EWA Manual

Author:	Jacob Smullyan
Contact:	jsmullyan@gmail.com
Organization:	WNYC New York Public Radio

Date:	2007-02-02
Revision:	164
Version:	0.62.1

Contents

Overview

Ewa (East-West Audio) is an application that manages the production of spliced mp3 files. It is meant to solve a common audio production problem facing producers of audio for the web who would like to add extra content such as credits and promotional or underwriting messages to their mp3s and be able to update those messages periodically without having to remaster all their mp3s from scratch.

Ewa makes a distinction between *content files* and *extra files*. The content files contain the material of main interest; the extra files are the promotional material. Ewa assumes that one resultant mp3 is normally an aggregation of exactly one content file with any number of extra files.[1]

In order to know what extra files to combine with a particular content file, ewa consults a *rule*, which is a function that takes the name of the requested file and returns an ordered list of files that should be combined. The rule ewa consults is usually a special kind of rule called a *rule list* that contains a list of sub-rules; the first sub-rule that matches, i.e., that returns a non-empty list, is the return value of the parent rule. Ewa provides a expressive mini-language for specifying such rule lists; with it, rules can apply only for filenames that match glob or regular expression patterns, or that match date formats, or combinations of such criteria with "and", "or", and "not" operators. Also, rules can be made active only for certain date ranges, so you can add configuration in January that will only become effective in February. If the rule system isn't flexible enough and you have special needs, it is also feasible to plug in your own, implemented in Python.

Ewa also manages transcoding the extra audio to match the content files with which it may be spliced. Master files for each piece of extra audio are placed in a directory managed by ewa, and ewa transcodes them as needed, leaving the transcoded files in another managed directory for future runs. The masters may be in mp3, wav, or aiff format.

Modes of Operation

Ewa can operate either in batch mode, in which case it produces combined files for the content files specified on the command line, or in server mode.

Batch Mode

The batch mode, controlled by the script ewabatch, is convenient if you can't run your own persistent processes on the web server serving your audio files; you can use it to produce static files on a machine under your own control and rsync them up to the web server however often you need. Batch mode can operate either on individual files or recursively on entire directories.

Server Mode

If you are using `ewabatch` to generate all your files, there is no need for any integration with the web server at all; you just need a cron job to generate the files periodically and perhaps rsync them. But if you have a large number of files, this can become unwieldy. Not only must each new file must be processed before going live, but the costs in time and bandwidth of changing intros for a large number

of mp3s and having to rsync them up to your webserver may be prohibitive if undertaken frequently. It is highly inefficient to have to move thousands of files just because an intro has changed. The ewa server gets around this problem.

The server mode, controlled by the script ewa, is a persistent daemon running a simple WSGI application, normally connected to a web server via FCGI or SCGI, which generates the composite files on demand, caching them on the filesystem and rebuilding them upon request with a configurable frequency. The path to the composite file is passed to the webserver via the X-Sendfile technique (which originated with lighttpd and is also supported by apache with the mod_xsendfile module; nginx also has a similar feature). The webserver is responsible for returning the actual file over HTTP, and ewa does not need to do IO; as a result, ewa processes each request extremely quickly, and files are served at almost the same speed as static files, with excellent scaleability.

Limitations

Ewa has a few limitations that the user should be aware of.

1. Mp3 is the only supported audio format.
2. Ewa only supports CBR (constant bit rate) encoding.
3. Ewa's rule system only takes into account the name of the requested content file and the current time and date in determining the list of files to splice; in particular, it isn't currently suited to personalizing mp3 downloads.
4. Ewa currently does not support the dynamic writing of id3 tags; it takes whatever id3 tags are on the main content file and transfers them verbatim to the composite.
5. Ewa relies on the model of one content file + multiple extra files; scenarios with multiple content files aren't supported.

Some or all of these may be addressed in future revisions, depending on community interest.

A Note on Mp3 Splicing

You will occasionally read that mp3s cannot be reliably spliced, as mp3 frames may store information used by later frames in the bit reservoir. This is not quite true; the reality is that mp3s cannot be reliably *cut and spliced*. In ewa, all the mp3s are spliced on preexisting mp3 boundaries; they are not cut (except to drop a bad frame at the end of a file). Obviously, the last frame in an mp3 does not store content in the bit reservoir for subsequent frames. Therefore, the bit reservoir does not present a problem for ewa.

