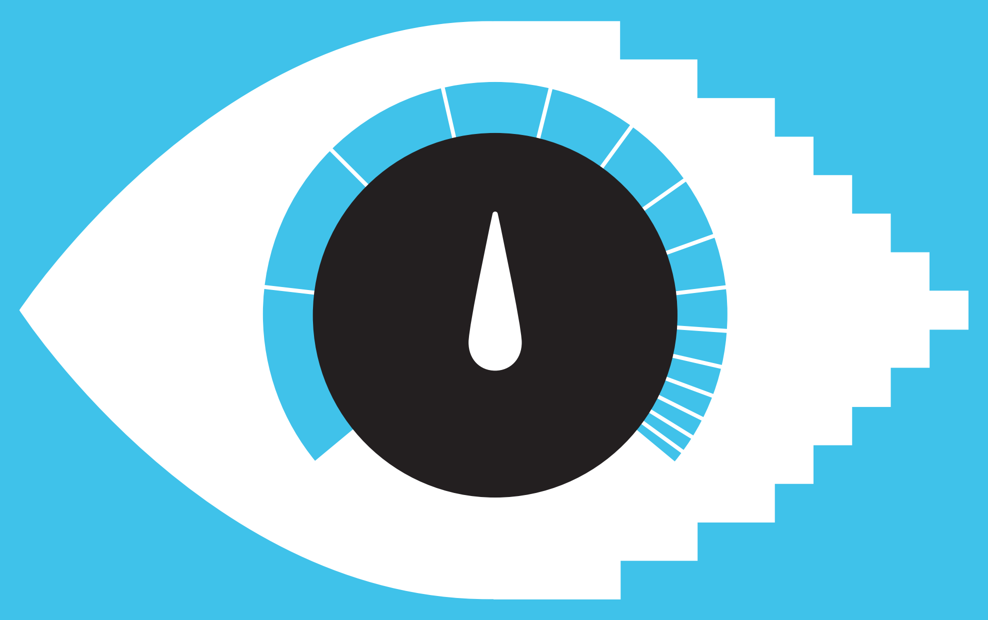
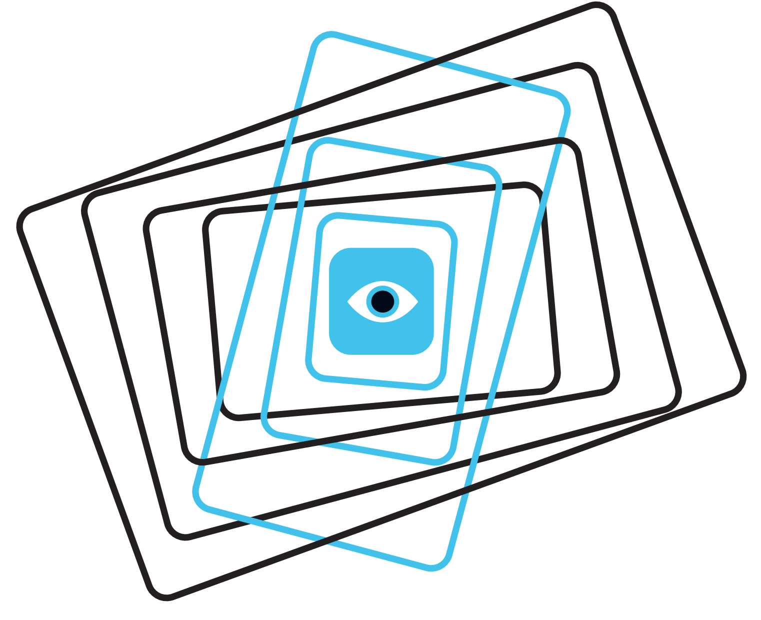


# Image Optimization

by Addy Osmani



# Image Optimization

by

Addy Osmani

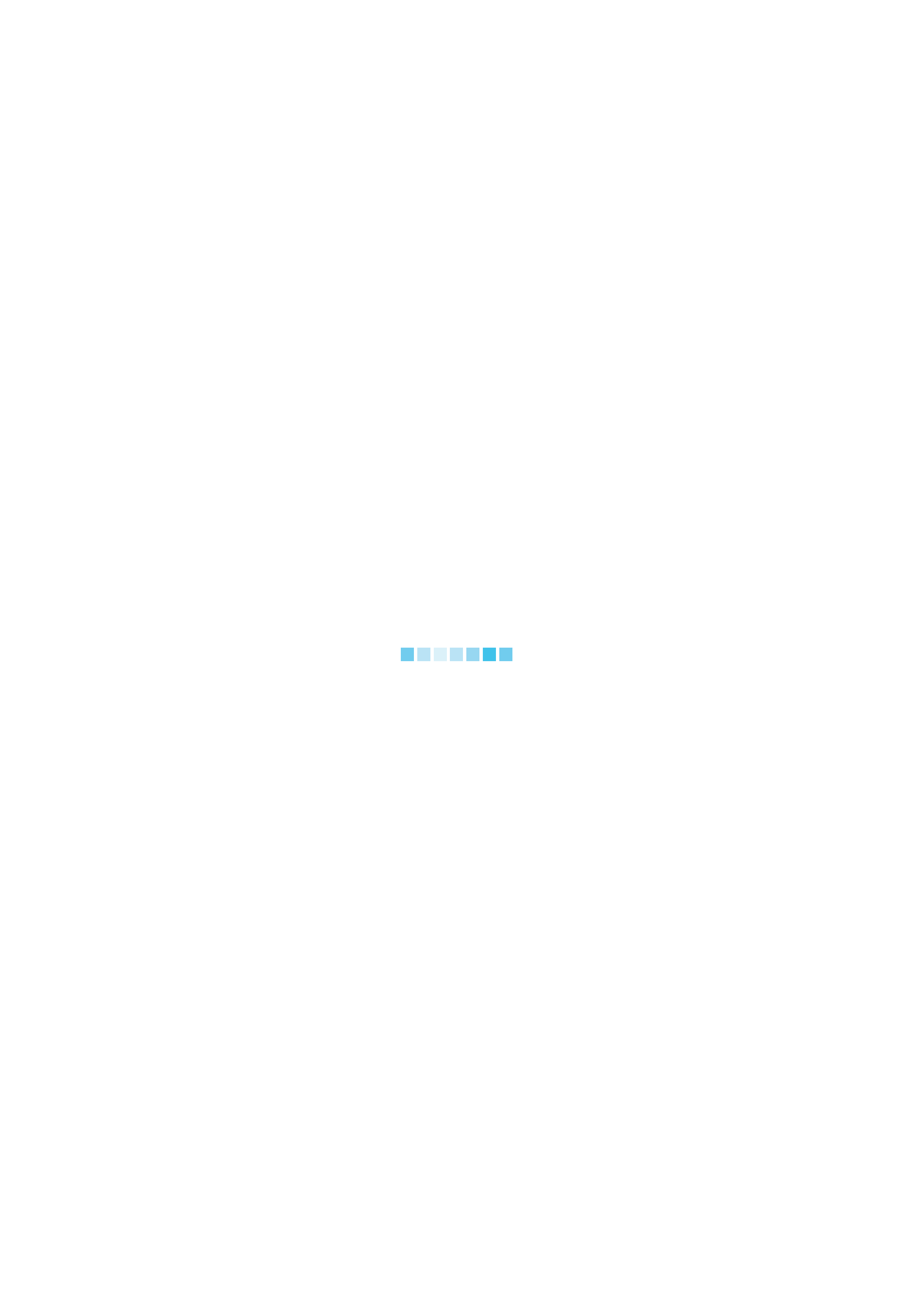
Published 2021 by Smashing Media AG, Freiburg, Germany.
All rights reserved.
ISBN: 978-3-945749-94-4

Technical editing: Milica Mihajlija and Colin Bendell
Copyediting: Owen Gregory
Cover and section illustrations: Espen Brunborg
Interior full-page illustrations: Nadia Snopek
Book design and indexing: Ari Stiles
Ebook production: Cosima Mielke
Typefaces: Elena by Nicole Dotin, Mija by
Miguel Hernández and Andalé Mono by Steve Matteson

*Image Optimization* was written by Addy Osmani.
Reviewers and contributors include Colin Bendell,
Kornel Lesiński, Estelle Weyl, Jeremy Wagner,
Tim Kadlec, Nolan O'Brien, Pat Meenan, Kristofer Baxter,
Henri (Helvetica) Brisard, Houssein Djirdeh, Una Kravets,
Ilya Grigorik, Elle Osmani, Leena Sohoni, Katie Hempenius,
Jon Sneyers & Mathias Bynens.

This book is printed with material from
FSC® certified forests, recycled
material and other controlled sources.

Please send errors to: errata@smashingmagazine.com



# Contents


*Foreword by Colin Bendell* . . . . . . . . . . . . . . vi

*Introduction* . . . . . . . . . . . . . . . . . . . . . . . xix

1 The Humble <img> Element . . . . . . . . . . . 29

PART ONE ■ **IMAGE QUALITY AND PERFORMANCE**

2 Optimizing Image Quality . . . . . . . . . . . . 43

3 Comparing Image Formats . . . . . . . . . . . . 54

4 Color Management . . . . . . . . . . . . . . . . . 68

5 Image Decoding Performance . . . . . . . . . . 80

6 Measuring Image Performance . . . . . . . . . . 100

PART TWO ■ **CURRENT IMAGE FORMATS**

7 JPEG . . . . . . . . . . . . . . . . . . . . . . . . . . 115

8 PNG . . . . . . . . . . . . . . . . . . . . . . . . . . 144

9 WebP . . . . . . . . . . . . . . . . . . . . . . . . . 170

10 SVG . . . . . . . . . . . . . . . . . . . . . . . . . . 200

PART THREE ■ **IMAGES IN BROWSERS**


11 Responsive Images . . . . . . . . . . . . . . . . 223

12 Progressive Rendering Techniques . . . . . . . . 238

13 Caching Image Assets . . . . . . . . . . . . . . 256

14 Lazy-Loading Images . . . . . . . . . . . . . . . 290

15 Replacing Animated GIFs . . . . . . . . . . . . . 314

16 Image Content Delivery Networks . . . . . . . . 337


PART FOUR ■ **NEW & EMERGING IMAGE FORMATS**


17 HEIF and HEIC . . . . . . . . . . . . . . . . . . 382

18 AVIF . . . . . . . . . . . . . . . . . . . . . . . 398

19 JPEG XL . . . . . . . . . . . . . . . . . . . . . 410

20 Comparing New Image Formats . . . . . . . . . 433


PART FIVE ■ **FURTHER OPTIMIZATION**


21 Data Saver . . . . . . . . . . . . . . . . . . . . 457

22 Optimize Images for Core Web Vitals . . . . . . 471

23 Case Study: Twitter . . . . . . . . . . . . . . . 490

Conclusion . . . . . . . . . . . . . . . . . . . . 505

# Foreword

*by Colin Bendell*

Images and animations are an important part of the web experience – arguably the most important part. Images and animations can tell complex stories in just one glance, they can attract and engage audiences, and they can provide artistic expression in consistent and unique ways that other web technologies cannot. Every web performance strategy culminates in bringing the visual content to the user. For this reason, even optimizing images, animations, and video is an essential last link in the chain to ensuring the success of our web pages.

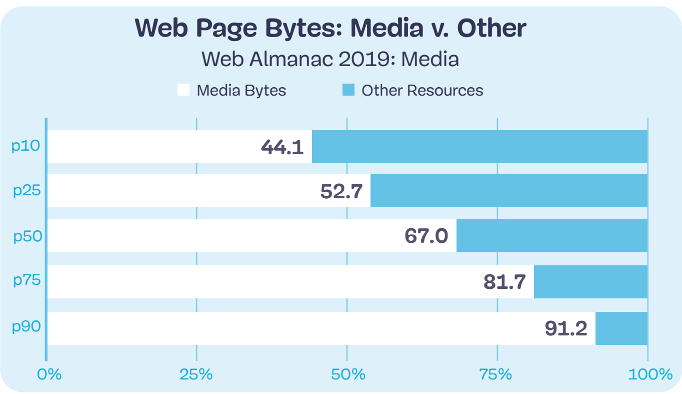
Examining the data from the HTTP Archive's Web Almanac,[1] we can see both how important media is for the web, and why image optimization can go a long way to lower the size of media on the web.

We can measure how important media resources are in two ways: by the sheer volume of bytes required to download for a page, and how much of the web page layout we devote to presenting these technologies.

From a pure bytes perspective, HTTP Archive has historically reported[2] an average of two-thirds of resource bytes

1 https://smashed.by/almanac
2 https://smashed.by/bytesperpage

associated with media. From a distribution perspective, we can see that virtually every web page depends on images and videos. Even at the tenth percentile, we see that 44% of the bytes are from media and can rise to 91% of the total bytes at the 90th percentile of pages.




*Web page bytes: image and video versus other.*

These bytes are important to render pixels on the screen. As such, we can see the importance of the images and video resources by also looking at the amount of pixels used per page. Typically, we express this volume in megapixels, but this can be a challenging metric to internalize because it is largely abstract, and the number grows geometrically, which is hard to easily compare.

One easy way to get a feel for the metric is to compare it to devices we use. For example, the Samsung Galaxy S10 has CSS dimensions of 360×740 pixels or 0.26 MP. But because we use the devices closer to our eyes, the S10 packs more pixels per CSS pixel (referred to as pixel density or device pixel ratio) and has an actual screen display of 1,440×3,040 pixels or 4.2 MP.

There are three metrics to consider when looking at pixel volume: CSS pixels, natural pixels, and screen pixels:

- **CSS pixel volume** is from the CSS perspective of layout. This measure focuses on the bounding boxes into which an image or video could be stretched or squeezed. It also does not take into account the actual file pixels nor the screen display pixels.

- **Natural pixels** refer to the logical pixels represented in a file. If you were to load this image in GIMP or Photoshop, the pixel file dimensions would be the natural pixels.

- **Screen pixels** refer to the physical electronics on the display. Prior to mobile phones and modern high-resolution displays, there was a 1:1 relationship between CSS pixels and LED points on a screen. However, because mobile devices are held closer to

> the eye, and laptop screens are closer than the old mainframe terminals, modern screens have a higher ratio of physical pixels to traditional CSS pixels. This ratio is referred to as the device pixel ratio, or colloquially referred to as Retina™ displays.

In the same Web Almanac data we can see that the median web page had a layout that would show (CSS) 0.65 MP. The device used to collect data for the Web Almanac had a (CSS) 0.18 MP display (similar to the Samsung Galaxy S10 above, with CSS 0.26 MP). This implies that images and media are so important that more than 3.6 screens' worth of content is used on the median web page layout. At the 90th percentile, web pages have 25 full displays' worth of content!

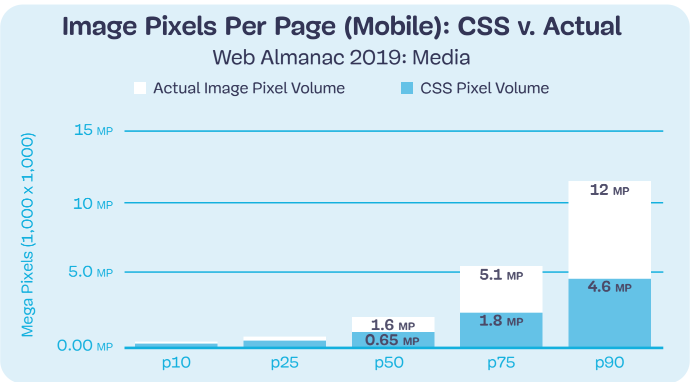



*Image pixels per page (mobile): CSS versus actual.*

Media resources are critical for the user experience. While media are critical for the visual experience, the impact of this high volume of bytes has two side effects.

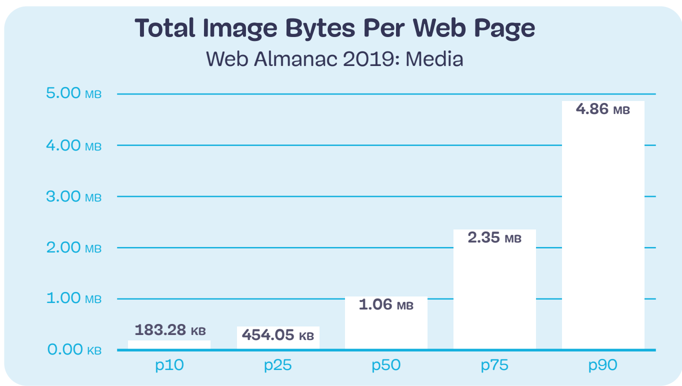
First, the network overhead required to download these bytes can be large, and in cellular or slow network environments (like coffee shop Wi-Fi) can dramatically slow down the page performance.[3] Images represent a lower priority request by the browser but can easily block CSS and JavaScript in the download. This by itself can delay the page rendering. Yet at other times, the image content is the visual cue to the user that the page is ready. Slow transfers of visual content, therefore, can give the perception of a slow web page.

The second impact is on the financial cost to the user. This is often an ignored aspect since it is not a burden on the website owner but a burden to the user. It has been shared anecdotally that some markets, like Japan,[4] see a drop in purchases by students near the end of the month when data caps are reached, and users cannot see the visual content.

Further, the financial cost of visiting these websites in different parts of the world is disproportionate. At the median and 90th percentile, the volume of image bytes is 1 MB and 1.9 MB respectively.

3 https://smashed.by/imageperf
4 https://smashed.by/datacaps

Using WhatDoesMySiteCost.com[5] we can see that the gross national income per capita cost to a user in Madagascar of a single web page load at the 90th percentile would cost 2.6% of daily gross income. By contrast, in Germany this would be 0.3% of daily gross income.




*Total image bytes per web page (mobile).*

This book covers managing and optimizing images to help reduce the bytes and optimize the user experience. It is an important and critical topic for many because it is the creative media that define a brand experience. Optimizing image and video content is a balancing act between applying best practices that can help reduce the bytes transferred over the network and preserving the fidelity of the intended experience.

5 https://smashed.by/gnicost

While the strategies for images, videos, and animations are broadly similar, the specific approaches can be very different. In general, these strategies boil down to:

- **File formats**: using the optimal file format.
- **Responsive**: applying responsive images techniques to transfer only the pixels that will be shown on screen.
- **Lazy loading**: to transfer content only when a human will see it.
- **Accessibility**: ensuring a consistent experience for all people.

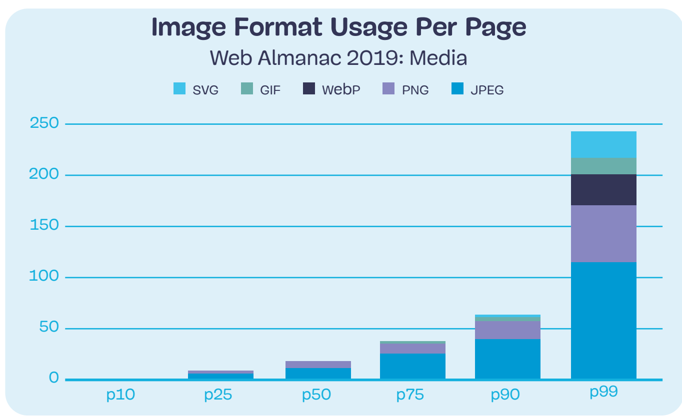
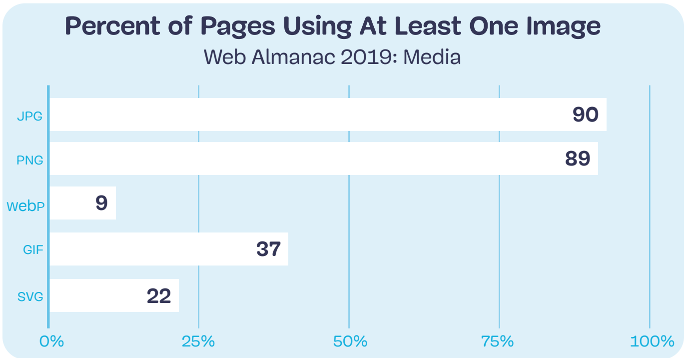



*Image format usage per page.*

These strategies are not new, and many thousands of words in books, blogs, and tutorials have been devoted to expanding awareness on how to reduce the cost and improve the performance of media. Yet, based on the Web Almanac data, there is still a lot of unrealized opportunity to improve the user experience through optimizing images.

Consider the following breakdown of image formats most commonly used in web pages. Each site is unique and the use of image content is not uniform. Some depend on images more than others. Look no further than the home page of google.com and you will see very little imagery compared with a typical news website. Indeed, the median website has




*Percentage of pages using at least one image. (The data collected by the HTTP Archive uses a Chrome browser. This explains why* JPEG 2000*,* JPEG XR*,* HEIC*, and* AVIF *are absent from the results.)*

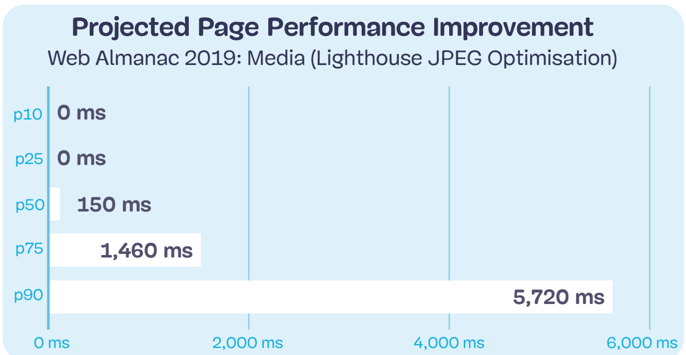
13 images, 61 images at the 90th percentile, and a whopping 229 images at the 99th percentile.

Most striking from this analysis is that the use of WebP does not materialize until the 99th percentile. WebP, as this book describes in detail, is one of the new(er) image formats that can dramatically reduce bytes per pixel and is, therefore, the easiest way to optimize images. While the median page has 9 JPEGs and 4 PNGs, and only in the top 25% of pages GIFs were used, the 90th percentile does not include any use of WebP . Only at the 99th percentile do we see WebP. However, this doesn't report the adoption rate. Pivoting this data, we can see the adoption and use of each format across all web pages.

This helps explain why – even at the 90th percentile of pages – the frequency of WebP is still zero; only 9% of web pages have even one resource. Even 9% might be too generous considering that even one ad or third-party iframe that includes a WebP would be counted.

Adoption of WebP is just one indicator of the pervasiveness of image optimization. Lighthouse[6] is a tool for auditing the quality of user experiences. The Lighthouse data here provides simulated projections for encoding optimization of JPEGs.

6 https://smashed.by/lighthouse




*Projected page performance improvements from image optimization from Lighthouse.*

Just re-encoding progressive JPEG images can save 150 milliseconds at the median and nearly 6 seconds at p90. The time savings from the Lighthouse tests indicate again that there is a lot of untapped potential even with existing images.

But don't be discouraged by this low adoption of image optimization techniques! It would be understandable to feel dismayed that only 9% of the web has adopted this simple image optimization strategy despite WebP being readily available for nearly ten years. Or that many JPEGs are not encoded optimally. Yet, the tools and services available today are making it increasingly easier to adopt and maintain great image optimization techniques.

The good news is that there is a lot of low-hanging fruit that web developers can pick to improve image optimization and increase user engagement For this reason, I am excited about the advances in the tools, knowledge, and services available to all web developers that make it easier to implement image optimizations.

Nearly all web pages use images and video to some degree to enhance the user experience and create meaning. Adopting alternative formats, lazy loading, responsive images, and image optimization can go a long way to lowering the size of media on the web. Throughout this book you will encounter many examples of each of these techniques, and testimonials from different organizations that have adopted these best practices.

You are not alone in this challenge to improve the user experience. We are in this together!

*—Colin Bendell*

# Introduction

Images are an important part of any web experience. They let us convey ideas much faster than text, help us tell powerful stories and engage with our users in ways that few other forms of content can. This is one reason why high-quality images often help increase conversions, raise user engagement, and add broader context to any page. While images are critical for a satisfying visual experience, delivering them to users efficiently isn't easy. The good news is this book will provide you with insights and best practices shared by industry experts to guide you towards success.

Gorgeous high-quality images have a cost. They first have to be downloaded, requiring a network overhead that can be large and, on slow internet connections (think coffee shops or when you're on the go), can significantly slow down how quickly web pages load. Often, image content is a primary visual cue that a page is ready to be used, so slow transfers of images imply the page is slow – think of any page with hero images, be it news articles or product pages.

Unoptimized images require massive amounts of bandwidth because they often have large file sizes. According to the HTTP Archive, 50% of the data transferred to fetch a web page comprises images of various formats.[1] Images also

1 http://httparchive.org/

account for a whopping 2 MB+[2] of the content loaded for websites at the 75th percentile. That's a lot. Slow images can also block CSS and JavaScript owing to network contention, which itself can delay page rendering.

Adding images to a page and making existing images larger have long been proved[3] to increase conversion rates. Research also shows that the quality and quantity of those images matters. Research undertaken by SOASTA and Google[4] in 2016 highlighted that images were the second highest predictor of conversions, with the best pages having 38% fewer images than those that didn't convert. The reason for this might be the performance impact of all those images. Faster websites often have higher conversion rates, so investing in an efficient compression strategy to minimize bloat from high-quality images is important. There's plenty of room for us to collectively optimize images better.

In the first part of the book, "**Image Quality and Performance**," we'll explore the challenges involved in defining image quality and improving how images perform. You will get a foundation of knowledge in how image compression works to help you squeeze out each unnecessary byte from your visuals.

2 https://smashed.by/bytesperpage
3 https://smashed.by/perfplanet
4 https://smashed.by/soasta

Many image formats allow you to control quality. In formats supporting lossy compression (where some image data is lost), you can control the compression by setting a value, usually between 0 and 100. Lower values mean greater compression yielding potentially lower quality images. Just how much lower, and whether this change can be noticed by the human eye, really depends on the type of image. Many developers err on the side of caution when setting image quality because they're worried about degrading visual quality.

Effective compression reduces the size of an image while still delivering a crisp level of detail . Adjusting quality levels doesn't have to reduce visual quality very much, and you'll leave this section understanding the trade-offs when it comes to image quality and how to fine-tune encoding settings to get the most out of image optimization tools. Smaller image sizes bring smaller, faster downloads to help retain users' attention, but also reduce the cost of storing and transferring these images.

This part of the book will also discuss color management. A mastery of color allows you to control how image colors are represented across a range of different contexts, like phone screens, monitors, cameras, and printers. You'll also be able to adjust and simplify an image's color palette to further reduce file sizes.

Part 2 looks closely at the most widespread "**Current Image Formats**" at our disposal: JPEG, PNG, WebP, and SVG.

Different formats are optimal for compressing different kinds of images (detailed photographs, illustrations, decorative content, and so on), and which format you choose has an impact on the final size and experience. Different kinds of images when saved to the same format with the same quality and configuration will have very different file sizes. This is because the content of an image can strongly affect how effective each format is.

Different formats use compression strategies tuned for specific types of content and this section will help you gain a deep understanding of what formats like JPEG, PNG, WebP, and SVG are most optimally used for. This section will also cover the practical tools, tips, and tricks for effectively using each of these formats in production.

Part 3, "**Images in Browsers**," focuses on techniques to improve how images are displayed in modern web browsers.

We'll cover where images fit in responsive design, where your site delivers an optimal visual experience on each user's device, irrespective of screen size or resolution. For an image to adapt appropriately, it will often need to adjust its resolution and sometimes even format, quality, or art direction for the best experience.

Sending the highest-resolution image and hoping the browser will resize it appropriately is a waste of bandwidth and can slow down the user experience. Instead, you'll learn how to prepare images for a variety of resolutions, so devices requesting it only load what's needed.

Sites that load fast prioritize the resources a user needs when they need it. Images are no different, yet most sites load all images even if users won't see them until they scroll down. This is where lazy loading can save the day – and your users' time, bandwidth, and CPU cycles. Lazy loading defers loading images that aren't needed right away. When done right, it can ensure that images not visible to users are never loaded.

