www.phparch.com

June 2016
VOLUME 15 - Issue 6

php[architect]

decoupled by design

Security Architecture: Securing your Web Services, Part Two

Distributed Workers and Events

The Middleware Awakens

The Art of Transduction

ALSO INSIDE

Professional Development for Professional Developers

MySQL's JSON Data Type

Community Corner:
PHPeople

Education Station:
Easy Audio and Video Manipulation with FFmpeg

finally{}:
Ups and Downs of an Entrepreneur

Easy Audio and Video Manipulation with FFmpeg

Matthew Setter

Wouldn't it be nice to be able to script the workflow, using something that takes the raw files and does it all for us—preferably in PHP? This month, I'm going to show you how to do just that, by using the php-ffmpeg library and the open-source package FFmpeg. We'll be able to script this functionality, so that we can do it repeatedly, whenever we need to, without relying on applications like iTunes, Windows Media Player, or VLC.

What would the Internet be without audio and video? Likely, it would be a medium that never took off or became an integral part of everyday life. And it's not just the Internet where audio and video are essential. What about the exploding world of podcasts? What about online tutorials and screencasts, spectacularly realized in the PHP world with Laracasts? What about creating compilation videos of the recent holiday, or storing all our music in MP3 format?

Without audio and video, the modern world would be a much sadder, quieter place than it is today. Sure, there is a range of websites and services that allow us to convert or transcode a video or audio file across a range of formats. We can download and install applications that can also extract slices of video or scale, rotate, and merge them together.

If you're anything like me, automating this process probably sounds enticing. To put it in context, I regularly create online training courses and produce two episodes a month of my podcast, http://freethegeek.fm. I've used a range of software and services, including Audacity[1], and auphonic[2], to perform all the post-processing I need.

What is FFmpeg?

But first, what is FFmpeg? Drawing directly from the project's website, FFmpeg[3] is:

> *A complete cross-platform solution for recording, converting, and streaming audio and video.*

Going further, FFmpeg can *decode*, *encode*, *transcode*, *mux*, *demux*, *stream*, *filter*, and *play* almost any video or audio format in existence. It's available on all major platforms, including *Mac OS X*, *Microsoft Windows*, *BSD*, and *Solaris*.

That's a lot to take in if you're new to working with audio and video formats other than listening to or watching them. So here's an explanation of some the terms mentioned above:

- **Transcode**: Change a file from one format to another, such as converting an MP3 to a WAV file.
- **Mux (multiplex)**: Combines different types of data in a single stream or file.
- **Demux (demultiplex)**: Split a video and audio into separate files.
- **Stream**: Listen to music or watch video in 'real time' on a different device.
- **Filter**: Apply some form of processing to the file. There are three types of filters: pre, post, and intra. Pre-filters run before encoding. Post filters run after encoding. Intra filters run during encoding.
- **Encode**: Digitize video or audio information into different video or audio standards from a source.

Installing FFmpeg

As the library we'll be using is, in effect, a wrapper over FFmpeg, we have to install the FFmpeg binaries first. Here's how to do that, depending on whether you're running *Mac OSX*, *Linux*, or *Windows*. If you're on Windows, then download the build for your version of Windows from Zeranoe FFmpeg[4]. They have options for 32- and 64-bit installations.

If you're on Linux, there are builds available for download, or you can use the package manager of your choice to install it. Look for a package named *ffmpeg* or similar.

If you're on Mac OSX, you can download and install a build for that as well, or use one of the package managers, such as MacPorts[5] and Homebrew[6], to do it for you.

Installing the Library

With the binaries installed, the next thing to do is to install php-ffmpeg. Like all modern PHP libraries, this can be done using Composer. To do so, from the root of your project directory, run the command:

```
composer require php-ffmpeg/php-ffmpeg·.
```

This will add php-ffmpeg as a dependency to an existing project or create a `composer.json` file and add it as the first dependency, if this is a new project. Regardless, the library will be available shortly after that in the vendor directory.

1 *Audacity:* http://www.audacityteam.org
2 *auphonic:* https://auphonic.com
3 *FFmpeg:* https://ffmpeg.org
4 *Zeranoe FFmpeg:* https://ffmpeg.zeranoe.com/builds/
5 *MacPorts:* https://www.macports.org
6 *Homebrew:* http://brew.sh

Converting a Stereo Track to Mono

Now that php-ffmpeg is installed, let's see how to convert a stereo track to mono. If you're not familiar with the difference between stereo and mono, when you're listening to most podcasts you're listening to audio in mono. When you're listening to a recording of a symphony orchestra, you're listening in stereo.

