



Jeremy Wagner

THE WebP MANUAL

#perfmatters

Imprint

© 2018 Smashing Media AG, Freiburg, Germany

ISBN (PDF): 978-3-945749-67-8

Cover Design: Ricardo Gimenes

eBook Production: Cosima Mielke

Tools: Elja Friedman

Syntax Highlighting: Prism by Lea Verou

Idea & Concept: Smashing Media AG

The WebP Manual was written by Jeremy Wagner.

Table of Contents

Foreword	4
WebP Basics	6
Performance	21
Converting Images To WebP	34
Using WebP Images	62
In Closing	78
Appendix	80
Thanks	83
About The Author	84

Foreword

Performance matters now more than ever. *What* we send over the wire, *how* we send it, and *how much* of it we send matters. Because users matter. Web performance is a vast subject with many nooks and crannies, and all deserve their due attention.

Of serious concern, however, is the role of media in performance, specifically images. Images are powerful. Engaging visuals evoke visceral feelings. They can provide key information and context to articles, or merely add humorous asides. They do *anything* for us that plain text just can't by itself. But when there's *too much imagery*, it can be frustrating for users on slow connections, or run afoul of data plan allowances. In the latter scenario, that can cost users real money. This sort of inadvertent trespass can carry real consequences.

Though images don't block rendering in typical usage like CSS and JavaScript can, they represent a disproportionate portion of a given page's weight. According to [HTTP Archive](http://smashed.by/bytesimg)¹, images comprise approximately 900 Kb of the median web page's total weight. For those users on slower connections and older devices, this can be an insurmountable obstacle between them and your content.

A whole niche area of web performance exists to explain how to best optimize images. We're constantly searching for new ways to transmit as little image data

1. <http://smashed.by/bytesimg>

as possible while also keeping a close eye on quality. While established formats such as JPEG, PNG and GIF are serviceable and highly optimizable, there's an additional image format at our disposal that helps us go a little further. That format is WebP.

In this short ebook, you'll learn all about WebP: what it's capable of, how it performs, how to convert images to the format in a variety of ways, and most importantly, how to *use* it.

WebP Basics

WebP is a relatively recent open source raster image² format first released by Google in 2010, and was derived from the VP8 video codec³. Though not as ubiquitous as established formats such as JPEG, PNG, or GIF which enjoy broad use on the web and browser support, WebP distinguishes itself with better suitability in high-performance web applications. This is chiefly because WebP images usually use less disk space when compared to other formats at reasonably comparable visual similarity.

While WebP doesn't enjoy universal browser support, a majority of internet users use browsers that do⁴. Depending on your site's audience and the browsers they use, an opportunity to deliver less data-intensive user experiences for a significant segment of your audience might exist.

Aside from its suitability in scenarios where performance is vital, WebP excels in its flexibility and bevy of features. WebP takes what's best about established formats and incorporates them into a single format well-suited for web use. Let's kick things off by covering some of those features.

2. <http://smashed.by/rastergraphics>

3. <http://smashed.by/vp8>

4. <http://smashed.by/caniusewebp>

Encoding Features

If you're totally new to WebP, you may not know what to expect. This is understandable, and your first question is likely to be "What format is WebP most like?"

The answer is not entirely straightforward, as WebP is a highly flexible format sporting a number of encoding features. Because of this, WebP resembles both JPEG and PNG in many ways. Let's step through each of these features, and learn what WebP has to offer.

LOSSY ENCODING

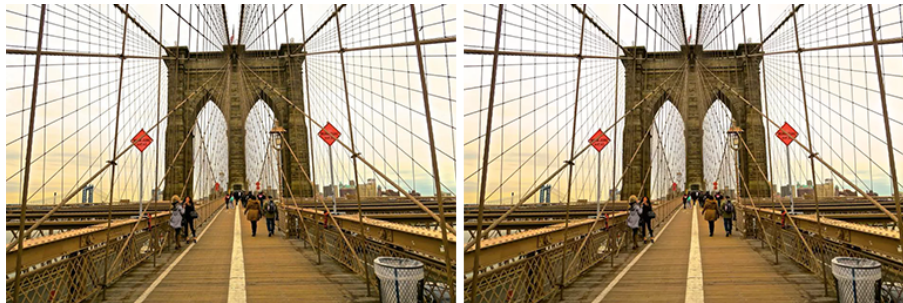
If you've used WebP at all, you might already see it as being somewhat similar to JPEG in that WebP can encode images using lossy compression⁵, which achieves lower file sizes by discarding some amount of image data during encoding. This flavor of WebP is arguably used most frequently, and often outperforms JPEG in output size while retaining reasonably comparable visual similarity.

Lossy WebP permanently discards data during encoding, so it's important to keep in mind that, as is the case with JPEG, the compression is not reversible.

Like JPEG, encoding quality for lossy WebP is expressed as an integer within the 0 to 100 range. The lowest setting of 0 provides the smallest possible file size, and consequently the worst possible visual quality. As you progress from 0 to 100, however, you're increas-

⁵. <http://smashed.by/lossy>

ing file size, but also increasing visual quality. In the figure on the next page, note the higher quality WebP image on the left has better visual quality along with a larger output file size, whereas the opposite is the case with the WebP image on the right.



JPEG (q75)
136.11 KB

Lossy WebP (q67)
94.69 KB

A JPEG encoded at a quality setting of 75 versus a lossy WebP image of similar visual quality (as measured by SSIMULACRA⁶) encoded at a quality setting of 67. The WebP version is roughly 30% smaller. Both images have been encoded from the same lossless PNG source.

If your experience with WebP has only been in passing, then lossy WebP is what you're likely familiar with. Tools that export images to WebP employ lossy encoding by default, including Google's own cwebp command line encoder.

Of course, WebP isn't only capable of lossy encoding. Where image quality is paramount, WebP provides a lossless encoding mode.

⁶. <http://smashed.by/ssimulacra>



Lossy WebP (q95)
36.81 KB



Lossy WebP (q5)
2.91 KB

A high-quality lossy WebP image exported using a quality setting of 95 versus a low-quality lossy WebP image exported using a quality setting of 5.

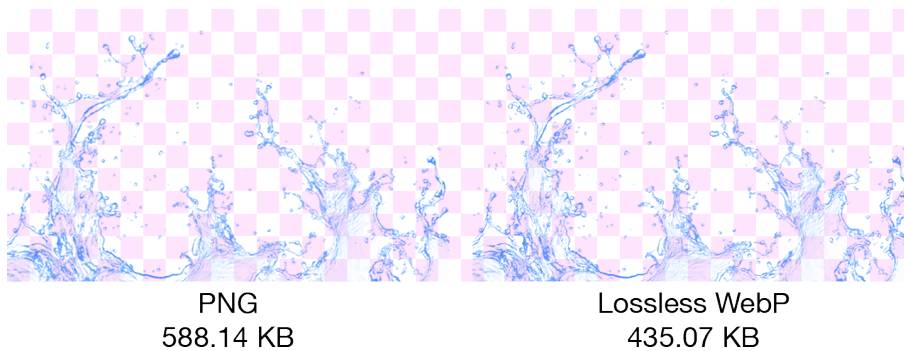
LOSSLESS ENCODING

What makes WebP so flexible is that it isn't strictly limited to one type of encoding. WebP images can also be encoded in lossless fashion⁷. Whereas lossy WebP is most like JPEG, lossless WebP is more like PNG. As is the case with lossless formats, no data is discarded during encoding (barring potential color quantizing⁸ optimizations, which discard colors above a determined threshold to achieve further reductions in file size). Consequently, the compression used is reversible, meaning that lower file sizes are achieved while maintaining the same visual quality as the source image.

In my practical (and admittedly anecdotal) experience, file size reductions achieved by lossless WebP over PNG can be quite significant:

⁷. <http://smashed.by/lossless>

⁸. <http://smashed.by/quantization>



A PNG image (left) compared to a lossless WebP image (right). In this case, lossless WebP offers an approximate 26% reduction in file size without sacrificing image quality.

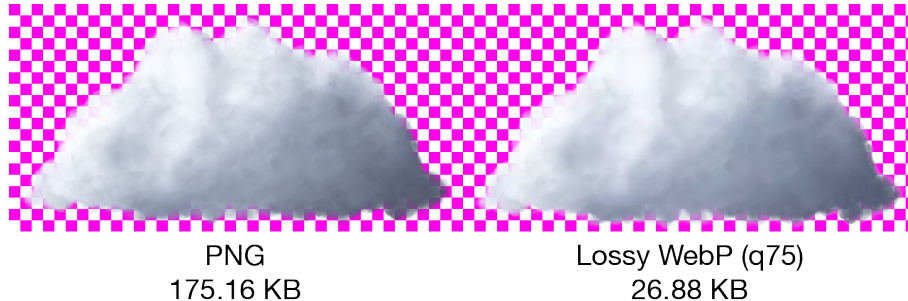
As with full-color PNG, lossless WebP images can be encoded in 24-bit color. Encoding tools can also adjust the compression ratio. Lower compression yields higher file sizes with faster compression speeds. Higher compression levels yield the inverse of this. Some PNG optimizers offer similar compression ranges, but the results of those optimizers don't yield quite the same reduction in file size that lossless WebP often can.

TRANSPARENCY FEATURES

When we think of transparency in raster image formats, it's often in two flavors: 1-bit (Boolean) transparency used in GIFs and 8-bit PNG images, and 8-bit (full) transparency often used in full color PNG images. Those who want transparency features in JPEG are out of luck.

WebP's transparency capabilities are highly flexible. As such, it supports full transparency for both lossless and lossy encoding types. Because of this, you can take 24-bit PNG images with transparency and realize sig-

nificant savings in file size by encoding them in lossy fashion.



Transparent lossless PNG versus transparent lossy WebP at a quality setting of 75. Transparency is retained while reducing file size by roughly 85%.

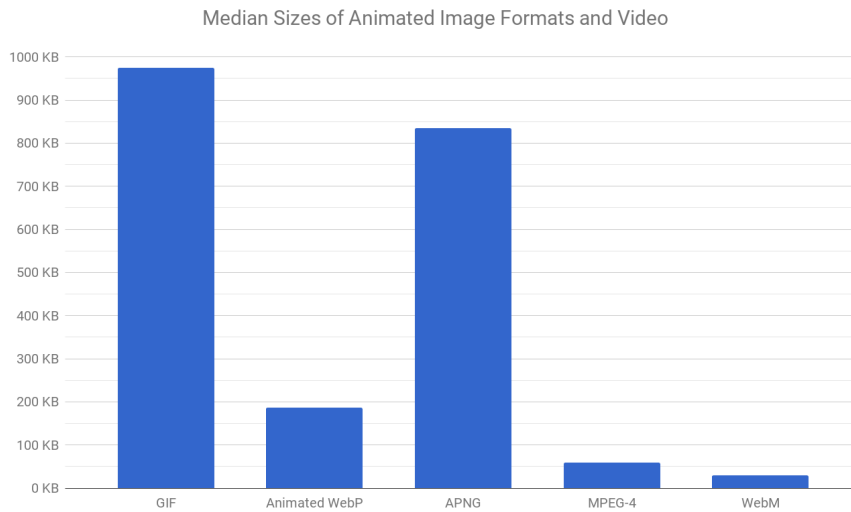
If you take away one thing about WebP's transparency capability, it's that you can have what's best about JPEG (smaller file sizes through lossy compression) *and* have full 8-bit transparency as with PNG. It's the best of both worlds combined in a single, versatile image format.