Ewa attempts to produce spliced files that are without bad frames; to do so, it looks at the frames preceding frame boundaries and discard broken ones. However, ewa also attempts to splice very quickly, and hence cannot scan entire mp3s to clean them; if the mp3s going into ewa are broken, the ones coming out will be too.

Installation

Supported Platforms

Ewa has been developed and tested on Linux, but should work fine on any flavor of BSD, including Mac OS X, and commercial UNIX implementations. It hasn't been tested on Windows, but in future might work there in whole or in part. Please note that some parts of this manual presuppose a UNIX platform.

Ewa is written in Python, and requires Python 2.4 or later. In addition, the following Python packages need to be installed:

- setuptools
- simplejson
- eyeD3
- ply (>=2.2)
- flup

To run tests you also need:

- nose

Ewa also requires that lame be installed for transcoding. To run the ewa server, you need to run an http server that supports X-Sendfile or something equivalent: either lighttpd, apache with mod_xsendfile, or possibly nginx.

Getting Ewa

Ewa releases are available in binary and source form from http://cheeseshop.python.org/pypi/ewa.

If you want to follow the bleeding edge development version, you can check out the latest source code from our subversion repository:

```
svn co svn://svn.wnyc.org/public/ewa/trunk ewa
```

Software Installation

To install, if you already have setuptools installed, you can simply do:

```
easy_install ewa
```

Or, if you have already installed the source tarball and have unpacked it, cd into it and type:

```
easy_install .
```

or equivalently:

```
python setup.py install
```

The latter will install setuptools if you don't already have it.

The Managed Audio Directory

Ewa expects audio to be stored in a directory structure like:

$basedir/main Your content mp3s go here; you manage this directory and can organize it however you like. Ewa needs read access to it.

$basedir/extra/masters Your "extra" files -- intros, outros, ads, etc. -- go here; you manage this directory also. Ewa needs read access to it also.

$basedir/extra/transcoded Ewa manages this directory and needs write access to it; it stores transcoded versions of the audio files extra/masters here.

$targetdir Ewa manages this directory and needs write access to it; this is where it stores the spliced files.

`basedir` and `targetdir` are configuration-defined. You must specify `basedir` in `ewa.conf`; `targetdir` will default to `$basedir/combined` if not otherwise specified.

Permissions Gotchas

Some care is necessary to ensure that file permissions will be right for your deployment, especially if you are running both the ewa server and ewa batch processes, as a variety of users may then be creating files in the managed directories.

One approach is to create a user and group that the ewa server will run as, give ownership of the managed directories to it, and make them both group-writeable and the group permissions sticky. On Linux, you might do this:

```
groupadd ewa
useradd -g ewa -s /bin/false  -d $targetdir -c "ewa user" ewa
chown -R ewa:ewa $targetdir $basedir/extra/transcoded
chmod -R g+ws $targetdir $basedir/extra/transcoded
```

While you are at it, creating directories for ewa's pid file and log file isn't a bad idea:

```
mkdir -p /var/{run,log}/ewa && chown ewa /var/{run,log}/ewa
```

In `ewa.conf` you'll want to set the `user` and `group` variables to match the user and group you created. If you do this, `ewa` and `ewabatch` will need to be run as root (in the case of `ewabatch`, most conveniently through `sudo`), but will drop credentials to your user/group before it creates any files.

Configuration

Ewa has two configuration files: `ewa.conf`, for adminstrative options, and a rule configuration file, which is used to determine the playlists.

The `ewa.conf` File

`ewa.conf` is written in Python; keys defined there that don't start with an underscore become attributes of the `ewa.config.Config` object. The following are meaningful keys:

basedir The path to to the base audio directory. Must be supplied, as there is no default.

rulefile The path to the file with ewa rules, either in Python, JSON or ewaconf. If the file ends with `.py`, it is assumed to be in Python; if with `.json` or `.js`, in JSON; otherwise ewaconf. This also must be supplied.

targetdir The path to the directory where ewa will place generated composite files. If not supplied, basedir + `/combined` will be used.

protocol what server protocol to use: one of `'fcgi'`, `'scgi'` or `'http'`, defaulting to `'fcgi'`. `'http'` is for development only and should not be used otherwise.