The difference is that in mono, the audio plays the same thing at the same time across all speakers, whether there is one or several speakers. In stereo, different audio tracks can be heard at different times from different speakers. For example, you might hear a guitar solo on your left side, and the lead singer on your right.

Let's say I've accidentally recorded my podcast in stereo format, and I want to change it to mono. This can be important as there will be extra audio information in the stereo file, making it significantly larger than it needs to be. Let's see how to convert it, and reduce the file size.

```
<?php

require_once('vendor/autoload.php');

$ffmpeg = FFmpeg\FFmpeg::create();
```

First off, we need to include Composer's autoloader and create a new `FFmpeg` class instance. There's no requirement to differentiate between audio and video on instantiation—the `FFMpeg` class can handle both types of files.

The static method will scan your system paths looking for the required binaries. Your package manager may have put them in a non-standard location, so if they're not found, you can use one or both of the following configuration options to tell the library where to find them.

- **ffmpeg.binaries**: the path to FFmpeg
- **ffprobe.binaries**: the path to FFprobe

```
$audio = $ffmpeg->open('audio.mp3');
$format = new FFmpeg\Format\Audio\Wav();
$format->on('progress', function ($audio, $format, $percentage) {
    printf("%s%% transcoded\n", $percentage);
});

$format->setAudioChannels(1);

$audio->save($format, 'track.wav');
```

With an `FFmpeg` object instantiated, let's convert the file. In the code above, I first open a sample audio file I created, called `audio.mp3`. I then specify a format to convert the file to, in this case, WAV.

Out of the box, php-ffmpeg supports *AAC*, *FLAC*, *MP3*, *OGG-Vorbis*, and *WAV*. To see what formats your installation supports, from the command line, run:

```
ffmpeg -protocols
```

That way if an error occurs, you can more readily make sense of it and install any extensions or codecs, as needed.

With the format object instantiated, I then add a listener on the progress information provided by the format object. Specifically, I'm listening to the progress event, by specifying the name of the event (that is, progress) and a callback to handle the event.

I do this because I want to print out the progress information—it's handy to keep track of how much progress has been made, and gives an indication of how much more time is required. Much preferable to watching a black screen with no indication.

After that, I set the audio channel to 1, which in effect creates a mono audio track. Finally, I save the file, passing in the `$format` object and the file name to save the new file as `track.wav`. In Figure 1, you can see example output, showing the progress as transcoding progresses

php-ffmpeg transcoding output **FIGURE 1**

```
ffmpeg — mattsetter@Matts-Mac-2 — ..gazine/ffmpeg — -zsh
➜  ffmpeg php convert-audio.php
10% transcoded
15% transcoded
21% transcoded
26% transcoded
32% transcoded
37% transcoded
42% transcoded
48% transcoded
53% transcoded
59% transcoded
64% transcoded
69% transcoded
75% transcoded
80% transcoded
86% transcoded
91% transcoded
97% transcoded
99% transcoded
➜  ffmpeg
```

Reduce the Bit Rate

Now that we can transcode a file to another format, what about changing the bit rate? According to the BBC, bit rate[7] is defined as:

> *… how many bits of data are processed every second. Bit rates are usually measured in kilobits per/second (kbps).*

The higher the bit rate, the greater the quality, and size, of the audio file. Conversely, the lower the bit rate, the lower the quality, and the smaller the accompanying file size. You can think of it a lot like working with images for the web. How much quality do you need? How high (or low) a level of quality is acceptable?

This is just as important for audio files, especially given the fact that not all countries have low-cost/high-data plans. And for some of those that do, the infrastructure may not always support streaming large files at the rates we desire.

So to ensure that we consider a broad spectrum of users, we're now going to reduce the bit rate of the audio file we used previously to 64Kbps, which is standard for podcasts. If you're interested in finding the right bit rate for your audio file, check out this post from Richard Farrar[8]. In general, lower bit rates are acceptable for conversations and discussions, while high bit rates will make music and performances sound richer.