It should be noted that the `cwebp` command line encoder also allows you to specify a quality setting for the alpha channel itself, lending further optimization opportunities. We'll cover this little tidbit in some detail in the encoding chapter.

ANIMATED WEBP

The short version: Don't use animated image formats unless you must. Use `<video>` instead⁹, as videos tend to be much smaller than animated GIFs.

⁹. <http://smashed.by/replacegifs>



Animated image formats (GIF, WebP and APNG) versus common video formats (MPEG-4 and WebM).

The long version: As with GIF, WebP is capable of animation. As you might guess, this simply means that WebP is capable of storing multiple images in the same file in frames, with viewers displaying frames in sequence. Compared to animated GIF, the file size of a comparable animated WebP can be quite competitive (especially where lossy compression is used). That said, while animated WebP images outperform animated GIFs by a significant margin where file size is concerned, the best vehicle for this type of content is usually video. WebM¹⁰ (a sister project of WebP) is particularly performant in this regard. Below is a comparison graph of three animated image formats and two video formats. The basis for these numbers comes from 15 an-

¹⁰. <https://www.webmproject.org/>

imated GIFs selected at random and converted to various formats for comparison.

If you *must* use an animated image format in lieu of video, animated WebP may be a performant alternative to animated GIF (especially if you employ lossy encoding, or use the `-mixed` option available in the [gif2webp](http://smashed.by/gif2webp) binary¹¹), but also consider Animated PNG (APNG), which is capable of displaying more colors than GIF. Otherwise, just stick with video. If you want embedded videos to exhibit the same behavior as an animated GIF or WebP image, the HTML below should suffice:

```
<video autoplay muted loop playsinline>
  <source src="example.webm" type="video/webm">
  <source src="example.mp4" type="video/mp4">
</video>
```

The key attributes to pay attention to here are `autoplay`, `muted`, `loop`, and `playsinline`, which represent the typical characteristics of animated images:

- automatic playback
- no audio track (muted)
- continuous looping
- inline media element (i.e., doesn't play in fullscreen)

¹¹. <http://smashed.by/gif2webp>

As always, order your sources by the size of each file from smallest to largest. Browsers will pick up whichever video source matches first. WebM is often smallest, but it's not supported everywhere¹². For every other browser, there's MPEG-4¹³.

As a side note, it is possible (at the time of writing) to use MPEG-4 files in `` tags¹⁴, but only in Safari. There is a currently an open ticket¹⁵ for implementation into Chrome, but until this approach is ubiquitous across browsers, it should be avoided in lieu of `<video>` to avoid compatibility issues.

Content Suitability

When it comes to what kind of visual content WebP works best for, use these two rules as a basic guideline:

1. Whatever's a JPEG probably works best as lossy WebP.
2. Whatever's a PNG probably works best as lossless WebP.

Of course, these are not hard and fast rules. You may get acceptable quality by encoding PNGs to lossy WebP, and realize more savings than a PNG to lossless WebP conversion (especially for images with full trans-

¹². <http://smashed.by/caniusewebm>

¹³. <http://smashed.by/caniusempeg4>

¹⁴. <http://smashed.by/animatedgif>

¹⁵. <http://smashed.by/imgsrcbug>

parency). If time allows, experiment a little to see what works best for you.

Beyond these simplistic guidelines, the *type* of visual content can be a good predictor of what encoding will work best. Photographs (or similar content) will work great as lossy WebP images. Line art, logos, illustrations or any imagery with flat, well-delineated shapes, colors and shading will tend to work quite well as lossless WebP images (assuming you don't want to use SVG).

Additionally, be aware that encoding JPEGs (which have already been encoded in lossy fashion) to lossy WebP will suffer further quality degradation. This phenomenon is known as *generation loss*¹⁶. When you re-encode an image that previously has had lossy compression applied to it, visual artifacts present in the lossy source will be compounded. As a consequence, new artifacts will be introduced as the image is re-encoded.

Some encoders (such as `jpeg-recompress`¹⁷) attempt to mitigate recompression issues by using a visual similarity scoring algorithm (for example, the structural similarity index method SSIM¹⁸) to compare output to the source. Even so, some additional quality loss may occur that may be unacceptable to you. It's best to test when you can.

¹⁶. <http://smashed.by/generationloss>

¹⁷. <http://smashed.by/recompress>

¹⁸. <http://smashed.by/similarity>



Generation loss introduced by re-encoding a JPEG several times. The intensity of recompression artifacts increases from left to right.

If quality is of paramount concern, you should always strive to encode lossy WebP images from lossless sources. However, this may not always be practical or even possible, such as when build systems are in place or when lossless or uncompressed sources are unavailable. In any case, you can specify a higher quality setting to minimize recompression artifacts, or check if your specific image encoder has an option that allows you to specify a visual similarity target value. For example, the command line WebP encoder allows you to specify a target similarity using the peak signal-to-noise ratio (PSNR) method¹⁹. While PSNR doesn't account for human perception of image quality quite like SSIM and similar algorithms, it's better than nothing. Finally, if time permits, perform a quick spot check to ensure the encoder's output is up to your standards.

¹⁹. <http://smashed.by/psnr>

Measuring WebP Support within Your Audience

The simplified answer to who in your audience can benefit from WebP is easy: it's whoever uses Chrome (including Chrome for Android). Given Chrome's large market share among browsers (at least at the time of writing), many of your users could benefit from WebP.

A somewhat more nuanced answer includes not only users of Chrome, but also users of Chromium-based browsers, such as Samsung Internet, Opera, and UC Browser. Opera Mini also supports the format, and is more popular in developing nations. There are signals that Firefox may eventually support WebP, but no significant movement has occurred on that front at the time of writing. Similarly, WebP support for Edge is in development²⁰, and could very well land by the time you read this.

Be aware, however, that browsers are anything but static in their adoption (and deprecation) of features and APIs. Browsers change constantly. To get the absolute latest state of WebP support, check out the Can I Use... feature support page²¹.

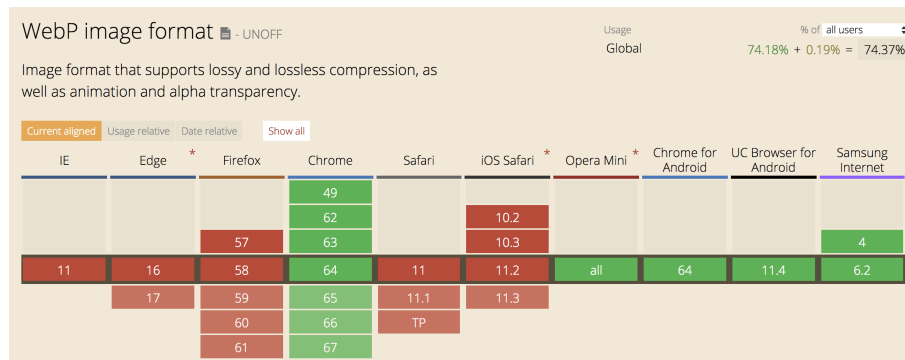
Of course, you *could* look at browser market share statistics and use them as the basis for a loose assumption of how much of your audience supports WebP. Tools such as StatCounter GlobalStats²² do an ad-

²⁰. <http://smashed.by/webpformat>

²¹. <http://smashed.by/caniusewebp>

²². <http://gs.statcounter.com/>

mirable job of providing this kind of data. The problem with relying on statistics such as these, however, is they don't effectively show the makeup of your audience and the browsers *they* use.

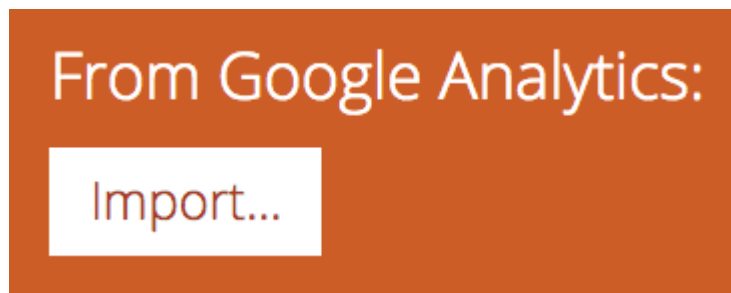


caniuse.com showing the status of WebP support in available browsers.

The most popular tool for gathering data on your visitors is Google Analytics²³. This tool alone can tell you everything you'd need to know about your users, including the browsers they use. Unfortunately, it doesn't translate those browsers into a segmented list of those in your audience who can use WebP. It's only when we feed analytics data into caniuse.com, however, that we actually get to see what WebP support looks like for a specific audience.

In the *Can I Use...* app, simply toggle the settings panel, which is found near the search bar at the top of the page. Once that panel is opened, you'll see a small prompt to import data from Google Analytics.

²³. <https://www.google.com/analytics/>



caniuse.com's prompt in the settings panel to import visitor data from Google Analytics.

After you import your site data, enter “WebP” into the search bar at the top. You’ll not only see the global support view in the statistics, but also an audience support view that shows what support for WebP (or any other browser feature you care to search for!) looks like for your site’s visitors.

Usage	% of all users		
Global	74.18%	+ 0.19%	= 74.37%
All Web Site Data	43.25%	+ 0.14%	= 43.39%

Can I Use... showing feature support for a specific audience from Google Analytics data, labeled “All Web Site Data.”

When the data is imported, a new data point will show up in the upper right-hand corner of the feature support table showing the degree of support for a particular feature for your site’s audience. When caniuse.com and Google Analytics are used together, you can make an educated decision on whether it’s worth the effort to serve alternate WebP images on your site. Should you

decide this is the correct course of action, you'll want to read the "Converting Images to WebP" and "Using WebP Images" sections to get started.

Before we touch on how to encode WebP images, however, let's first cover how the format performs in terms of encoding/decoding speed, output size, and visual similarity to established formats.

Performance

You've learned about WebP's features, and while that might answer questions you have about its suitability for your sites and applications, you likely want to know how it performs compared to JPEG and PNG. To get a reasonably broad picture of WebP performance, comparisons were performed on a large set of images, with statistics averaged where appropriate. To find out more about how the testing was done, check out the appendix.

File Size

The most attractive feature of WebP is its ability to outperform established formats in the realm file size. After all, reducing the amount of bytes transferred over the wire (while retaining a reasonable degree of quality, of course) is the primary goal of any image optimization effort. To this end, WebP performs well.

In this section, we'll cover how both lossy and lossless WebP compare to JPEGs and PNGs exported by a number of image encoders. If your experience of exporting images is limited to imaging software, you might not be aware that there are a number of command line encoders that each export JPEGs and PNGs in different (and sometimes novel) ways. Each of these encoders performs differently than others. As such, we'll compare the output from a handful of encoders to WebP to gain a broader understanding of the format's performance.

LOSSY WEBP

Lossy WebP is probably what you'll use most often, as it tends to be the default setting for the official WebP encoder (cwebp) as well as interfaces that implement it. Let's examine how lossy WebP output file sizes compare to the output of a few JPEG encoders:



Output file sizes of popular JPEG encoders versus lossy WebP.

In this graph, three JPEG encoders are represented, with both progressive and baseline modes for mozjpeg²⁴ and cjpeg plotted. Since Google's Guetzli encoder²⁵ doesn't export JPEGs lower than a quality of 84, its output performance is only represented from 84–100 on the quality scale. Additionally, Guetzli only outputs JPEGs in baseline mode.