interface an ip address like `'127.0.0.1'`, which is the default.

port default: `5000`.

unixsocket if you want to use a UNIX rather than a TCP/IP socket, put the path to the socket file here; e.g., `'/var/run/ewa.socket'`.

umask if you are using a UNIX socket, this will determine its permissions; e.g., `0600`.

logfile path to logfile. By default there is no logfile and hence no logging.

loglevel how much to log -- should be one of `'debug'`, `'info'`, `'warn'`, or `'critical'`, defaulting to `'critical'`.

logrotate if you want to rotate your logfiles, set this to one of the following:

- `True`. This will result in a logfile that rotates when the file reaches 10M in size; up to 10 backups will be kept.
- an integer meaning the maximum number of bytes that should be stored before rollover; up to 10 backups will be kept.
- a two-tuple of integers specifying the maximum number of bytes that should be stored before rollover and the number of backups to retain: e.g., `(1e7, 5)`.
- `'daily'` (rotates every day at midnight regardless of size)
- `'weekly'` (rotates on Monday at midnight)
- a value accepted for the `when` constructor parameter of `logging.handlers.TimedRotatingFileHandler` (see Python's logging documentation for details): e.g., `"D"`.
- a `when` parameter, as above, followed by a colon and a value accepted for `TimedRotatingFileHandler`'s `interval` parameter (an integer); e.g, `"D:3"`.

daemonize whether the server process should daemonize (default: `True`).

use_xsendfile whether to send an X-Sendfile or equivalent header from the server process to the front-end web server (default: `True`).

sendfile_header what flavor of X-Sendfile-ish header to send. `'X-Sendfile'` is the default, but lighttpd in versions <=‘.4.11 requires `'X-LIGHTTPD-send-file'` instead, and nginx uses `'X-Accel-Redirect'` (with slightly different semantics).

stream whether to stream the concatenated file directly rather than saving to disk. This is not a production-quality option; don't use it.

refresh_rate how often to refresh combined files, in seconds. Default is `0` (never refresh).

pidfile if daemonizing, where to put a pidfile (default: `None`).

content_disposition if you want a `Content-Disposition: attachment` header, set this to `'attachment'`. Default is `None`.

user If you run in either server or batch mode as root and want to drop credentials to another user/group, set this.

group Same as for user.

engine What splicing engine to use. You don't want to change this or even know about it.

use_threads Whether to use a pool of threads rather than a pool of forked processes. If the platform supports `fork()`, this will default to `False`; otherwise (that is, on Windows) to `True`.

lame_path The path to the `lame` executable, for transcoding. Default is `/usr/bin/lame`.

min_spare For the FCGI and SCGI backends, the minimum number of spare threads or processes. Defaults to 1.[2]

max_spare For the FCGI and SCGI backends, the maximum number of spare threads or processes. Defaults to 5.

max_threads For the FCGI and SCGI threaded backends, the maximum number of threads. Default is unlimited.

max_children For the FCGI and SCGI preforked backends, the maximum number of child processes. Default is 50.

max_requests For the FCGI and SCGI preforked backends, the maximum number of requests a child process handles before it is killed. Default is 0 (unlimited).

The EWA Rule Configuration File

The rule file can be written either in Python or in a special configuration mini-language, ewaconf.[3]

A rule file in Python format gives you maximum flexibility, at the cost of requiring you to know Python and understand the ewa API. The Python file can contain anything as long as it defines a global with the name `rules`, which should be a Python callable that, when called, returns an iterator that yields symbolic names for the files that should be combined. (These names will be interpreted as file paths relative to the `extra/masters` managed directory, unless they have the Python attribute `is_original` set to a true value, in which case, they will interpreted as file paths relative to the `main` managed directory.) With this hook you can load into ewa just about any sort of rule system that you might like to devise.

The Ewaconf Configuration Language

Ewa's default rule configuration format is designed to make it easy to define a list of rules that say, for a given mp3 file, what files ewa should combine to make an aggregate file, and in what order. The rules are consulted in order, and checked to see if they match the input mp3 file; the first one that matches returns a list of files to combine, and those are then combined. `ewaconf` only supports a limited number of rule types, but nonetheless the system is quite powerful.