To lower the bit rate, we only need to add one further call on the `$format` object, which you can see below.

```
$format->setAudioChannels(1)->setAudioKiloBitrate(64);
```

7 *BBC: Bitrate definition:* http://phpa.me/bbc-bit-rate
8 *Richard Farrar:* http://www.richardfarrar.com/choosing-bit-rates-for-podcasts/

The rest of the code remains the same. Now this won't be a fair comparison of file sizes, as WAV files are a lot larger than equivalent MP3 files.

So to give you an apples-to-apples comparison, I converted an MP3 file with a bit rate of 192 kbps to 64 kbps: the file size dropped from 86mb to 29mb. Not bad, considering that the two files, to the listener, would likely sound almost identical.

Extracting an Image from a Video File

Now that we've played around with audio files, let's work with some video files. We'll ease into it by extracting an image from a video file at an arbitrary point in its timeline.

There are several reasons you may be interested in doing this, including having a thumbnail for the video that the user can see before they start playback. Let's say that's what we're doing. To extract an image we only need the code below.

```
$video = $ffmpeg->open('video.mp4');
$frame = $video->frame(
    FFmpeg\Coordinate\TimeCode::fromSeconds(2)
);
$frame->save('image.jpg');
```

Here, as before, we first open the file. We then call the `frame()` method, passing in a `TimeCode` object, which specifies the time at which to extract the exerpt. Finally, we call `$frame`'s save method, passing in the filename to save the image as `image.jpeg`. Try this with a video of your own, and pick an arbitrary point in the timeline where you want to extract a frame.

Resizing a Video File

Now that we've extracted a frame, let's do something slightly more complex, and scale a video down to a smaller resolution. Let's say that we have a video site, something like Laracasts[9], and we need to create smaller versions of our video for a preview page.

Our original video, the one that will be viewed by users when they're watching the screencast, has a resolution of 1504 x 846, but that the thumbnail video will have a resolution of 320 x 240. To do that, we can use the code sample below.

```
$ffmpeg = FFmpeg\FFmpeg::create();
$video = $ffmpeg->open('video.mp4');
$video
    ->filters()
    ->resize(new FFmpeg\Coordinate\Dimension(320, 240))
    ->synchronize();
$video->save(new FFmpeg\Format\Video\X264(),
        'video-320x240.mp4');
```

Here, we've opened the file, applied a resize filter to it, specifying the dimensions to resize the video to, and then saved it again, with a name containing the dimensions of the file.

Notice that we've also made a call to the `synchronize()` method. While this is not necessary in this example, it's worth mentioning, because it ensures that there's no lag between the video's video and audio content.

9 *Laracasts:* http://www.laracasts.com

Adding a Watermark to a Video File

Continuing with the screencast analogy, let's say that you want to add a watermark to your videos so that it's harder for people to rip off your work and pass it off as their own.

To do that, you could add a watermark like the one in Figure 2, likely at the bottom left or right of the video, showing your name or your company's name.

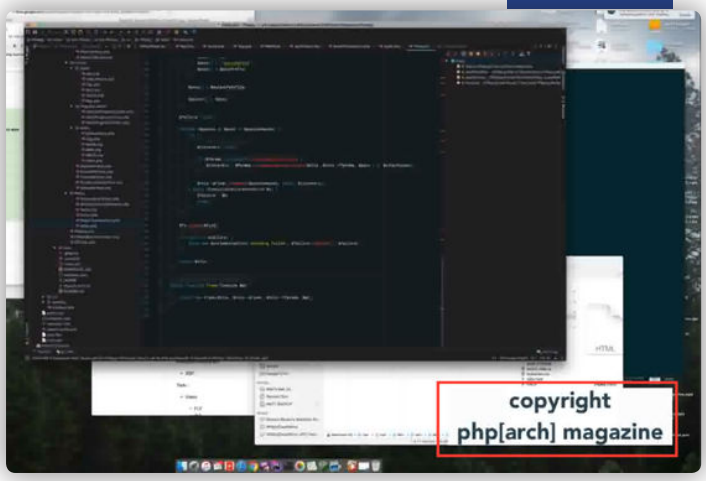
A video watermark **FIGURE 2**

To do this, we can use the same code as before, replacing the call to `resize()` with a call to `watermark()`, as you can see in the code sample below.

```
$video->watermark('./watermark.png', [
    'position' => 'relative',
    'bottom' => 50,
    'right' => 50,
])
```

Here, we've specified the location of the watermark file, along with the details of its placement. When we run the script again, the video will now have a small watermark on it, which you can see in Figure 3.