²⁴. <http://smashed.by/mozjpeg>

²⁵. <http://smashed.by/guetzli>

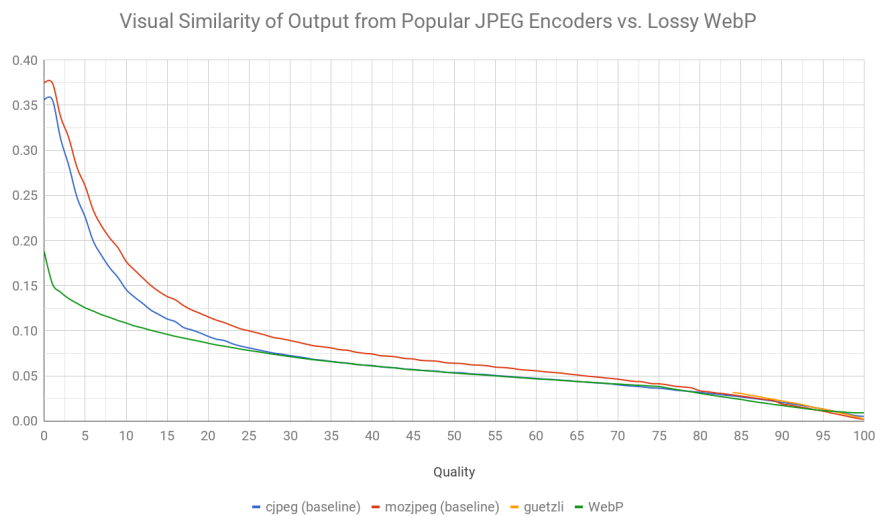
What we see in the graph is that some encoders at the lowest end of the quality spectrum slightly outperform WebP. However, as quality increases into usable ranges, WebP begins to outperform other encoders. In our image set, this begins to happen at quality levels in the mid 60s. Most JPEG encoders are outdone by at least 16 kilobytes starting at a quality range of 80 to 90, which is not necessarily an insignificant improvement. As quality approaches 95, gains were at least 32 KB over at least Guetzli. Guetzli competes well at lower quality settings, but output size begins to skyrocket at quality settings higher than 95, as is the case with other JPEG encoders. Truthfully, you're not likely to encode at quality settings higher than 95, but WebP does *extremely* well in this quality range.

It's worth noting that while Guetzli performs well among JPEG encoders, it takes a *long* time to encode images, as its similarity scoring algorithm²⁶ is very CPU-intensive. Still, if you're not encoding images on the fly (which you should avoid anyway), this shouldn't be a problem for your users. It just means you'll need to be more patient at encode time, and that converting very large sets of images could take several hours.

Bear in mind, though, that it's not enough to compare output file size. You should take visual similarity into account. A reduction in file size from JPEG to WebP may not be acceptable if visual similarity is low (that is, too degraded). Because WebP's lossy encoding

²⁶. <http://smashed.by/butteraugli>

algorithm is different from that of JPEG encoders, its output at a given quality setting may not be directly comparable to that of a JPEG at the same quality setting. In other words, a WebP at a quality of 75 may differ substantially from a JPEG at a quality of 75.



Visual similarity of various JPEG encoders and lossy WebP.

To get an idea of how different JPEG encoders compare visually to WebP, we leaned on SSIMULACRA²⁷, an image similarity scoring program developed by Cloudinary²⁸. SSIMULACRA's algorithm improves on similar visual similarity comparison methods (such as SSIM) by focusing on how people perceive images, as opposed to simply calculating differences between pixel values within two similar images. When SSIMULACRA calcu-

²⁷. <http://smashed.by/psychovisual>

²⁸. <https://cloudinary.com/>

lates the difference between two images, it returns a score. Lower scores indicate high similarity, whereas higher scores indicate low similarity.

To calculate visual similarity, we once again compared the output of the same JPEG encoders, and plotted their similarity scores against WebP:

At lower quality ranges, WebP has better similarity scores than both cjpeg and mozjpeg, but as quality increases, WebP starts to fall in line with other JPEG encoders. For the set of images this data was generated from, this means that WebP's quality is generally comparable with JPEG. This signifies that in most cases, the format should offer some benefit in file size gains in typical quality ranges while retaining reasonably good visual similarity to lossless sources.

That said, it's not always possible to encode from lossless sources (for reasons we'll cover soon). To that end, I also wanted to see how WebP performed with other formats when recompressed from lossy JPEG sources.

RECOMPRESSING FROM LOSSY SOURCES

While it's ideal from a quality standpoint to encode images from lossless sources, that's not always a choice you'll have. Sometimes you don't have access to design documents or other lossless sources, and must make do with whatever is available. In times like these, it's not uncommon to recompress lossy images to achieve further savings in file size. This is especially true in automated build systems (gulp, Grunt, and so on) where im-

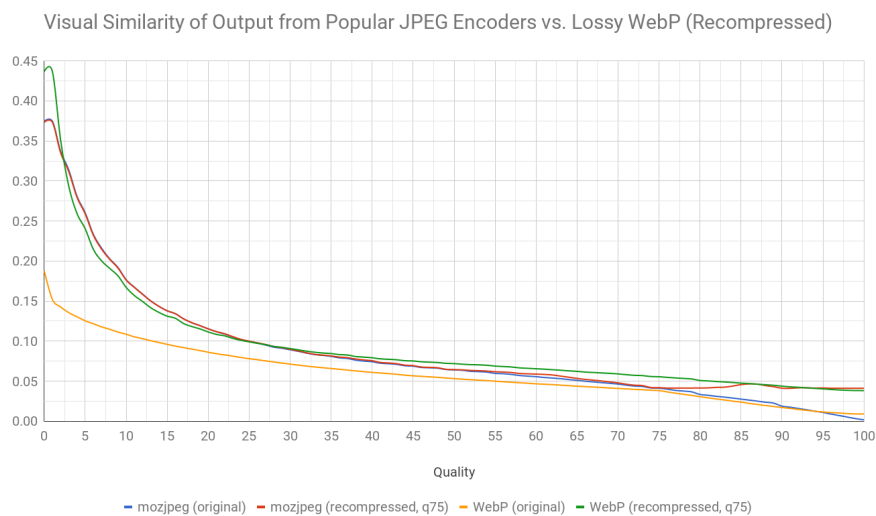
ages in a source directory are uniformly optimized and written to a destination directory.

To figure out the impact of lossy recompression on file size and visual similarity, I recompressed JPEGs in the research image set from the various JPEG encoders represented in the initial output size graph at mozjpeg's default quality setting of 75. I also did the same for lossy WebP images at the same quality setting of 75. The result looked something like this:



Comparisons of mozjpeg and lossy WebP images both compressed from lossless sources, and recompressed from lossy sources.

What we can see in this graph is not too surprising: recompressing lossy images yields even further file size reduction for both JPEG and WebP, but WebP still wins out by a bit. But how much do these recompressed images resemble their lossy sources?



Visual similarity of compressed and recompressed JPEG and lossy WebP images to their lossless sources.

Unsurprisingly, we see that recompressed images in our image set are less similar to their lossless sources for most of the quality range, especially as we approach the higher end of the quality spectrum. But let's put this into perspective: SSIMULACRA's command line help text states that if a "value is above 0.1 (or so)" then "the distortion is likely to be perceptible/annoying"; and if a "value is below 0.01 (or so)" then "the distortion is likely to be imperceptible." Given that images in typical quality ranges are well within the 0.01 to 0.1 range, it's safe to assume that a measure of recompression may be acceptable. Thus, WebP should afford some savings in file size without a hugely significant degradation in perceptible visual quality (at least within the set of images this data draws on anyway).

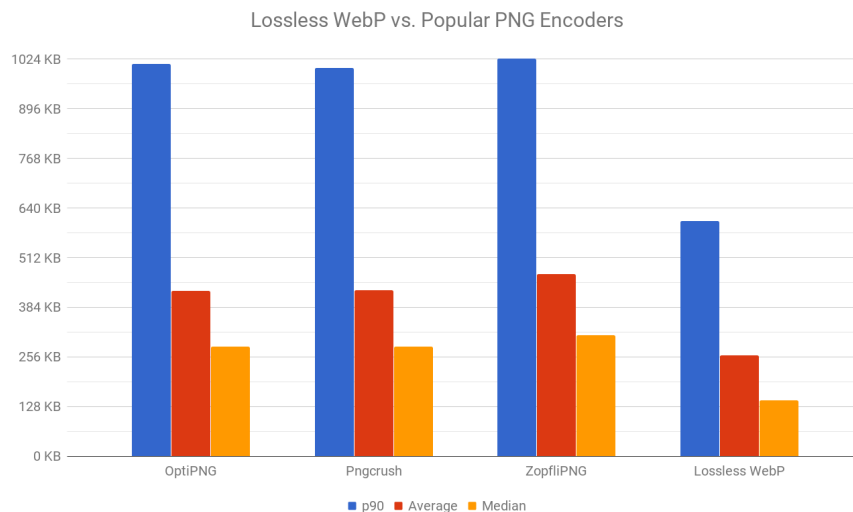
It's important to also note that the nature of how people perceive images on the web is subjective. We're

used to having high quality sources to compare against, but our users don't have that ability. As always, though, verify visual quality for yourself when possible.

With the performance benefits of lossy WebP understood, let's move on and cover how lossless WebP competes with PNG.

LOSSLESS WEBP

As I pointed out earlier, WebP is also capable of encoding images in lossless format. WebP excels at delivering lower file sizes when compared to PNG, even when those PNGs have been optimized. Let's take a quick look at a graph of various PNG optimizers versus lossless WebP.



Output file size of various PNG encoders versus lossless WebP.

In this test, the 90th percentile, average, and median output file sizes are measured across 50 PNG samples.

The images encoded in the WebP test were taken from the highest performing optimizer, which was OptiPNG. As you can see, lossless WebP does extremely well. All PNG optimizers do a good job of reducing image sizes relative to their source, but lossless WebP goes a step further, and shaves off even more. Because the optimizations are lossless, these images are visually identical to their PNG counterparts. There's no risk of sacrificing visual quality to achieve lower file sizes. If you serve a lot of PNG images and want to reduce the amount of data you send to your users, seriously consider lossless WebP as an alternative format.

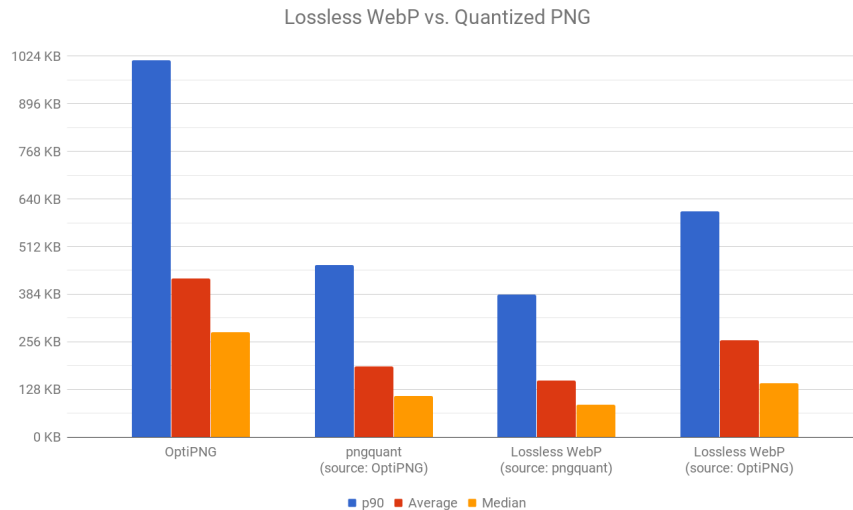
That said, the optimizers in this test did not use color quantization. Color quantization²⁹ is when colors are removed from an image at a specified threshold, or in a selective fashion as to be less noticeable to the eye. Quantizing optimizers (such as pngquant³⁰) excel at reducing file size, but the output may not be suitable for all applications. While these optimizers don't discard visual information the same way JPEG or lossy WebP do, they still discard *some* data to achieve further reductions in file size. If your goal is to export images that are visually identical to their lossless sources, you'll want to avoid color quantization.