We'll cover how to use browsers' native image lazy-loading features, JavaScript libraries for lazy loading (for browsers that don't support it yet), as well as advanced techniques, like displaying low-quality placeholders as final images are loaded in.

Throughout the book, you'll learn how to automate image optimization through modern tools. While these tools are great, staying on top of image optimization best practices and regularly updating your toolchains can be time-consuming. These tools are also often focused on optimizing static images at build time (like logos, or site assets known ahead of time), while your site may also need to support

user-generated content, where users upload often large, uncompressed images from their phones or desktops.

There is where content delivery networks (CDNs) focused on image delivery can be a great option. Using an image CDN can result in 40–80% savings[5] in image file sizes, and many image CDNs support automatic selection of the best quality and format , image transformation, and media management.

This section will cover how to set up a self-managed CDN as well as third-party image CDNs, which offer image optimization and delivery as a service. Both options have their trade-offs, but we'll equip you with the knowledge to pick what makes sense.

In Part 4, we survey "**New Image Formats**" that are emerging online: AVIF, JPEG XL, and HEIF. These next-generation formats typically deliver far better quality and compression than traditional formats. Delivering modern image formats to users whose browsers support them (and fallbacks to those that do not) allows you to optimize for image quality and storage while ensuring images can be viewed by all of your users. One such example is delivering AVIF to modern browsers and WebP or JPEG everywhere else. In this section, we cover the main features, pros, and cons of new image formats.

5 https://smashed.by/savings

The final part moves into "**Further Optimization**" and includes tips and tricks for optimizing Google's Core Web Vitals, adaptive image delivery with Data Saver mode, and a production case study of how Twitter improved its image optimization pipelines at scale.

Core Web Vitals is an initiative by Google to encourage sites to identify opportunities to improve user experience. In this section, we'll cover image-specific guidance to improve your Core Web Vitals to ensure things like your largest content-ful elements load fast and don't cause layout shifts. Google's research shows that when sites meet the Core Web Vitals thresholds, users are 24% less likely to leave a page before any content has been painted.

Users with slow connections deserve a beautiful user experience too. If a user has opted into a browser's Data Saver mode, you can use this as a signal to fine-tune image delivery to serve fewer image bytes so pages still load fast for them.

We'll cover adaptive image delivery using Data Saver, where sites can remove unnecessary images, switch to lower-resolution images, lower-quality images or vid-eos, or simply trigger a "lite" experience by checking the browser's data-savings hints.

Finally, bringing it all together we will take a look at a production case study of Twitter's image optimization pipeline. With a huge user base spread across a number of platforms, form factors, and network conditions, Twitter has evolved its ability to handle image delivery in a number of ways that improve user experience.

We cover device pixel ratio (DPR) capping to provide high-resolution responsive images that account for details the human eye can see, without serving more bytes when it can't. We'll also look at how Twitter optimizes image uploads, supports special cases like pixel art, and implements a Data Saver mode for maximal image savings.

Images help us tell a story and engage with our users at a deep level. They also have a low barrier to access, with anyone being able to drop an `<img>` element into a page and begin building more beautiful experiences. Let's begin our journey to faster-loading images by first looking at some of the superpowers that have recently come to `<img>` – you might be surprised at how much the browser can now do out of the box.

CHAPTER ONE

# The Humble <img> Element

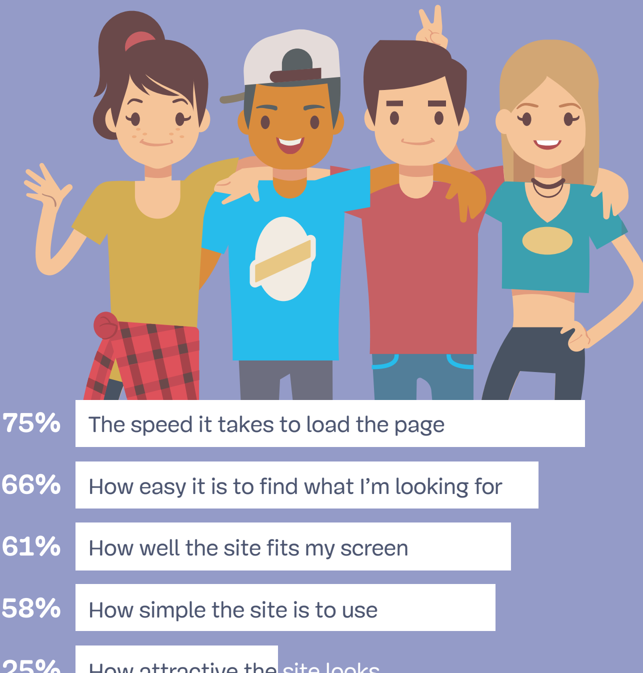
The humble `<img>` element has gained some superpowers over the years. Given how central it is to image optimization on the web, let's catch up on what it can do.

## The Basics

To place an image on a web page, we use the `<img>` element. This is an empty element – it has no closing tag – requiring a minimum of one attribute to be helpful: `src`, the source. If an image is called *donut.jpg* and it exists in the same location as your HTML document, it can be embedded as follows:

```
<img src="donut.jpg">
```

To ensure our image is accessible, we add the `alt` attribute. The value of this attribute should be a textual description of the image, and is used as an alternative to the image when it can't be displayed or seen; for example, a user accessing




# HOW IMPORTANT IS SPEED?

Users rated speed highest in the UX hierarchy according to Google's Speed Matters Vol. 3

your page via a screen reader. The above code with an `alt` specified looks as follows:

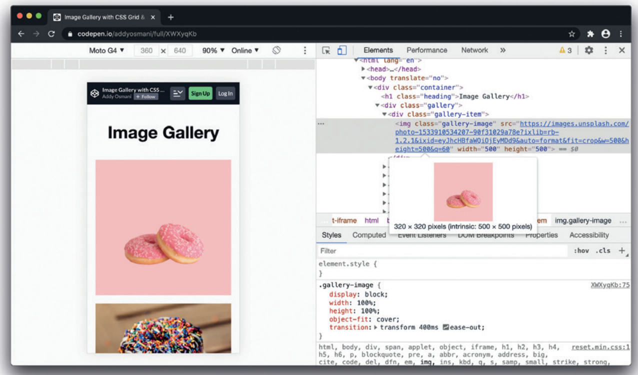
```
<img src="donut.jpg"
     alt="A delicious pink donut.">
```

Next, we add `width` and `height` attributes to specify the width and height of the image. The dimensions of an image can usually be found by looking at this information via your operating system's file explorer (**Cmd + I** on macOS).

```
<img src="donut.jpg"
     alt="A delicious pink donut."
     width="400"
     height="400">
```

When width and height are specified on an image, the browser knows how much space to reserve for the image until it is downloaded. Forgetting to include the image's dimensions can cause layout shifts, as the browser is unsure how much space the image will need.

Modern browsers now set the default aspect ratio of images based on an image's `width` and `height` attributes, so it's valuable to set them to prevent such layout shifts.




*Hovering over an image in the Chrome DevTools **Elements** panel displays the dimensions of the image as well as the image's intrinsic size.*

## Swapping Out Images

What about switching image resolution? A standard `<img>` only allows us to supply a single source file to the browser. But with the `srcset` and `sizes` attributes we can provide many additional source images (and hints) so the browser can pick the most appropriate one. This allows us to supply images that are smaller or larger.

```
<img src="donut-800w.jpg"
     alt="A delicious pink donut."
     width="400"
```

```
height="400"
srcset="donut-400w.jpg 400w,
        donut-800w.jpg 800w"
sizes="(max-width: 640px) 400px,
       800px">
```

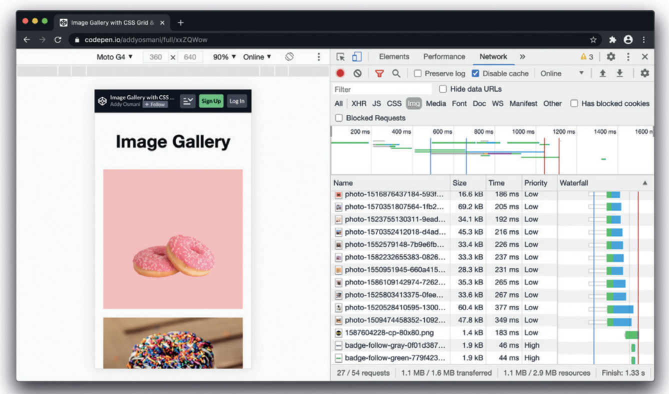
The `srcset` attribute defines the set of images the browser can select from, as well as the size of each image. Each image string is separated by a comma and includes: a source filename (`donut-400w.jpg`); a space; and the image's intrinsic width specified in pixels (`400w`), or a pixel density descriptor (`1x`, `1.5x`, `2x`, etc.).

The `sizes` attribute specifies a set of conditions, such as screen widths, and what image size is best to select when those conditions are met. Above, `(max-width:640px)` is a media condition asking "if the viewport width is 640 pixels or less," and `400px` is the width the image is going to fill when the media condition is true.

Even those images which are responsive (that is, sized relative to the viewport) should have width and height set. In modern browsers, these attributes establish an aspect ratio that helps prevent layout shifts, even if the absolute sizes are overridden by CSS. (Chapter 11 covers responsive images.)

## Image Loading

What about offscreen images that are not visible until a user scrolls down the page? In the example below, all the images on the page are “eagerly loaded” (the default in browsers today), causing the user to download 1.1 MB of images. This can cause users’ data plans to take a hit in addition to affecting performance.



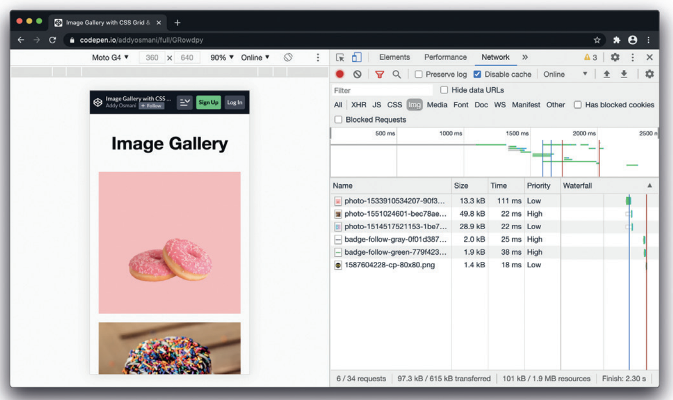

*An image gallery eagerly loading all the images it needs up front, as shown in the Chrome DevTools* ***Network*** *panel. 1.1 MB of images have been downloaded, despite only a small number being visible when the user first lands on the page.*

Using the `loading` attribute on `<img>`, we can control the behavior of image loading. `loading="lazy"` lazy-loads images, deferring them loading until they reach a calculated distance from the viewport. `loading="eager"` loads images

right away, regardless of their visibility in the viewport. `eager` is the default and can be ignored (that is, just use `<img>` for eager loading).

Below is an example of lazy-loading an `<img>` with a single source:

```
<img src="donut.jpg"
     alt="A delicious pink donut."
     loading="lazy"
     width="400"
     height="400">
```




*An image gallery using native image lazy-loading on images outside of the viewport. As seen in the Chrome DevTools **Network** panel, the page now only downloads the bare minimum of images users need up front. The rest of the images are loaded in as users scroll down the page.*

With native `<img>` lazy-loading, the earlier example now downloads only about 90 KB of images! Just adding `loading="lazy"` to our offscreen images has a huge impact.

Lazy loading also works with images that include `srcset`, as `<img>` is what drives image loading:

```
<img src="donut-800w.jpg"
     alt="A delicious donut"
     width="400"
     height="400"
     srcset="donut-400w.jpg 400w,
             donut-800w.jpg 800w"
     sizes="(max-width: 640px) 400px,
            800px"
     loading="lazy">
```

We'll cover lazy loading in full in chapter 14.

## Image Decoding

Browsers need to decode the images they download in order to turn them into pixels on your screen. However, how browsers handle deferring images can vary. At the time of writing, Chrome and Safari present images and text together – synchronously – if possible. This looks correct visually, but images have to be decoded, which can mean text isn't

shown until this work is done. The `decoding` attribute on `<img>` allows you to signal a preference between synchronous and asynchronous image decoding.

```
<img src="donut-800w.jpg"
     alt="A delicious donut"
     width="400"
     height="400"
     srcset="donut-400w.jpg 400w,
             donut-800w.jpg 800w"
     sizes="(max-width: 640px) 400px,
            800px"
     loading="lazy"
     decoding="async">
```

`decoding="async"` suggests it's OK for image decoding to be deferred, meaning the browser can rasterize and display content without images while scheduling an asynchronous decode that is off the critical path.

As soon as image decoding is complete, the browser can update the presentation to include images. `decoding="sync"` hints that the decode for an image should not be deferred, and `decoding="auto"` lets the browser do what it determines is best. (There's more on the `decoding` attribute in chapter 5.)

## Placeholders

What if you would like to show the user a placeholder while the image loads? The `background-image` CSS property allows us to set background images on an element, including the `<img>` tag or any parent container elements. We can combine `background-image` with `background-size: cover` to set the size of an element's background image and scale the image as large as possible without stretching the image.

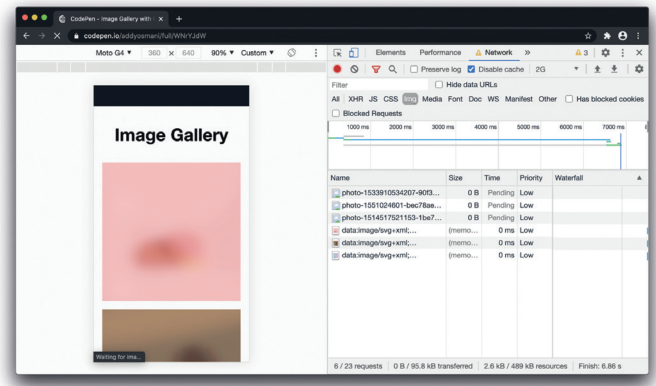
Placeholders are often inline, Base64-encoded data URLs which are low-quality image placeholders (LQIP) or SVG image placeholders (SQIP). This allows users to get a very quick preview of the image, even on slow network connections, before the sharper final image loads in to replace it.

```
<img src="donut-800w.jpg"
     alt="A delicious donut"
     width="400"
     height="400"
     srcset="donut-400w.jpg 400w,
             donut-800w.jpg 800w"
     sizes="(max-width: 640px) 400px,
            800px"
     loading="lazy"
```

```
    decoding="async"
    style="background-size: cover;
           background-image:
              URL(data:image/svg+xml;base64,[svg
text]);">
```

Note: Given that Base64 data URLs can be quite long, `[svg text]` is denoted in the example above to improve readability.

With an inline SVG placeholder, here is how the example from earlier now looks when loaded on a very slow connection. Notice how users are shown a preview right away prior to any full-size images being downloaded:




*Images loaded on a simulated slow connection, displaying a placeholder approximating the final image as it loads in. This can improve perceived performance in certain cases.*

Chapter 12 has much more on progressive rendering techniques, including placeholder images.

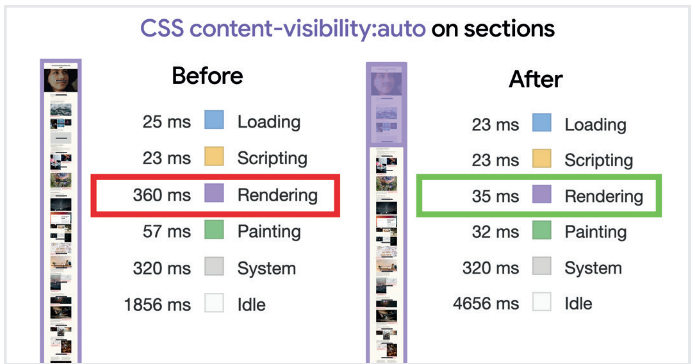
## Lazy-Render Offscreen Content

Next, let's discuss the CSS `content-visibility` property, which allows the browser to skip rendering, layout, and paint for elements until they are needed. This can help optimize page load performance if a large quantity of your page's content is offscreen, including content which uses `<img>` elements.

```
section {
  content-visibility: auto;
}
```

The `content-visibility` property[1] can take a number of values; auto is the one that offers performance benefits. Sections of the page with `content-visibility:auto` get containment for layout, paint, and style. Should the element be offscreen, it would also get size containment.

1 https://web.dev/content-visibility/




*When chunking up a page into sections with* `content-visibility:auto`*, developers have observed a 7–10x improvement in rendering times as a result. Note the reduction in rendering times above of 937ms to 37ms for a long HTML document.*

Browsers don't paint the image content for `content-visibility` affected images, so this approach may introduce some savings.

```
section {
  content-visibility: auto;
  contain-intrinsic-size: 700px;
}
```

One option is to pair `content-visibility` with `contain-intrinsic-size`, which provides the natural size of the element if it is affected by size containment. The 700px value in this example approximates the width and height of each chunked section.

## Maintain a Consistent Aspect-Ratio

The aspect ratio of an image is the ratio of its width to its height. This is often represented by two numbers separated by a colon (such as 4:3 or 16:9). Maintaining a consistent aspect ratio can be important in responsive web design where the dimensions of images can vary and introduce layout shifts depending on how much space is available in the page.

In our image gallery, we might wish to create responsive space for images that vary by dimension, are in more complex elements like cards, or require a placeholder container to avoid layout shifts when the images load and occupy space.

Historically, developers have used the padding-top hack to maintain aspect ratio using an image's width. This involves

using two containers: a parent container, and a child container that gets absolutely positioned. The aspect ratio is then computed as a percentage for the `padding-top` value.

For example, a 16:9 aspect ratio = 9 ÷ 16 = 0.5625 = CSS `padding-top: 56.25%`. For the following container:

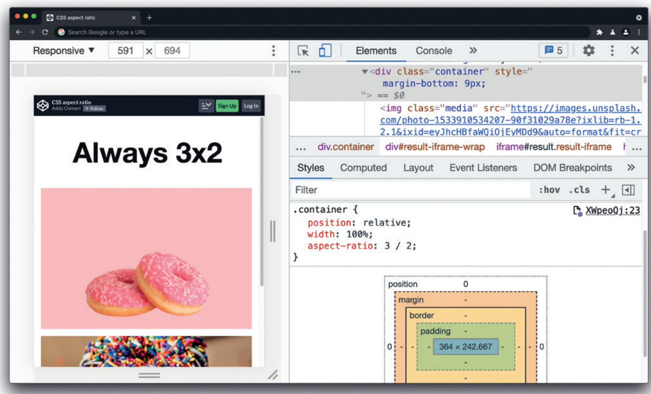
```
<div class='container'>
     <img style='position: absolute; top:0;'>
</div>
```

This is the CSS for the padding-top hack to maintain aspect ratio:

```
.container {
  position: relative;
  padding-top: 56.25%; /* Aspect ratio of 16:9  */
  width: 100%;
}
```

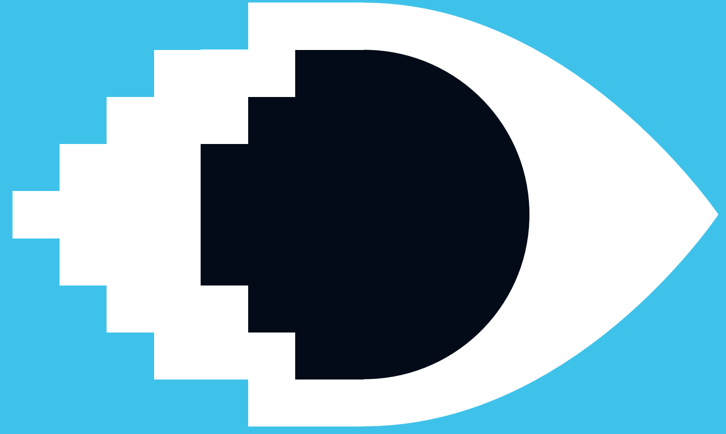
Thanks to the new CSS `aspect-ratio` property,[2] a more intuitive alternative to the padding-top hack is now available.[3] This enables replacing `padding-top:56.25%` with `aspect-ratio:16/9` to clearly specify the width to height ratio.

2 https://web.dev/aspect-ratio/
3 https://css-tricks.com/aspect-ratio-boxes/




*The new* CSS `aspect-ratio` *property, available in modern browsers, is clearer than the padding-top hack and doesn't involve more manual calculation for positioning. In the example above, a 3:2 aspect ratio = 2 ÷ 3 = 0.66666 =* `padding-top:66.67%`*. Thanks to the* CSS `aspect-ratio` *property, this can just be defined as aspect-ratio: 3 / 2.*

Throughout this book, we will cover advanced image optimization techniques, as well as how to best use elements like `<img>` and `<picture>` to make your images on the web shine. Now that we've covered the foundations of the modern `<img>` tag, let's turn our attention to understanding image quality and how it affects web performance.



## Part One

# Image Quality and Performance

CHAPTER 2 Optimizing Image Quality . . 43
CHAPTER 3 Comparing Image Formats . . 54
CHAPTER 4 Color Management . . . . . . . 68
CHAPTER 5 Image Decoding Performance . . . . . . . . . . . . . . 80
CHAPTER 6 Measuring Image Performance . . . . . . . . . . . . . 100

CHAPTER 2

# Optimizing Image Quality

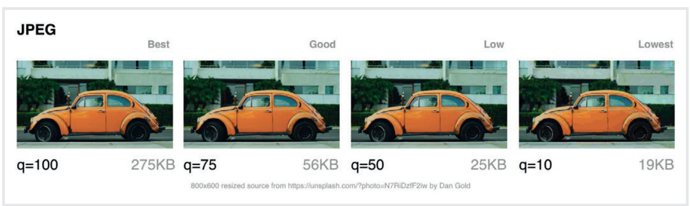
Most optimization tools allow you to set the level of quality you're happy with. Lower quality reduces file size but can introduce artifacts, halos, or blocky degrading.

Squoosh.app[1] is a free, web-based tool that reduces image size through modern image compression techniques. It supports many of the formats discussed in this book. If you need to compress multiple images, ImageOptim[2] and the ImageOptim plug-in for Sketch[3] are also both free and are equally excellent.

*Tools like Squoosh and ImageOptim can compress images with savings of over 50% without a perceivable loss in quality.*

1 https://squoosh.app
2 https://imageoptim.com
3 https://smashed.by/optimsketch

The quality index[4] you choose informs the level of compression that the optimization tool will use.




*JPEG compression artifacts can be increasingly perceived as we shift from best quality to lowest.*

Perceived image quality is subjective and depends on things such as image content, screen size, resolution, and the person observing the image. It's possible we sometimes overestimate the image quality that our users need. For best performance results, remember that less is more.

When choosing the quality setting, consider which quality bucket your images fall into:

- **Best quality**: when quality matters more than bandwidth. This may be because the image has high prominence in your design or is displayed at full resolution.
- **Good quality**: when you care about shipping smaller file sizes but don't want to negatively impact image quality too much. Users still care about some level of image quality.

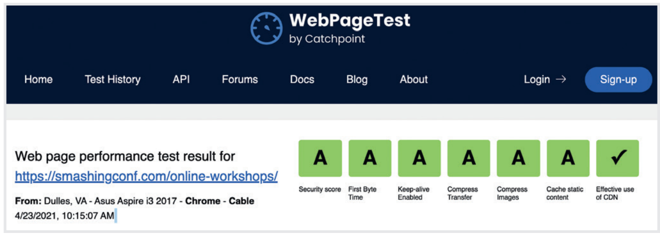
4 https://smashed.by/qualityindex

- **Low quality**: when you care enough about bandwidth that image degradation is OK. These images are suitable for spotty or poor network conditions.

- **Lowest quality**: bandwidth savings are paramount. Users want a decent experience but will accept pretty degraded images for the benefit of pages loading more quickly.

## Audit Your Images

Perform a site audit through WebPageTest[5] and it will highlight opportunities to better optimize your images (see "Compress Images").




*WebPageTest supports auditing for image compression via the "Compress Images" section.*

The "Compress Images" section of a WebPageTest report lists images that can be compressed more efficiently and the estimated file-size savings of doing so.

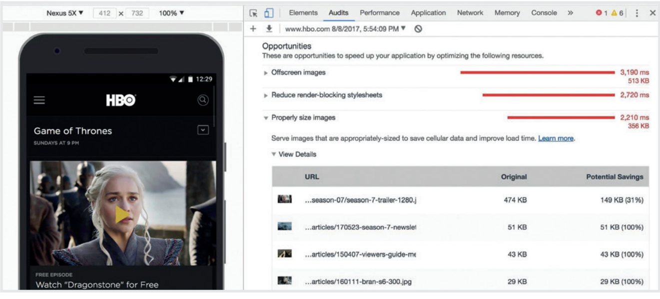
5 https://www.webpagetest.org/

**Compress Images: 58/100**

1,118.4 KB total in images, target size = 644.8 KB - potential savings = **473.5 KB**

*Image compression recommendations from WebPageTest*

Use Lighthouse[6] audits for performance best practices. It includes audits for image optimization that suggest which images could be compressed further and which images are off-screen and could be lazy-loaded. In Chrome 60 and above, Lighthouse powers the Audits panel[7] in Chrome DevTools. (Lighthouse audits are discussed in more detail in chapter 6.)




*Lighthouse audit for HBO.com, displaying image optimization recommendations.*

Other popular performance auditing tools are PageSpeed Insights[8] and Website Speed Test[9] by Cloudinary which includes a detailed image analysis audit.

---

6 https://smashed.by/lighthouse
7 https://smashed.by/auditspanel
8 https://smashed.by/pagespeedinsights
9 https://smashed.by/speedtest

## Measuring Image Quality

The image quality indexes you see in optimization tools are approximations of human perception. And quality index in one tool can be very different to quality index in another, and often they cannot be compared directly. Some tools, like ImageMagick,[10] for example, use a 0–100 scale, while Photoshop uses a 0–12 scale.

Quality index doesn't map directly to compression levels, and the mapping equations are different for different image formats. If you set the quality of a JPEG and a PNG image to 75 in ImageMagick, the compression levels and perceived image quality will differ.

To make a fair comparison[11] between images, you can't rely on quality index alone. Be sure to compare images in the same format, convert both from high-quality sources and pay attention to encoder settings.

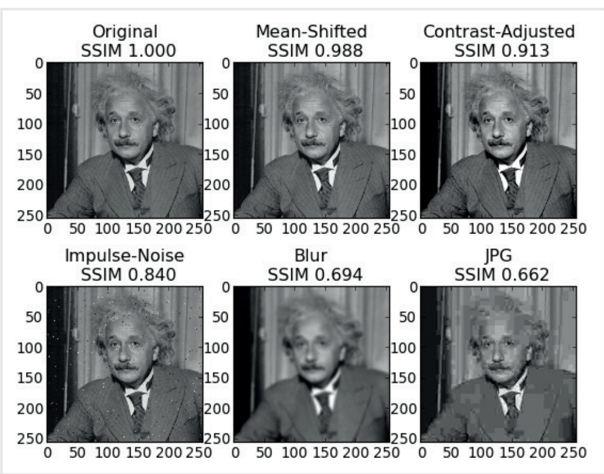
Quality measurements in encoding tools often aren't very consistent between images or proportional to quality perceived by humans. There is no ideal image quality measurement, however, but some, such as the structural similarity index measure (SSIM) and Butteraugli, stand out.

10 https://www.imagemagick.org/
11 https://smashed.by/faircomparison

## SSIM

The structural similarity index measure[12] is a method for measuring the similarity between two images. It does not judge which of the two is better; it just tells us how far away in quality an image is from its original reference image.

The SSIM algorithm considers three key components of our visual system: luminance, contrast, and structure.




*Increasing degrees of distortion and associated SSIM values.*

Open-source tools, like DSSIM[13] by Kornel Lesiński, and the Node.js module img-ssim[14] are available for comparing images using SSIM.

12 https://smashed.by/structuralsimilarity
13 https://smashed.by/dSSIM
14 https://smashed.by/imagesSIM

## BUTTERAUGLI

Butteraugli[15] is a tool developed by Google for measuring perceived differences between images. It estimates the point when a person might notice visual image degradation (the psychovisual similarity) between two images, and gives a score for the images that is reliable in the domain of barely noticeable differences. Butteraugli provides both a scalar score as well as a spatial map of the level of difference.