A rule is normally written in the form:

```
condition [options]:
   pre:  [file1,file2...]
   post: [file1,file2...]
```

where a condition is a glob pattern, a regex pattern, or a date specification, or combinations of the above with with the logical operators `and`, `or`, and `not`. The `pre` and `post` lists indicate what files should go before or after the main content file in the aggregate file ewa produces. Condition options are put in brackets after the condition and separated by commas; they can either be a single symbol, such as `F` or `I`, or a name-value pair, separated by `=`. For example:

```
bigband*.mp3 [I]:
  pre: [bigbandintro.wav]
  post: [bigbandoutro.wav]
regex:schwartz.*:
  pre: []
  post: []
and(09/01/2006 - 11/01/2006 [F,fmt=YYYYMMDD],
    or(lopate/*, bl/*):
  pre: []
  post: [specialoutro.mp3]
```

The regular expression follows Python regular expression rules. If you want a regex to ignore case, you can pass the `I` option. Two other regex options are supported: `U` (unicode) and `L` (locale). These correspond to the same options in the Python `re` module. For more information, see the official Python documentation.

Globs support only one option: `I`. By default, globs are case-sensitive, but if this option is passed they will ignore case. (Globs are implemented with Python's fnmatch module.)

Both globs and regexes can contain arbitary characters if they are delimited with either single or double quotation marks. They can also be written without quotation marks, with some restrictions. Spaces are not permitted for either; for regexes, colons and commas must be escaped with a preceding backslash. Unquoted globs are furthermore restricted to alphanumeric characters, forward slashes, asterisks, question marks, underscores, and periods. When in doubt, quote.

Hint

Both globs and regexes need to match the *entire path* to requested file, relative to the main content file directory (`$basedir/main`); and furthermore globs and regexes have different matching behavior, in that a regex will match as long it matches against the beginning of the target string, but a glob needs to match all the way to the end. So if someone requests `http://bozoland.org/dingdong/frogling.mp3`, the path against which your pattern will be matched will be `dingdong/frogling.mp3`, *without* a leading forward slash. `*frogling*` would match it, as would `regex:.*frogling`; `frogling.mp3` wouldn't, and neither would `dingdong`, but `regex:dingdong` would.

The date options are `F`, `T`, and the name-value option `fmt`. `F` and `T` are incompatible. `T` is the default (so its use is actually not necessary except perhaps for readability); it means that the condition will return true only if the current time matches against the date range specified.

`F` means that the date is matched against the filename using a regular expression derived from a format (the `fmt` option); the default format is `MMDDYYYY`. Formats may be specified with the following symbols:

- MM (months)
- DD (days)
- YY (2-digit year)
- YYYY (4-digit year)
- HH (hours, 24 hour clock)
- mm (minutes)
- PM (AM or PM)
- hh (hours, 12 hour clock)

Any additional characters in the format become a literal part of the regular expression. The `fmt` option has no meaning and may not be used when matching against the current time.

If the pre and post lists are both empty, the special form `default` may be used. Also, if a rule applies unconditionally, the condition may be omitted. Therefore, the following four forms are equivalent:

```
*: pre: [], post: []
*: default
pre: [], post: []
default
```

For regex rules, it is possible for the filenames in the pre and post lists to back-reference named groups in the matching regex. Named or numbered group references can be used, with either a shell-like interpolation style:

```
regex:^/shows/(?P<showname>[^/]+)/.*\.mp3:
   pre:  ["intro/$showname.mp3", "ad/${showname}.mp3"]
   post: ["notices/$1.mp3", "outro/${1}.mp3"]
```

or the style used by backreferences in Python regular expression expansions:

```
regex:^/shows/(?P<showname>[^/]+)/.*\.mp3:
   pre:  ["intro/\\g<showname>.mp3"]
   post: ["outro/\\1.mp3"]
```

Note that these forms need to be quoted.