If you *do* find quantizing optimization acceptable, be aware that you can take the optimizations they provide even further by re-encoding the quantized output into

²⁹. <http://smashed.by/quantization>

³⁰. <https://pngquant.org/>

lossless WebP. The following graph shows what kind of gains you may realize.



Output sizes of OptiPNG and OptiPNG-derived lossless WebP versus quantized PNG and pngquant-derived lossless WebP.

If you're cool with quantizing your PNGs, you can clearly see that there's further benefit to be had in converting them to lossless WebP.

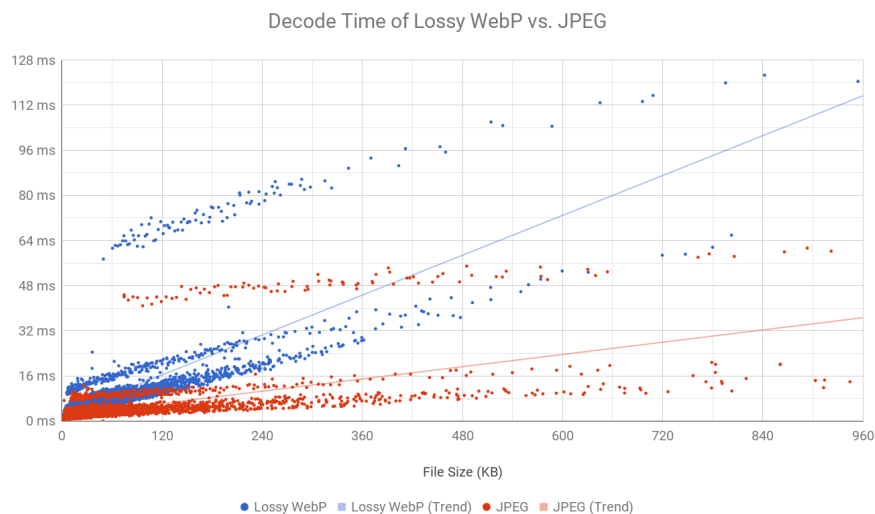
Now that we've covered the file size-related performance aspects of the WebP format, let's focus a bit on decoding speed which, while not as important a performance factor as file size, is still something that deserves attention.

Decode Time

Image decode time is not something you often think about, but it is an aspect of performance worth some minor attention. When *any* kind of resource finishes

downloading, be it CSS, JavaScript or images, the browser's job doesn't stop there. In the case of image resources, the browser still needs to decode the image before it can display it to the browser window.

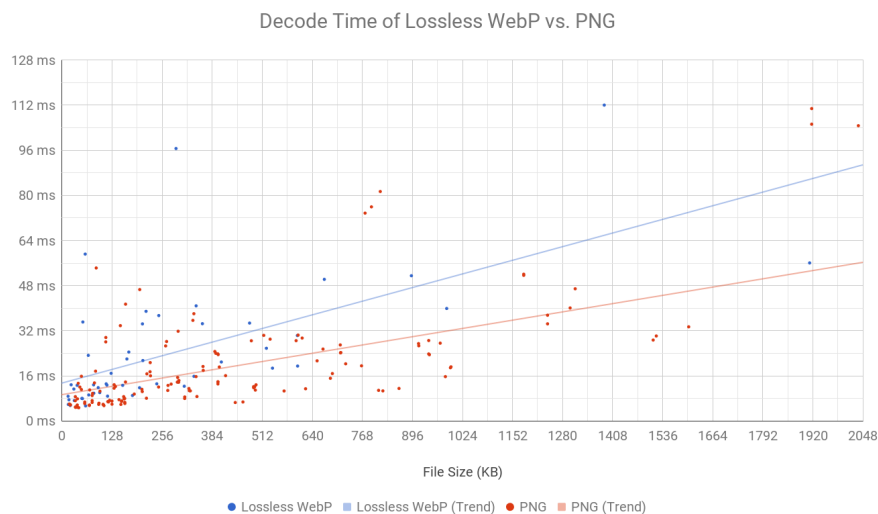
Established formats such as JPEG and PNG have had a long history of development. They've both benefited from many years of research and optimization. By comparison, WebP has had less time in development. As such, WebP's decoding speed is not *quite* as fast as these two established formats. First, let's take a look at average decode times for JPEG versus WebP:



Average decode times across various file sizes for JPEG versus lossy WebP.

As you can see here, WebP takes somewhat longer to decode at all quality levels as compared to JPEG. As quality level increases, WebP's decoding performance falls behind that of JPEG. Here's my personal take on this: even though this test was performed on modern,

high-end hardware, decode time for either WebP or JPEG is often not the deciding factor on which image is painted to the browser window first. In most situations, *especially* where latency is high and bandwidth is low, a smaller image is going to win out over a much larger one. Lossy WebP has some very high decode times at the highest end of the file size spectrum, but those are for *extremely* large files you would be unlikely to see on most web pages. Now let's tip the hand to lossless WebP and look at its decoding performance compared to PNG.



Average decode times across various file sizes for PNG versus lossless WebP.

As you might expect, lossless WebP's performance against PNG is similar to that of lossy WebP versus JPEG. WebP often takes longer to decode, especially in extreme cases where file sizes are quite large. Relative

to other formats, though, the time is not significantly greater.

In short, don't sweat decode time too much. Smaller images decode faster than larger ones, and according to HTTP Archive data³¹ queryable from BigQuery, the median and average image sizes on the web are quite small:

Format	Number of Requests	Median Size	Average Size	p90 Size
JPEG	9,153,030	19.06 Kb	57.54 Kb	105.09 Kb
PNG	3,284,588	16.26 Kb	69.14 Kb	134.15 Kb
WebP	128,766	17.91 Kb	33.87 Kb	59.69 Kb

Furthermore, in scenarios where bandwidth or data plan allotments are of high paramount concern, any potential performance issues that *could* arise with image decode time are always second to conserving bandwidth.

That was quite a bit of noodling around with performance numbers, but I believe it makes a nice case for WebP's suitability in high-performance web sites. Let's continue and cover the variety of ways you can convert your existing images to WebP!

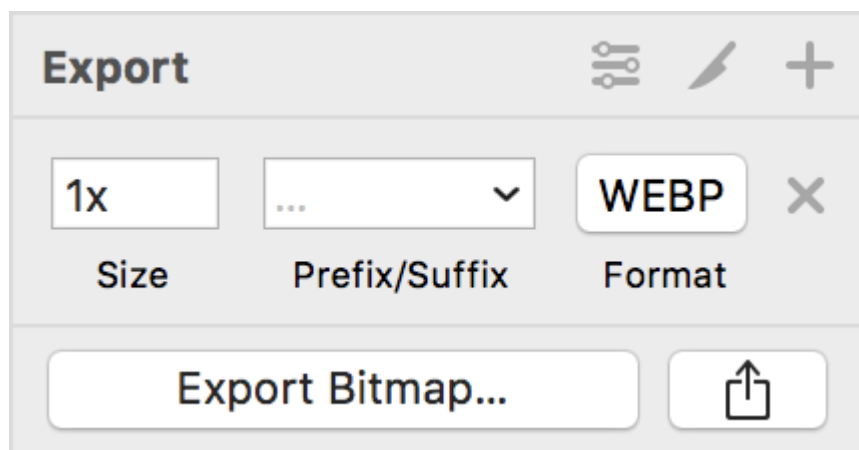
³¹. <http://smashed.by/httparchivedata>

Converting Images To WebP

To use WebP, you'll first need to convert your existing images to the format. This can be done in a myriad of ways, from something as simple as exporting from your preferred design program, to cloud services, to the official cwebp encoder, and even in Node.js-based build systems. Here, we'll cover all avenues.

Sketch

Sketch is able to export any resource in a design document to WebP natively. To export an image to WebP, select a resource on the canvas, open the **Export** panel on the right hand side, and choose “WEBP” in the format dropdown.



The resource export panel in Sketch, with the WebP format chosen in the format dropdown.

After you make your selection, click the **Export Bitmap...** button. The resulting dialog will predictably

ask where you want the image to be exported to. Within that dialog, a slider will appear at the bottom that prompts you to specify the quality of the WebP image from 0 to 100, implying the output is lossy WebP.



The quality slider in the export dialog when exporting a resource to WebP.

If you're using design software to export to WebP, Sketch is probably one of the easier programs to use. Although, there *are* other programs capable of doing this, such as Photoshop.

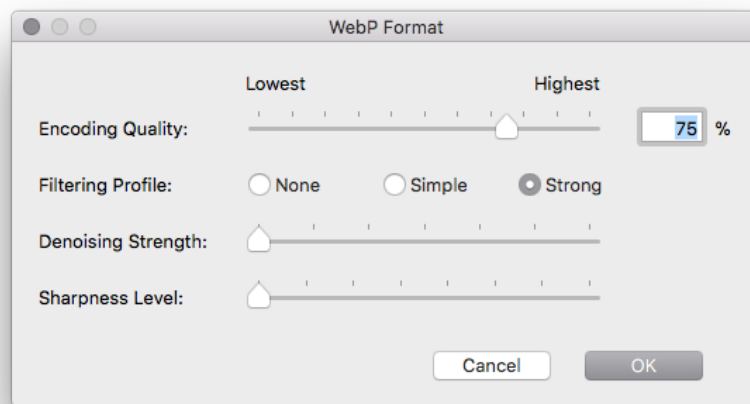
Photoshop

Exporting images to WebP in Photoshop is possible, but is not as convenient as in Sketch. You'll need to rely on a plug-in to get the job done. The good news, however, is that the Photoshop plug-in does give you a bit more flexibility than Sketch does. To obtain the Photoshop plug-in for exporting WebP images, visit the [Telegraphics site](http://smashed.by/telegraphics)³² and grab the version for your system. Once installed, start Photoshop and open an image. Once opened, you can export an image to WebP through the **Save As...** dialog. At the bottom of the dialog where you

³². <http://smashed.by/telegraphics>

choose a format, you'll notice two options: WebP, and WebP Lossless.

What happens from here depends on what you choose in the dropdown. If you choose WebP Lossless, the file will be exported, and that will be that. If you choose WebP, however, you'll be presented with a dialog with several configuration options:



The Photoshop plug-in's lossy WebP export dialog.

In most cases, you'll merely adjust the encoding quality, but feel free to experiment with the filtering, denoising, and sharpness options to obtain the desired result. Unfortunately, this plug-in lacks the ability to show a preview of the image before you save it. If you're accustomed to using the **Save for Web** tool, this is kind of a bummer, as you'll have to go at it blind.