While SSIM looks at the aggregate of errors from an image, Butteraugli looks at the worst part.

*Butteraugli validating an image of a parrot.*

15 https://smashed.by/butter

Above is an example that used Butteraugli to find the minimal JPEG quality threshold before visual degradation was bad enough for a user to notice something wasn't clear. It resulted in a 65% reduction in total file size.

In practice, you would define a target goal for visual quality, and then run through a number of different image optimization strategies, looking at your Butteraugli scores, before choosing something that fits the best balance of file size and level.

```
~/projects/butteraugli/dist

~/projects/butteraugli/dist master*
❯ ./butteraugli fruit.jpg fruit-guetzli.jpg
1.900138

~/projects/butteraugli/dist master*
❯ ./butteraugli aziz-acharki.jpg aziz-acharki-guetzli.jpg
2.091388

~/projects/butteraugli/dist master* 37s
❯
```

*Butteraugli being run from the command line.*

All in all, it took me about half an hour to set up Butteraugli locally after installing Bazel[16] and getting a build of the C++ sources to correctly compile on my Mac. Using it is relatively straightforward: specify the two images to

16 https://smashed.by/bazel

compare (a source and a compressed version) and it will give you a score to work from.

A comment[17] from a Guetzli project member suggests Guetzli (a JPEG encoder from Google; see chapter 7 on JPEG for more details) scores best on Butteraugli, worst on SSIM, and MOZJPEG (Mozilla's JPEG encoder) scores about as well on both. This is in line with the research I've put into my own image optimization strategy. I run Butteraugli and a Node module like img-ssim over images comparing the source with their SSIM scores before and after Guetzli and MOZJPEG processing.

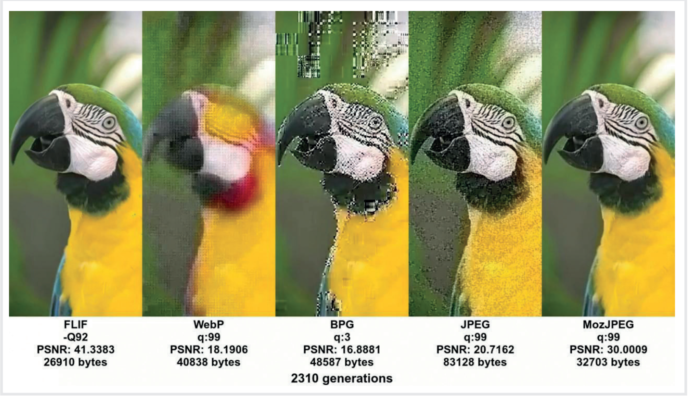
## Avoid Recompressing Images with Lossy Codecs

Recompressing images has consequences. For best results, always compress from the original image.

Let's say you take a JPEG that's already been compressed with a quality of 60. If you recompress this image with lossy encoding,[18] it will look worse. Each additional round of compression is going to introduce generational loss – information will be lost and compression artifacts will start to build up – even if you're recompressing at a high quality setting.

17 https://smashed.by/guetzliissue
18 https://smashed.by/lossy

To avoid this trap, set the lowest good quality you're willing to accept in the first place and you'll get maximum file savings from the start. Re-encoding a lossy file will almost always give you a smaller file, but this doesn't mean you're getting as much quality out of it as you may think.




*Generational loss when re-encoding an image multiple times.*

The example above,[19] from Jon Sneyers' "Why JPEG is like a photocopier,"[20] shows the generational loss impact of recompression using several formats. You may have run into this problem when saving (already compressed) images from social networks and re-uploading them, causing recompression and quality-loss buildup.

MozJPEG (perhaps accidentally) has a better resistance to recompression degradation thanks to trellis quantization.[21]

19 https://smashed.by/genloss
20 https://smashed.by/photocopier
21 https://smashed.by/trellis

(Quantization is the process of mapping continuous infinite values, like colors or light levels, to a smaller set of discrete finite values, thereby reducing the amount of data.) Instead of compressing all discrete cosine transform (DCT) values as they are exactly, it can check close values within a +1 to −1 range to see if similar values compress to fewer bits. (DCT is a process that compresses images by breaking them into different visual frequencies.)

TRIVIA ■ ImageMagick is often recommended for image optimization. It's a fine tool, but its output generally requires further optimization, and other tools can offer better output. ImageMagick has also historically had security vulnerabilities you may want to be aware of. We recommend trying libvips instead. It is lower-level, however, and requires more technical skill to use.

Lossy FLIF[22] (free lossless image format) has a hack similar to lossy PNG in that prior to (re)compression, it can look at the data and decide what to throw away.

When editing your source files, store them in a lossless format like PNG or baseline TIFF v6, so you preserve as much quality as possible. Your build tools or image compression service can then output the compressed version you serve to users with minimal loss in quality.

22 https://smashed.by/flif

CHAPTER 3

# Comparing Image Formats

It is easy to declare that WebP is 30% smaller[23] than JPEG. This is a great headline, but it glosses over real-world experiences. In truth, each new format has a range of effectiveness if you compare and align on a consistent "quality" of experience. There are two fallacies when converting and comparing image formats:

1. That *quality* is a consistent term that means the same thing across all formats.

2. That your eyes have the same composition of photoreceptor cells (cones and rods) and that your perception of "same" and "different" is the same as everyone else's.

To address the second fallacy, we can use math to compare the differences between images. Some of the more common algorithms that can produce an experience score include peak signal-to-noise ratio (PSNR), structural dissimilarity (DSSIM), Butteraugli, and SSIMULACRA[24] (developed by Cloudinary).

Each comparison algorithm is slightly different and focuses on different aspects of how humans perceive images.

23 https://smashed.by/webp
24 https://smashed.by/SSIMulacra

Traditionally, these algorithms are color-blind and focus on the structure of pixels in *luma* (the brightness in an image). Butteraugli and SSIMULACRA differ by attempting to also consider the *chroma* (color) channels.

In this section we will explore the challenge of defining quality and setting expectations when converting between formats. As with every technology involving humans, it's complicated and messy. The data provided in this chapter comes from an image comparison study conducted by Cloudinary that was used to tune its own algorithms for better image optimization.

## Image Context

Choosing the optimal image format for lowest bytes has three dimensions:

- Does the format have the functionality required (such as transparency or animation)?
- Can the target audience view the file?
- Is the format effective at optimizing this specific image content?

# LESS IS MORE

Is that image really needed? Fewer images can create more conversions according to research by Google/SOASTA

Each format uses algorithms and compression, and quantization strategies that are tuned for specific types of image content. Additionally, each format has many different switches and features that can change the total number of bytes and final experience of the image. This is why two different images, saved to the same image format with the same settings will have different byte sizes. The content and context of the image will dramatically impact the effectiveness of each format.

To distill this, there are two dimensions that predominantly affect the effectiveness of formats:

1. Number of colors and color depth: grayscale, 8-bit, 10-bit wide gamut, etc.

2. Photograph versus illustration: how much of the image is generated by a computer or a camera's image sensor?

For example, an image with hard, straight lines from generated text will appear blurry when chroma subsampling is applied. (We'll look more closely at subsampling in chapter 7, about the JPEG format.) A pastoral scene with mountains and trees with the text "Top 10 Camping Destinations" bonded on top will appear blurry if lossy WebP is used. Comparing a JPEG and WebP is not appropriate in this case, since the WebP format does not have the ability

to produce full chroma images, while JPEG can. An image of just the mountains and trees – without the text – would be a fair comparison.

## JPEG Quality !== JPEG Quality

The quality factor used in JPEG can create a lot of confusion. For convenience, each JPEG encoder exposes an easy to understand number that internally aligns to a quantization matrix definition. However, this means that JPEG quality 80 with libjpeg-turbo (a JPEG encoder) will not use the same definition as quality factor 80 in MozJPEG. In both of these cases the index of 0 to 100 is not a grade or a consistent score, but rather 100 placeholders that internally map to a 32×32 quantization matrix.

SUCCESS STORIES ■ Yelp investigated potential optimizations on image file size it could apply without loss of quality. It found that “switching to MozJPEG was responsible for 13.8% of the savings,” which helped make “the website faster for users and saved terabytes a day in data transfer.” (June 2017)

These factors are generally arranged in increasing aggressiveness as you go down to 0. Even between JPEG encoders, the quality factor is not consistent.

## Comparing Formats and Aligning Experience Scores: A Study from Cloudinary

Cloudinary conducted a longitudinal study comparing different image formats with a large corpus of over 10 million images including content from open-source corpora. In this study, each image was saved with all the different quality options and then matched up with SSIMULACRA scores.

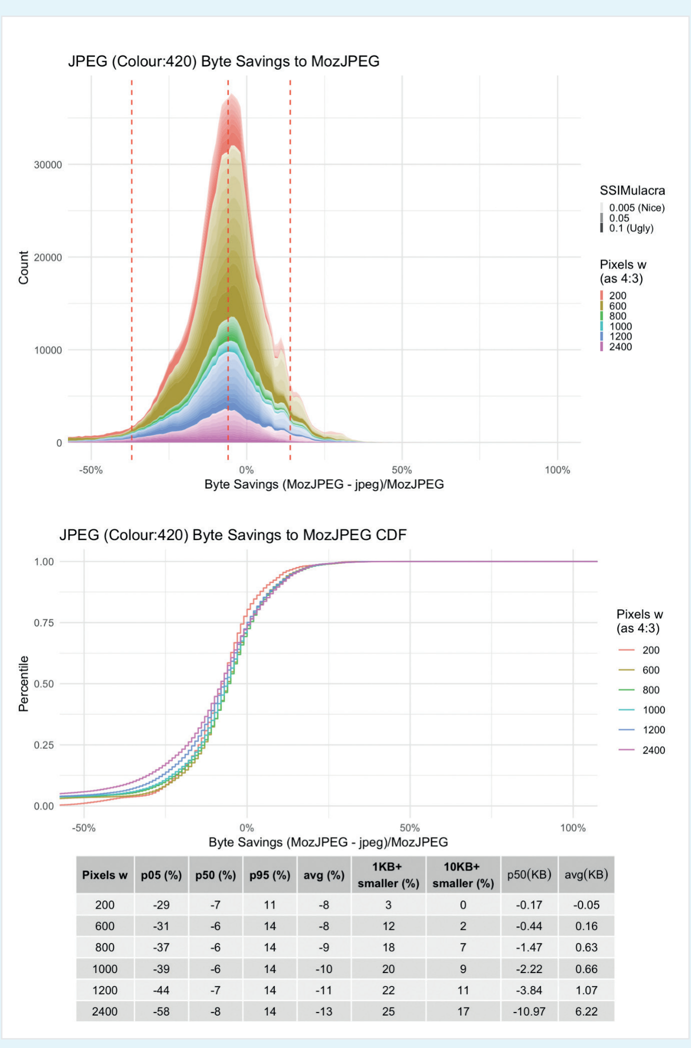
The smallest file that matches the SSIMULACRA experience is then compared from one format to another.

### LIBJPEG-TURBO VS. MOZJPEG

At the 95th percentile, MOZJPEG produced an image that was 15% smaller than the same experience JPEG produced using libjpeg-turbo. Yet, at the median (p50), the MOZJPEG file was slightly larger.

Note: Negative numbers indicate the file size got larger comparatively, meaning that the MOZJPEG file was larger than the libjpeg-turbo equivalent. Positive numbers means the file was smaller in MOZJPEG.

For this reason, when comparing image formats you must first compare all the different JPEG variants for an expe-




| Pixels w | p05 (%) | p50 (%) | p95 (%) | avg (%) | 1KB+ smaller (%) | 10KB+ smaller (%) | p50(KB) | avg(KB) |
|---|---|---|---|---|---|---|---|---|
| 200 | -29 | -7 | 11 | -8 | 3 | 0 | -0.17 | -0.05 |
| 600 | -31 | -6 | 14 | -8 | 12 | 2 | -0.44 | 0.16 |
| 800 | -37 | -6 | 14 | -9 | 18 | 7 | -1.47 | 0.63 |
| 1000 | -39 | -6 | 14 | -10 | 20 | 9 | -2.22 | 0.66 |
| 1200 | -44 | -7 | 14 | -11 | 22 | 11 | -3.84 | 1.07 |
| 2400 | -58 | -8 | 14 | -13 | 25 | 17 | -10.97 | 6.22 |

*libjpeg-turbo vs. MozJPEG.*

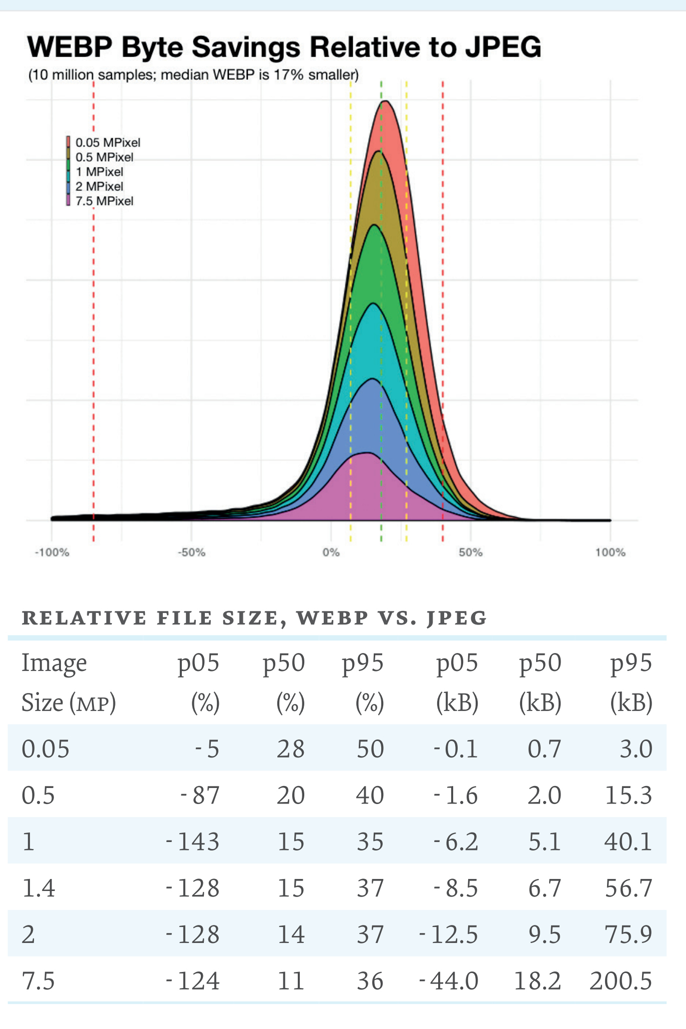
rience score. Unfortunately, most comparisons are made simply using the older libjpeg libraries and do not compare the more modern JPEG encoders like MOZJPEG.

### OTHER FORMATS: JPEG 2000, WEBP, HEIF

Similar claims of superiority have been made for each new format. From this study, however, we can see that there are clearly some scenarios where each format is better, and other scenarios where the results can be larger compared to an equivalent JPEG.

In the comparisons below, each format is compared with the best JPEG file for the experience. For each experience score, the smallest JPEG (libjpeg-turbo or MOZJPEG) is compared with the smallest equivalent for the other formats.

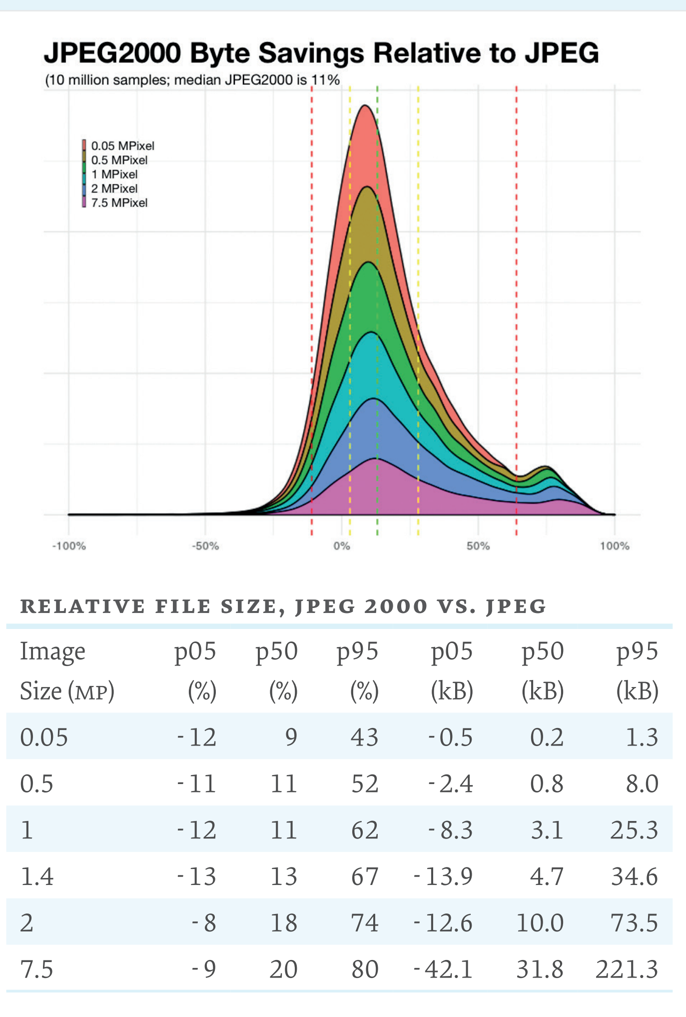
Note: Also worth mentioning is that this comparison uses pure photographic images where chroma subsampling is not a concern. Similar comparisons can be done where subsampling should not be used. However, this immediately disqualifies WebP and HEIF (we'll look at them in later chapters).




RELATIVE FILE SIZE, WEBP VS. JPEG

| Image Size (MP) | p05 (%) | p50 (%) | p95 (%) | p05 (kB) | p50 (kB) | p95 (kB) |
|---|---|---|---|---|---|---|
| 0.05 | -5 | 28 | 50 | -0.1 | 0.7 | 3.0 |
| 0.5 | -87 | 20 | 40 | -1.6 | 2.0 | 15.3 |
| 1 | -143 | 15 | 35 | -6.2 | 5.1 | 40.1 |
| 1.4 | -128 | 15 | 37 | -8.5 | 6.7 | 56.7 |
| 2 | -128 | 14 | 37 | -12.5 | 9.5 | 75.9 |
| 7.5 | -124 | 11 | 36 | -44.0 | 18.2 | 200.5 |

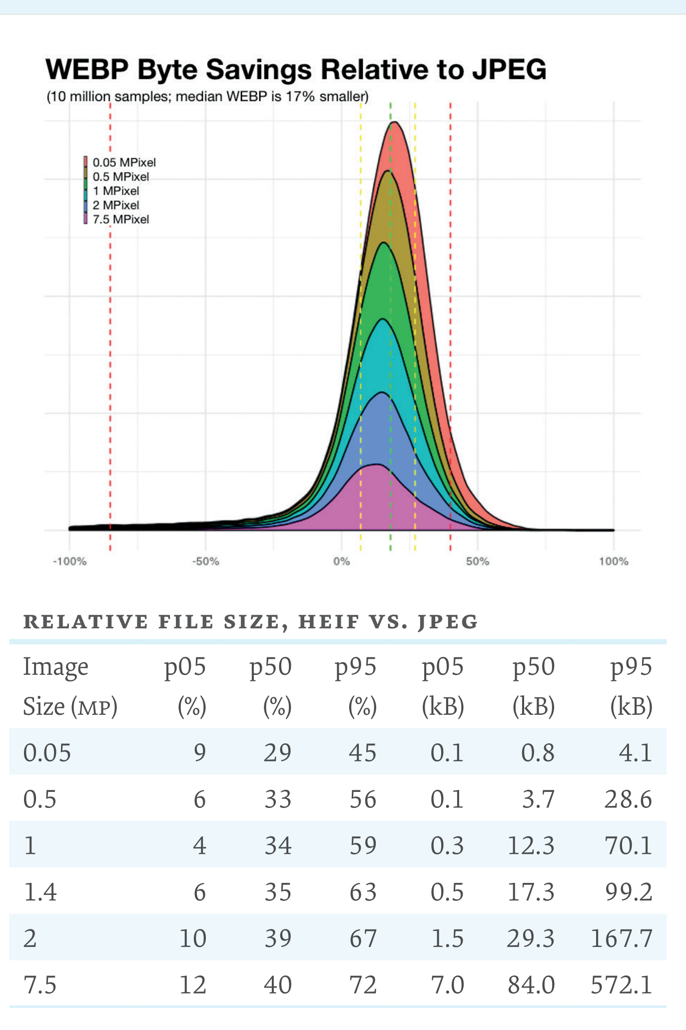
*WebP vs. *JPEG.*




**RELATIVE FILE SIZE, JPEG 2000 VS. JPEG**

| Image Size (MP) | p05 (%) | p50 (%) | p95 (%) | p05 (kB) | p50 (kB) | p95 (kB) |
|---|---|---|---|---|---|---|
| 0.05 | -12 | 9 | 43 | -0.5 | 0.2 | 1.3 |
| 0.5 | -11 | 11 | 52 | -2.4 | 0.8 | 8.0 |
| 1 | -12 | 11 | 62 | -8.3 | 3.1 | 25.3 |
| 1.4 | -13 | 13 | 67 | -13.9 | 4.7 | 34.6 |
| 2 | -8 | 18 | 74 | -12.6 | 10.0 | 73.5 |
| 7.5 | -9 | 20 | 80 | -42.1 | 31.8 | 221.3 |

*JPEG 2000 vs. *JPEG.*




RELATIVE FILE SIZE, HEIF VS. JPEG

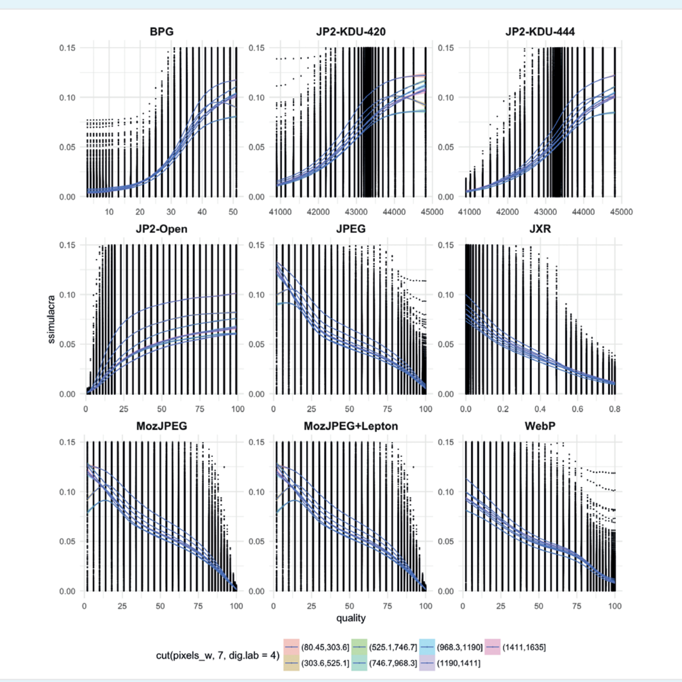
| Image Size (MP) | p05 (%) | p50 (%) | p95 (%) | p05 (kB) | p50 (kB) | p95 (kB) |
|---|---|---|---|---|---|---|
| 0.05 | 9 | 29 | 45 | 0.1 | 0.8 | 4.1 |
| 0.5 | 6 | 33 | 56 | 0.1 | 3.7 | 28.6 |
| 1 | 4 | 34 | 59 | 0.3 | 12.3 | 70.1 |
| 1.4 | 6 | 35 | 63 | 0.5 | 17.3 | 99.2 |
| 2 | 10 | 39 | 67 | 1.5 | 29.3 | 167.7 |
| 7.5 | 12 | 40 | 72 | 7.0 | 84.0 | 572.1 |

*HEIF vs. *JPEG.*

A few observations from the charts above:

- No single format is always a winner (although HEIF comes close).
- WebP appears to produce the most savings with smaller (in pixels) images. The file size savings on a large product image will be less compared to the same image used as a thumbnail.
- In contrast, JPEG 2000 and HEIF become more effective with byte savings with more pixels.
- Since HEIF is based on a video codec, with use cases of 1080p and 4K videos in mind, it makes sense that the algorithms for pixel deduplication will become more effective with larger pixel volume.

Not shown in the charts is the quality factor or settings for each equivalent experience. Depending on the image, each quality factor or quality setting can yield an experience score from really good to really bad. Selecting one quality factor can yield inconsistent results in experience and why you need to accompany the output with post-analysis.




*Visual experience vs. format quality factor.*

When converting an image from one format to another, it is important to ensure consistency in the generated output. Each format uses a different set of options to save or compress an image which can produce different experiences.

For best results you should:

- Establish an experience benchmark or high-water mark using DSSIM or SSIMULACRA.
- Analyze the image context to determine the applicability of features for each format. Does it require transparency or animation? Does the image have computer-generated characteristics and require full chroma subsampling?
- Save the image as a JPEG using both libjpeg-turbo and MozJPEG using a variety of quality factors.
- Select the smallest JPEG (in bytes) that matches the experience metric.
- Repeat the procedure for WebP, JPEG 2000, HEIF, or any other image formats.
- When serving the image to different users, determine which format the user can accept and use the smallest option. Sometimes JPEG will still be the winner.

CHAPTER 4

# Color Management

There are at least three possible perspectives to take on color: biology, physics, and print. In biology, color is a perceptual phenomenon: objects reflect light in different combinations of wavelengths, and light receptors in our eyes translate these wavelengths into the sensation we know as color. In physics, it's the light that matters – light frequencies and brightness. Print is all about color wheels, inks, and color models and modes.

Ideally, every screen and web browser in the world would display color in exactly the same way. Unfortunately, due to a number of inherent inconsistencies, they don't. Color management allows us to reach a compromise on displaying color through color models, spaces, and profiles.

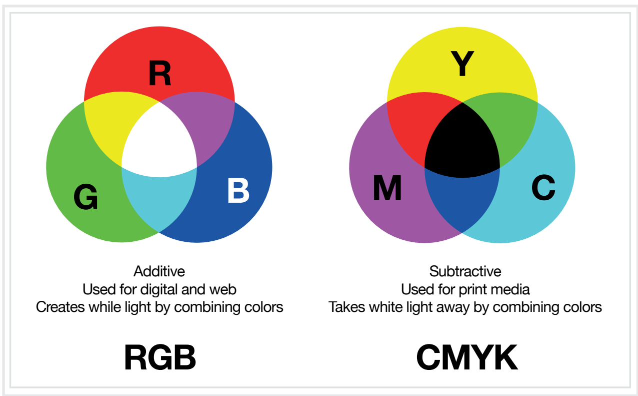
## Color Models

Color models[25] are systems for generating a complete range of colors from a smaller set of primary colors. There are different types of color models which use different parameters to control colors. Some color models have fewer control

25 https://smashed.by/colormodel

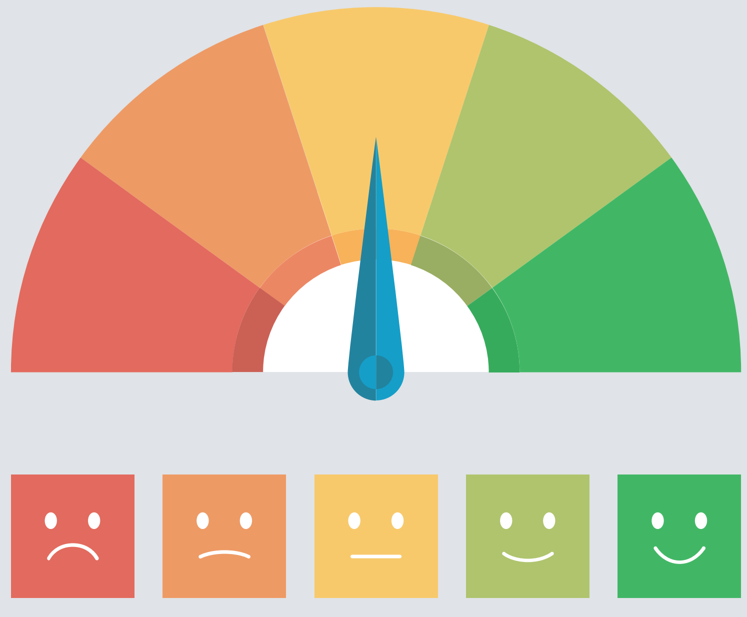
parameters than others; for example, grayscale only has a single parameter for controlling brightness between black and white colors.