Warning

Back-references can be used with compound conditions only if they refer to the last matching element in the compound condition -- and if the last element is itself compound, the last matching element of it, etc. For instance, the first of the next two rules will work, and the second will not:

```
# if this matches, the match result of the regex will be
# returned, and the back-reference will work
and(>01-01-2001, (and(*nougat*, regex:"(foo|bar)"))):
    pre: ["$1.mp3"]
    post: []

# if this matches, the match result of the date match
# will be returned, and back-references don't work
# with those, so the literal string '$1.mp3' will be
# used instead -- probably not what you want
and((and(*nougat*, regex:"(foo|bar)"), >01-01-2001)):
    pre: ["$1.mp3"]
    post: []
```

With `and`, the last matching element will always the very last element. With `or`, however, that is not the case -- as soon as one of an `or` compound condition's sub-matchers matches, that match is returned and subsequent sub-matchers are ignored.

It is convenient under some circumstances to nest lists of rules, with a conditional qualifier shared by all of them. To do this, enclose the nested list of rules in matching brackets:

```
regex:shows/(?P<showname>[^/]+)/.*: [
    <=09-01-2005 [F]: default
    09-02-2005 - 10-14-2006 [F]:
       pre: ["intro/$showname.mp3"]
       post: []
    >10-15-2006 [F]:
       pre: [current.mp3]
       post: [current.mp3]
    ]
```

For a complete reference, see the grammar specification below.

Checking the Syntax of the Configuration Files

The ewabatch script, when run with the `-t` option, will perform a syntax check on both `ewa.conf` and the rulefile, and either exit with a `Syntax OK` message or blow up with a possibly helpful traceback.

Configuring the Web Server

Two recommended options for integrating ewa with a web server are discussed below.[4]

Lighttpd

First of all, enable `fastcgi` in `ewa.conf`. If you are using lighttpd in version 1.4.11 or lower, set `sendfile_header` to `'X-LIGHTTPD-send-file'`.

Then use something like the following lighttpd configuration:

```
server.modules = ( "mod_access",
                   "mod_fastcgi",
```

```
                    "mod_accesslog",
                    "mod_staticfile" )

server.document-root = "/path/to/basedir"
server.errorlog      = "/var/log/lighttpd/error.log"
server.port          = 80
accesslog.filename   = "/var/log/lighttpd/access.log"

fastcgi.server = (
                  "/" =>
                    ( "127.0.0.1" =>
                      (
                        "host" => "127.0.0.1",
                        "port" => 5000,
                        "check-local" => "disable",
                        # note: this is for lighttpd < 1.5.
                        # for 1.5, apparently you do instead:
                        # proxy-core.allow-x-sendfile = "enable"
"allow-x-send-file" => "enable"
                      )
                    )
                )
```

Apache with mod_xsendfile

TBD. This should be a fairly straightforward combination of mod_scgi and mod_xsendfile.

Using The CLI Programs

Below are summaries of the commandline options of `ewa` and `ewabatch`, and also for a third less important program, `ewasplice`, which provides lower-level access to ewa's splicing facilities.

`ewabatch`

usage: `ewabatch` [options] [files]
Produces a combined MP3 file according to the specified rules.

options:

-h, --help	show this help message and exit
-c CONFIGFILE, --config=CONFIGFILE	path to ewa config file
-r, --recursive	recurse through directories
--rulefile=RULEFILE	specify a rulefile
-d, --debug	print debugging information
-n, --dry-run	don't do anything, just print what would be done
-e ENGINE, --engine=ENGINE	which splicing engine to use (default ewa splicer, mp3cat, or sox)
-a, --absolute	interpret file paths relative to the filesystem rather than the basedir (default: no)
-t, --configtest	just test the config file for syntax errors

Hint

With both `ewabatch` and `ewa`, if you don't specify a config file, ewa will look for it in `~/.ewa/ewa.conf` and `/etc/ewa.conf`.

`ewa`

usage: `ewa` [options]

Starts ewa's WSGI application that produces combined MP3 files according to the specified rules.

options:
-h, --help show this help message and exit
-c CONFIGFILE, --config=CONFIGFILE path to ewa config file
-D, --nodaemonize don't daemonize, regardless of config settings

`ewasplice`

usage: `ewasplice` [options] files

This utility splices MP3 files together using the ewa splicer, but doesn't use the managed directories or perform automatic transcoding. You have to specify a file as "tagfile" so it knows where to get id3 tags.

options:
-h, --help show this help message and exit
-o OUT, --output=OUT output file (default: stdout)
-t TAGFILE, --tagfile=TAGFILE tag file
-d, --debug print debugging information
-s, --sanitycheck sanity check the input mp3 files
-e ENGINE, --engine=ENGINE which splicing engine to use (default ewa splicer, mp3cat, or sox)