Of course, if you're not a fan of tinkering around in imaging software, the easiest possible option for using

WebP might just be to rely on an image optimization CDN. Cloudinary is one such service.

Cloudinary

Using WebP is not a frictionless experience, and the easiest way to use it is to never have to convert to the format on your own in the first place. Some online services can do this work for you, and Cloudinary is one of them.

You only need to upload images to their service. From there, Cloudinary will optimize your images for you. These optimizations are highly customizable, and one such optimization is to automatically serve whatever image format is optimal for your users. If you use Cloudinary to serve your site's images and your visitors are using WebP-capable browsers, Cloudinary takes on the hassle of serving WebP images for you, provided you choose the proper URL parameters. When you upload an image, Cloudinary's control panel will provide a URL to your image that looks something like this:

```
https://res.cloudinary.com/drp9iwjqz/image/upload/  
v1508291830/jeremywagner.me/using-webp-images/  
tacos-2x.jpg
```

Using any number of URL parameters, you can change how Cloudinary serves images to you, including serving WebP images automatically to clients that can best handle them. The parameter that controls this behavior is *f_auto*, which tells Cloudinary to automatically pick

the best format for the client issuing the request. Parameters are added after the `/upload/` portion in the URL like so:

```
https://res.cloudinary.com/drpg9iwjqz/image/upload/f_auto/  
v1508291830/jeremywagner.me/using-webp-images/  
tacos-2x.jpg
```

Though Cloudinary preserves the extension, you can tell what the content type of the image is by looking at the image's Content-Type response header. If you're using a browser that supports WebP, that header will have a value of `image/webp`. Of course, Cloudinary is capable of more than simply serving the best format for a given browser. You can combine multiple parameters by separating them with commas. For example, here's how you could tell Cloudinary to automatically pick the best format *and* the best quality setting (represented by `q_auto`):

```
https://res.cloudinary.com/drpg9iwjqz/image/upload/  
f_auto,q_auto/v1508291830/jeremywagner.me/using-webp-im-  
ages/tacos-2x.jpg
```

To learn more about what Cloudinary is capable of, check out their [documentation](https://cloudinary.com/documentation)³³.

While Cloudinary is a convenient tool that does the work of image optimization for you, you may yet feel

³³ <https://cloudinary.com/documentation>

compelled to encode your own images. And there are plenty of good reasons to do so! Beyond using imaging software, you can accomplish this with perhaps the most flexibility by using the Google's official WebP command line encoder.

The Official cwebp Command Line Encoder

The official tool for encoding images to the WebP format is Google's own command line utility. While a command line program may not be as easy to use as a graphical interface, it offers much more flexibility in controlling the output. Most graphical front-ends that export to WebP abstract away features that don't neatly fit into a dialog, and thus aren't the best tools for achieving the best result for your site. Before you can use the command line encoder, however, you'll need to install it.

INSTALLING

Your installation method will depend on your operating system. The easiest way to install the WebP encoder will be through your operating system's package manager. macOS users can install the encoder and related tools using the Homebrew³⁴ package manager:

```
brew install webp
```

³⁴. <https://brew.sh/>

Windows users can install the encoder using Chocolatey³⁵:

```
choco install webp
```

Users of Red Hat and Red Hat-derived Linux distros such as CentOS or Fedora can use yum to install the encoder:

```
yum install libwebp-tools
```

Other platforms may use different package managers (such as apt-get for Debian Linux), but the mere presence of a package manager doesn't mean it will provide a way to install the encoder. If your operating system package manager somehow fails you, the WebP encoder is available via Node.js's own package manager (npm):

```
npm install -g cwebp-bin
```

If *none* of these options work for you, you can always download the source³⁶ and compile your own binaries. It's a bit onerous, but the option is there.

With the encoder installed, let's take a look at some quick examples of how you can use it.

³⁵. <https://chocolatey.org/>

³⁶. <http://smashed.by/libwebp>

SIMPLE COMMAND LINE EXAMPLES

The WebP encoder can be invoked with the `cwebp` command. For starters, let's examine what is arguably the most common use case: converting an image to lossy WebP:

```
cwebp -q 75 source.png -o output.webp
```

This command takes a PNG image and converts it to lossy WebP by way of the `-o` parameter. By default, the encoder outputs lossy WebP images, and the quality of the output can be set from 0 to 100 via the `-q` parameter. The default lossy quality setting is 75.

You may remember reading earlier that WebP is capable of full alpha transparency, even for lossy images. You can control the quality of this transparency in the same fashion via the `-alpha_q` parameter:

```
cwebp -q 75 -alpha_q 10 source.png -o output.web
```

`-alpha_q` applies lossy compression to the transparency in an image. The default for `-alpha_q` is 100, which is lossless. You can further reduce output size by lowering this value, which will apply lossy compression to the transparency, but lowering it *too* much can significantly degrade the quality of transparent regions in the image.

If you're conservative in adjusting `-alpha_q`, you can reduce the size of transparent images even further, but be sure to examine the output to ensure it's acceptable to you. If you're automating conversion of images

(through a shell script, for example), you might want to shy away from setting this parameter at all. It's also worth noting that `-alpha_q` has no effect when encoding lossless WebP images.



A lossy transparent WebP with best transparency quality (left), and a lossy transparent WebP with low transparency quality (right). Note the loss of fine details at the cloud's edges.

We've discussed encoding lossy WebP images with `cwebp`, but what if you want to export to lossless WebP? Just use the `-lossless` option:

```
cwebp -lossless source.png -o output.webp
```

Depending on image content, you may not realize as much of a reduction in output file size when compared to lossy WebP. You *can* control the aggressiveness of the compression via the `-z` parameter, though:

```
cwebp -lossless -z 9 source.png -o output.webp
```

The `-z` parameter accepts a value between 0 (no compression) and 9 (most compression). Higher compression yields lower file sizes, but requires more time to encode images. You can also set the `-m` parameter to in-

fluence output size, which specifies a compression method from 0 to 6:

```
cwebp -lossless -m 6 -z 9 source.png -o output.webp
```

You can also use the `-q` parameter to tell cwebp how aggressively to compress the image, which can be slightly confusing because `-q` means something entirely different for lossless WebP than it does for lossy images. For lossless WebP, `-q 100` applies the most compression, whereas `-q 0` applies the least. You can use `-q` in tandem with the `-m` and `-z` parameters to achieve very high compression, like so:

```
cwebp -lossless -m 6 -z 9 -q 100 source.png -o  
output.webp
```

This level of compression takes by far the longest, but it can go quite a ways further in reducing your lossless image payloads. Experiment to find what works best for you. You might shave off more kilobytes than you thought possible!

Of course, these are just some quick examples of what's possible. cwebp is an amazingly flexible encoder. To get the full list of available options, type `cwebp -longhelp` and poke around.

There's a good chance you'll need to convert a whole bunch of images at once. Next, I'll show you a couple ways you can convert multiple files with short commands for Unix-like systems on bash.

BULK CONVERSION IN BASH

If you're using Bash on a Unix-like operating system such as macOS or Ubuntu, and you have a directory of images you need to convert to WebP, the `find` command does an excellent job. As the name would imply, `find` finds objects in the file system. The syntax for finding files by a specific extension is very straightforward:

```
find ./ -type f -name '*.png'
```

In case you're not familiar with how `find` works, let's step through this example command.

1. The first argument is the file system path to search, which in this case, is the current directory.
2. The second argument specifies the type of file system object. You can find directories with `-type d`. We're finding files in this case, however, which means we're going with `-type f`.
3. Lastly, the `-name` argument is the pattern by which to find files. In this instance, we're finding all files ending in `.png`. When we run this command in a directory tree containing PNG files, the output might look something like this:

```
./sources/5-1024w.png  
./sources/14-768w.png  
./sources/old/15-768w.png
```

This might not seem very useful by itself, but here's where the magic happens: you can take the files found by `find` and redirect them as input to any other program you want via the `-exec` parameter. Here's how you could use `-exec` with `cwebp` to encode all PNGs in a subtree to lossy WebP images at a quality setting of 75:

```
find ./ -type f -name '*.png' -exec sh -c 'cwebp -q
75 $1 -o "${1%.png}.webp"' _ {} \;
```

This may look a bit convoluted, but let's step through what's going on so we can better understand it together:

1. The initial part of the command finds all files ending with a `.png` extension in the current directory (and sub-directories), just as we described earlier.
2. Files found are sent to the `-exec` parameter, which invokes an instance of the `sh` shell. The `-c` parameter accepts a command to run.
3. The command invoked is the `cwebp` encoder. The `-q` argument is the quality setting.
4. The `$1` placeholder represents the file found by `find` as an argument passed into `sh`. The output file name is the `-o "${1%.png}.webp"` portion. The pattern expressed replaces the `.png` extension with a `.webp` extension.

5. The final part passes the reference to the file found by `find` (represented by `{}`) to `sh`, and the entire command is terminated by `\;`.

While this is admittedly a bit unwieldy, it's quite useful when you need to convert a number of images to WebP quickly. When invoking `cwebp` in this fashion, feel free to adjust parameters as necessary to achieve the desired results. Just be aware this example command dumps the converted images into the same folder as the source images, so adjust as needed to control the location of the output.

The only drawback of this approach is it may take a long time to finish if you're processing a large number of files, since images are processed one after the other rather than concurrently. If you need to speed up processing a bit, you might consider parallelizing image processing, which we'll talk about next.

CONCURRENT BULK CONVERSION IN BASH

Here's a potential scenario. You have hundreds, perhaps thousands of images you need to convert to WebP.

While `cwebp` is reasonably fast, it can still take a very long time if you're not processing images concurrently. Serialized processing doesn't make the best use of your CPU's potential like concurrent processing does. To get around this, you can augment the earlier command used for serialized batch conversion with `xargs` like so:

```
find ./ -type f -name '*.png' | xargs -P 8 -I {} sh
-c 'cwebp -q 75 $1 -o "${1%.png}.webp"' _ {} \;
```

This isn't much different than the command from earlier, except with some key differences:

1. The `-exec` parameter is notably absent. Instead, the output from `find` is piped directly to `xargs`, which handles processing in lieu of `-exec`.
2. The `xargs` command immediately follows the `|` character with three specific parts. The first is the `-P` parameter, which specifies the maximum amount of concurrent processes (8 in this example). The second is the `-I` parameter, which is the input `xargs` acts on (the value of which is the file found by `find`, represented by the `{}` placeholder). The final argument is the command `xargs` will execute, which is exactly the same as in the serialized bulk conversion command from before.

If you want to increase the amount of concurrent processes, simply change the value you pass to the `-P` parameter. Unlike other asynchronous approaches, `xargs` allows you to throttle concurrency to a specified maximum so you don't render your system unresponsive. While this approach may only shave off a few seconds for small batches of images, it shines when operating on very large ones. For example, when converting a batch of roughly 10,000 JPEGs to WebP, a serialized approach using only `find` took 21 minutes and 26 seconds, whereas a concurrent approach using *both* `find`

and `xargs` took **9 minutes and 8 seconds**. This is approximately a **57%** decrease in processing time, and with a relatively conservative concurrency of 8 processes at a time. If you can afford to bump up concurrency, you may realize further reductions in processing time.