Two common color models are additive and subtractive. Additive color models (like RGB, used for digital displays) add light to show color, while subtractive color models (like CMYK, used in printing) work by taking light away.




*The additive model of RGB compared with CMYK's subtractive model.*

In the RGB color model, red, green, and blue light are added in different combinations to produce a broad spectrum of colors. CMYK (cyan, magenta, yellow, black) works through different colors of ink subtracting brightness from white paper.



# SPEED MATTERS

53% of mobile visits are abandoned if pages take longer than 3 seconds to load

"Understanding Color Models and Spot Color Systems"[26] has a good description of other color models and modes, such as hue, saturation, lightness (HSL), hue, saturation, value (HSV), and CIELAB, a color space defined by the International Commission on Illumination (Commission internationale de l'éclairage, CIE).

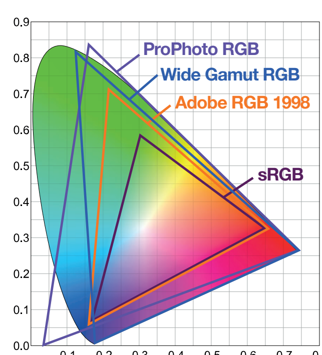
## Color Spaces

The terms *color space* and *color model* are often used interchangeably, though they are not quite the same thing. When a color model is associated with a precise description of how its color components are to be interpreted, the resulting set of colors is called a color space. Color spaces[27] are specific ranges of colors that can be represented for a given image. For example, if an image contains up to 16.7 million colors, different color spaces allow the use of narrower or wider ranges of these colors.

sRGB[28] was designed to be a standard[29] color space for the web and is based on the RGB color model. It's a small color space that is typically considered the lowest common denominator and the safest option for cross-browser color management because it is ubiquitous across most web browsers, games, and monitors. Other color spaces, such as Adobe RGB[30] or

26 https://smashed.by/understandingcolor
27 https://smashed.by/colorspace
28 https://smashed.by/srgb
29 https://smashed.by/standard
30 https://smashed.by/adobergb

ProPhoto RGB[31] used in Photoshop and Lightroom, can represent more vibrant colors but are less widely used.




*A visualization of gamut (the range of colors a color space can define) in sRGB, Adobe RGB and ProPhoto RGB.*

Color spaces have three channels (red, green, and blue). There are 255 colors possible in each channel under 8-bit mode, bringing us to a total of 16.7 million colors. 16-bit images can show trillions of colors.

*A comparison of sRGB, Adobe RGB, and ProPhoto RGB using an image from Yardstick.*

31 https://smashed.by/prorgb

It's incredibly hard to show this concept in sRGB, when you can't show colors that can't be seen. A regular photo in sRGB vs. wide gamut should have everything identical, except the most saturated "juicy" colors. The above image sources are from Clipping Path Zone.[32]

The differences in color spaces are their gamut (the range of colors they can reproduce with shades), illuminant (the nature of its theoretical light source, like incandescent light or natural sunlight) and gamma[33] curves. sRGB is about 20% narrower than Adobe RGB, and ProPhoto RGB is about 50% wider[34] than Adobe RGB.

Wide gamut[35] is a term describing color spaces with a gamut larger than sRGB. These types of displays are becoming more common. That said, many digital displays are still simply unable to display color profiles that are significantly better than sRGB. When saving for the web in Photoshop, consider using the "Convert to sRGB" option unless targeting users with higher-end wide-gamut screens.

> When working with original photography, avoid using sRGB as your primary color

32 https://smashed.by/clippingpath
33 https://smashed.by/gamma
34 https://smashed.by/gamut
35 http://www.astramael.com/

space. It's smaller than the color spaces most cameras support and can cause clipping. Instead, work in a larger color space (like ProPhoto RGB) and output to sRGB when exporting for the web.

### ARE THERE ANY CASES WHERE WIDE GAMUT MAKES SENSE FOR WEB CONTENT?

Yes. If an image contains a fluorescent highlighter color, then you'll have an easier time with wide gamut. Another good use case is images that contain very saturated and vibrant color, if you care about them being just as juicy on screens that support it.

However, in most photos it's often easy to tweak color to make it appear vibrant, without it actually exceeding sRGB's gamut. That's because human color perception is not absolute but relative to our surroundings – and it's easily fooled. That said, you should still strive to deliver the most realistic images possible. With the technology constantly advancing and new image formats that support wide gamut already available, that's becoming easier.

## Gamma Correction and Compression

Gamma correction[36] (or just gamma) controls the overall brightness of an image. Changing the gamma can also alter the ratio of red to green and blue colors. Images without gamma correction can look like their colors are bleached out or too dark.

In video and computer graphics, gamma is used for compression, similar to data compression. This allows you to squeeze useful levels of brightness in fewer bits: 8, rather than 12 or 16. Human perception of brightness is not linearly proportional to the physical amount of light.

Representing colors in their true form would be wasteful when encoding images for human eyes. Gamma compression is used to encode brightness on a scale that is closer to human perception.

With gamma compression, a useful scale of brightness fits in 8 bits of precision (0–255, used by most RGB colors). This stems from the fact that if colors used some unit with a 1:1 relationship to physics, RGB values would be from 1 to 1 million, and values 0 to 1,000 would look distinct, but values between 999,000 and 1,000,000 (and well before this range too) would look identical.

36 https://smashed.by/gammacorrection

Imagine being in a dark room lit by a single candle. Light a second candle and you'd notice a significant increase in brightness. Add a third and the room will be brighter still. Now imagine being in a room with 100 candles. Light the 101st candle, then the 102nd. You won't notice a change in brightness, even though in both cases, exactly the same amount of light was added. Because eyes are less sensitive when light is bright, gamma compression "squeezes" brightness values, so in physical terms brightness levels are less precise, but the scale is adjusted so from our perspective all values are equally precise.

> Gamma compression is different to the image gamma curves you might configure in Photoshop. When gamma compression works as it should, it doesn't "look" like anything.

## Color Profiles

A color profile is the information describing the color space of a device. It's used to convert between different color spac-

es. Profiles attempt to ensure an image looks as similar as possible on different kinds of screens and media.

Images can have an embedded color profile as described by the International Color Consortium[37] (ICC) to represent precisely how colors should appear. This is supported by different formats including JPEG, PNG, SVG, and WebP and most major browsers support embedded ICC profiles. When an image is displayed in an app and it knows the monitor's capabilities, these colors can be adjusted based on the color profile.

> Some monitors have a color profile similar to sRGB and cannot display much better profiles, so depending on your target users displays, there may be limited value in embedding them. Check who your target users are.

Embedded color profiles can also heavily increase the size of your images (over 100 KB occasionally) so be careful with embedding. Tools like ImageOptim will automatically remove color profiles[38] if they find them. In contrast, with the ICC profile removed in the name of size reduction, browsers

37 https://smashed.by/webpcontainer
38 https://smashed.by/colorprofiles

will be forced to display the image in the monitor's color space, which can lead to differences in expected saturation and contrast. Evaluate the trade-offs that make sense for your use case.

Nine Degrees Below[39] has an excellent set of resources on ICC profile color management if you are interested in learning more about profiles.

### COLOR PROFILES AND WEB BROWSERS

Earlier versions of Chrome did not have great support for color management, but this is improving with color-correct rendering.[40] Displays that are not sRGB (newer MacBook Pros) will convert colors from sRGB to their profile. With this, colors should look more similar across different systems and browsers. Safari, Edge, and Firefox can now also take ICC profiles into account, so images with a different color profile can now be displayed correctly whether your screen has wide gamut or not.

> JPEG images that do not contain embedded color profiles can be problematic for consistency in this mode. Also note that some versions of Chrome on Android have color management disabled.

39 https://smashed.by/9degrees
40 https://smashed.by/colorcorrect

For a great guide on how color applies to a broader spectrum of ways we work on the web, see "A Nerd's Guide to Color on the Web"[41] by Sarah Drasner.

41 https://smashed.by/nerdsguide

CHAPTER 5

# Image Decoding Performance

How quickly an image can be decoded determines how soon browsers can show it to the user. Keeping this efficient helps ensure a good user experience. We need to minimize the time it takes for a compressed image to be translated back into an uncompressed bitmap a browser can render to the screen.

At a high level, browsers process images in a series of steps:

1. Image is loaded from the server.
2. Image data is read or "decoded."
3. Image pixels are painted on the screen based on the decoded data.

Decoding image data is a key step, and decode time is a major component of the overall image load time after downloading. The size and format of the image as well as the user's hardware (CPU and GPU) can all affect decoding time. Larger images take longer to decode. Formats with easily available and efficient decoders outperform those that do not have an efficient decoder on the client device.

Let's dig deeper into image decoding to understand how browsers perform behind the scenes and how you can control decoding using:

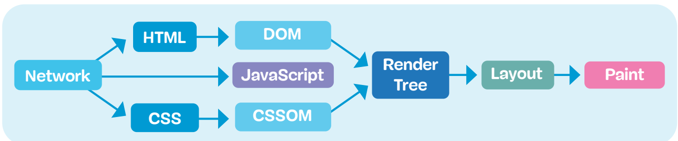
- `<img decoding>` for async image decoding
- `img.decode()` to pre-decode images
- web workers to decode in a worker thread

## Browsers: Behind the Scenes

The basic building blocks of a web page are text, images, markup, styles, and script. Web pages are really just thousands of lines of HTML, CSS and JavaScript, and images delivered over the network. This simplicity was key to the early success of the web.

Before a browser can paint a web page on the screen, it has to go through multiple steps to translate the content these resources represent into a bitmap that can be painted. Getting pixels on the screen involves using the graphics libraries provided by the underlying operating system. On most platforms, this is done with a standardized called OpenGL. So rendering is turning HTML, CSS, JavaScript, and images into OpenGL calls to display pixels on a screen.

Typically, browsers start by parsing the HTML to construct the DOM tree. The DOM tree combines with styling information and visual instructions to result in the render tree. The layout is computed based on the geometry of each node in the render tree. This layout is then used to paint each of the nodes on the screen.

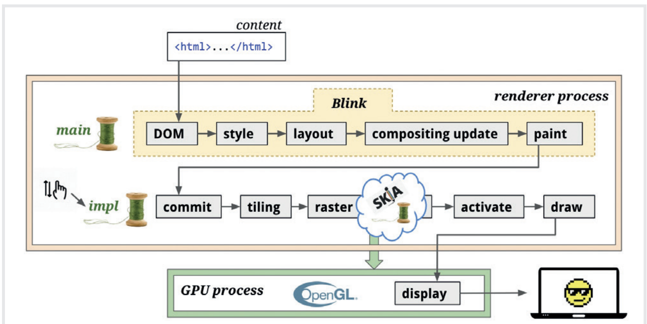



*The process browsers typically follow to render content on screen.*

This process of painting the pixels, also known as rasterization, takes a series of draw calls. The draw calls are based on the layout of each element and can handle all supported HTML elements.

> We won't go into the detail of how any specific browser renders. However, if you are interested in learning how Chrome turns web content into pixels in more depth, I recommend the excellent "Life of a Pixel"[42] series by Steve Kobes and Philip Rogers.

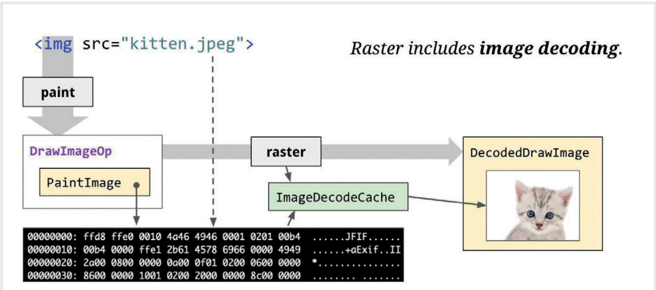
42 https://smashed.by/lifeofapixel




*The process Chromium-based browsers (such as Chrome, Edge, and Opera) use to render web pages in further detail (Source: "Life of a Pixel" by Steve Kobes and Philip Rogers)*

## IMAGE RASTERIZATION

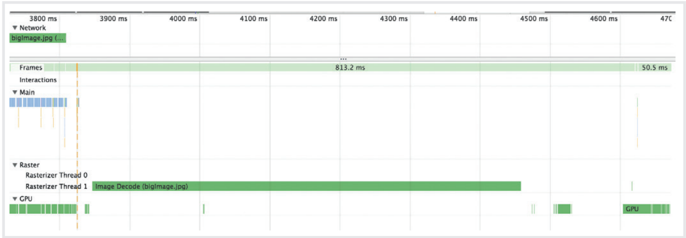
When the browser rasterizer comes across a draw call for an image, it has to get the corresponding encoded image file (JPEG, PNG, and so on) and decode it to generate the pixels




*Rasterization ("raster") also decodes image resources in the page. "Paint" references compressed image data (e.g. JPEG) and "raster" invokes the correct decoder to decompress it.*

to be painted. Most images need to be resized, based on the dimensions of the screen, the size of the parent elements in the HTML, and the width specified by the developer.

Resizing might also happen when a user pinches or zooms the image. Most algorithms decode the images to the required output size. Since both decoding and resizing are expensive operations, it takes longer to paint frames with large or multiple images.




*The DevTools Performance panel illustrates*[43] *how expensive long image decode operations can be compared with other phases of the page load life cycle.*

Most modern browsers support multithreading: that is, they allow multiple operations to be carried out on different threads. However, JavaScript itself is single-threaded and runs on the main thread. In a single-threaded environment, both JavaScript execution and operations such as layout and rasterization occur on the main thread.

43 https://smashed.by/perfpanel

In a multithreaded environment, content is divided into tiles that can be rasterized on multiple threads simultaneously. However, because of the atomicity of updates guaranteed by the browser, all content is shown at the same time after all the threads have completed their respective tasks. This means that until the frame containing the image is completely rasterized and presented, the browser does not display any subsequent frames. A later smaller image has to wait for the larger image to finish painting first.

Thus, rendering of larger images causes a significant blip in performance not only due to the higher consumption of memory and processing power but also because of the way they are handled by browsers.

### HOW IMAGE DECODERS WORK

Earlier we saw that rasterization invokes the correct image decoder to decompress an image. But how do image encoders and decoders work?

The process of encoding or compressing an image is composed of a number of phases.

- **Color Transform** attempts to produce an efficient mathematical representation of colors optimized for

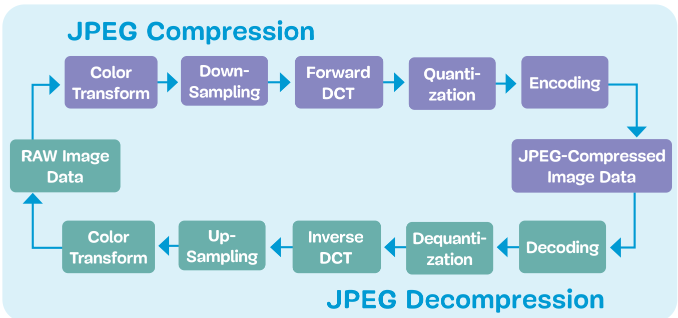
# COMBAT BLOAT

Avoid image bloat with a performance budget. Budgets constrain a site based on performance goals. e.g to load in under 3 seconds, include less than 100KB of images.

what the eye can perceive. It converts the image from an RGB color space to a YCbCr color space. The JPEG encoder will convert the RGB components of the image into three components: monochrome (luminance), red and blue chroma. This separation of RGB into monochrome and color enables additional processing on both the luminance and chroma channels.

- **Downsampling (or chroma subsampling)** attempts to resize certain color channels to a fraction of their size. It takes advantage of the eye's reduced sensitivity to color by using fewer pixels for the two chroma channels. The luminance channel, however, is kept at its original resolution. It's common for both chroma channels to be downsampled horizontally by 2:1 and vertically by 2:1 or 1:1. For a 500×500 pixel image, the luminance channel remains at 500×500, but the chroma channels would either be 250×500 or 250×250 pixels. Downsampling can give us a high level (50%) of compression with low perceived loss of quality in photos.

- **Forward Discrete Cosine Transforms (DCTs)** assume any numeric signal can be reproduced using a combination of cosine functions. We divide the luminance and chroma components of the image into 8×8 blocks of pixels as we don't expect there to be much variance over these blocks. The idea for this step is any 8×8 block can be represented as the sum of weighted cosine




*The different phases of compressing an image into JPEG and then decompressing it.*

transforms. There's a lot of self-similarity in different areas of an image which can help with compression. The output of the forward DCT is a set of 64 values representing the strength of each frequency component.

- **Quantization** is a key step in lossy image compression. As the human eye is less sensitive to losses in high-frequency detail (noise), this information is discarded while low-frequency information is preserved in the quantization table. The quantization process reduces the total quantity of bits required to store an integer by reducing integer precision.

- **Encoding.** The last phase of JPEG compression is to use a statistical encoder – the Huffman coding algorithm to encode each of the DCT coefficients into variable-length

code. Huffman does this using statistical probabilities. Any symbols that are frequently used are encoded using a code that occupies only a few bits, while symbols more rarely used are represented by code that takes more bits to encode. The output of this overall process should now be JPEG-encoded.

JPEGs have up to four Huffman tables which include the mapping between code that is variable-length and code values. While most JPEG encoders will use the Huffman tables found in the JPEG standard, some allow optimizing such tables to be more efficient.

As a quick recap of JPEG *compression*:

1. Transform an image into the appropriate color space.
2. Downsample components (Cb, Cr) as our eyes can't perceive the full brightness of an image.
3. Split the image into smaller blocks (8×8 pixels) for processing, and apply a DCT to each of the 8×8 pixel blocks.
4. Quantize each 8×8 block with a weighting function optimized for human perception.
5. Rearrange coefficients in each 8×8 block, and encode the coefficients based on quality needs.

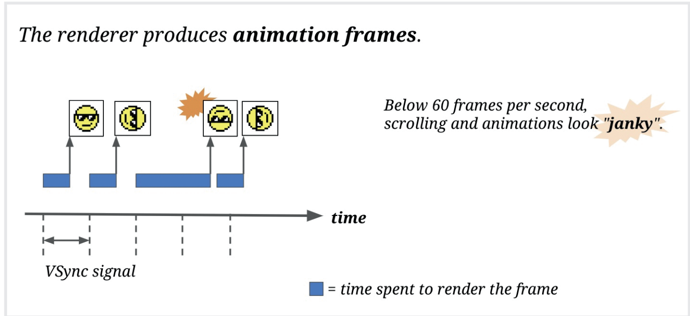
> As a reminder, RGB and YCbCr are color models used in color space conversions. RGB represents colors in combinations of red, green and blue signals. YCbCr represents colors as combinations of a brightness signal and two chroma signals. YCbCr includes: Y for luminance (brightness); Cb for blue minus luma (B – Y); and Cr for red minus luma (R – Y). Luma (Y) is an approximation of monochrome image content, while the Cb and Cr chroma channels represent color difference.

To *decompress* a JPEG, a decoder inverts the steps from above:

1. The image data goes through a Huffman decoding process.
2. The output of that step goes through an inverse DCT.
3. This goes through a dequantization process to return the image from the frequency space to the color space.
4. Chroma upsampling is applied to restore downsampled components to their full size.
5. Finally, the image gets converted from YCbCr to RGB.

## Performance and Jank

Jank can be defined as a perceptible pause in the smooth rendering of a software application's user interface.




*An animated sun experiencing jank when the third phase of the animation takes longer to render, dropping its frame rate below 60 frames a second. This makes the animation look janky. (Source: "Life of a Pixel" by Steve Kobes and Philip Rogers)*

Tom Wiltzius has described in detail various symptoms that might lead to jank[44] and categorized them as follows:

1. **Incomplete rendering**: Also known as checkerboarding, this is the situation when parts of a page are not rendered or rendered in low resolution, especially during a fast scroll. This may result in a checkerboard pattern to appear on the screen.

44 https://smashed.by/jank

2. **Low frame rate**: Imagine your web page has an embedded video or animated content. When the network speed is low, you will observe a perceptible break between frames causing the video to render unevenly. This is due to a low frame rate, which causes the frames to change slowly when compared to a normally rendered video.

3. **Latency**: This implies a longer delay between any input event and the corresponding frames rendered on screen; for example, when you touch the screen to scroll but observe a delay in the actual scroll.

Due to the heavy processing cost associated with image decoding and resizing operations (in particular on low-end mobile devices), jank can be a problem in image-heavy pages and disturbs not just the images being rendered but also the other contents of the page owing to layout shifts. In the following sections we will see how we can control image decoding to improve performance and avoid jank.

### REDUCE UNNECESSARY RESIZE COSTS

We've all shipped images that are larger or at a higher resolution than our users need. As you've learned, decoding and resizing are expensive operations for a browser on average mobile hardware. Sending large images and rescaling using CSS or `width` or `height` attributes can negatively impact performance.

Omitting the `width` or `height` attributes on an image can also hinder performance. Without them, a browser assigns a smaller placeholder region for the image until sufficient bytes have arrived for it to know the correct dimensions. At that point, the document layout must be updated. The more elements a page has, the longer this process can take.

Sending images that a browser can render without needing to resize at all is ideal. Serve the smallest images for your target screen sizes and resolutions, taking advantage of `srcset` and `sizes`.[45]

When building its new mobile web experience,[46] Twitter improved image decoding performance by ensuring it served appropriately sized images. Decode time for many images

*Chrome DevTools Timeline/Performance panel highlighting image decode times before and after Twitter Lite optimized its image pipeline. Before was considerably higher.*

45 https://smashed.by/srcset
46 https://smashed.by/twittermobile

in the Twitter timeline was reduced from approximately 400 ms all the way down to 19!

### DEVELOPER-CONTROLLED DECODING

There are two ways an image can be loaded on a web page. There's the one you are most familiar with: specifying an image source in your HTML:

```
<img src="bigImage.jpg">
```

In some cases, images can also be loaded by client-side JavaScript, like when you need to dynamically create and inject images returned from an response. The following example shows how this is done:

```
const img = new Image();
 img.src = "bigImage.jpg";
 img.onload = () => {
    document.body.appendChild(img);
 };
img.onerror = () => {
    throw new Error('Could not load the big image.');
};
```

In the first scenario the image is decoded and painted while the page is loaded; in the second case, it is done after the page is loaded or when a specific event occurs. Two updates to the HTML specification help us control image decoding in each of the above scenarios:

- `<img decoding>` attribute
- `decode()` method

### HTMLImageElement: <img decoding> attribute

When image sources are known before page load (e.g. `<img src="cat.jpg">`), they may be loaded with the other content on the page or after the content has been displayed. Synchronous image decoding prevents the other content from rendering until the decoding is completed. This causes the image and other content to be presented atomically at the same time. There might be a delay in rendering the page if the time taken for decoding the image is higher.

Asynchronous image decoding, on the other hand, does not block the other content from being rendered. The image content is updated on the screen once the decode finishes. In both cases the total time taken for rendering all of the content on screen is the same.

In some cases developers might want both images and text to be presented together to achieve the desired user experience, and they can use synchronous decoding. This will also ensure that there is no flicker or pop due to the delayed display of images. In situations where the images are not really that relevant to the context of the text, developers may prefer asynchronous decoding of images.

Before 2018 there was no way for developers to control image decoding. However, the `decoding` attribute of the `HTMLImageElement` is supported in Chrome, Edge, and Firefox, and it allows developers to indicate their preference for decoding images.

This attribute can be used as follows:

| EXAMPLE USE | EXPLANATION |
|---|---|
| `<img decoding=async src="…">` | Developer prefers to delay image decoding and render other content first. |
| `<img decoding=sync src="…">` | Developer prefers that this image and other content be rendered atomically together. |
| `<img decoding=auto src="…">` | No preference indicated by developer. Browser can choose sync or async decoding. |

**HTMLImageElement: decode() method**

For images that need to be decoded at runtime by JavaScript, the `decode()` method has been added to the HTML specification.[47] The method allows an asynchronous decode in parallel, and provides a success or failure callback for when the image is loaded and decoded. It can be used to add the image to the DOM without causing a decoding delay when it is painted on the screen. The `decode()` method is supported[48] in Chromium browsers, Firefox, and Safari.

The following code (from Stephan Köpp's "Image loading with image.decode()")[49] illustrates how the `decode()` method can be used with callback functionality to decode and load the image in JavaScript:

```
const img = new Image();
img.src = "bigImage.jpg";
img.decode().then(() => {
    document.body.appendChild(img);
}).catch(() => {
    throw new Error('Could not load/
decode big image.');
});
```

The performance improvement with this method may not be significant for small images, but it can help to reduce jank when loading large images and inserting them into the DOM.

47 https://smashed.by/HTMLspec
48 https://smashed.by/decode
49 https://smashed.by/decodeloading

When using `requestAnimationFrame`,[50] however, it is not a good idea to issue decode requests for multiple images in the same frame. Since the guarantees the image will remain cached until the next `requestAnimationFrame()` to give the developer a chance to draw the decoded image, it puts memory pressure on the system to cache all these images simultaneously.

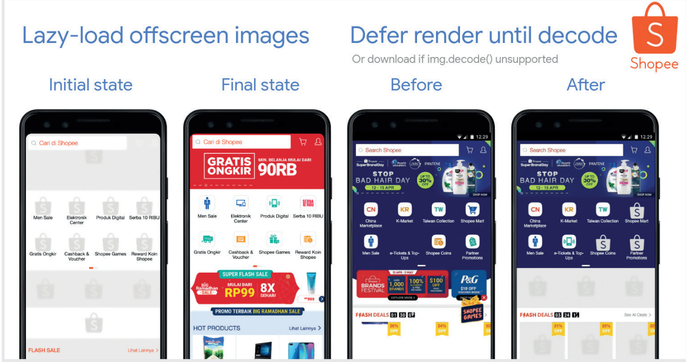
E-commerce platform Shopee[51] has incorporated the `decode()` method as part of its progressive image-loading strategy. They implemented an image component that displays a placeholder by default. When the image is inside the viewport (tracked using Intersection-Observer),[52] a network call is triggered to download the image in the background. Browsers that do not support the `decode()` method download the image synchronously.

TRIVIA ■ Blink (the rendering engine used by Chrome) decodes images off the main thread. Moving the decoding work to the compositor thread frees up the main thread to work on other tasks. This is called deferred decoding. With deferred decoding, the decoding work remains on the critical path for presenting a frame to the display, so it can still cause animation jank. The `img.decode()` should help with that.

The `<img>` tag is rendered after the image is decoded or downloaded. To enhance the user experience, Shopee also

50 https://smashed.by/requestanimation
51 https://shopee.co.id/
52 https://smashed.by/intersectionobssample

includes the fade-in animation effect as the actual image appears. When the image is huge in size or the network is slow, users first see the placeholder, and then the fully rendered image will fade in without jank. The above illustration shows how the Shopee images were rendered before and after the changes.




*Shopee defers the rendering of the actual image until the image has been downloaded and decoded. Users see either a placeholder or a fully rendered image.*

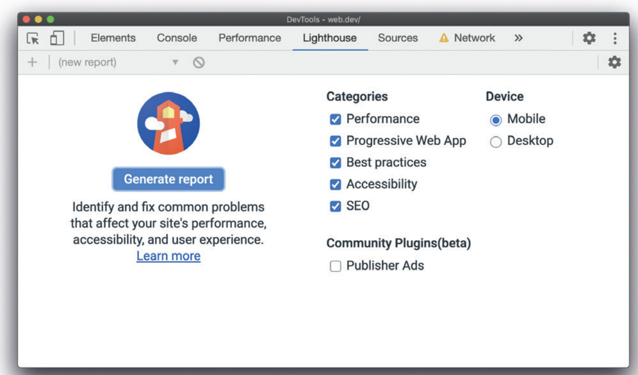
Image decoding performance is key to the overall performance of web pages and improving this can be significant in reducing jank. Browser engineering teams are constantly coming up with solutions and workarounds that offer more control to web developers over how they want to render images. The techniques listed above should provide a little more insight and control over image decoding in the browser.