A watermarked video file **FIGURE 3**



Transcoding a Video File to Several Formats

Let's finish up by seeing how to transcode our video to several different file formats. To do that, we only need to call video's `save()` method for each format that we want to save the file to.

To each call to `save()`, we pass in a video format object and a filename. Currently, php-ffmpeg supports five formats: *OGG*, *WebM*, *WMV*, *WMV3*, and *X264*. Let's take a look at how to save the file to *OGG*, *WMV* and *WebM*.

```
$video
    ->save(new FFmpeg\Format\Video\OGG(), 'video.ogg')
    ->save(new FFmpeg\Format\Video\WMV(), 'video.wmv')
    ->save(new FFmpeg\Format\Video\WebM(), 'video.webm');;
```

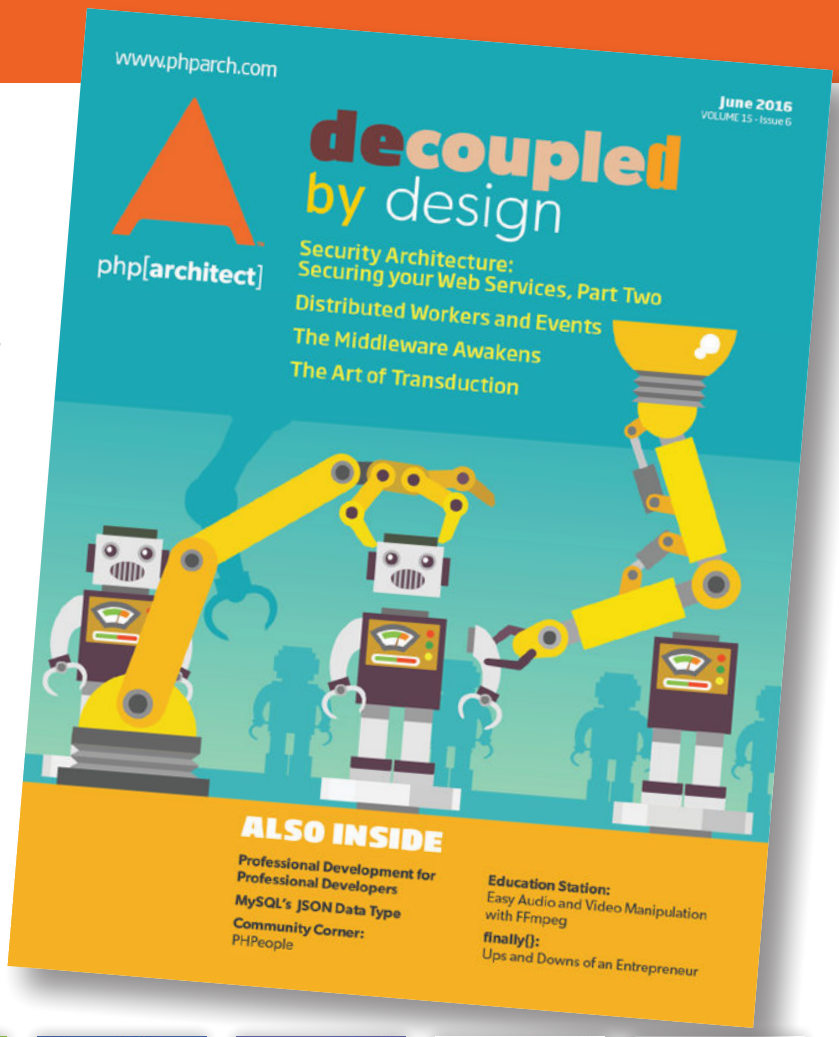
We've already seen this in action, in the earlier example, where we resized the file. But it never hurts to be explicit.

Conclusion

That's your introduction to using the php-ffmpeg library and FFmpeg, for manipulating audio and video content. If you're a podcaster or screencaster like I am, then do yourself a favor and read up on both FFmpeg and php-ffmpeg to see how you can script up various parts of your workflow, saving time and effort and making your work much more consistent.

Matthew Setter is a software developer specializing in PHP, Zend Framework, and JavaScript. He's also the host of http://FreeTheGeek.fm, the podcast about the business of freelancing as a software developer and technical writer, and editor of Master Zend Framework, dedicated to helping you become a Zend Framework master? Find out more http://www.masterzendframework.com.

Want more articles like this one?



Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine
books
conferences
training

www.phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.