Of course, these commands aren't limited to merely converting images to WebP. Whatever shell commands you can think of that will work with `find` and `xargs` will be just as serviceable. Experiment and see what you can pull off.

If converting images to WebP this way isn't to your liking, maybe you'd prefer to accomplish the task in JavaScript. Next, we'll talk about how to convert your images to WebP using Node, as well as within the various build systems available in the Node.js ecosystem.

Convert Images to WebP with Node.js

Node.js's only use isn't just a JavaScript application stack. Even in environments where JavaScript isn't used in an application back-end, Node.js still proves useful as a build tool. The tool used for exporting images to WebP in Node.js or any Node.js-based build system is `imagemin`. `Imagemin` is a tool that converts and optimizes images of all formats, but it can be extended to support WebP conversion. Whether you're writing scripts to run with Node.js, or using one of many Node.js-based build systems (`gulp`, et al.), `imagemin` is the ticket. Let's get started by demonstrating how you can convert images to WebP in a simple Node.js script.

USING A NODE.JS SCRIPT

Perhaps you're not the type to reach for an opinionated build system first, and prefer to use Node.js scripts instead. That's reasonable enough, and if you know how to write JavaScript, the learning curve isn't very steep, since you don't need to learn the syntax of a build system. To get started on converting images to WebP in Node.js, install the `imagemin` and `imagemin-webp` modules in your project root directory:

```
npm install imagemin imagemin-webp
```

This command installs two modules: the first is `imagemin` itself, and the second is the `imagemin-webp` plugin that extends `imagemin` so it can convert images to WebP. With these two modules installed locally in our project, we can then write a small script that will process JPEG images in a directory and convert them to WebP:

```
const imagemin = require("imagemin");
const webp = require("imagemin-webp");

imagemin(["sources/*.png"], "images", {
  use: [
    webp({
      quality: 75
    })
  ]
}).then(function() {
```

```
console.log("Images converted!");
});
```

This short script imports the `imagemin` and `imagemin-webp` modules into the current script as two constants (`imagemin` and `webp`, respectively). We then run `imagemin`'s main method, which takes three arguments:

1. The location of the source images. In this case, we're converting all files ending in `.png` in the *sources* directory, which is assumed to be located in the same directory as the script itself.
2. The destination directory, which in this case is *images*. This directory is created if it doesn't already exist.
3. An options object for the `imagemin` program. In this case, we're specifying the `use` option, which allows us to pass plug-ins into `imagemin`. The only plug-in we're using here is an instance of `imagemin-webp` (represented by the `webp` constant). The plug-in itself also accepts a configuration object, which we have used to specify that we want all PNGs converted to lossy WebP (implied, as lossy encoding is the default) with a `quality` setting of 75.

`Imagemin` will convert images for us, and when it's finished, it will return a `Promise`³⁷. In that `Promise`, we output to the console that all of the images have been

³⁷. <http://smashed.by/promiseobject>

converted. If we save this script as *webp.js*, we can run it with the node command like so:

```
node webp.js
```

Assuming everything is successful, a message will appear in the console:

```
Images converted!
```

When all is done, WebP images should now exist in the *images* directory, relative to the location of where you saved *webp.js*. If you want to tweak the output, you can do so through a variety of options³⁸. For example, if you wanted to generate lossless WebP images, you would instead use the `lossless` option like so:

```
webp({  
  lossless: true  
})
```

Just be aware that as stated earlier, the quality setting for lossless mode adjusts the compression. The compression is still lossless, but higher settings will generate smaller files.

Pro tip: the options you can pass to this plug-in are the same, no matter where you invoke the *imagemin-webp* plug-in, be it in a Node.js script, or the build system of your choice. Speaking of build systems, let's next cover how you might convert images using *gulp*.

³⁸. <http://smashed.by/imagemin>

USING GULP

As far as build systems go, [gulp](https://gulpjs.com/)³⁹ is pretty common. Thanks to a huge ecosystem of plug-ins, gulp can do almost anything you'd need it to, including converting images to WebP! If you have gulp installed and configured for your project, you only need to install a few extra node modules at the command line:

```
npm install gulp-imagemin imagemin-webp  
gulp-ext-replace
```

gulp-imagemin is a plugin that allows imagemin to interface with gulp. As in the previous example using a plain Node.js script, imagemin-webp is the plug-in that allows imagemin to export images to the WebP format. The final plug-in, gulp-ext-replace, just allows us to change the extension of files output by gulp. By default, gulp outputs files with the same extension as their input source. gulp-ext-replace helps us to overcome this so we can write WebP images to the disk with the proper *.webp* extension.

Once you have these plugins installed, you'd just need to write a quick gulp task that reads source images from the disk and outputs them to WebP format. That task might look something like this:

```
const gulp = require("gulp");  
const imagemin = require("gulp-imagemin");  
const webp = require("imagemin-webp");
```

³⁹. <https://gulpjs.com/>

```

const extReplace = require("gulp-ext-replace");

gulp.task("exportWebP", function() {
  let src = "src/images/**/*.png"; // Where your
  PNGs are coming from.
  let dest = "dist/images"; // Where your WebPs are
  going.

  return gulp.src(src)
    .pipe(imagemin([
      webp({
        quality: 75
      })
    ]))
    .pipe(extReplace(".webp"))
    .pipe(gulp.dest(dest));
});

```

There's a lot going on here, so let's break it down:

1. Like any Node.js-driven program, the first part of the script imports the modules necessary for the script to work.
2. Using the `gulp.task` method, we create a task named `exportWebP`, which starts with two variables: `src` points to the image files we want to process (PNG files in this example), `dest` points to the directory where the resulting WebP images will be written to.

3. The `gulp.src` method reads the PNG images from the location specified by the `src` variable.
4. Images are ferried to `imagemin` by the `gulp.pipe` method. The sole argument passed to `imagemin` is an array of `imagemin` plug-ins, which in this case is the sole `imagemin-webp` plug-in. As is the case with using `imagemin` in any environment, the arguments passed to the `imagemin-webp` plug-in (or any `imagemin` plug-in) will follow the same format.
5. Before we write the converted WebP images to the disk, we want to ensure the resulting files are written with a `.webp` extension by using the `gulp-ext-replace` plug-in. Using `gulp`'s `dest` method, we write all the converted images to the location specified earlier in the `dist` variable.
6. Using `gulp`'s `dest` method, we write all the converted images to the location specified earlier by the `dist` variable.

When everything's in place, we can invoke `gulp` on the command line to convert images in the `src/images` directory like so:

```
gulp exportWebP
```

Once the command finishes, your destination directory will contain WebP images you converted from whatever `gulp` finds in the source directory. That simple!

If you prefer Grunt over gulp, you're in luck. Next, we'll show an example of how you can use Grunt to convert images to WebP.

USING GRUNT

Much like gulp, Grunt⁴⁰ is a task runner, albeit with a different syntax. It can be used for many of the same things gulp is used for, including optimization and conversion of images to different formats via imagemin. Unsurprisingly, it too can convert images to WebP by way of imagemin's imagemin-webp plug-in. If you have a project currently using Grunt that you would like to modify to generate WebP images, it's a relatively trivial task. In the directory containing *Gruntfile.js*, simply install these two modules like so:

```
npm install grunt-contrib-imagemin imagemin-webp
```

This command installs the imagemin plug-in for Grunt (represented as grunt-contrib-imagemin), as well as the familiar imagemin-webp plug-in used to convert images to WebP. Once the installation finishes, you'll need to set up a Grunt task in *Gruntfile.js* like so:

```
const grunt = require("grunt");
const webp = require("imagemin-webp");

grunt.initConfig({
  imagemin: {
```

⁴⁰. <https://gruntjs.com/>

```

    dist: {
      options: {
        use: [webp({
          quality: 75
        })]
      },
      files: [{
        expand: true,
        cwd: "src/images/",
        src: ["**/*.png"],
        dest: "dist/images",
        ext: ".webp"
      }]
    }
  }
});

grunt.loadNpmTasks("grunt-contrib-imagemin");

```

Once again, let's step through this code and find out what's going on:

1. Initially, we require the necessary grunt and imagemin-webp modules that we'll need for the imagemin Grunt task.
2. In the imagemin task, we create a dist target, which is what we'll run from the command line. This target contains an options object for Grunt's imagemin plug-in that allows us to specify configuration options. In this

case, we supply an instance of the `imagemin-webp` plug-in and pass the usual options to it.

3. The `files` object is the guts of the operation. `cwd` specifies which directory we want to work within. `src` specifies the files within `cwd` that we want to convert to WebP. `dest` specifies the directory we want to output the converted images to. Finally, `ext` specifies that we want the converted images to be saved with a `.webp` extension.
4. The last line of code loads the `grunt-contrib-imagemin` plug-in so we can use it in the Gruntfile.

Once you have this in place, you can convert images to WebP like so:

```
grunt imagemin:dist
```

Once this command finishes, images ending in `.png` in the directory specified in `files.cwd` will be converted to WebP, and output to the directory specified in `files.dest`.

Grunt is not quite as intuitive as `gulp` is, and is falling out of use somewhat. However, it's still a serviceable choice for a build system, and you may yet encounter it in some situations.

To round out our documentation of how to convert images to WebP within the Node.js ecosystem, let's discuss how you might accomplish that same task in `webpack`.

USING WEBPACK

If you're developing modern JavaScript applications, chances are high that webpack⁴¹ is in your midst. In contrast to gulp and Grunt, which style themselves as task runners, Webpack is a module bundler that analyzes your code starting from one or more entry points and generates optimized output. While much of what webpack does is accomplished through loaders, it *does* have a large ecosystem of plug-ins, too. imagemin is represented in that space by imagemin-webpack-plugin.

How webpack works and how to write a config for it are complex subjects, especially if you're used to using task runners like gulp. I'm going to assume you have at least some familiarity with webpack basics, and just show you how to add imagemin to an existing webpack config to convert images to WebP. Like gulp and Grunt before, you'll need to use npm to install a few necessary Node.js modules:

```
npm install imagemin-webpack-plugin imagemin-webp  
copy-webpack-plugin
```

Similar to how gulp has its own imagemin plugin, webpack has one as well in the form of imagemin-webpack-plugin. And as is the case with all imagemin-related use cases, the imagemin-webp plug-in provides the WebP conversion functionality. The copy-webpack-plugin

⁴¹. <https://webpack.js.org/>

module is used in this case to help us copy images from a source folder, and tell imagemin to process those images for us. Webpack can be rigged up with loaders (such as file-loader) to pipe the file it encounters in its dependency graph to imagemin intelligently, but it may be more utilitarian to specify a source directory on the disk and let imagemin churn through everything it finds and spit out images to the destination directory. Such code might look something like this:

```
ImageminWebpackPlugin =  
require("imagemin-webpack-plugin").default;  
ImageminWebP = require("imagemin-webp");  
CopyWebpackPlugin = require("copy-webpack-plugin");
```

```
module.exports = {  
  // Omitted required entry, output, and loader  
  configs for brevity...  
  plugins: [  
    new CopyWebpackPlugin([  
      {  
        from: "./src/images/*.png",  
        to: "./images/[name].webp"  
      }  
    ]),  
  
    new ImageminWebpackPlugin({  
      plugins: [  
        ImageminWebP({  
          quality: 75  
        })  
      ]  
    })  
  ]  
}
```

```
    })
  ]
};
```

As we have with other code examples, let's step through everything and get a handle on what's happening:

1. As noted, the required entry, output, and loader configurations have been omitted for brevity and relevance, as we're assuming you have *some* familiarity with webpack.
2. The `plugins` config is merely an array of plug-ins we want to use. The first plug-in in the array is an instance of `copy-webpack-plugin`. To start, we specify in `from` the files we want to copy from the source directory. In this example, we include all files ending in `.png` from a specific source directory.
3. Next, we tell `copy-webpack-plugin` where we want the optimized images to go in the `to` option. It's important to note in this option that we're using a placeholder of `[name]` to tell `copy-webpack-plugin` that we want the name of the output file to be the same as its corresponding source. However, because we're outputting WebP files, we don't want to preserve the source file's extension, so we're hardcoding the output file's extension to `.webp`. It's also important to remember that files will be written to a location relative to whatever is set in `output.path` earlier in your webpack config.