CHAPTER 6

# Measuring Image Performance

## Audit for Unoptimized Images Using Lighthouse

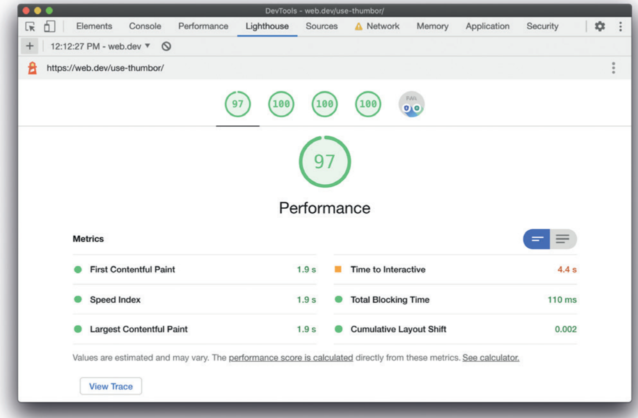
Lighthouse[53] is an open-source tool from the Chrome team for auditing and improving the quality of your web pages. You can run it against any web page whether it is public or requires authentication. Lighthouse includes audits for several best practices including web performance and image optimization opportunities.




*The Lighthouse panel in Chrome DevTools.*

53 https://smashed.by/lighthouse

You can run Lighthouse from the Audits panel in Chrome DevTools, from the command line, or as a Node.js module from npm. You provide Lighthouse a URL that it can run a number of audits against. It then generates a report on how well the page performed as well as suggestions on how to improve the page.




*A Lighthouse audit result highlighting the performance of a page in the lab (on your local machine). Higher up in the report are performance metrics while specific opportunities to improve are presented lower down.*

Each audit, including those for image optimization opportunities, links up to documentation explaining the issue and how best to fix it.

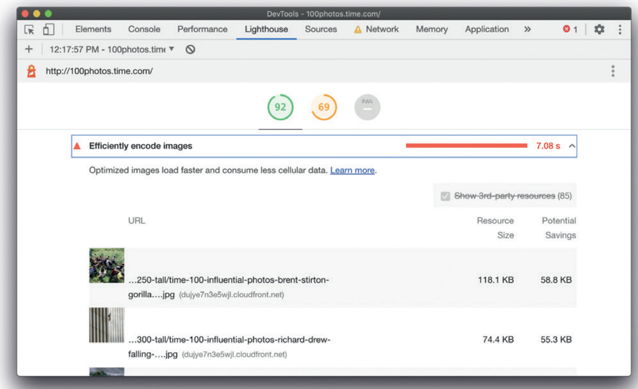
# IMAGES COST

Images increase download times. The median web page includes 800KB of images according to HTTP Archive

Running Lighthouse on a web page can highlight some of the following image optimization suggestions.

### OPTIMIZE IMAGES

JPEGs on the web can often be compressed at a lower quality without a perceivable difference compared to the original source. Lighthouse checks each JPEG image on the page, attempts to recompress it at a quality of 85, and then compares the first with the compressed version. If the potential savings are 4 KB or greater, Lighthouse includes the image in the report. This is part of the "efficiently encode images"[54] audit.




*The results of a Lighthouse audit including a suggestion to efficiently encode images.*

54 https://smashed.by/optimizeimages

#### PROPERLY SIZE IMAGES

For each image, Lighthouse compares the size of the rendered image against the size of the actual image. The rendered size also accounts for device pixel ratio. If the rendered size is at least 25 KB smaller than the actual size, the image fails the audit.

#### SERVE IMAGES IN NEXT-GENERATION FORMATS

A number of modern image formats,[55] such as WebP, can offer better compression and quality characteristics compared to their older JPEG and PNG counterparts. Encoding your images in these formats, rather than JPEG or PNG, means they can load faster and consume less data. (We'll look at some emerging image formats in part 4.)

#### DISPLAY IMAGES IN THE CORRECT ASPECT RATIO

If a rendered image has a significantly different aspect ratio[56] from that of its source file (the "natural" aspect ratio), then the rendered image may look distorted, possibly creating an unpleasant user experience. When possible, it's good practice to specify the image's width and height in HTML, so that the browser can allocate space for the image, which prevents it from jumping around as the page loads.

---

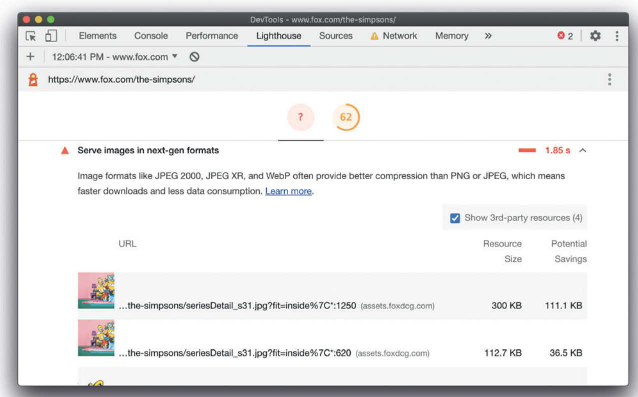
55 https://smashed.by/formats
56 https://smashed.by/offscreen

## LAZY-LOAD OFFSCREEN IMAGES

Offscreen images[57] are those that appear below what used to be called the "fold." Since users can't see offscreen images when they load a page, there's no reason to download them as part of the initial page load. Lazy-loading offscreen images can speed up page load time and reduce time-to-interactive. (Chapter 14 examines lazy-loading in detail.)

## LIGHTHOUSE WORKFLOW

A good workflow for using Lighthouse is to run it once to discover image optimization opportunities, and then, if performance needs some work, check the suggested improve-




*The results of a Lighthouse audit including a suggestion to serve images in next-gen formats.*

57 https://smashed.by/offscreen

ments and read the documentation referenced for each audit. This should hopefully guide you towards a fix for the issues. Once you've got a fix, rerun Lighthouse against the web page and with luck you can celebrate your page being faster.

## Web Performance Budgets for Images

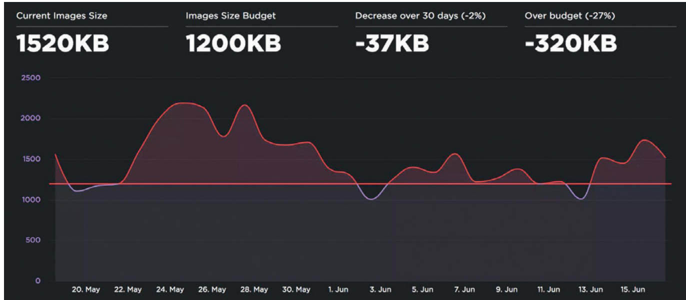
A performance budget is a set of limits imposed on metrics that affect site performance. For example: "Images will not exceed 200 KB on any page" or "The user must be able to interact with the page in under 3 seconds." When a budget isn't met, explore why this happens and how you can get back on target.

Budgets provide a useful framework for discussing performance with stakeholders. When a design or business decision might influence site performance, consult the budget. It's a reference for pushing back or rethinking the change when it can harm user experience.

I've found teams have the best success with performance budgets with automated monitoring. Rather than manually inspecting network waterfalls for budget regressions, automation can flag when the budget is crossed. Two useful services for performance budget tracking are Calibre[58] and SpeedCurve.[59]

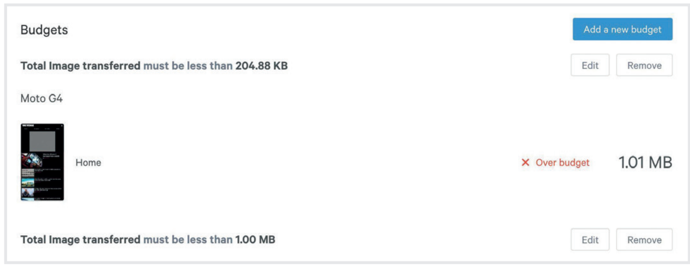
58 https://smashed.by/calibre
59 https://smashed.by/speedcurve

Once a performance budget for image sizes is defined, SpeedCurve starts monitoring and alerts you if the budget is exceeded:

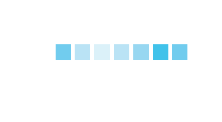


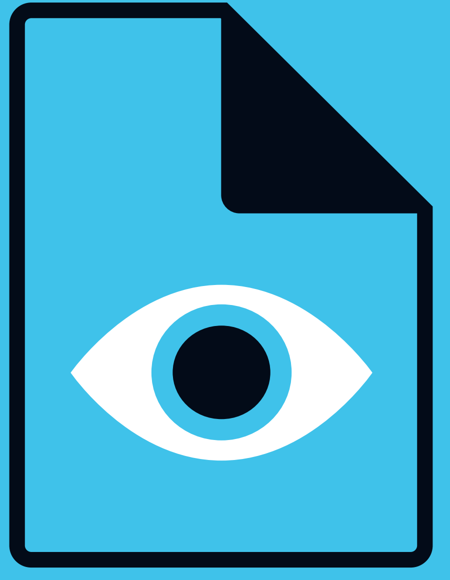

*SpeedCurve image size monitoring.*

Calibre offers a similar feature with support for setting budgets for each device class you're targeting. This is useful as your budget for image sizes on desktop over Wi-Fi may be different than your budget on mobile.




*Calibre supports budgets for image sizes.*





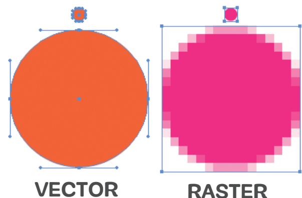
Part Two

# Current Image Formats

CHAPTER 7 JPEG .................... 115
CHAPTER 8 PNG .................... 144
CHAPTER 9 WebP .................... 170
CHAPTER 10 SVG .................... 200

## How to Choose the Best Image Format

The "right format" for an image is a combination of the desired visual results and functional requirements.[1] Are you working with raster or vector images?




**Raster graphics**[2] present images by encoding the values of each pixel within a rectangular grid of pixels. They are not resolution- or zoom-independent: if you stretch raster images to a width and height beyond their resolution, they begin to lose quality and clarity as there aren't enough pixels to organically fill the larger dimensions. Photorealistic scenes are almost always represented in raster. WebP and widely supported formats like JPEG and PNG handle these graphics well. MozJPEG, Guetzli, and other ideas discussed in this book apply well to raster graphics.

**Vector graphics**[3] use points, lines, and polygons to present images that consist of simple geometric shapes (such as logos). Vector graphics offer high resolution, and as the

1 https://smashed.by/optimizationguide
2 https://smashed.by/rastergraphics
3 https://smashed.by/vectorgraphics

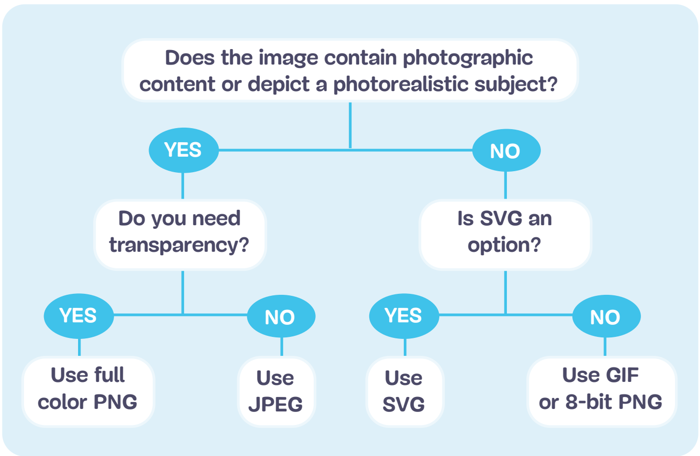
presentation of vectors isn't based on a fixed number of pixels, this type of image can be scaled to any size needed with zero loss in image quality and clarity. Formats like SVG handle this use case better.

Each format has its own merits and ideal uses for the web. A simplified summary could break down as follows:

| | HIGHLIGHTS | DRAWBACKS |
|---|---|---|
| **JPEG** | • Ubiquitously supported.<br>• Ideal for photographic content. | • There is always quality loss.<br>• Most decoders cannot handle high bit-depth photographs from modern cameras (> 8 bits per channel).<br>• No support for transparency. |
| **PNG** | • Like JPEG and GIF, enjoys wide support.<br>• It is lossless.<br>• Supports transparency, animation, and high bit-depth. | • Much bigger files compared to JPEG.<br>• Not ideal for photographic content. |

| | | |
|---|---|---|
| **GIF** | • The predecessor to PNG, most known for animations.<br>• Lossless. | • Because of the limitation of 256 colors, there is always visual loss from conversion.<br>• Very large files for animations. |
| **SVG** | • A vector-based format that can be resized without increasing file size.<br>• It is based on math rather than pixels and creates smooth lines. | • Not useful for photographic or other raster content. |
| **WebP** | • A newer file format that can produce lossless images like PNG and lossy images like JPEG.<br>• It boasts a 30% average file reduction compared to JPEG, while other data suggests that median file reduction is between 10 and 28% based on pixel volume. | • Unlike JPEG, it is limited to chroma subsampling which will make some images appear blurry.<br>• Not universally supported: only Chrome, Firefox, and Android ecosystems.<br>• Fragmented feature support depending on browser versions. |

Jeremy Wagner has covered trade-offs[4] worth considering when evaluating formats in his image optimization talks.




*One way to choose an image format by Jeremy Wagner.*

Using the wrong format can cost you. Choosing the right format is not always straightforward, so be careful when you experiment with the savings different formats can afford.

4 https://smashed.by/tradeoffs

CHAPTER 7

# JPEG

The JPEG[5] may well be the world's most widely used image format. As noted earlier, 45% of the images[6] seen on sites crawled by HTTP Archive are JPEGs. Your phone, your digital SLR, that old webcam – everything pretty much supports this codec. It's also very old, dating all the way back to 1992 when the standard was first released by the Joint Photographic Experts Group.[7] Since then, there's been an immense body of research into attempts to improve what it offers.

JPEGs are best suited to photographs or images with a number of color regions. JPEG is a lossy compression algorithm that discards information to save space. Many of the efforts that came after it tried to preserve visual fidelity while keeping file sizes as small as possible. Let's examine JPEG's compression modes as these can have a significant impact on perceived performance.

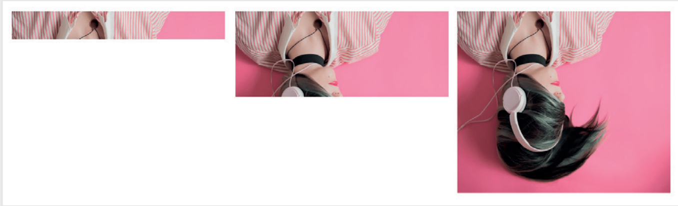
## JPEG Compression Modes

The JPEG image format has a number of different compression modes.[8] Two popular modes are baseline (sequential) and progressive JPEG (PJPEG).

5 https://smashed.by/jpeg
6 https://smashed.by/stateofimages
7 https://jpeg.org/
8 https://smashed.by/compressionmodes (PDF)

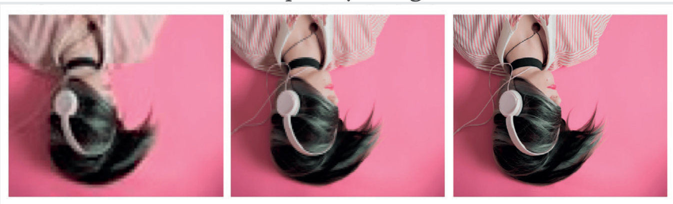
JPEG

**Baseline JPEGs** (the default for most image editing and optimization tools) are encoded and decoded in a relatively simple manner: top to bottom. When baseline JPEGs load on slow or spotty connections, users first see the top of the image with more being revealed as the image loads. Lossless JPEGs are similar but have a smaller compression ratio.



*Baseline JPEGs load top to bottom.*

**Progressive JPEGs** divide the image into a number of scans. The first scan shows the image in a blurry or low-quality setting and subsequent scans improve image quality. You can think of this as progressively refining it: each scan of an image adds an increasing level of detail. When combined, the scans create the full-quality image.



*Progressive JPEGs load from low resolution to high resolution.*

To test and learn about progressive JPEG scans, try Pat Meenan's interactive tool.[9]

> Higher-fidelity JPEG optimization can be achieved by: removing EXIF (exchangeable image file format) data[10] added by digital cameras or editors; optimizing an image's Huffman tables;[11] or rescaling the image. Tools like jpegtran[12] achieve higher-fidelity compression by rearranging the compressed data without image degradation. JPEGrescan,[13] jpegoptim[14] and MOZJPEG[15] (which we'll cover shortly) also support this kind of JPEG compression.

## The Advantages of Progressive JPEGs

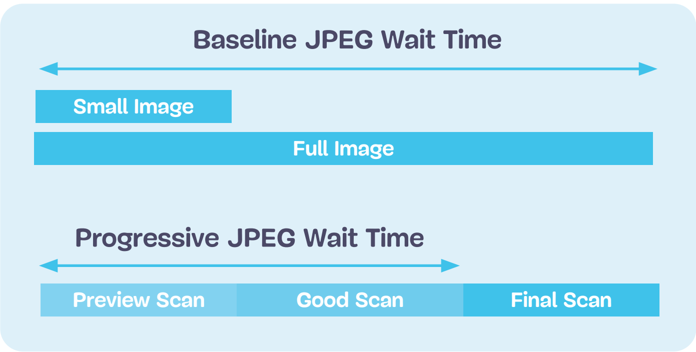
The ability for PJPEGs to offer low-resolution "previews" of an image as it loads improves perceived performance – users feel like the image is loading faster compared to traditional image loads. On slower 3G connections, this allows users to see (roughly) what's in an image when

9 https://smashed.by/progressivejpeg
10 http://www.verexif.com/en/
11 https://smashed.by/huffman
12 https://smashed.by/jpegtran
13 https://smashed.by/jpegrescan
14 https://smashed.by/jpegoptim
15 https://smashed.by/mozjpeg

JPEG

only part of the file has been received and make a call on whether to wait for it to fully load. This can be more pleasant than the top-to-bottom display of images offered by baseline JPEGs.

Because PJPEG's first scan has the same dimensions as the final image, the browser engine can calculate the page layout[16] sooner. This also means there will be less content shifting[17] during page load, which provides better user experience.




*Impact to wait time of switching to progressive JPEG.*

In 2015, Facebook switched to PJPEG (for its iOS app) and saw a 10% reduction in data usage. They were able to show a good-quality image 15% faster than previously, optimizing perceived loading time, as shown in the figure above.

16 https://smashed.by/pagelayout
17 https://smashed.by/pjpegios

PJPEGs can improve compression, consuming 2–10%[18] less bandwidth compared to baseline JPEGs for images over 10 KB. Their higher compression ratio is thanks to each scan in the JPEG being able to have its own dedicated optional Huffman table. Modern JPEG encoders (e.g. libjpeg-turbo,[19] MOZJPEG, etc.) take advantage of PJPEG's flexibility to pack data better.

Why do PJPEGs compress better? Baseline JPEG blocks are encoded one at a time. With PJPEGs, similar discrete cosine transform[20] coefficients across more than one block can be encoded together leading to better compression.

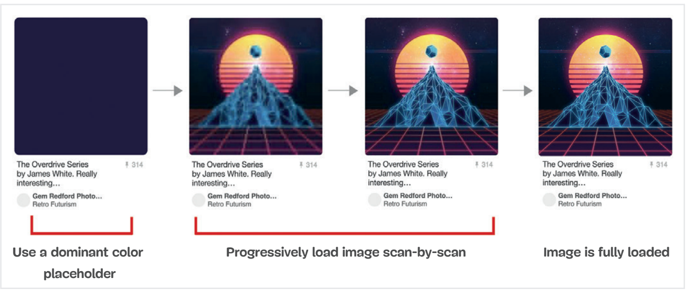
Another advantage of PJPEGs is that on HTTP/2 their first scan layers load simultaneously, which improves the speed with which users can see initial image contents[21] and enables browsers to lay out the page elements faster. Combining this with customized scan layers for PJPEGs (by providing a custom scans file to MOZJPEG,[22] for example, or using Cloudinary's custom PJPEG options[23]) will render truly meaningful image contents faster.

18 https://smashed.by/bookofspeed
19 https://smashed.by/libjpegturbo
20 https://smashed.by/cosine
21 https://smashed.by/pjpegshttp2
22 https://smashed.by/scans
23 https://smashed.by/martians

JPEG

### WHO'S USING PROGRESSIVE JPEGS IN PRODUCTION?

- Twitter.com ships progressive JPEGs[24] with a baseline quality of 85%. They measured user-perceived latency (time to first scan and overall load time) and found overall that PJPEGs were competitive at addressing their requirements for low file sizes, and acceptable transcode and decode times.
- Facebook ships progressive JPEGs for its iOS app.[25] They found it reduced data usage by 10% and enabled showing a good-quality image 15% faster.
- Yelp switched to progressive JPEGs[26] and found it was in part responsible for approximately 4.5% of their image size reduction savings. They also saved an extra 13.8% using MozJPEG.




*Pinterest's JPEGs are all progressively encoded. This optimizes the user experience by loading them each scan by scan.*

24 https://smashed.by/twitterpjpegs
25 https://smashed.by/fbjpegs
26 https://smashed.by/yelpjpegs

Many other image-heavy sites, like Pinterest,[27] also use progressive JPEGs in production.

## The Disadvantages of Progressive JPEGs

Progressive JPEGs are not always smaller. For very small images (like thumbnails), progressive JPEGs can be larger than their baseline counterparts. However, for such small thumbnails, progressive rendering might not really offer much value.

PJPEGs can also be slower to decode than baseline JPEGs as decoding the image multiple times to display different layers takes more work. On desktop machines with powerful CPUs this is not a big concern, but it can be on underpowered mobile devices with limited resources.

> There is no extensive benchmark that compares decoding speed, but when testing a small sample of images on desktop and mobile, I've noticed PJPEG decode time sometimes being up to three times as long as baseline JPEG.

27 https://pinterest.com

JPEG

When deciding whether or not to ship PJPEGs, you'll need to experiment and find the right balance of file size, network latency, and use of CPU cycles.

> All JPEGs (including progressive JPEGs) can sometimes be hardware decodable on mobile devices. It doesn't improve on RAM impact, but it can negate some of the CPU concerns. Not all Android devices have hardware acceleration support, but high-end devices do, and so do all iOS devices.

Some users may consider progressive loading to be a disadvantage as it can be hard to tell when an image has completed loading. As this can vary heavily per audience, evaluate what makes sense for your users.

## Creating Progressive JPEGS

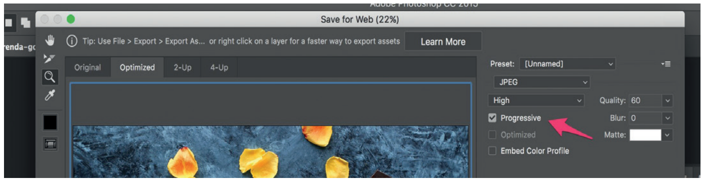
Tools and libraries like ImageMagick,[28] libjpeg,[29] jpegtran,[30] jpeg-recompress,[31] and imagemin[32] support exporting progressive JPEGs. If you have an existing image optimization

28 https://www.imagemagick.org/
29 https://smashed.by/libjpeg
30 https://smashed.by/jpegtran
31 https://smashed.by/jpegarchive
32 https://smashed.by/imagemin

pipeline, there's a good likelihood that adding progressive loading support will be straightforward:

```
const gulp = require('gulp');
const imagemin = require('gulp-imagemin');

gulp.task('images', function () {
    return gulp.src('images/*.jpg')
        .pipe(imagemin({
            progressive: true
        }))
        .pipe(gulp.dest('dist'));
});
```

Most image editing tools save images as baseline JPEG files by default.




*Photoshop supports exporting to progressive JPEG from the **File > Export** menu.*

You can save any image you create in Photoshop as a progressive JPEG by going to **File** > **Export** > **Save for Web** (legacy)

and then clicking on the **Progressive** option. Sketch also supports exporting progressive JPEGs: export as *.jpg* and check the **Progressive** checkbox while saving your images.

## Chroma (or Color) Subsampling

Our eyes are more forgiving to loss of color detail (chroma) in an image than they are luminance (luma for short – a measure of brightness). Chroma subsampling[33] is a form of compression that reduces the precision of color in a signal in favor of luma. This reduces file size, in some cases by up to 15–17%,[34] without adversely affecting image quality and is an option available for JPEG images. Subsampling can also reduce image memory usage.

*Signal = luma + chroma.*

Luma is very important because it defines contrast, which is responsible for forming the shapes we see in an image. Older, or filtered, black and white photos may not contain color, but thanks to luma, they can be just as detailed as their color counterparts. Chroma (color) has less of an impact on visual perception.

33 https://smashed.by/subsampling
34 https://smashed.by/usingsubsampling

> Recent research suggests that men and women see[35] colors differently.[36] Since women have greater color sensitivity, they might be able to notice image degradation from chroma subsampling more easily.

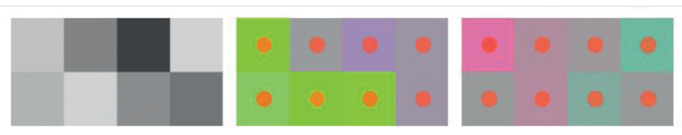
JPEG supports a number of different subsampling types: none, horizontal, and horizontal and vertical. There are a number of common examples discussed when talking about subsampling: 4:4:4, 4:2:2, and 4:2:0. But what do these represent? Let's say a subsample takes the format A:B:C. A is the number of pixels in a row and for JPEGs this is usually 4. B represents the amount of color in the first row, and C the color in the second.

- 4:4:4 has no compression, so color and luma are transported completely.
- 4:2:2 has half sampling horizontally and full sampling vertically.
- 4:2:0 samples colors out of half the first row's pixels and ignores the second row.

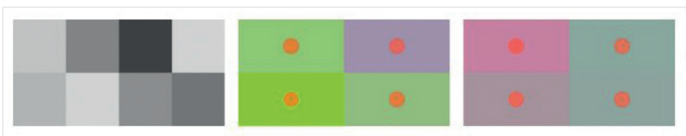
4:2:0 subsampling is used in all video codecs and it's the recommended setting for photos.

35 https://smashed.by/gender
36 https://smashed.by/colorvision

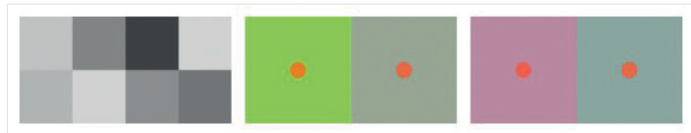
JPEG



*4:4:4 (1×1) No subsampling. All the chroma samples are kept as is. To ensure maximum picture quality, chroma subsampling remains an optional feature within JPEG. (Source for subsampling images: "JPEGs for the horseshoe crabs"* [37] *by Frédéric Kayser)*



*4:2:2 (2×1) Horizontal. Two horizontally contiguous chroma samples are merged into a single one, and horizontal chroma definition is halved. This type of subsampling is often used by default.*



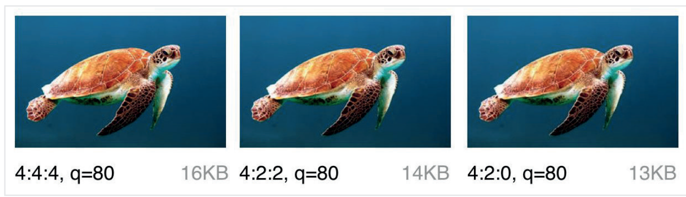
*4:2:0 (2×2) Horizontal and vertical. A square of four chroma samples is merged into a single one, and chroma definition is divided by four. This is WebP's (lossy mode) mandatory subsampling type and a common type for highly compressed JPEGs.*

37 https://smashed.by/horseshoecrabs (PDF)

jpegtran and cjpeg support separate quality configuration of luminance and chroma. This can be done by adding the `-sample` flag (e.g. `-sample 2x1`).



