Init buckets                                ($_, -s $_);
20 my $b = Algorithm::Bucketizer-              47
21 >new(                                          48 my $path = sprintf
22     bucketsize => 700_000_000,                "$POD_DIR/%03d/$_",
23     algorithm => 'simple',                      $bucket->serial();
24 );                                             49
25 # Prefill buckets with                       50
26     existing Pods                             unless(-d dirname($path)) {
27 while(<$POD_DIR/*>) {                          51     mkdir dirname($path) or
28     my($idx) = /(\\d{3})/;                    52         die "Cannot mkdir " .
29     while(<$POD_DIR/$idx/*.mp3>)              53         dirname($path);
30     {                                           54
31         my $base = basename($_);              55 }
32         if(exists $seen{$base}) {            56
33             print "Dupe detected:            57 move($_, $path) or
34                                             58     die "Cannot move $_
35                                             to $path";
36                                             59 }
37                                             60
```

new MP3s and store them in the latest bucket, or create a new bucket. To reflect this virtual order, the script creates new subdirectories (line 52) and drops new MP3 files into them (line 57) in the real world. *Algorithm::Bucketizer* enumerates the buckets starting at 0; the directories in the real world are called 001, 002, and so on. The *add_item()* method in line 46 expects the name and size of the MP3 file. The file test operator *-s* is used to ascertain the size.

add_item returns the bucket object that stored the MP3 file, and the *serial()* method returns the index number of the bucket (0, 1, 2, ...). Adding one and prepending one or two zeros (as in 007) will give you the matching pod directory.

Views

Who would want to root around the pod jungle, just to find a track. Instead I opted for a three-tier hierarchy with a top directory of all artists, an album level below this, and then the tracks on each album in the right order.

Now *crip* has already ensured that the individual MP3 files contain the tag information I need for this task. And the file names provide additional clues, as in:

```
The_Strokes_-_ITI02_The_Modern
_Age.mp3
```

The artist (in this case *The_Strokes*) comes first, followed by a dash and the first letters of the words in the album title (ITI = Is This It), followed by the

track number (02), and title (*The_Modern_Age*).

And this is how this song would be stored in the *by_artist* hierarchy I opted for:

```
by_artist
Strokes,_The
Is_This_It
01 .....
02_The_Modern_Age.mp3
03 .....
```

To do this, the *mktree* listing iterates through all the pods, using the *MP3::Info* module to read the embedded info tags of the MP3 files stored in the pods to create the required subdirectory below the *by_artist* tree ("Strokes,_The/Is_

Listing 2: mktree

```
001 #!/usr/bin/perl
002 #####
003 # mktree
004 # Mike Schilli, 2003
    (m@perlmeister.com)
005 #####
006 use warnings;
007 use strict;
008
009 my $POD_ROOT =
    "/ms1/SONGS/pods";
010 my $TREE_ROOT =
    "/ms1/SONGS/by_artist";
011 my $MP3_PATTERN = qr/\.mp3$/;
012 my %ARTIST_MAP = ();
013 my $ARTIST_FILE =
    "artistmap.gdbm";
014
015 use Log::Log4perl qw(:easy);
016 use GDBM_File;
017 use File::Find;
018 use MP3::Info;
019 use File::Basename;
020 use File::Path;
021 use File::Spec;
022 use Getopt::Std;
023
024 Log::Log4perl->easy_init(
025     { level => $INFO,
026       layout => '%m%n' });
026
027 getopts("du", \%opts);
028
029 tie %ARTIST_MAP, 'GDBM_File',
    $ARTIST_FILE,
030     &GDBM_WRCREAT, 0640 or
    031     die "Cannot tie
        $ARTIST_FILE";
    032
    033 if($opts{d}) {
    034     # Dump artist map
    035     for(sort keys
        %ARTIST_MAP) {
    036         print "$_ =>
            $ARTIST_MAP{$_}\n";
    037     }
    038 } elsif($opts{u}) {
    039     # Undump artist map
    040     %ARTIST_MAP = ();
    041     while(<>) {
    042         chomp;
    043         my($k, $v) = split /
            => /, $_, 2;
    044         $ARTIST_MAP{$k} = $v;
    045     }
    046 } else {
    047     # Link hierarchy
    048     entry to pod entry
    049     find(sub {
    050         mmlink($File::Find::name)
    051         }, $POD_ROOT);
    052 }
    053 #####
    054 sub mmlink {
    055     #####
    056     my($file) = @_;
    057     my $tag =
        get_mp3tag($file);
    060
    061     if(!$tag) {
    062         warn "No TAG info in
            $file";
    063         link_path($file,
    064             "Lost+Found/" .
            basename($file));
    065         return;
    066     }
    067
    068     for(qw(ARTIST ALBUM TITLE
        COMMENT)) {
    069         unless($tag->{$_} =~
            /\S/) {
    070             warn "No $_ TAG
            in $file";
    071             link_path($file,
    072                 "Lost+Found/" .
            basename($file));
    073             return;
    074         }
    075     }
    076
    077     my ($track_no) =
    078         ($tag->{COMMENT}
    079             =~ /(\d+)/);
    080
    081     $track_no = "XX" unless
    082         defined $track_no;
    083
    084     my $artist = $tag-
    >{ARTIST};
```

This_It”), and as the song “The Modern Age” is stored in pod 017, to then create the following symbolic link:

```
ln -s ../../pods/018/The_Strokes
  _-_ ITI02_The_Modern_Age.mp3
  by_artist/Strokes,_The/Is_This
  _It/02_The_Modern_Age.mp3
```

Of course, human error does tend to raise its ugly head in the CD data available from the <http://www.freedb.org> database: a simple typo, such as leaving out the second “z” in “Eros Ramazzotti”, will store your database entry in the wrong spot. Or maybe somebody has entered “Tom Waits”, where “Waits, Tom” would be preferable in an alphabetical list.

Brain Power to the Rescue

Now this is difficult to automate: What is the difference between “John Cale”, who we would preferably index under “Cale, John” in our collection, and a famous group such as “Judas Priest”, who we would like to keep just like that?

Use your brain! *mktree* makes a few useful suggestions for each new artist and leaves it up to the user to decide:

```
[1] Judas Priest
[2] Priest, Judas
[1]>
```

You can simply hit “Enter” to accept the first option; if you type a number, *mktree* selects the entry with this number. If

none of the current suggestions makes sense, *mktree* even allows text input at this point, storing your selection persistently until called in a GDBM database.

Starting in line 9 *mktree* first defines a number of installation specific parameters: *\$POD_ROOT* is the root for the pod directories where your MP3 files will be stored, *\$TREE_ROOT* specifies the directory in which artists, albums and tracks will be stored. The persistent *%ARTIST_MAP* hash specifies how to correct an artist’s name after reading it from the MP3 file.

Line 16 calls the GDBM_File module, which is used by the *tie()* command in line 29 to store the *%ARTIST_MAP* hash persistently. Line 24 initializes *Log::Log4perl*; I kept this in for old times’ sake, but

Listing 2: mktree

```
085         unless(exists
086             $ARTIST_MAP{$artist}) {
087             $ARTIST_MAP{$artist}
088             =
089             warp_artist($artist);
090         }
091         $artist =
092         $ARTIST_MAP{$artist};
093         my $relpath = File::Spec->
094             catfile(
095                 map { s/[\s\\]/_/g;
096                     $_;
097                     } $artist, $tag->
098                     {ALBUM},
099                     "${track_no}_$tag->
100                     {TITLE}.mp3");
101         link_path($file,
102                 $relpath);
103     }
104     my $path = File::Spec->
105         rel2abs(
106             $relpath, $TREE_ROOT);
107     my $dir = dirname($path);
108     unless(-d $dir) {
109         die "Cannot
110             mkpath $dir";
111     }
112     unless(-l $path) {
113         die "Cannot
114             symlink $file
115             to $path";
116     }
117     or
118         die "Cannot
119             symlink $file";
120 }
121 sub warp_artist {
122     my($artist) = @_;
123     my @choices = ();
124     my @c = split ' ',
125             $artist;
126     if(@c == 1) {
127         @choices = ();
128     } elsif($c[0] =~
129             /^the$/i) {
130         my $the = shift @c;
131         @choices = ("@$c,
132             $the");
133     } elsif(@c == 2) {
134         @choices = ("$c[1],
135             $c[0]");
136     } elsif(@c == 3) {
137         @choices = ("$c[2],
138             $c[0] $c[1]");
139     }
140     return pick($artist,
141                 @choices);
142 }
143 sub pick {
144     my(@options) = @_;
145     my $counter = 1;
146     for(@options) {
147         print "[",
148             $counter++, " ] $_\n";
149     }
150     $| = 1;
151     print "[1]>";
152     chomp(my $input =
153             <STDIN>);
154     $input = 1 unless $input;
155     if($input =~ /\^d+$/) {
156         return
157             $options[$input-1];
158     } else {
159         return $input;
160     }
161 }
162 }
```

it does allow you to make *mktree* more or less talkative. `%m%n` simply outputs the log message on a newline character.

Thanks to line 27, *mktree* also understands the `-d` (Dump) and `-u` (Undump) options, which output or set the contents of the persistent `%ARTIST_MAP` hash.