Appendix I. `ewaconf` Formal Grammar Specification

Normative EBNF

The below is an EBNF grammar for the rule configuration format:

```
grammar        := cond_rule [','? cond_rule]*
rulelist       := '[' cond_rule [','? cond_rule]* ']'
cond_rule      := [cond ':']? rule
rule           := simplerule | rulelist
simplerule     := prelist ','? postlist | postlist ','? prelist | 'default'
prelist        := 'pre' ':' speclist
postlist       := 'post' ':' speclist
speclist       := '[' [specifier [',' specifier]*]? ']'
specifier      := string
string         := BAREWORD | QWORD
cond           := cond_expr | simple_cond
cond_expr      := cond_op '(' cond [',' cond]+ ')'
cond_expr      := NOT '(' cond ')'
cond_op        := 'and' | 'or'
simple_cond    := regex | glob | datespec
```

```
regex          := BAREREGEX condopts? | QREGEX condopts?
glob           := string condopts?
datespec       := daterange condopts?
daterange      := [date '-' date] | [ datecompare date ] | date
datecompare    := '<' | '<=' | '>' | '>=' | '='
date           := DATE | DATETIME
condopts       := '[' condopt [',' condopt]* ']'
condopt        := BAREWORD | BAREWORD '=' BAREWORD
```

Lexical Details

Significant Tokens

The tokens that the lexer must produce will be:

BAREWORD an unquoted string with alphanumeric characters, asterisks, backslashes, question marks, underscores, or periods.

QWORD a string delimited by single or double quotation marks. Internal quotation marks of the same type used as the delimiter must be escaped.

BAREREGEX a string that matches a regex; should start with `regex:`, followed by an unquoted string with the same restrictions as BAREWORD above.

QREGEX like a BAREREGEX, but the regex, after the `regex:` prefix, is delimited by single or double quotation marks, and escaping (except of quotation marks) is not necessary.

DATE MM-DD-YYYY format. The separator can also be a slash (/) or a period (.), but the same separator must be used in both positions.

DATETIME MM-DD-YYYY HHMM format. The separator can also be a slash or period, as with DATE, and the space before the hour can be either a space or the previously used separator.

DEFAULT 'default'

PRE 'pre'

POST 'post'

AND 'and'

OR 'or'

OP '<', '<=', '>', '>=', '='

DASH '-'

COMMA ','

COLON ':'

LBRACK '['

RBRACK ']'

LPAREN '('

RPAREN ')'

Ignored Tokens

Any text on a line after a pound sign (#) is a comment and is ignored. Whitespace, including line returns, is ignored between tokens. Indentation may be freely used to clarify patterns.

Complete Example

```
# test rule file.

# comments and blank lines are ignored.
08/01/2005-12/01/2006: default

>08/08/2006: pre: [lumpy.mp3], post: []

shows/bl/*:
  pre: [intro/bl.mp3, ad/generic.mp3]
  post: [outro/bl.mp3]

regex:"^/shows/(?P<nick>[a-z][a-z0-9]+)/(?P=nick).*\.mp3" [I]:
  pre: [intro/newyear.mp3], post: []

=01/20/2001 [F, fmt=MMDDYYYY]: [
   shows/studio/*:
      pre: []
      post: [outro/studio.mp3]
   shows/pingpong/*:
      pre: [foomanchu.mp3,bingo.mp3]
      post: []
   pre: [plop.mp3], post: []
   ]
default
```

[1] There are use cases in which you might want more than one content file -- one for each segment of a radio program, for instance -- but this usage is not currently supported.

[2] The stated default value of this config variable, and of the several following which refer to the configuration of the FCGI and SCGI daemons, are actually enforced by flup; the value help in `Config` object for all of them is actually `None`.

[3] Actually, there is a third format -- a special dialect of JSON -- but it isn't very useful and may be dropped in a future release.

[4] Other options are possible. In addition to nginx, mentioned elsewhere, it would be possible run ewa's WSGI application in another WSGI container or even a CGI. With Apache's `mod_rewrite` it is possible to detect whether a static file is available and serve it directly if so, and only call a splicing backend if not, which, if X-Sendfile were not available, could accomplish much the same thing with an external redirect.