By reducing pixels in our chroma components, it's possible to reduce the size of color components significantly, ultimately reducing byte size.




*Chroma subsampling configurations for a JPEG at quality 80.*

Chroma subsampling is worth considering for most types of images. It does have some notable exceptions: because subsampling relies on limitations in our eyes, it is not great for compressing images where color detail may be as important as luminance (medical images, for instance).

Sharper edges are harder to compress with JPEG as it was designed to better handle photographic scenes with softer transitions. That's why images containing typefaces are poor candidates for subsampling (it can decrease legibility), as are images with patterns, diagrams, banners, buttons, or logos.

| Source | Subsampling 1x1 | Subsampling 2x2 |
|---|---|---|
| Color Subsampling | Color Subsampling | Color Subsampling |

*In his article "Finally understanding JPEG,"[38] Christoph Erdmann recommends sticking with a subsampling of 4:4:4 (1×1) when working with images containing text.*

The exact method of chroma subsampling wasn't specified in the JPEG specification, so different decoders handle it differently. MOZJPEG and libjpeg-turbo use the same scaling method. Older versions of libjpeg use a different method that adds ringing artifacts in colors. Photoshop sets chroma subsampling automatically when using the "Save for web" feature. When image quality is set between 51 and 100, no subsampling is used at all (4:4:4). When

38 https://smashed.by/understandingjpeg

quality is below this, a 4:2:0 subsampling is used instead. This is one reason a far greater file size reduction can be observed when switching quality from 51 to 50.

In subsampling discussions the term YCbCr[39] is often mentioned. Like RGB,[40] YCbCr is a way to mathematically represent how humans view color. While RGB represents colors as combinations of red, green, and blue signals, YCbCr represents colors as combinations of a brightness signal and two chroma signals. Y is gamma-corrected luminance, Cb is the blue color's chroma component, and Cr is the red color's chroma component. If you look at EXIF data, you'll see YCbCr next to sampling levels.

The color transformation from RGB to YCbCr[41] is reversible in principle, but in practice involves small losses of data due to round-off errors. For lossless conversion to and from RGB, you can use YCoCg[42] and CIELAB[43] color models.

39 https://smashed.by/ycbcr
40 https://smashed.by/rgbmodel
41 https://smashed.by/compressionhandbook
42 https://smashed.by/ycocg
43 https://smashed.by/cielab

For a further read on chroma subsampling, see "Why aren't your Images using Chroma-Subsampling?"[44] by Colin Bendell.

## How Far Have We Come Since the JPEG?

Here's the current state of image formats on the web. (These image formats typically aim to offer more efficient replacements for handling photographic images, but in many cases they also offer better handling for illustrations too.)

tl;dr: there's a lot of fragmentation. We often need to conditionally serve different formats to different browsers to take advantage of anything modern.

- **JPEG 2000**[45] (2000): an improvement to JPEG, switching from a discrete cosine-based transform to a wavelet-based method. Browser support: Safari macOS and iOS

- **JPEG XR**[46] (2009): an alternative to JPEG and JPEG 2000, supporting high dynamic range[47] (HDR) and wide gamut[48] color spaces. Produces smaller files than JPEG at slightly slower encode/decode speeds. Browser support: Edge, IE 9+.

44 https://smashed.by/usingsubsampling
45 https://smashed.by/jpeg2000
46 https://smashed.by/jpegxrwiki
47 https://smashed.by/hdr
48 https://smashed.by/gamutwiki

- **WebP**[49] (2010): developed by Google and based on block prediction with support for lossy and lossless compression. Offers byte savings associated with JPEG and the transparency support that byte-heavy PNGs are often used for. Lacks chroma subsampling configuration and progressive loading. Decode times are also slower than JPEG decoding. Browser support: Chrome (and Chromium-based browsers like Edge and Opera), Firefox and Safari. (Chapter 9 covers WebP in more detail.)

- **HEIF**[50] (2015): a format for images and image sequences for storing HEVC-encoded images with constrained inter-prediction applied. Apple announced at WWDC[51] (June 2020) that it would explore switching to HEIF over JPEG for iOS, citing up to 50% savings on file size. Browser support: Since iOS 12, supported in web view of iOS apps. (We'll take a close look at HEIF in chapter 17.)

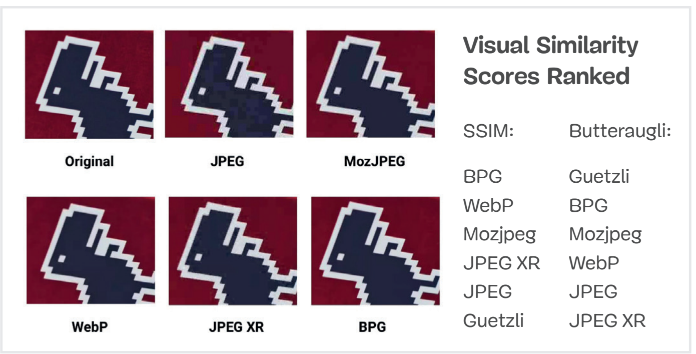
- **AVIF**[52] (2019): a highly efficient compression format[53] based on HEIF for storing still and animated images compressed with AV1[54] video codec. It has lossy and lossless modes for compression and supports transparency, HDR, and wide color gamut. Browser support: None at the time of writing. Chrome and Firefox currently support AV1 video decoding, but they cannot display AVIF images.

49 https://smashed.by/webpwiki
50 https://smashed.by/heifwiki
51 https://smashed.by/wwdc
52 https://smashed.by/avif
53 https://smashed.by/nextgenavif
54 https://smashed.by/av1

JPEG

- **JPEG XL**[55] (2019): a work-in-progress that offers substantially better compression efficiency than existing image formats (e.g. >60% over JPEG), fast decoding and encoding configurations, and a rich feature set for web distribution, particularly optimized for responsive web environments. JPEG XL encoders can produce backwards-compatible JPEG files and existing JPEG files can be losslessly transcoded to JPEG XL with file-size reduction. Browser support: None at the time of writing.

To see for yourself, try a visual image format comparison tool.[56] Here are different modern image formats (and optimizers) used to demonstrate what is possible:




*Modern image formats compared based on quality. At a target file size of 26 KB it's possible to obtain a higher perceived quality with modern options compared with lossy JPEG alone. (Source: 800×600 SRGB 1.2 MB file. Shown above is a crop of the final image.)*

55 https://smashed.by/jpegxl
56 https://smashed.by/comparison

We can compare quality using the structural similarity index measure[57] (SSIM) or Butteraugli,[58] both of which we covered in more detail in chapter 2.

**BPG**[59] (Better Portable Graphics, 2015) is another interesting format that was intended to be a more compression-efficient replacement for JPEG, but it's unlikely to get broad traction due to licensing issues. Like HEIF, it's based on HEVC (high efficiency video coding) and appears to offer better file size compared to MozJPEG and WebP. It's not currently supported in any browser, though there is a JS in-browser decoder.

So, browser support is fragmented and if you wish to take advantage of any of the formats above you'll likely need to conditionally serve fallbacks for each of your target browsers. At Google, we've seen some promise with WebP so we'll dive into it in more depth in chapter 9.

You can also serve image formats (WebP and JPEG 2000, for example) with a *.jpg* extension (or any other) as the browser can render an image it can decide the media type for.

---

57 https://smashed.by/ssim
58 https://smashed.by/butteraugligit
59 https://smashed.by/bpg

JPEG

This allows for server-side `Content-Type` negotiation[60] to decide which image to send without needing to change the HTML at all.

Next, let's talk about an option for when you can't conditionally serve different image formats: optimizing JPEG encoders.

## Optimizing JPEG Encoders

To maintain compatibility with existing browsers and image processing apps, modern JPEG encoders that produce smaller, higher-fidelity JPEG files were created. They avoid the need to introduce new image formats or changes in the ecosystem in order for compression gains to be possible. Two such encoders are MozJPEG and Guetzli.

MozJPEG is a good choice for most web assets, while Guetzli achieves higher image fidelity, but at the cost of very long encode times.

There are also configurable optimization tools that apply perception analysis (imitating the human visual system) in addition to JPEG encoding. Based on this analysis they apply the maximum amount of compression that will not cause

60 https://smashed.by/negotiation

visible artifacts. Under the hood, most of them use libjpeg, libjpeg-turbo, or MOZJPEG.

Popular ones are:

- **jpeg-recompress**[61] (uses MOZJPEG under the hood).
- **JPEGmini**[62] chooses the best quality automatically and aims at a quality range suitable for the web.
- **ImageOptim**[63] (with free online interface[64]) is unique in its handling of color. You can choose color quality separately from overall quality. It automatically chooses a chroma subsampling level to preserve high-res colors in screenshots, but avoid wasted bytes on smooth colors in natural photos.

### MOZJPEG

Mozilla offers a modern JPEG encoder in the form of MOZJPEG.[65] It claims[66] to shave up to 10% off JPEG files. Files compressed with MOZJPEG work cross-browser and some of its features include progressive scan optimization, trellis quantization[67] (discarding details that compress

61 https://smashed.by/jpegrecompress
62 https://smashed.by/jpegminitech
63 https://smashed.by/imageoptimapi
64 https://smashed.by/imageoptimonline
65 https://smashed.by/mozjpeg
66 https://smashed.by/mozjpegclaims
67 https://smashed.by/trellis

the least), and a few decent quantization table presets[68] that help create smoother high-DPI images (although this is possible with ImageMagick if you're willing to wade through XML configurations).

MozJPEG is supported in ImageOptim[69] and there's a relatively reliable configurable imagemin plug-in[70] for it.

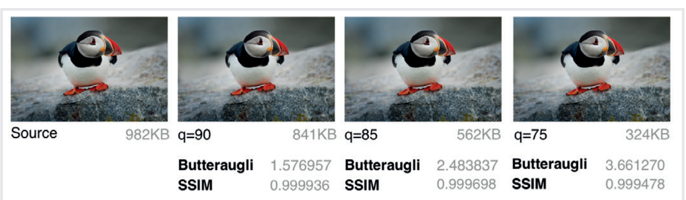
Here's a sample implementation with Gulp:

```
const gulp = require('gulp');
const imagemin = require('gulp-imagemin');
const imageminMozjpeg = require('imagemin-mozjpeg');

gulp.task('mozjpeg', () =>
    gulp.src('src/*.jpg')
    .pipe(imagemin([imageminMozjpeg({
        quality: 85
    })]))
    .pipe(gulp.dest('dist'))
);
```

```
[12:20:29] gulp-imagemin: Minified 1 image (saved 657 kB - 67%)
[12:20:29] Finished 'mozjpeg-75' after 841 ms
[12:20:29] gulp-imagemin: Minified 1 image (saved 419 kB - 42.7%)
[12:20:29] Finished 'mozjpeg-85' after 1.14 s
[12:20:29] gulp-imagemin: Minified 1 image (saved 141 kB - 14.4%)
```

*MozJPEG being run from the command line.*

68 https://smashed.by/mozjpegquant
69 https://smashed.by/imageoptimsupport
70 https://smashed.by/imageminplugin




*MozJPEG: a comparison of file sizes and visual similarity scores at different qualities (q = 90, 841 KB, 14.5% saving; q = 85, 562 KB, 42.8% saving; q = 75, 324 KB, 67% saving). Unsplash photo by Ray Hennessy.*

I used jpeg-compress[71] from the JPEG Archive[72] project to calculate the SSIM scores for a source image. SSIM is a method for measuring the similarity between two images, where the SSIM score is a quality measure of one image given the other is considered perfect.

In my experience, MozJPEG is a good option for compressing images for the web at a high visual quality while delivering reductions on file size. For small to medium sized images, I found MozJPEG (at quality = 80–85) led to 30–40% savings on file size while maintaining acceptable SSIM, offering a 5–6% improvement on jpeg-turbo. It does come with a slower encoding cost[73] than baseline JPEG, but you may not find this a showstopper.

71 https://smashed.by/imageminrecompress
72 https://smashed.by/jpegarchive
73 https://smashed.by/encodingcost

JPEG

> If you need a tool supporting MOZJPEG with additional configuration support and some complementary utilities for image comparison, check out jpeg-recompress.[74]

### GUETZLI

Guetzli[75] is a promising, if slow, perceptual JPEG encoder from Google that tries to find the smallest JPEG that is perceptually indistinguishable from the original. It performs a sequence of experiments that produces a proposal for the final JPEG, accounting for the psychovisual error of each proposal. Out of these, it selects the highest-scoring proposal as the final output. This way, Gutezli achieves image-size reduction with minimal quality loss.

To measure the differences between images, Guetzli uses Butteraugli, an objective image quality assessment metric. Guetzli can take into account a few properties of vision that other JPEG encoders do not. For example, there is a relationship between the amount of green light seen and sensitivity to blue, so changes in blue in the vicinity of green can be encoded a little less precisely.

74 https://smashed.by/jpegarchive
75 https://smashed.by/guetzli

> Image file size is much more dependent on the choice of quality than the choice of codec. There are far, far larger file size differences between the lowest and highest quality JPEGs compared to the file size savings made possible by switching codecs. Using the lowest acceptable quality is very important. Avoid setting your quality too high without paying attention to it.

Guetzli claims[76] to achieve a 20–30% reduction in data size for images for a given Butteraugli score compared to other compressors. A large caveat to using Guetzli is that it is extremely slow. From the README, we should note that Guetzli requires a large amount of memory – it can take 1 minute + 200 MB RAM per megapixel. There's a good thread on real-world experience with Guetzli to be found on GitHub.[77]

### What Is Guetzli Good For?

Guetzli offers excellent compression and output quality but at a high cost in terms of encoding time. This makes it impractical to use for optimizing images on demand, but it

---

76 https://smashed.by/guetzlisize
77 https://smashed.by/guetzlirealworld

can be considered for optimizing images as a part of a build process for a static site, and archiving photos.

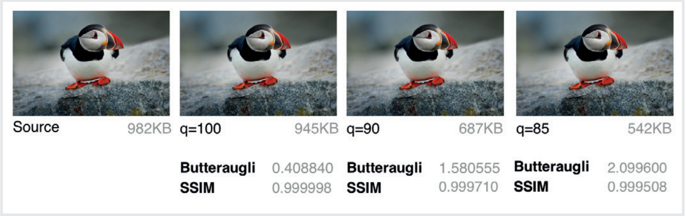
Tools like ImageOptim support Guetzli optimization. Here is an example of how to use Guetzli with the imagemin-guetzli package.

```
const gulp = require('gulp');
const imagemin = require('gulp-imagemin');
const imageminGuetzli = require('imagemin-guetzli');

gulp.task('guetzli', () =>
    gulp.src('src/*.jpg')
    .pipe(imagemin([
        imageminGuetzli({
            quality: 85
        })
    ]))
    .pipe(gulp.dest('dist'))

);
```

```
addyo-macbookpro:guetzli addyo$ gulp goo
[11:38:42] Using gulpfile ~/projects/guetzli/gulpfile.js
[11:38:42] Starting 'guetzli-100'...
[11:38:42] Starting 'guetzli-90'...
[11:38:42] Starting 'guetzli-85'...
[11:42:38] gulp-imagemin: Minified 1 image (saved 36.9 kB - 3.8%)
[11:42:38] Finished 'guetzli-100' after 3.92 min
[11:45:27] gulp-imagemin: Minified 1 image (saved 295 kB - 30.1%)
[11:45:27] Finished 'guetzli-90' after 6.73 min
[11:45:38] gulp-imagemin: Minified 1 image (saved 439 kB - 44.7%)
[11:45:38] Finished 'guetzli-85' after 6.92 min
```

*Guetzli being run from Gulp for optimization.*

It took almost seven minutes (and high CPU usage) to encode three 3 MP images using Guetzli with varied savings. For archiving higher-resolution photos, I could see this offering some value.




*Guetzli: a comparison of file sizes and visual similarity scores at different qualities (q = 100, 945 KB, 3.7% saving; q = 90, 687 KB, 30% saving; q = 85, 542 KB, 45% saving). Unsplash photo by Ray Hennessy.*

It's recommended to run Guetzli on high-quality images (e.g. uncompressed input images, PNG sources, or JPEGs of 100% quality or close). While it will work on other images (e.g. JPEGs of quality 84 or lower), results can be poorer.

JPEG

While compressing an image with Guetzli is very (very) time-consuming and will make your fans spin, for larger images it is worth it. I have seen a number of examples where it saved anywhere up to 40% on file size while maintaining visual fidelity. On small to medium sized images, I have still seen some savings (in the 10–15 KB range), but they were not quite as pronounced. Guetzli can introduce more liquify-esque distortion on smaller images while compressing.

You may also be interested in Eric Portis's research comparing Guetzli to Cloudinary's auto-compression[78] for a different data point on effectiveness.

### HOW GUETZLI COMPARES WITH MOZJPEG

Comparing different JPEG encoders is complex – you need to compare both the quality and fidelity of the compressed image as well as the final size. As image compression expert Kornel Lesiński notes, benchmarking one but not both of these aspects could lead to invalid conclusions.[79]

How does Guetzli compare with MOZJPEG? Kornel's take:

- Guetzli is tuned for higher-quality images. (Butteraugli is said to be best for q = 90+; MOZJPEG's sweet spot is around q = 75.)

78 https://smashed.by/guetzlivscloudinary
79 https://smashed.by/conclusions

- Guetzli is much slower to compress (both Guetzli and MozJPEG produce standard JPEGs, so decoding is fast as usual).
- MozJPEG doesn't automagically pick a quality setting, but you can find optimal quality using an external tool, like jpeg-archive.[80]
- Unlike MozJPEG, Guetzli doesn't support progressive image loading or color profiles (only sRGB with gamma 2.2).

MozJPEG is a beginner-friendly encoder for web assets that strikes a good balance between speed, compression, and image quality. As Guetzli is very resource-intensive its practical use is limited, but if you have large, high-quality images you need to optimize, the results might be worth the wait.

### COMBINING ENCODERS

For larger images, I found combining Guetzli with lossless compression in MozJPEG (jpegtran, not cjpeg, to avoid throwing away the work done by Guetzli) can lead to a further 10–15% decrease in filesize (55% overall) with only very minor decreases in SSIM. This is something that requires experimentation and analysis, but it has been tried by others in the field, like Ariya Hidayat,[81] with promising results.

80 https://smashed.by/jpegarchive
81 https://smashed.by/ariya

CHAPTER 8
# PNG



PNG,[82] or portable network graphics, is a raster image format that uses lossless compression.[83] It was developed around 1995 as an alternative for static GIFs.[84] Several popular tools allow you to export PNGs including Photoshop, Sketch and most image converters.

PNG has advantages such as the ability to use transparency or opacity (compared to JPEG, which retains background colors) and has broad support for color palettes.

Because of wide browser support, PNGs are the most popular choice for:

- Graphics that have sharp contrast, like line art, text, large areas of solid color, screenshots, illustrations, and logos.
- Graphics that need to preserve opacity and transparency.
- Graphics that are to have multiple edits (since PNGs do not accrue any generation loss when opened and saved again and again).

Most computers come with a built-in image viewer that can open a PNG with ease, but because all web browsers support

82 https://smashed.by/pngwiki
83 https://smashed.by/lossless
84 https://smashed.by/gifwiki

PNG files, there is no need for external apps or programs to view them; simply go to **File > Open** from your browser and select your file.



## PNG Basics

Every single PNG image has an identical 8-byte signature identifier. If this signature is altered in any way, the file will show as corrupt.

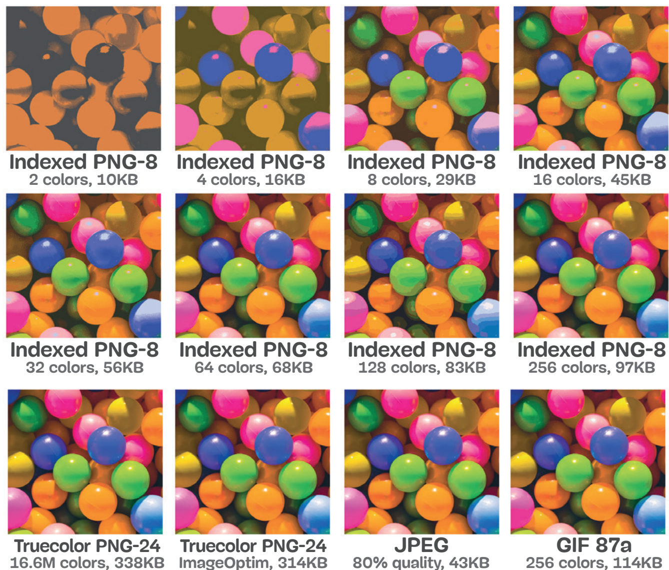
### CHUNKS

After the PNG signature identifier, come the specific file's chunks, which are essentially the building blocks of the PNG format. Each chunk is composed of a set of four components:

- **Length field**: takes up 4 bytes and refers to the length of the chunk's data field.
- **Type field**: takes up 4 bytes and indicates to the decoder what type of data the chunk contains.
- **Chunk data**: bytes of data that range from 0 to 2,147,483,647 bytes in size
- **Cyclic redundancy code (CRC)**: a 4-byte check calculated using the chunk data and type.

## PNG Palette Modes

A PNG uses one of three different palette modes: PNG-8, PNG-24, and PNG-32. The number of colors you can have in a PNG-8 image can be anything from 2 to 256. PNG supports color depths up to 48-bit (also known as deep color), providing you billions of different colors. However, the vast majority of consumer monitors and screens do not support more than 24-bit, so this is, for the most part, useless to us.




*Since the color information is stored separately in the palette, reducing the number of colors in your PNG-8 files makes a noticeable impact on file size.*

> The terms *indexed* and *paletted* can be used interchangeably.



### PNG-8

Supports 8-bit color and can handle up to 256 colors while retaining a small file size. Background transparency is available, although any round edges of the artwork will appear jagged. If a matte (background color) is applied when saving, the matte edges will become jagged, but can give the illusion of smooth artwork edges when placed on a background matching the matte color.

### PNG-24

Supports 24-bit colors[85] and can handle 16 million colors. This makes it a good choice for images containing gradients because the higher color range will reduce banding. Owing to the amount of information PNG-24 files hold, their file size is significantly larger than a PNG-8 and they should only be used when saving more complex graphics or photos where detail retention is important. Similar to PNG-8, PNG-24 files support background transparency best used with a matte to give the illusion of smooth, rounded edges.

85 https://smashed.by/colordepth

PNG

#### PNG-32

Can you work out how many colors a PNG-32 supports? The answer might surprise you. A PNG-32 supports 24-bit colors but has an extra 8-bit alpha channel used when we need transparency in our images. This can mean the highest quality PNG output and, in turn, the largest file size. It's only necessary when saving complex graphics containing gradients, rounded edges, and transparency with varying opacity; for example, images with drop shadows or outer glows. There are also instances where the background you need to place your PNG over is complex and a matte sticks out like a sore thumb, such as gradient blend backgrounds.

What if output doesn't offer PNG-32? Chances are it does, but it's disguised as an additional option under PNG-24 mentioning something like "full alpha transparency." To learn more about the difference between the various PNG bit-depths, read "The Difference Between PNG8, PNG24, and PNG32"[86] from Beamtic.

### Transparency: Index versus Alpha

In digital imagery, transparency is the idea that certain areas of an image are transparent or invisible. This unlocks

86 https://smashed.by/bitdepth

a wide range of use cases, such as displaying logos seamlessly against any colored background, all the way up to presenting composite images that take advantage of partial transparency with multiple images or backgrounds to produce special effects.



Several image formats support transparency, the most basic being index (or binary) transparency, and more advanced kinds using an alpha channel in which transparency information can be represented on a per-pixel basis with intermediate levels of opacity.

### INDEX TRANSPARENCY (OR TRANSPARENT PIXELS)

Index transparency is the simpler of the two kinds of transparency, allowing only a single color to be transparent. Any pixels with an index transparency color are not shown, and whatever content is in the background "behind" the pixel is visible instead.

As a reminder, images use a color table to represent colors where each color is designated a number. Pixel data is for the image and each pixel is given a number pointing to its color in the color table. If color #5 in a PNG is yellow and has been chosen to be (index) transparent, pixels that are color #5 will not be displayed and the background will be visible instead.

Index transparency can often be found in GIFs and PNG-8 files as the edges around transparent areas are often pixelated and there are no pixels that are partially transparent.

Index transparency can be useful when a unique symbol is not available for a block of text (perhaps you are using a web font or system font that doesn't support that character) and you wish to use an image with a transparent background to match the background of the text.

*Lorem Ipsum* is simply the dummy text of the printing and typesetting industry.

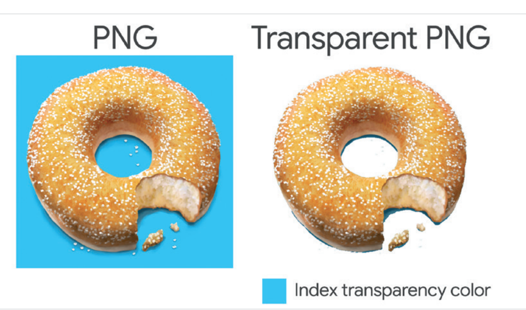
*Lorem Ipsum* is simply the dummy text of the printing and typesetting industry.

*Lorem Ipsum* is simply the dummy text of the printing and typesetting industry.

Index transparency color

*An image or symbol with a solid color background that we wish to make transparent so it better matches the rest of the text it is displayed alongside.*

It can also be useful when you need to display a non-rectangular image (a donut, for example) with a transparent background matching other content on a page.




*A donut PNG where the blue background color is set as transparent.*

Take care when selecting transparent colors so that the index-transparent PNGs are rendered as seamlessly as possible. Note how the index-transparent donut PNG might render acceptably if the background color is similar to the original blue, but might not look (entirely) seamless against a white background.

When relying on index transparency, avoid images with shades of gray on the edges of characters/illustrations with transparency backgrounds. These are commonly used between the color of the letter or image and that of the backdrop for intermediate colors.

Shades of gray often have issues in the intermediate between a darker colored letter and a lighter background. As can be seen in the example below, an unclean result occurs when giving the encoder an image with gray edge pixels. For such images, I recommend relying on alpha channel transparency instead.

PNG

*This image uses index transparency to make a white background transparent. Unfortunately, it has grayscale edges with anti-aliasing. It looks awkward even on a white background, and has a very visible ghosting effect on a colored background.*

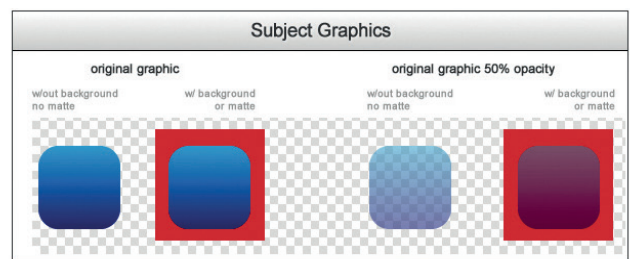
## ALPHA TRANSPARENCY

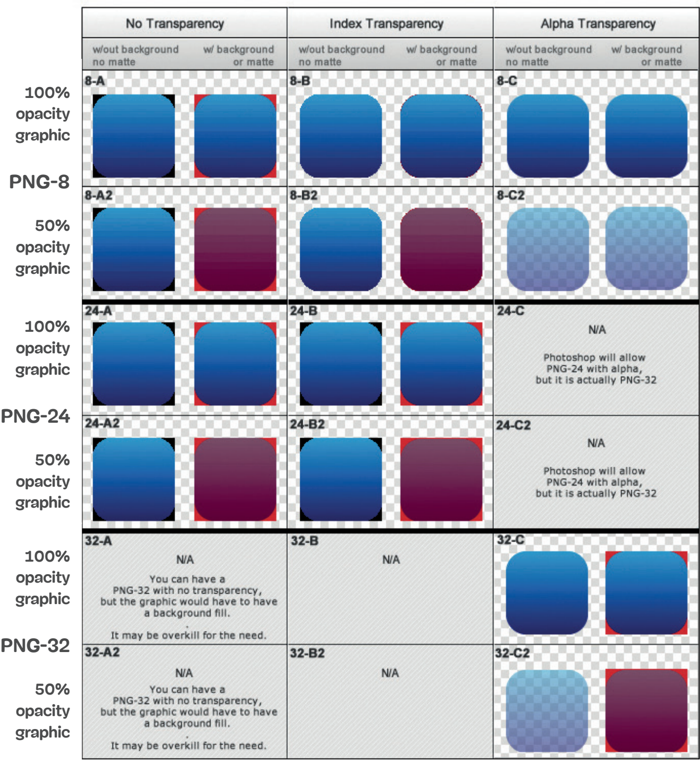
Image formats that support partial transparency do so through an alpha channel. This allows transparency to be represented on a per-pixel basis where each color value can indicate how transparent it needs to be. A colorspace such as RGBA has colors represented as red, green, blue, and alpha for controlling color opacity. An alpha value of 0 indicates complete transparency (the full background is visible, as it were), while 1 means fully opaque (full render of the pixel's color without any background being visible).