```
mktree -d > data
```

creates the artist list in the *data* file, as in:

```
The Beatles => Beatles, The
Salt 'N' Pepa => Salt 'N' Pepa
Zucchero Sugar ➤
Fornaciari => Zucchero
```

If you use a text editor to manipulate *data* manually, you can type:

```
mktree -u < data
```

to load the whole kit and caboodle back into the binary GDBM file, and *mktree* will automatically repair itself, the next time you call it. This is very practical if I mistype an entry when prompted by *mktree*.

Extracting MP3 Tags

The `MP3::Info` module helps to read the tag information stored in the MP3 files. The `get_mp3tag()` function exported by the module, expects an MP3 filename and returns a reference to a hash containing the CD data entries for all of the `ARTIST`, `ALBUM`, `TITLE` and `COMMENT` keys.

The `mklink()` function (its definition starts in line 55 of *mktree*) expects the complete path for an MP3 file in a pod, using `MP3::Info` to extract the corresponding CD data and ascertain the album title and the normalized artist name – this can involve some user interaction.

`link_path()` then uses `symlink` to create a symbolic link in `by_artist/interpret/album/song.mp3` that points at the actual MP3 file in the pod.

In the case of confusing or missing MP3 tag data, a link is created in the `Lost + Found` directory.

The script uses a regular expression to extract the track number from the MP3 `COMMENT` field, which contains

something along the lines of “track11”. Space characters and illegal slashes are replaced by simple underscores using the `map` command in line 94.

As `s/[\\s\\/_]/_/g;` does not return the resulting string, but the number of replacements, you need to add `$_;` to allow the `map` command to pass the individual components to *catfile*. This function, which is part of the `File::Spec` collection, concatenates the elements to finally create a pathname.

Starting at line 123, `warp_artist()` attempts to put forward more-or-less intelligent suggestions based on an artist’s name passed to it. If you pass “The Red Hot Chili Peppers” to the function it will generate both “Red Hot Chili Peppers, The” and “The Red Hot Chili Peppers”, and allow you to choose between these two. When confronted with “Rory Gallagher”, it will suggest both “Rory Gallagher” and “Gallagher, Rory”.

Finally (as of line 146), `pick` expects a list of suggestions, offering the user an enumerated list of strings, and returning the string that matches the number selected by the user, if any. On the other hand, if the user enters a text string, `pick` will pick it up (sorry about the pun) and hand it back to the caller.

Installation

After customizing the configuration lines to reflect your local environment, simply launch *topod* and *mktree* from the command line. *topod* will locate ripped MP3 files in the current working directory, *mktree* can run anywhere. After setting up the `by_artist` tree, we simply have to set up an Apache Webserver to serve it up.

You will need a `mod_perl` capable Apache version (see the Howto at [3]). Your local Perl installation also needs the `Apache::MP3` module by CPAN. My installation worked with Apache 1.3.37 – but `mod_perl` should now run reliably on 2.0. The following lines in `httpd.conf` should enable your music server:

```
<Location /songs>
  SetHandler perl-script
  PerlHandler Apache::MP3::Sorted
  PerlSetVar SortFields Artist, ➤
```

```
Album,comment
</Location>
```

The `/songs` directory below the document root, *htdocs*, of the Apache server, must point to (or at least use a symbolic link to) the `by_artist` directory, created by *mktree* earlier on. The configuration file will need a few lines, such as the following, to allow Apache to follow the link:

```
<Directory />
  Options FollowSymLinks
  AllowOverride None
</Directory>
```

If you then restart and point your browser to:

```
http://localhost/songs
```

you can browse around your collection to your heart’s delight. Clicking on a “stream” link for a song will launch the Linux MP3 player, *xmms* ([4]), and play one or more songs in sequence or random order, as specified. And this is just the start of a beautiful new relationship. ■

INFO

- [1] Crip Homepage:
<http://bach.dynnet.com/crip/>
- [2] Apache::MP3:
<http://namp.sourceforge.net>
- [3] Apache mod_perl homepage:
<http://perl.apache.org>
- [4] xmms on Redhat 8.0/9.0 no longer plays MP3s, a fix is available from:
<http://www.gurulabs.com/downloads.html>
- [5] Steve Litt, “Installing Your ATAPI CDRW Drive in Linux”: <http://www.troubleshooters.com/linux/cdrw.htm>

THE AUTHOR

Michael Schilli works as a software 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>.