4. The next plug-in in the sequence is an instance of `imagemin-webpack-plugin`, which should look familiar to other `imagemin` use cases. Here, we're simply passing an instance of the `imagemin-webp` plug-in to the `imagemin` instance.

Using this configuration, all images ending in `.png` found in `./src/images` will be converted to WebP and output to the `images` directory relative to your configuration's `output.path` directory.

While there are multiple approaches to converting images to WebP within webpack, this aims to be the most straightforward. Compared to other build systems, webpack is an incredibly rich (and at times complex) tool. This example is not meant to be *the* authoritative approach to generating WebP images within webpack, but rather to show that it's possible.

With an explanation of how to convert images to WebP within Node.js and its numerous build systems out of the way, let's move on to what is perhaps the most pressing matter: how to *use* WebP.

Using WebP Images

Because WebP isn't supported in all browsers, you'll need to learn how to use it that sites and applications gracefully fall back to established formats when WebP support is lacking. Here, we'll discuss the many ways you can use WebP responsibly, starting by detecting browser support in the Accept request header.

Detecting Support on the Server

When a browser that supports WebP happens by, it will advertise WebP support in the Accept HTTP request header. If you're not familiar with Accept, its purpose is to advertise what content types a given browser supports. The value of this header is contextual. As such, it will change depending on the resource type being requested. For example, a request for a CSS file may carry a different value for a corresponding Accept header than a request for an image might.

Browsers supporting WebP advertise their support for it in relevant contexts, which is often in requests for HTML documents and images. The value of Accept in these cases may look something like this:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
```

Within the Accept header content, you'll notice the image/webp content type is advertised, which signals the browser supports WebP images. If your site is powered

by a back-end language, you can easily inspect the content of this header to modify image delivery. In PHP, for example, that could look something like this:

```
<?php
$webpSupport = striestr($_SERVER["HTTP_ACCEPT"],
"image/webp") !== false;
$imageSource = $webpSupport ? "example.webp" :
"example.jpg";
?>
```

```

```

In this example, we use PHP's striestr function⁴², which performs case-insensitive substring matching on a string input. If it finds the `image/webp` substring in the `Accept` header (accessible in the `$_SERVER["HTTP_ACCEPT"]` global variable), we can change the `src` value of the `` tag accordingly. The logic used here should be largely similar in any other back-end language.

The main advantage of serving WebP images this way is it simplifies your markup. Sending fewer bytes down the wire is always preferable to sending more. If you can serve WebP images in this fashion, you absolutely should.

That said, it's not always possible to inspect the `Accept` header and modify markup based on its value. For

⁴². <http://smashed.by/striestr>

example, you might be running a static site with no access to a back-end language. In cases like these, you may have to rely on a Rewrite rule, or the `<picture>` element to provide adequate fallbacks.

As a Rewrite Rule

Sometimes it's desirable to separate WebP delivery logic from your application code entirely, and rely on the web server configuration to rewrite requests for appropriate image types to WebP. This may be desirable to you if you like to keep WebP detection out of your application code, and turn it into a web server concern. To achieve this on Apache, you'd use `mod_rewrite`⁴³ in an `.htaccess` or core configuration file to rewrite the request:

```
RewriteEngine On
RewriteCond %{HTTP:Accept} image/webp [NC]
RewriteCond %{HTTP:Content-Disposition} !attachment
[NC]
RewriteCond %{DOCUMENT_ROOT}/$1.webp -f [NC]
RewriteRule (.+)\.(png|jpe?g|gif)$ $1.webp [T=image/
webp,L]
```

Let's step through each line of this rewrite chain and talk a little about what's going on here.

1. The first line turns on the the rewrite engine. If this line is omitted, no URL rewriting will occur.

⁴³ <http://smashed.by/modrewrite.com/>

2. Similar to the server-side detection mechanism discussed previously, the first rewrite condition (`RewriteCond`) checks if the `Accept` request header advertises WebP support by looking for the `image/webp` substring. The `[NC]` portion is a rewrite flag that says the check is not case-sensitive.
3. The second condition checks if the `Content-Disposition` request header has been set. If you've ever right-clicked on an image to save it, the browser sends a `Content-Disposition` header with a value of `attachment` to tell the server that the user wants to download the image to their computer. Because WebP images aren't viewable in most operating systems without installing extra software, we *don't* want to send a WebP image if the user wants to save an image to their computer. We want to send them whatever the request was originally for so they can work with the image more easily. To do this, we ensure `Content-Disposition` doesn't contain a value of `attachment`.
4. The third and final condition checks to see if the image exists on the server. This is useful if you don't have WebP alternates for every image on your site. If a WebP version of the image is found, the server will send it. If not, the request for the original image will be fulfilled. From here, we can specify the URL pattern that will be rewritten if (and *only* if) this and all prior conditions have been met.

5. The `RewriteRule` specifies the URL pattern that the rewrite engine will act on. `RewriteRules` are composed of two to three parts: a regular expression that matches the incoming request; what the request should be rewritten to; and any optional flags to modify the matching behavior. In this case, we're using a regular expression to match any request for resources ending in `.png`, `.jpg`, `.jpeg`, or `.gif`. If the request pattern matches, the resource request is rewritten to point to an image file ending in `.webp`. To cap it off, we use two flags: the `T` flag specifies the content type of the response (`image/webp` in this case). Since rewrite rules can be chained together, the `L` flag states that this is the last rule in the chain.

Apache isn't the only server capable of rewriting URLs, it's just one of the most commonly used. Many popular web servers have rewrite capability either bundled in or available in a dynamically loadable extension. Servers with rewrite functionality include Nginx⁴⁴, IIS⁴⁵, and lighttpd⁴⁶.

If you're going to use a URL rewriter other than `mod_rewrite`, be aware of potential caching issues when rewriting requests. For example, when this rewrite rule works it doesn't cause a browser to *redirect* from `https://example.com/my-awesome-image.jpg` to

⁴⁴. <https://www.nginx.com/>

⁴⁵. <https://www.iis.net/>

⁴⁶. <https://www.lighttpd.net/>

https://example.com/my-awesome-image.webp. If the rule kicks in and the requesting user is sent a WebP image, the URL for that resource still points to a resource with a .jpg extension. You *could* use a 300-level redirect in a rewrite flag (e.g., [R=302]) to force the browser to redirect to the WebP image, but this is a performance anti-pattern in that redirects add latency to requests. For this reason, redirects should be avoided when possible.

To prevent caching issues with two types of content originating from the same URL for different browsers, `mod_rewrite` will automatically adjust the `Vary` response header to pay attention to whatever header is involved in a rewrite condition. This ensures that browsers (as well as proxies and CDNs) understand how to cache the resource properly. If you're not using `mod_rewrite`, but your rewrite rules are similar to the one used above, ensure the `Vary` header is set to pay attention to heads such as `Accept` or `Content-Disposition`. Otherwise, clients without WebP support could receive WebP responses in some circumstances (such as when CDNs are involved).

URL rewriting can seem kind of scary if you're new to it — and that's OK! It's not the easiest thing to understand at first, especially if you have limited experience with configuring web servers. If your comfort level with `mod_rewrite` in particular is limited to copying and pasting rewrite rules into an `.htaccess` file, the above snippet should do the trick. If you don't like the idea of using them at all, though, there are plenty of other

methods for using WebP responsibly with fallbacks. So don't fret!

Using WebP in <picture>

If you don't have access to a back-end language or the ability to change a server's configuration, you might feel left in the cold when it comes to using WebP in a responsible way. Fortunately for everyone, browser vendors have provided a use case by way of the `<picture>` element.

Use cases for `<picture>` are numerous, and one of them covers using WebP images with a fallback. Let's take a look at some example HTML code that uses a WebP image with a JPEG fallback:

```
<picture>
  <!-- The preferred WebP source image -->
  <source srcset="some-image.webp" type="image/
webp">
  <!-- The fallback image used if WebP or <picture>
isn't supported -->
  
</picture>
```

If this looks similar to the contents of a `<video>` element, you're not wrong. In much the same way `<source>` elements allow you to specify a preferred order of video resources in a `<video>` element, `<source>` elements also allow you to do the same thing within a

`<picture>` element. In the above example, the first (and only) `<source>` specifies a WebP image, and identifies it as such via the `type` attribute. The `type` attribute accepts a content (or [MIME](http://smashed.by/mimetypes)⁴⁷) type the browser uses to determine if the `<source>` is ultimately usable. If the lone `<source>` element isn't usable in this case, the JPEG image specified in the `` element's `src` attribute is used instead. If `<picture>` is not supported in a particular browser, the content of the contained `` element's `src` attribute is always used, and the `<source>` element is ignored. As such, an `` element within a `<picture>` element should *always* contain a `src` value that points to an image usable by all browsers.

Also be aware that, as is the case with the `<video>` element, multiple `<source>`s can be used in `<picture>`. Let's assume a more complex use case of `<picture>` with multiple `<source>`s targeting high- and low-pixel density screens (for example, Retina versus standard DPI) with WebP, [JPEG-XR](http://smashed.by/jpegxr)⁴⁸ and JPEG images:

```
<picture>
  <!-- This source is examined first -->
  <source srcset="some-image-2x.webp 2x,
    some-image-1x.webp 1x" type="image/webp">
  <!-- This source is examined last -->
```

⁴⁷. <http://smashed.by/mimetypes>

⁴⁸. <http://smashed.by/jpegxr>

```

<source srcset="some-image-2x.jxr 2x,
some-image-1x.jxr 1x" type="image/jxr">
<!-- The fallback -->

</picture>

```

The order of these `<source>`s is optimal, because it specifies preferred formats (WebP and JPEG XR) *first*. Getting the order correct is important, because browsers evaluate `<source>`s in `<picture>` from top to bottom. As is the case with `<video>`, specify your most optimal media resource(s) first, *then* specify additional preferences and a fallback that will work everywhere else.

While images are often used inline in HTML, they're also used quite often in CSS in various properties. Next, let's talk a bit about how you can use WebP images in CSS with fallbacks in mind.

In CSS

CSS allows you load images via the `url()` resource pointer in various properties, such as `background`, `list-style` and so forth. What's problematic about using CSS in WebP is that there's no support detection capability for WebP built into the language. You *can* get around this by using using an image CDN service (such as Cloudinary, mentioned earlier), or by using a URL

rewriter (also mentioned earlier). But what if neither of those are options?

Depending on your host and the tools you have access to, you may have to rely on a back-end language to detect support and modify markup on delivery. PHP is one such potential tool you might come across. You might remember this WebP detect code from a section earlier on where we detect WebP support by inspecting the Accept header:

```
<?php $webpSupport =  
stristr($_SERVER["HTTP_ACCEPT"], "image/webp") !==  
false; ?>
```

In an earlier example, we acted on this `$webpSupport` variable to change the image source in HTML. While it's *possible* to execute PHP in contexts other than HTML (such as a CSS file), you may not want to architect your application to do so. Another strategy could be to expose WebP support via a class, like so:

```
<?php $webpClass = $webpSupport === true ? "webp" :  
"no-webp"; ?>  
<html class="<?php echo($webpClass); ?>">
```

With this class added to the `<html>` element, we now have a hook we can use in CSS to change how we reference images:

```
.no-webp body {  
    background-image: url("/images/background.jpg");  
}
```

```
.webp body {
  background-image: url("/images/background.webp");
}
```

Easy as that. By using the `no-webp` and `webp` classes, we can modify the delivery of images in CSS so browsers capable of using WebP get WebP. Those that don't will use the proper fallback.

In a Service Worker

Ideally, you'd want to take care of serving WebP on the server side or via the `<picture>` element whenever possible, because handling image requests this way would be both most efficient and convenient. *But*, one web developer's situation and restrictions could be quite different from another's. You might not have access to a back-end language, or even the server configuration itself. In cases like these, you *could* use a service worker to rewrite image requests to WebP on the client.

Service workers are extremely powerful, and can provide app-like offline experiences and push notification functionality. At their core, however, they're simply a sort of a request-and-response handler similar to a URL rewriter, except for one hugely fundamental difference: the service worker's context is that of the client rather than the server, and the capability is available to front-end developers. Below is a highly simplified example of how you could use a service worker to intercept a network request in a `fetch`⁴⁹ event to return a

WebP image if the client is capable of using WebP images:

```
self.addEventListener("fetch", function(event) {
  const imageRequest = /\.?(png|jpe?g|gif)$/i;
  const url = new URL(event.request.url);

  if (url.origin === location.origin &&
  imageRequest.test(url.href) === true) {
    if (event.request.headers.has("accept") &&
    event.request.headers.get("accept").includes(
    "image/webp")) {
      let webp = url.href.replace(imageRequest,
      ".webp");
      event.respondWith(fetch(webp));
    }
  }
});
```

Once again, let's step through the code bit by bit so we can get a handle on what's going on:

1. The first line attaches an event listener to the service worker's fetch event. The attached code will run every time the service worker detects an outgoing request to the network.








49. <http://smashed.by/fetchevent>

2. `imageRequest` is simply a regular expression we can use to check if the request is for a resource ending in `.png`, `.jpg`, `.jpeg`, or `.gif`.
3. Using the `event.request.url` value, we create a new instance of the `URL` object⁵⁰. `URL` provides convenient methods that makes working with URL data easier.
4. Before we rewrite any image requests, we want to be sure of two things. First, the request needs to be for the primary origin, not a cross-origin request. We ensure this by checking that `url.origin` and `location.origin` are the same. Second, we want to ensure the request is for an image, by using the `imageRequest` regular expression to check if the requested resource's extension is for an image.
5. If the request is for the primary origin, and is for an image, we can continue on. At this point, we check if the event object's request headers includes an entry for the `Accept` header, *and* if it contains the `image/webp` substring. The mechanism here is exactly the same as when we checked the `Accept` header in our back-end code, only this time, we're performing the check in JavaScript.
6. Finally, if the browser supports WebP, we'll rewrite the URL to point to a WebP image, and we respond with a fetch request to retrieve it from the network. Any requests that don't meet the conditions described above

⁵⁰. <http://smashed.by/urlinterface>




just pass through to the browser as normal without any intervention from the service worker.

If a user happens by in a browser that supports service workers *and* supports WebP, requests for PNG, JPEG and GIF images will be rewritten to WebP. You can verify that this works in your browser dev tools. In Chrome's dev tools, that process may look something like this in the network panel:

Name	Size	Content-Type	Waterfall	40.00 ms
 localhost	520 B	text/html; charset=UTF-8		
 test.jpg	(from ServiceWorker)	image/webp		
  test.webp	59.0 KB	image/webp		

A request for a JPEG image rewritten to a WebP image by a service worker.

In the figure above, you can see that the request from *test.jpg* has been rewritten to *test.webp* by observing two things in the network panel contents. First, the **Size** column for *test.jpg* contains a value of “(from ServiceWorker),” which plainly indicates that the request has been handled by the service worker. Second, the request for *test.webp* will have a small gear icon next to the resource in the **Name** column. This also indicates that the request was handled by the service worker. If you hover over the waterfall entry for *test.jpg*, you'll notice the breakdown of timings includes service worker-related entries within the request/response category:

Request/Response		TIME
ServiceWorker Preparation		1 μ s
Request to ServiceWorker		22.34 ms
Waiting (TTFB)		33 μ s
Content Download		9.83 ms
Explanation		36.99 ms

Service worker timings exposed in Chrome's network panel for a service worker-initiated network request.

It may seem unintuitive at first that these service worker-specific timings appear under the entry for the original JPEG request, but once you remember that this specific request was what the service worker intercepted, it makes sense that timings for it would appear in this location.

In my opinion, this approach *does* present some flaws. First, it doesn't do anything for users on browsers that support WebP, but don't *also* support service workers. Currently, this only includes Opera Mini. You may be think this is an edge case, but Opera Mini sees some usage in emerging markets where users could benefit the most from WebP images, but would miss out if an application depended on service workers to serve them. Secondly, because of the service worker life cycle, image requests may not get rewritten until after the service worker has been installed. In this case, the first page navigation will request images in established formats (JPEG, GIF, or PNG), and *then* rewrite requests to use WebP images on subsequent requests.

For these reasons, I still recommend rewriting URLs on the server side instead of in a service worker, as it will work for *everyone* who supports WebP. Still, this approach gives you a reasonably transparent means of rewriting requests, and may prove useful in some contexts. As with any approach, weigh the benefits against the drawbacks, and make the best decision for your specific application and audience.

In Closing

The role of performance in the user experience is critical for a number of reasons, and media play a big role, specifically in loading performance. Images account for such a large portion of the total data we consume, and for those on high latency and low bandwidth connections, it's incumbent on us to do all we can to make the experience as fast as possible.

WebP isn't perfect, and this guide is not an advertisement for the format. If you're already well-served by established image formats, and your site is already quite fast, WebP may not be for you.

But WebP *does* offer potential benefits to a large portion of internet users. To say that it doesn't is simply false. In my experience, WebP has substantially improved loading performance for many of my clients, and continues to be a tool I reach for when I want to make pages as lean as possible. While it's certainly true that augmenting applications to serve WebP takes effort, I have found in project after project that the effort is worth it.

While image formats represent a constantly evolving space with new contenders continually vying for implementation in browsers, WebP currently stands as the most viable alternative format, owing to the collective ubiquity of browsers that support it. Simply put, if you're not currently using WebP, you could be missing out on a chance to make your site faster than it already is for a substantial portion of your users. Hopefully,

this guide will encourage you to experiment and see what's possible with WebP, and help you to draw your own conclusions. 🐼

Appendix

If you're reading this, chances are good that you arrived here after reading the "Performance" section of the ebook, and are curious as to how the statistics used in that section were generated. This appendix will answer those questions.

Lossy WebP

For lossy WebP images, this process was followed:

1. 50 photographic images were selected at random from the internet. All images were selected from lossless PNG sources.
2. 101 lossy WebP images were generated from each PNG source, one for every quality setting between and including 0 and 100 (representing a pool of 5,050 images). In addition to exporting to the lossy WebP format, images were exported to JPEG across the entire quality range from PNG sources using the following JPEG encoders and methods:
 - cjpeg version 9b, converting to both baseline and progressive modes with the `-optimize` parameter.
 - mozjpeg version 3.2, converting to both baseline and progressive modes, also using the `-optimize` parameter.

- Guetzli version 1.0.1. Because Guetzli has a lower quality limit of 84, JPEGs converted using this encoder were exported at a quality range between 84 and 100.
3. As each set of images was exported, their file names, target quality, file size and SSIMULACRA score were recorded to a SQL database.

Lossless WebP

For lossless WebP tests, this process was followed:

1. 50 PNG images were selected at random from the internet. Most of the images selected were logos, line art, and other content best suited to the PNG format. Most images used transparency.
2. Each PNG source was optimized in the OptiPNG, ZopfliPNG, and Pngcrush optimization tools using the most optimal settings possible.
3. Lossless WebP images were created from the output of the highest performing PNG optimizer (OptiPNG). In addition to the -lossless parameter, further cwebp parameters used were -q 100, -m 6 and -z 9, which created the smallest possible output.
4. As each set of images was exported, their file names, and file size were recorded to a SQL database. SSIMULACRA scores were not recorded for lossless WebP or

PNG images because lossless encoding creates images that are visually identical to their source.

Measuring Image Decode Time

One aspect of image performance is decode time in the browser. I wanted to measure the decode times for the images I generated, but where thousands of images are concerned, this can be a brutally monotonous task if it's not automated.

Thankfully, tools like [Puppeteer](http://smashed.by/puppeteer)⁵¹ helped a *ton* on this front. With the help of [Paul Irish](https://www.paulirish.com/)⁵² and [Tim Kadlec](https://timkadlec.com/)⁵³, I was able to put together a hybrid Bash and Node.js solution that automated the measurement of image decodes. If you're interested in seeing the code behind this, [check out this GitHub gist](http://smashed.by/decodedecode)⁵⁴.

⁵¹. <http://smashed.by/puppeteer>

⁵². <https://www.paulirish.com/>

⁵³. <https://timkadlec.com/>

⁵⁴. <http://smashed.by/decodedecode>

Thanks

Many people were involved in the development and production of this book, and their efforts are sincerely appreciated.

Thank you to Rachel Andrew, Rey Bango, Markus Seyfferth, and Vitaly Friedman of Smashing Magazine for their work in managing the project, procuring editors, and publishing this work.

Thank you to Drew McLellan⁵⁵ for tech editing the book and ensuring the final text was as technically accurate as possible.

Thank you to Paul Irish⁵⁶ and Tim Kadlec⁵⁷ for helping me to find a way to automate image decode measurements in the browser. Without them, this data would not be present in the book.

Thank you to Paul Calvano⁵⁸ for wrangling HTTP Archive⁵⁹ data from BigQuery to help me find stats on median image sizes on the web.

Thanks for The WebM Project⁶⁰ for making WebP a thing so I could write about it!

Finally, thanks to my wife and stepdaughters for continuing to support me in my technical writing endeavors.

⁵⁵. <https://twitter.com/drewm>

⁵⁶. https://twitter.com/paul_irish

⁵⁷. <https://twitter.com/tkadlec>

⁵⁸. <https://twitter.com/paulcalvano>

⁵⁹. <https://httparchive.org/>

⁶⁰. <https://www.webmproject.org/>

About The Author

Jeremy Wagner is a developer, writer, and speaker from Minnesota. He's the author of the book *Web Performance in Action*⁶¹, a web developer's guide for creating fast websites. You can find him on Twitter [@malchata](https://twitter.com/malchata)⁶² or on the web at jeremywagner.me⁶³.

⁶¹. <http://smashed.by/perfinaction>

⁶². <https://twitter.com/malchata>

⁶³. <https://jeremywagner.me/>