One of the nice things about alpha channels is that they can represent transparency as a gradient of values; you can use values between 0 and 1 to have full control over how much background is visible and this can be mixed in with colors. The closer the value gets to 0, the more visible the background is, while the closer to 1, the less background is shown. This level of control enables images to better blend or fade into their backgrounds with much smoother edges. Alpha channels are supported in truecolor and grayscale PNGs.

PNG

# PNG Type Matrix




by Patrick Hansen

PNG

*On the top row is a PNG using index transparency; below it is a PNG using an alpha channel. Note how the alpha channel version blends into the background much more smoothly without any of the awkward fringes around the edges.*

Alpha channels help make the edges of complex artwork like logos blend better into the rest of the page, enabling smoother transitions between different image layers, or making images more translucent (a cleaner glass-like effect).

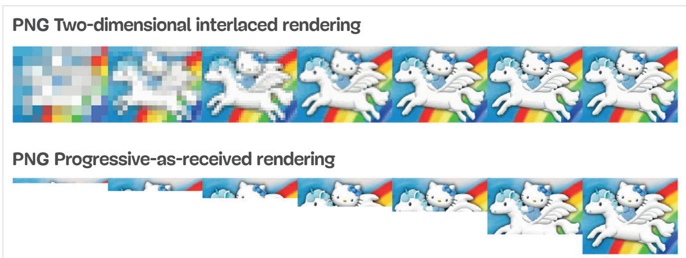
Alpha transparency allows many color levels to be transparent, which will smooth any blends along curved or anti-aliased[87] edges. A PNG-24 with alpha transparency is effectively a PNG-32. The comparison by Patrick Hansen[88] provides an excellent visualization of the same artwork saved using different color palettes and transparency options.

## Progressive versus Interlaced Display

Progressive and interlaced display options refer to how a PNG loads on a website.

Both options provide a way to give users early visual feedback while a PNG is loading. This can be useful for improving the perception that an image is loading quickly.

87 https://smashed.by/antialiasing
88 https://smashed.by/png8




*The difference between interlaced and progressive PNG rendering. (Adapted from "Progressive Image Rendering"[89] by Jeff Atwood)*

## PROGRESSIVE

A PNG saved without the interlaced option[90] is considered a progressive image, which loads one line at a time from top to bottom. When the connection speed is fast, this is often not even noticeable. However, when the connection is slow, a website visitor may grow frustrated with the slow build of the graphic.

## INTERLACED

A PNG saved with the interlaced option will begin displaying by loading the whole image in a degraded condition and then gradually build up quality.

Choosing interlaced when saving your PNG means a website's visitor will immediately see the image (at a suboptimal quality) when first opening a site rather than only a line

89 https://smashed.by/progressiveimage
90 https://smashed.by/interlaced

at a time. Though this may seem like the obvious choice, interlacing does increase file size slightly and that should be considered when deciding how to save your PNG.



## Optimizing PNGS

To understand how, it's important to first understand why a PNG should be optimized. Optimization basically refers to saving only the parts of a PNG that are necessary for its intended use, resulting in a compressed file size and quicker loading on the web. Finding the optimal settings for each PNG requires careful analysis along many dimensions: capabilities, content of the encoded data, quality, dimensions, and more.

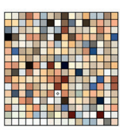
The following items discuss the behind-the-scenes process involved in optimizing PNGs that result in images with a low file size while retaining the highest quality possible.

### REDUCE UNIQUE COLORS

Remember the difference between PNG-8, PNG-24, and PNG-32, and don't automatically default to PNG-32 to retain the highest quality as that's often overkill. Weigh all of the options and choose the base output needed for that particular image.

This can mean manually reducing many slightly different colors into one color. This act alone directly influences the

compression potential of an image. Because reducing the number of unique colors is essentially applying a loss effect to an image, it is important to do this manually to ensure perceptual quality is unaffected.




*A Google doodle exported from 32-bit PNG to PNG-8.*

The Google doodle shown here was exported to PNG-8, which created the color palette to the right. This is a good example of moving from a 32-bit true color PNG file to an 8-bit indexed PNG file. By moving to an indexed image, we've replaced the unique color at each pixel to a pointer in the palette. The result is a reasonable size reduction. It works well as we're dealing with an illustration.

> Resizing an image with reduced unique colors can sometimes result in an increase of image colors due to anti-aliasing.

PNG

## OPTIMIZE DEFLATE COMPRESSION

PNG compression is a three-stage process. First, the pixels pass through a lossless arithmetic transformation called filtering, which does not compress or reduce the size of the data but makes it more compressible. Filtering is powered by how adjacent pixel colors differ from one another. In the second stage, that filtered data is passed through the Lempel-Ziv-Welch algorithm,[91] which produces LZ77[92] codes that are further compressed by the Huffman algorithm,[93] which makes up the third and final stage. The second and third stages are referred to as the Deflate compression[94] which is a universal form of lossless data compression. By reducing the number of unique colors, the range of values after filtering is decreased, and as a result the Deflate compression, which is made up of the second and third stage, will find more duplicate values and, therefore, be able to compress better.

Although Deflate is what PNG uses internally to compress pixel data, it may not compress as well as newer codecs. Advanced compressors like Zopfli[95] or 7-zip[96] can produce gzip (*.gz*) files which will generate smaller files overall. (Note: Gzip is not supported in IE11.) Tools like AdvPNG can repack PNG data to generate a smaller PNG file. We'll discuss this and other options for generating leaner PNGs shortly.

91 https://smashed.by/lempel
92 https://smashed.by/lz77
93 https://smashed.by/huffman
94 https://smashed.by/deflate
95 https://smashed.by/zopfli
96 https://smashed.by/7zip

### REMOVE UNNEEDED CHUNKS

Remember that a PNG file is made up of its signature identifier and image-specific chunks containing all sorts of data. For example, a header chunk can contain simple data such as an image's height, width, bit depth, and color type.

Then there is a chunk reserved for image data which contains the actual image information itself. This image information can actually be hidden in other chunks as well. Color-paletted images have a separate chunk specifically for that.

Finally, there is a chunk at the end of each PNG file that is considered to be the conclusion.

PNGs are notorious for containing a lot of other chunks that may not be necessary to your PNG's visible pixel data at all and may be responsible for unwanted added bytes. These chunks could be related to your file's default background color, chroma coordinates that control the display of white points, gamma spec, histogram information,[97] text data containing language or metadata, color space information, stereo-image data, notes dating previous edits, and transparency data.

These extra, sometimes unnecessary chunks represent opportunities for trimming because your image-editing

97 https://smashed.by/histogram

PNG

program automatically generates additional data behind the scenes. For example, saving a PNG file from Photoshop will result in a chunk that states that the image was made in Photoshop. That chunk has absolutely nothing to do with the visible pixel data, yet it's included in the file. Removing these useless chunks is a critical step in ensuring small file sizes. While Photoshop may add unnecessary chunks to your PNG, it also includes a feature to eliminate those chunks by simply selecting the Export option from the File menu.

Colt McAnlis's "Reducing PNG File Size"[98] and "How PNG Works"[99] explain all the steps listed above in further detail.

## Compression Tips for PNGS

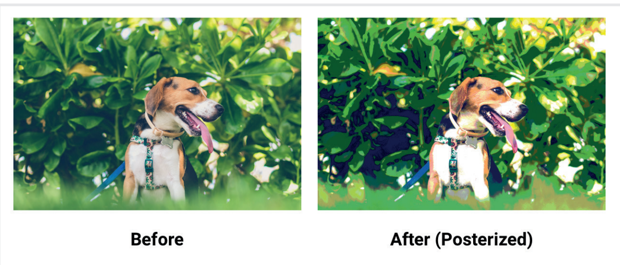
There are a few tricks you can perform manually to further compress a PNG.

### IMAGE POSTERIZATION

Posterization[100] can lower the file size of an image without harming the perceived image quality too much. It works by converting continuous color gradients into non-continuous segments that require fewer colors to render.

---

98 https://smashed.by/reducingpng
99 https://smashed.by/howpngworks
100 https://smashed.by/posterization




*A photograph before and after posterization.*

Posterization influences the appearance of a photo in a similar way to how the print process can limit the number of color inks in a poster. The effect can range from subtle to strong.

PNG

### ALIASING AND ANTI-ALIASING

The term aliasing[101] refers to the process of sampling a smooth and continuous item using a series of small measurements; in other words, taking a vector image and converting it to a raster image. If this is done without using anti-aliasing then unwanted, jagged artifacts not present in the original will appear.

*The letter on the left is aliased; on the right, anti-aliasing has been applied to make the edges smoother.*

101 https://smashed.by/aliasing

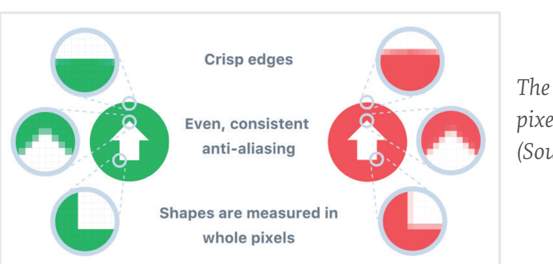
The appearance of these artifacts is called aliasing. Because pixels are set in a square grid system, aliasing isn't visible when dealing with square or rectangular objects. But as soon as an image deviates from the square pixel grid, unwanted artifacts represented by jagged curves or diagonal lines appear. Anti-aliasing is a method of reducing the visibility of those jagged edges by blurring them slightly, which creates a slightly higher file size.

### PIXEL FITTING

Pixel fitting[102] (also known as pixel hinting) is a practical technique for ensuring high-quality results for vector graphics that are converted over to raster graphics such as PNG.

Simple images like icons, wordmarks, and logos are best created as vector graphics, because doing so allows us to scale them to different sizes without any loss of quality. However, when converting that vector file into a browser-friendly, raster PNG, a problem often occurs when an image editor tries to smooth out edges during anti-aliasing. The result of that process varies but is often applied not only to the curves and diagonal lines where it's needed, but also unnecessarily along straight lines.

102 https://smashed.by/pixelfitting




*The results of pixel fitting. (Source: spec.fm[103])*



The illustration shows the results of pixel-fitting an icon from spec.fm. The red icon uses automatic anti-aliasing, leaving many of the important decisions to computer graphics tools. To the left, the green icon has its pixels aligned to the grid, ensuring they are sharp with crisp edges.

Pixel fitting or hinting is the process that bumps those blurry pixels along vertical or horizontal lines back into their place within the pixel grid.

### SPLIT BY TRANSPARENCY

Sometimes it is necessary to save an image as the "heavy" PNG-24 because of a few semitransparent pixels. You can reduce file size, though, if you split the image into two separate images: one with solid pixels saved as a PNG-8, and the other with semitransparent pixels saved as necessary.

103 https://smashed.by/specfm

“How to Optimize PNG”[104] by Sergey Chikuyonok is a good resource that lists a bunch of different ways to reduce file size manually.

PNG

## PNG Optimization Tools

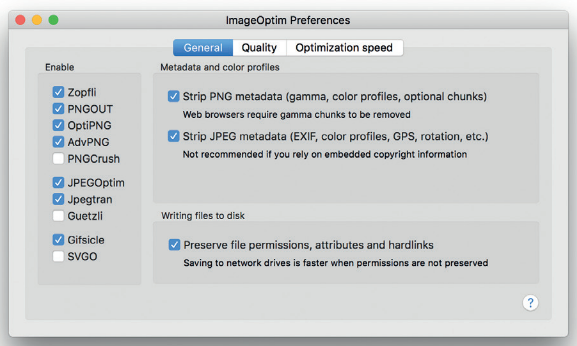
Several tools exist for optimizing PNGs in batch, with ImageOptim desktop being strongly recommended on desktop for Macs and ImageOptim[105] online for Windows. On the web, I personally also like Squoosh.app[106] for one-off compression of individual images.

### IMAGEOPTIM

ImageOptim[107] is a free desktop tool for macOS that seamlessly combines several image optimization tools, including pngcrush, pngquant, and PNGOUT (see individual entries in this list for more details).

In addition to stripping PNG metadata such as gamma, color profiles, and optional chunks, it also provides easy configuration over many of the PNG tools listed in the rest of this section.

104 https://smashed.by/optimizingpng
105 https://imageoptim.com
106 http://squoosh.app/
107 https://imageoptim.com/mac




*The preferences pane for ImageOptim showing its integrated compression tools.*

A number of alternatives to ImageOptim[108] for Windows and Linux are listed on the website.

### ADVDEF

An open-source, lossless deflate stream recompressor, advdef[109] does not run compression trials or change file specifications, but the bit depth is always rewritten to 8 bits per pixel.

### EFFICIENT COMPRESSION TOOL (ECT)

ECT[110] is an open-source and visually lossless JPEG and PNG optimizer that uses OptiPNG[111] for the fast mode and zopfliPNG[112] for stronger compression. It uses an optimized

108 https://smashed.by/imageoptimversions
109 https://smashed.by/advdef
110 https://smashed.by/ect
111 https://smashed.by/optipng
112 https://smashed.by/zopflipng

PNG

version of the libraries used by those tools, which makes ECT much faster. It includes nice improvements for dirty transparency (image regions which are 100% transparent, but still contain color information in the other channels) and lets the user combine optimization trials with order of entries in the palette.

#### MEDIAN CUT PNG POSTERIZER

This open-source and lossy PNG compressor has two distinct modes: the lossy blurizer, which changes data to make it more compressible with the average filter; and the posterizer, that reduces the number of colors without conversion to palette mode. In both cases, the median cut PNG posterizer's operation[113] is lossy but should reduce file size significantly.

#### OPTIPNG

An open-source and visually lossless PNG optimizer mainly based on pngcrush, OptiPNG[114] recompresses image files to a smaller size without losing information. If using Gulp, you can install gulp-imagemin[115] to add OptiPNG to your work-flow. To install, run:

```
$ npm install --save-dev gulp-imagemin
```

113 https://smashed.by/mediancut
114 https://smashed.by/optipng
115 https://smashed.by/gulpimagemin

Then, to configure and use OptiPNG:

```
const gulp = require('gulp');
const imagemin = require('gulp-imagemin');

gulp.task('default', () =>
    gulp.src('src/images/*')
        .pipe(imagemin([
            imagemin.optipng({optimizationLevel:
5})]))
        .pipe(gulp.dest('dist/images'))
);
```

### PINGO

Pingo[116] is an experimental, closed-source, and visually lossless or lossy JPEG and PNG optimizer for Windows. It processes files or folders recursively with a file-based multiprocessing system.

### PNGCRUSH

An open-source and visually lossless PNG optimizer, pngcrush[117] is probably one of the first public PNG optimizers, and most of the other tools are inspired by it. To make use of it, you actually have to know how PNG works, because the tool does not provide profiles, but there are lots of options.

116 https://smashed.by/pingo
117 https://smashed.by/pngcrush

PNG

### PNGOPTIMIZER

Open-source and visually lossless, PngOptimizer[118] exists as a command-line interface and a very simple GUI that allows drag-and-drop of files or folders. It performs all of the major reductions, along with color-type trials and basic dirty transparency support.

### PNGOUT

When used with the right options, PNGOUT usually compresses better than other tools that use zlib, thanks to its compression algorithm ks-flate. PNGOUT is slow but offers better compression with an advanced block splitter.

### PNGQUANT

This open-source and lossy PNG converter transforms RGBA PNG to indexed color PNG (256 colors maximum) using a nicely modified median-cut algorithm. The file size is usually reduced by half or more thanks to the conversion to palette type. Pngquant[119] works fine on most simple images, but owing to the color limitation, it requires some quality control.

---

118 https://smashed.by/pngoptimizer
119 https://pngquant.org/

### PNGWOLF

Pngwolf[120] is an open-source and lossless PNG filter finder tool that uses a generic algorithm to find filter combinations and selects the one that compresses better. Used well, it should be able to find a better filtering solution than most other PNG optimizers.

### ZOPFLIPNG

ZopfliPNG[121] is an open-source and visually lossless PNG optimizer that uses the lodePNG[122] library and zopfli compression algorithm. The tool is able to do some important reductions for web-based images, including dirty transparency. As a pure compressor, the tool is usually able to compress a bit more than PNGOUT's ks-flate algorithm.

For a detailed comparison of the free optimization tools listed above, see Cédric Louvrier's "PNG tools overview."[123]

120 https://smashed.by/pngwolf
121 https://smashed.by/zopflipng
122 https://smashed.by/lodepng
123 https://smashed.by/pngtools

CHAPTER 9

# WebP



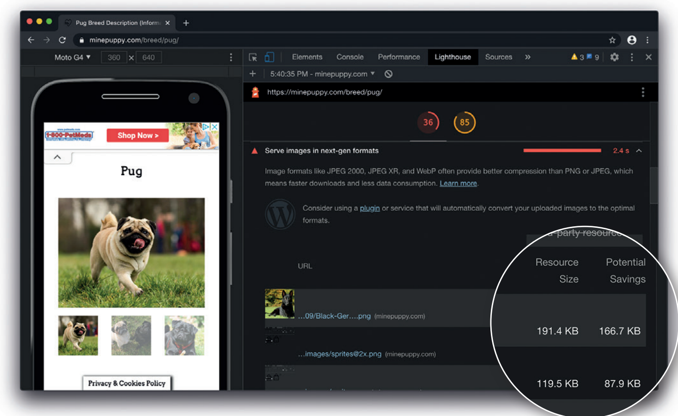
WebP[124] is a relatively recent image format developed by Google with the aim of offering lower file sizes for lossless and lossy compression at an acceptable visual quality. It includes support for alpha-channel transparency and animation.

Through 2019, WebP became a few percent better compression-wise in lossy and lossless modes, and the algorithm got twice as fast with a 10% improvement in decompression. WebP is not a tool for all purposes, but it has some standing and a growing user base in the image compression community.

*Comparison of WebP file sizes at different quality settings: q=90, 646 KB; q=80, 290 KB; q=75, 219 KB; q=70, 199 KB. Unsplash photo by Ray Hennessy.*

Is a more modern image format worth considering? Lighthouse in Chrome DevTools highlights images in older image formats, showing potential savings gained by serving WebP versions of those images.

124 https://smashed.by/webpspeed




*Lighthouse displays potential byte savings for images served in newer formats.*

The "Serve images in next-gen formats" audit can be a quick gut check that formats like WebP are worth investing your time in. As always, manual verification is also recommended if time allows.

## WebP in Performance

### LOSSY COMPRESSION

WebP lossy files, using a VP8 or VP9 video key frame encoding variant,[125] are cited by the WebP team as being 25–34%[126] smaller than JPEG files on average.

125 https://smashed.by/vp9
126 https://smashed.by/webpstudy

In the low-quality range (0–50), WebP has a large advantage over JPEG because it can blur away ugly blockiness artifacts. A medium quality setting (-m 4 -q 75) is the default, balancing speed and file size. In the higher range (80–99), the advantages of WebP shrink. WebP is recommended where speed matters more than quality.



### LOSSLESS COMPRESSION

WebP lossless files are 26% smaller than PNG files.[127] The lossless load-time decrease compared with PNG is 3%. That said, you generally don't want to deliver lossless on the web. There's a difference between lossless and sharp edges (i.e. non-JPEG). Lossless WebP may be more suitable for archival content.

### TRANSPARENCY

WebP has a lossless 8-bit transparency channel with only 22% more bytes than PNG. It also supports lossy RGB transparency, which is a feature unique to WebP.

### METADATA

The WebP file format supports EXIF photo metadata and extreme memory profile (XMP) digital document metadata. It also contains an ICC color profile.

127 https://smashed.by/webplossless

WebP offers better compression at the cost of being more CPU-intensive. Back in 2013, the compression speed of WebP was about ten times slower than JPEG, but it is now mostly negligible (some images may be two times slower). For static images processed as part of your build, this shouldn't be a large issue. Dynamically generated images will likely cause a perceivable CPU overhead and will be something you will need to evaluate.



> WebP lossy quality settings are not directly comparable to JPEG. A JPEG at a quality setting of 70 will be quite different to a WebP image at the same setting because WebP achieves smaller file sizes by discarding more data.

## WebP Feature Comparison

WebP combines good performance with rich features. WebP images can use either lossy or lossless compression, with or without transparency, while having consistently smaller[128] file sizes[129] than other image formats.

128 https://smashed.by/pngtowebp
129 https://smashed.by/animatedwebp

| | WEBP | PNG | JPEG | GIF |
|---|---|---|---|---|
| Lossy Compression | Yes | Yes | Yes | No |
| Lossless Compression | Yes | Yes | Yes | Yes |
| Transparency | Yes | Yes | No | Yes |
| Animation | Yes | No | No | Yes |



Support for animation makes WebP a good alternative to GIF:[130]

- WebP supports 24-bit RGB color with an 8-bit alpha channel, compared with GIF's 8-bit color and 1-bit alpha.

- Animated GIFs converted to lossy WebPs are 64% smaller, while lossless WebPs are 19% smaller.

In 2019, Tumblr[131] and Giphy[132] started serving animated images in WebP format to supporting browsers, with GIFs as a fallback for the rest. Other websites, including Twitter, use videos to display animated content, which we'll discuss in chapter 15.

## WebP in Production

Many large companies are using WebP in production to reduce costs and decrease web page load times.

130 https://smashed.by/animatedwebp
131 https://www.tumblr.com/
132 https://giphy.com/

Google reports 30–35% savings using WebP over other lossy compression schemes, serving 43 billion image requests a day, 26% of those being lossless compression. This can be seen in large sites like YouTube and Google Play. That's a lot of requests and significant savings. Savings would undoubtedly be larger if WebP's browser support[133] were more widespread.

At the time of writing, Netflix, Amazon, Quora, Yahoo, Walmart, Ebay, the Guardian, Fortune, and USA Today all compress and serve WebP images for browsers that support it. VoxMedia shaved between one and three seconds off load times[134] for the Verge by switching over to WebP for Chrome users. 500px saw an average 25% reduction in image file size[135] with similar or better image quality when switching to serving WebP to Chrome users.

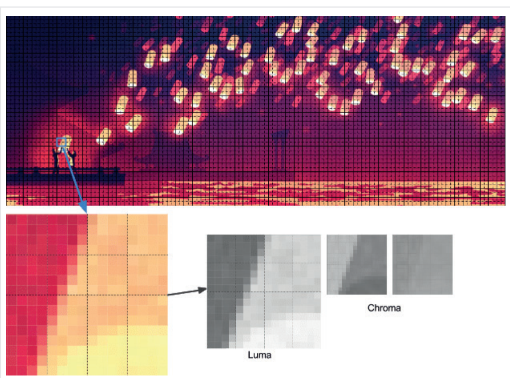


## WebP Encoding

WebP's lossy encoding is designed to compete with JPEG for still images. There are three key phases to WebP's lossy encoding.

**Macroblocking**: splitting an image into 16×16 (macro) blocks of luma pixels and two 8×8 blocks of chroma pixels. This may sound similar to the idea of JPEGs doing color space conversion, chroma channel downsampling, and image subdivision.

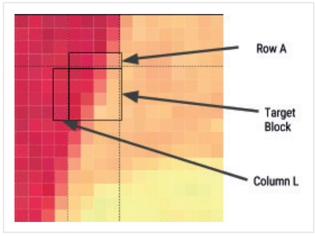
133 https://smashed.by/webpsupport
134 https://smashed.by/boogaloo
135 https://smashed.by/500px




*Macroblocking example of a Google doodle where a range of pixels is broken down into luma and chroma blocks.*

**Prediction**: every 4×4 subblock of a macroblock has a prediction model applied that effectively does filtering. This defines two sets of pixels around a block: A (the row directly above it), and L (the column to the left of it).

Using these two sets, the encoder fills a test block with 4×4 pixels and determines which creates values closest to the original block. Colt McAnlis talks about this in more depth in "How WebP Works (lossly mode)."[136]




*Google doodle example of a segment displaying the row A, target block, and column L when considering a prediction model.*

136 https://smashed.by/lossly

Finally, a **discrete cosine transform (DCT)** is then applied with a few steps similar to JPEG encoding. A key difference is the use of an arithmetic compressor[137] rather than JPEG's Huffman coding. If you want to dive deeper, the Google Developers article "WebP Compression Techniques"[138] goes into this topic in depth.



## Browser Support for WebP

WebP is supported in most major browsers, though support for different features has been added gradually. Here are the major browsers and support information[139] for WebP:

| | LOSSY WEBP SUPPORT | LOSSLESS WEBP | ANIMATION |
|---|---|---|---|
| Chrome | 17 | 23<br>25 on Android | 32 |
| Edge | 18 | 18 | 18 |
| Firefox | 65 | 65 | 65 |
| Internet Explorer | 11.10 (Presto) | 12.10 (Presto) | Not Supported |
| Opera | 15 (Blink) | 15 (Blink) | 19 (Blink) |
| Safari | 14 | 14 | 14 |

137 https://smashed.by/compressor
138 https://smashed.by/webpcompression
139 https://smashed.by/webpbrowser

It's left to developers to provide fallbacks for browsers that do not support this image format. Depending on the WebP features used, it might also be necessary to detect older browser versions that don't support them and provide fallback for those too. WebP is not without its downsides:



- It lacks wide gamut and full chroma subsampling.
- Lossy WebP works exclusively with an 8-bit YCbCr 4:2:0 (see also chapter 7, "JPEG"), while lossless WebP works exclusively with the RBGA format.
- It does not support progressive decoding.

That said, the tooling for WebP is decent, and browser support may well cover enough of your users for it to be worth considering with a fallback.

> Avoid converting low or average quality JPEGs to WebP. This common mistake can generate WebP images displaying JPEG compression artifacts. Using WebP for this is less efficient: it has to save the image *and* the distortions added by JPEG, meaning you lose on quality twice. When converting to WebP, use the best quality source file available, preferably the original.

## Viewing WebP Images

While you can drag and drop WebP images to Blink-based browsers (Chrome, Opera, Brave) to preview them, you can also preview them directly in your OS using an add-on for either macOS or Windows. When Facebook experimented with WebP[140] a few years ago, it found that users who tried to right-click on photos and save them noticed they could only display them in their browsers.

SUCCESS STORY ■ "Switching from jpeg to webP thumbnails decreased their size by about 30%. On desktop, this decreased mean thumbnail load time by about 13% in supported browsers"
—YouTube (Nov, 2018)

There were three key problems here:

- Users were able to save WebP images to their local filesystem using the browser's "Save As" functionality. Unfortunately, once saved, they were unable to view these WebP files locally. This was fixed by Chrome registering itself as a *.webp* handler.
- "Save As" then attaching the image to an email and sharing with someone without Chrome. Facebook solved this by introducing a prominent "download" button in their UI and returning a JPEG when users requested the download.

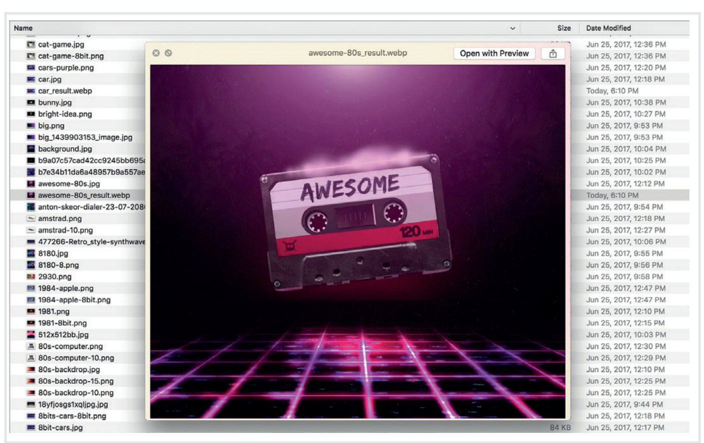
140 https://smashed.by/squawk

WebP

- Users who would right click a WebP image to copy the URL and share it on the web with other users (who may not have a WebP-supporting browser) found that sharing was broken. This was solved by content-type negotiation[141] and modern browsers now all supporting WebP.

These issues might matter less to your users, but it is an interesting note on social shareability in passing. Thankfully, utilities now exist for viewing and working with WebP on different operating systems.

On macOS, try the Quick Look plug-in for WebP[142] (qlImageSize). It works pretty well:




*Desktop on macOS showing a WebP file previewed using the Quick Look plug-in for WebP files.*

141 https://smashed.by/newformats
142 https://smashed.by/quicklook

On Windows, you can download the WebP codec package[143] that enables WebP images to be previewed in File Explorer and Windows Photo Viewer.

## Converting Images to WebP



Several free, open-source, and commercial image editing tools support WebP.

### SQUOOSH

For one off conversions to WebP, I personally enjoy using Squoosh[144] (by the Google Chrome team) on the web:

*The online interface for Squoosh. (Pug image source: Charles Deluvio[145])*

143 https://smashed.by/webpcodec
144 https://squoosh.app
145 https://smashed.by/deluvio

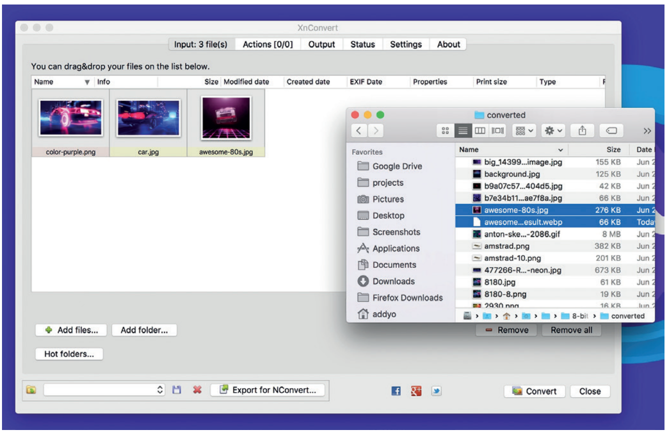
It supports an advanced mode, affording you more control over compression settings for each format, such as how alpha channels should be handled.

### XNCONVERT



A great desktop application for converting to WebP is XnConvert:[146] it's free, cross-platform, and a great batch image processing converter.

XnConvert enables batch image processing and is compatible with over 500 image formats. You can combine over 80 separate actions to transform or edit your images in multiple ways.




*Batch image editing using XnConvert on macOS.*

146 https://smashed.by/xnconvert

In addition to compression, XnConvert can also help with stripping and editing metadata, cropping and resizing, brightness and contrast, customizing color depth, blurring and sharpening, masks and watermarks, and other transforms.

### NODE.JS MODULES



Imagemin[147] is a popular image minification module that also has an add-on for converting images to WebP (imagemin-webp[148]). The add-on supports both lossy and lossless WebP modes.

To install imagemin and imagemin-webp run:

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

We can then `require()` in both modules and run them on any images in a project directory. The code below uses lossy encoding with a WebP encoder quality of 60:

```
const imagemin = require('imagemin');
const imageminWebp = require('imagemin-webp');

imagemin(['images/*.{jpg}'], 'images', {
    use: [
```

147 https://smashed.by/imagemingit
148 https://smashed.by/imageminwebp

```
        imageminWebp({quality: 60})
    ]
}).then(() => {
    console.log('Images optimized');
});
```



Similar to JPEGs, it's possible to notice compression artifacts in our output. Evaluate what quality setting makes sense for your own images. Imagemin-webp can also be used to encode lossless WebP images (supporting 24-bit color and full transparency) by passing `lossless: true` to options:

```
const imagemin = require('imagemin');
const imageminWebp = require('imagemin-webp');

imagemin(['images/*.{jpg,png}'], 'build/images', {
    use: [
        imageminWebp({lossless: true})
    ]
}).then(() => {
    console.log('Images optimized');
});
```

A WebP plug-in for Gulp[149] by Sindre Sorhus built on imagemin-webp, and a WebP loader for webpack[150] are

149 https://smashed.by/webpgulp
150 https://smashed.by/webploader

also available. The Gulp plug-in accepts any options the imagemin add-on does:

```
const gulp = require('gulp');
const webp = require('gulp-webp');

gulp.task('webp', () =>
    gulp.src('src/*.jpg')
    .pipe(webp
        quality: 80,
        preset: 'photo',
        method: 6
    }))
    .pipe(gulp.dest('dist'))
);
```

Or lossless:

```
const gulp = require('gulp');
const webp = require('gulp-webp');

gulp.task('webp-lossless', () =>
    gulp.src('src/*.jpg')
    .pipe(webp({
        lossless: true
    }))
    .pipe(gulp.dest('dist'))
);
```

WebP

## BATCH IMAGE OPTIMIZATION USING BASH

XnConvert supports batch image compression, but if you would prefer to avoid using an app or a build system, Bash and image optimization binaries keep things fairly simple. You can bulk convert your images to WebP using cwebp:[151]

```
find ./ -type f -name '*.jpg' -exec cwebp -q 70 {} -o
{}.webp \;
```

Or bulk optimize your image sources with MozJPEG using jpeg-recompress:[152]

```
find ./ -type f -name '*.jpg' -exec jpeg-recompress
{} {} \;
```

And trim those SVGs using SVGO[153] (which we'll cover in the next chapter):

```
find ./ -type f -name '*.svg' -exec svgo {} \;
```

Jeremy Wagner has published a comprehensive post on image optimization using Bash[154] and another on doing this work in parallel.[155]

151 https://smashed.by/cwebp
152 https://smashed.by/jpegarchive
153 https://smashed.by/svgo
154 https://smashed.by/bash
155 https://smashed.by/bulk

## OTHER WEBP IMAGE PROCESSING AND EDITING TOOLS

As well as those described above, other WebP tools include:

- **Leptonica**:[156] an entire website of open source image processing and analysis apps by Dan Bloomberg.
- **Sketch**:[157] supports exporting directly to WebP.
- **GIMP**:[158] the free, open-source image editor can export to WebP.
- **ImageMagick**:[159] a free command-line app that allows you to create, compose, convert, or edit bitmap images, including WebP.
- **Pixelmator**:[160] a commercial image editor for macOS exports WebP images.
- **Photoshop WebPShop plug-in**:[161] a free Photoshop plug-in from Google that enables importing and exporting WebP images (in the latest version of Photoshop).

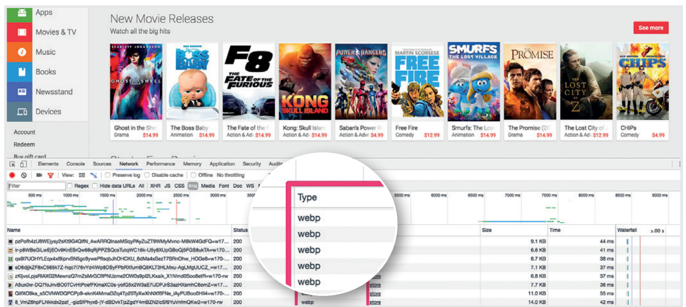
Android users can convert existing BMP, JPEG, PNG, or static GIF images to WebP format using Android Studio. For more information, see "Create WebP Images"[162] in the user guide.



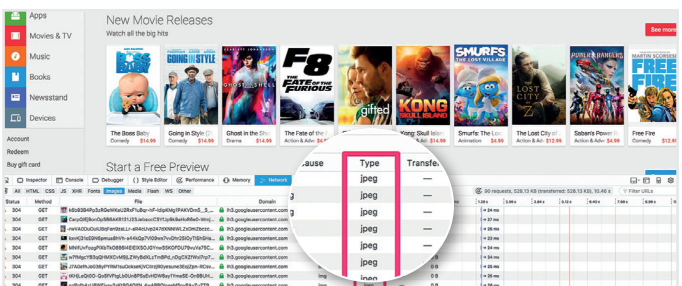
156 http://www.leptonica.org/
157 https://www.sketch.com/
158 https://www.gimp.org/
159 https://imagemagick.org/
160 https://www.pixelmator.com/
161 https://smashed.by/webpshop
162 https://smashed.by/convertwebp

## Serving WebP

Browsers without WebP support can end up not displaying an image at all, which isn't ideal. To avoid this there are a few strategies we can use for conditionally serving WebP based on browser support.




*The Chrome DevTools Network panel highlighting WebP files conditionally served to Blink-based browsers under the 'Type' column.*




*While the Play store delivers WebP to Blink, it falls back to JPEGs for browsers like Firefox.*

Let's look at some of the options for getting WebP images from your server to your user.

### USING THE <PICTURE> ELEMENT

We'll take a more comprehensive look at the `<picture>` element in chapter 11 on responsive images.

WebP

The browser can choose which image format to display using the `<picture>` element. The `<picture>` element supports multiple `<source>` elements, which can reference sources for formats like WebP.

```
<picture>
    <source srcset="puppy.webp" type="image/webp">
    <source srcset="puppy.jpg" type="image/jpeg">
    <img src="puppy.jpg" alt="Cute puppy">
</picture>
```

In this example, the browser will begin to parse the sources and will stop when it has retrieved the first supported match. If no match is found, the browser loads the source specified

in `<img>` as the fallback. This approach works well for serving any modern image format not supported in all browsers.

> Be careful with ordering `<source>` elements as order matters. Don't place `image/webp` sources after legacy formats, but instead put them before. Browsers that understand it will use them and those that don't will move on to more widely supported frameworks. You can also place your images in order of file size if they're all the same physical size (when not using the `media` attribute). Generally this is the same order as putting legacy last.



### USING .HTACCESS TO SERVE WEBP

Here's how to use an *.htaccess* file to serve WebP files to supported browsers when a matching *.webp* version of a JPEG/PNG file exists on the server. Vincent Orback recommended this approach:

Browsers can signal WebP support explicitly[163] via an Accept header.[164] If you control your back end, you can return a

163 https://smashed.by/signalwebp
164 https://smashed.by/accept

WebP version of an image if it exists on disk rather than formats like JPEG or PNG. This isn't always possible, however (for example, for static hosts like GitHub pages or Amazon S3), so be sure to check before considering this option.

Here is a sample *.htaccess* file for the Apache web server:

```
<IfModule mod_rewrite.c>

RewriteEngine On

# Check if browser support WebP images
RewriteCond %{HTTP_ACCEPT} image/webp

# Check if WebP replacement image exists
RewriteCond %{DOCUMENT_ROOT}/$1.webp -f


# Serve WebP image instead
RewriteRule (.+)\.(jpe?g|png)$ $1.webp
[T=image/webp,E=accept:1]

</IfModule>

<IfModule mod_headers.c>

Header append Vary Accept env=REDIRECT_accept

</IfModule>

AddType  image/webp .webp
```

If there are issues with the *.webp* images appearing on the page, make sure that the `image/webp` MIME type is enabled on your server.

Apache: add the following code to your *.htaccess* file:

```
AddType image/webp .webp
```



Nginx: add the following code to your *mime.types* file:

```
image/webp webp;
```

> Vincent Orback has a sample *.htaccess* config[165] for serving WebP for reference and Ilya Grigorik maintains a collection of configuration scripts for serving WebP[166] that can be useful.

#### AUTOMATIC CDN CONVERSION TO WEBP

Some content delivery networks (CDNs) support automated conversion to WebP and can use client hints[167] to serve up

165 https://smashed.by/htaccess
166 https://smashed.by/detect
167 https://smashed.by/clienthints

WebP images whenever possible.[168] Check with your CDN to see if WebP support is included in the service. You may have an easy solution just waiting for you.

### WEBP SUPPORT IN WORDPRESS

Jetpack,[169] a popular WordPress plug-in, includes a CDN image service called Site Accelerator, which gives you seamless WebP image support. The drawback is that Site Accelerator resizes your image, puts a query string in your URL, and there is an extra DNS lookup required for each image.

If you are using WordPress, there is at least one halfway open-source option from KeyCDN.[170] Its open-source plug-in Cache Enabler[171] has a menu checkbox option for caching WebP images to be served if they're available and the user's browser supports them. This makes serving WebP images easy. There is a drawback, however: Cache Enabler requires the use of a sister program called Optimus,[172] which has an annual fee for WebP conversion (the free version doesn't include it).

SUCCESS STORY ■ "With a migration to WebP, we saw a 30% reduction in data consumed per million requests."
—Navbharat Times
(November, 2018)

168 https://smashed.by/automatingimages
169 https://jetpack.com/
170 https://www.keycdn.com/
171 https://smashed.by/cacheenabler
172 https://optimus.io/

An alternative that works with Cache Enabler—also at a cost—is ShortPixel.[173] Short Pixel functions much like Optimus, and you can optimize up to 100 images a month for free.

## WebP and Chroma Subsampling



"Why does red text on a dark background look blurry in WebP images?"

Good question. Glad you asked. The short answer is chroma subsampling, which is great for saving bandwidth, but unfortunately reduces the resolution of the red channel in images. The long answer goes something like this.

As mentioned in chapter 7, "JPEG," the human eye is more sensitive to brightness (luma) than it is to color (chroma). Many image codecs take advantage of this to unlock more efficient forms of lossy image compression as they can avoid representing color in full resolution. Many lossy codecs will lower the chroma resolution to half or a quarter of full resolution. This means you only get one pixel of colour for every four pixels of brightness, which can significantly decrease how much data is needed with a low cost to quality.

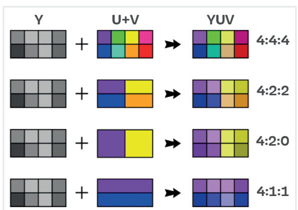
173 https://shortpixel.com/

Brightness is composed of the sum of red, green, and blue. Because encoding pixels as RGB would generate larger files requiring more bandwidth, they're encoded as YUV (see chapter 5). Y is roughly the green component; U is Y minus the red component (blue); and V is Y minus the blue component (red). It's an approximation.



Many codecs will sample the U and V components at a lower resolution than Y. This is conveyed as a ratio between the rate at which brightness (luma) and colour (chroma) values are sent. The ratio is often based on four luma values, taking the form A:B:C, where A is the number of pixels in the row (e.g, 4), and B and C are the number of chroma values in rows of a 4×2 pixel block. You may see these ratios often expressed in image compression circles as 4:1:1, 4:2:2, 4:2:0, and so on.

4:2:2 means that each horizontal scanline has 2 chroma values for every 4 luma values. 4:1:1 means 1 chroma value for every 4 luma values, and 4:4:4 means no chroma subsampling. This isn't very consistent. 4:2:0 implies that for each 4 luma values, there would be 2 for the first chroma component and none for the second, but this would not produce images of full color. In practice, 4:2:0 means there are two of each chroma sample per scanline.

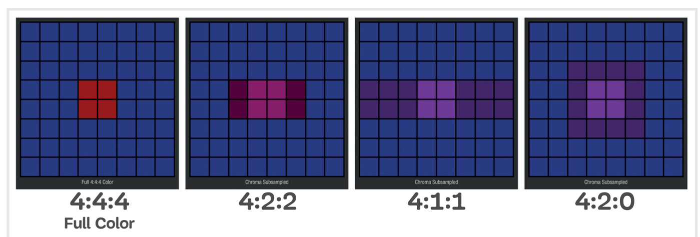



*The most widely used chroma subsampling formats (Source: shutterangle.com[174])*



On the web, much of the media we come across uses 4:2:0. In each 4×2 rectangle of pixels, there are only two U+V samples. This means the color portion of the image is made up of chunks 2×2 pixels in size, or one quarter the full resolution. The red channel alone has one quarter the resolution of the overall image, meaning if the edges of overlaid red text appear pixelated, it is because they are.

What do these artifacts look like up close? As visualized in the example from red.com, because chroma subsampling effectively reduces colour resolution, it is most visible near the edges of sharp color transitions. This is what this looks like with an 8×8 pixel image.



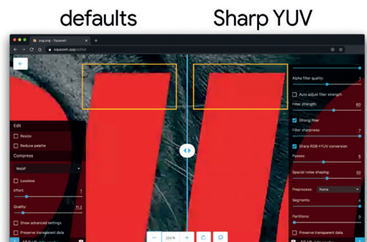

*(Source: https://www.red.com/red-101/video-chroma-subsampling[175])*

174 http://www.shutterangle.com
175 https://smashed.by/videochroma

To confirm if the artifacts are an issue with YUV 4:2:0, experiment with JPEGs and switch between 4:2:0 and 4:4:4 encoding. If you observe the same artifacts as WebP, this is a good indication that YUV 4:2:0 is the culprit.

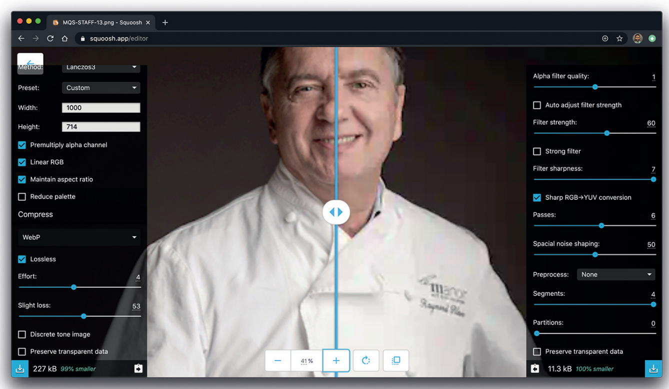
How do you address this with WebP? The answer is with the `-sharp_yuv` option with cwebp (see "Batch Image Optimization Using Bash" above). The idea with `-sharp_yuv` is to invent U and V values around the edges that are not necessarily correlated to the source image, but should produce sharper edges when smeared by the decoder's upsampling process. To demonstrate this workaround, let's take a (super cute) image of a pug with a red text overlay:

Below is the before/after of compressing the pug image using WebP. The version on the right has `-sharp_yuv` enabled:




*Using Squoosh to compare the edges of the red text overlay before and after applying* `-sharp_yuv`*. Notice how the edges of the red text overlay are subtly sharper.*

Your mileage may vary with `-sharp_yuv`. It's certainly not a panacea, so do spend time analyzing how well it suits the kinds of images you work with. Be careful not to use this mode for archival imagery.




*Images with gradient backgrounds can experience a banding effect with* `-sharp_yuv` *turned on. While this isn't simply* `-sharp_yuv`*'s fault, it highlights the limitations of lossy WebP only supporting 8-bit YUV 4:2:0, which may cause color loss on images with thin contrast elements.*

## The Future for WebP

With modern browsers now featuring broad support for WebP, the team behind the codec has set their sights on WebP 2.[176]

176 https://smashed.by/libwebp2

At the time of writing, WebP 2 is considered an experimental codec that explores how much further WebP compression can evolve. The team aims to achieve another 30% better than WebP to offer savings as close to AVIF as possible. Other features to be explored include:

- better visual degradation at low bit rates
- lightweight incremental decoding
- better transparency and lossless compression
- support for HDR10
- support for very lightweight image previews

As it will likely take time for browser and tooling support for AVIF and JPEG XL (see chapters 18 and 19) to mature, WebP is a great choice for anyone using JPEG or PNG that would like greater compression with support for wider color depths, transparency, or animation.

CHAPTER 10

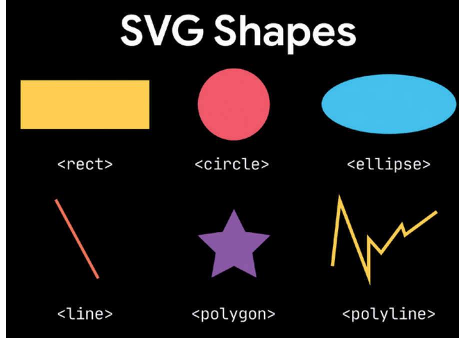
# SVG

SVG is an XML (extensible markup language) based vector image format for the web and other platforms. Think of SVG code as being quite similar to HTML but with a stricter subset of features. Here is a preview of the SVG markup to draw a simple blue rectangle with a black border around it:



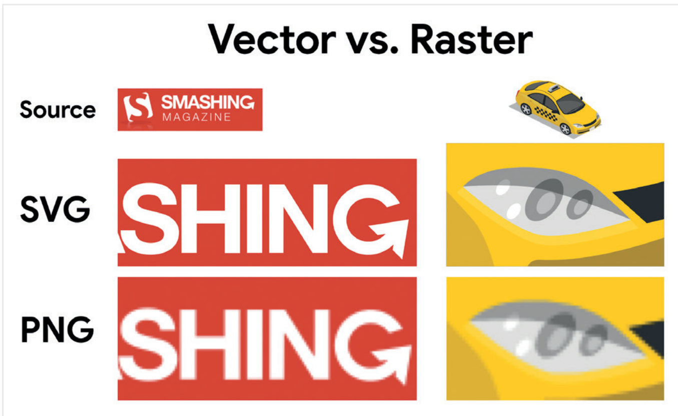
```
<rect x="14" y="23" width="200" height="50"
fill="blue" stroke="black" />
```

Under the hood, SVG is just plain text that uses shapes, lines, numbers, and coordinates to render graphics, instead of a grid of colors and pixels.




*The shapes supported by SVG include rectangles (`<rect>`), circles (`<circle>`), ellipses (`<ellipse>`), straight lines (`<line>`), polygons (`<polygon>`) and polylines (`<polyline>`).*

Because it is a vector format, SVG can be infinitely scaled without any loss in image quality, unlike raster-based formats with fixed dimensions. A JPEG, GIF, or PNG will eventually pixelate when it scales, but SVGs remain crisp in detail at any size.




*When you scale up a raster image, you are likely to see pixels. When you scale up an SVG, however, you'll continue to see detailed curves or lines. This makes SVG ideal for illustrations, logos, icons, and UI elements that need to maintain crispness on high-definition screens.*

Being text-based lends well to SVG files being small in file size. But keep in mind that this depends on the complexity of the image and shapes used. In web pages, CSS and JavaScript can be applied to both HTML and SVG. This means you can adjust the color of SVG elements using CSS, or add

interactivity using JavaScript, making it powerful for building user interface elements.

SVG[177] is a W3C web standard, and it can typically be used across modern browsers with a decent level of consistency for core features. These different qualities make SVG a great option for the web as it can deliver sharp vector graphics on different screen sizes with minimal bandwidth.

SVG

> SVG has become the de facto vector format used on the web. As such it has strong support across popular design tools such as Sketch,[178] Figma[179] and Adobe Illustrator[180] to name but a few. This allows you to import SVG elements between design tools by copying and pasting them.

## Adding SVGs to a Page

There are a number of ways to implement SVGs in a web page. They can be used via the `<img>` element, embedded inline, or embedded via an SVG map.

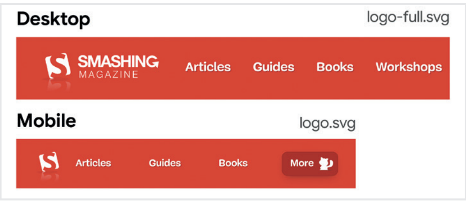
177 https://smashed.by/aboutsvg
178 https://www.sketch.com/
179 https://smashed.by/figma
180 https://smashed.by/illustrator

### USING THE <IMG> ELEMENT

An SVG image saved to a file from a design tool, text editor, or the web can be used directly inside of the `<img>` element. Like JPEGS or PNGS, you can adjust the SVG image's `width` or `height`, and browser support is relatively strong (IE9+ and all modern browsers).

```
<img src="smashing.svg" alt="Smashing Magazine logo"
width="300" height="100">
```

Similarly, it is possible to use SVGS with `<img>` within the `<picture>` element for responsive design use cases. Smashing Magazine delivers either a simplified or a more detailed SVG illustration depending on a media query in the `media` attribute:




```
<picture>
<source media="(min-width: 1450px)" srcset="/images/
logo-full.svg">
<img src="/images/logo/logo.svg" alt="Smashing
Magazine logo">
</picture>
```

There are several benefits to using the `<img>` approach to reference SVGs, including full support for `alt` and `title` attributes, browser caching, search indexability, and compression. The downsides, unfortunately, are:

- Lack of control over styling the inner elements of the SVG with CSS.
- If you need to add interactivity to your SVG, you will need to use a different approach, such as inline SVG embeds or the `<object>` element.
- Limited ability to rely on external resources like web fonts. Because of security concerns,[181] you might need to convert web fonts to outlined shapes to ensure visual fidelity with your source.

SVGs can also be used as background images in CSS with very similar pros and cons to SVGs in an `<img>` element. To ensure that outer areas of a larger SVG file are not visible we also set the dimensions and `background-size`:

```
#logo {
     display: block;
     width: 300px;
     height: 100px;
     background-image: url(smashing.svg);
     background-size: 300px 100px;
}
```

181 https://smashed.by/svgsecurity

### EMBEDDING SVGS INLINE

Inlining SVG directly in HTML is relatively straightforward and can often be seen with SVG icons. The primary benefit of inlining is that you can skip a network request to obtain the SVG image, speeding up how quickly a user can see it.

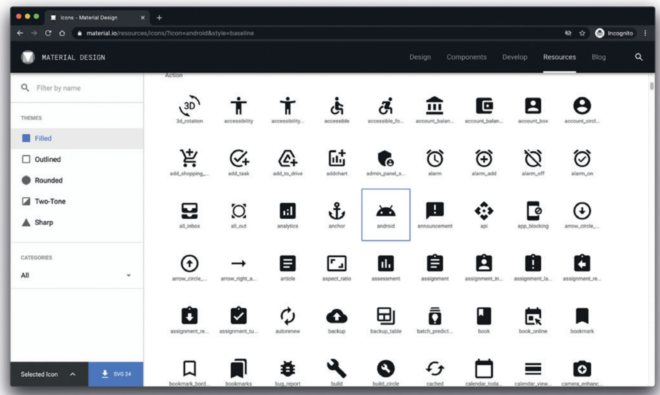
```
<a href="/blog/">
    Visit Blog
    <svg xmlns="http://www.w3.org/2000/svg"
version="1.1" class="triangle" width='100'
height='100'>
      <polygon points="100,50 0,100 0,0"/>
    </svg>
  </a>
```

The downsides are:

- Inline SVGs don't benefit from caching because the browser will need to redownload the embedded code each time you navigate to a new document that uses it.
- Inline SVGs need to be re-embedded each time you wish to use them, making maintenance and update more difficult.
- Additional care must be taken over accessibility.[182] If there is no text in the SVG that describes it, a `<title>image title</title>` should be used as the first

182 https://smashed.by/svgaccessibility

child of its parent element, and you might wish to add a `<desc>` for a description.




*SVG icons have become mainstream, such as Google's Material Design icons[183] collection. Individual icons can either be referenced via an* `<img>` *file, pasted as inline SVG into HTML or imported via other methods.*



### USE THE <OBJECT> ELEMENT

If the cacheability issues with inline SVGs are an issue, or you require a deeper level of interaction or customization using CSS than is available with the `<img>` approach, it's possible to link to an SVG file using the `<object>` element:

```
<object type="image/svg+xml" data="smashing.svg"></
object>
```

183 https://smashed.by/materialicons

This has strong cross-browser support but there are a few gotchas:

- You cannot use `<style>` in your document or an external style sheet to style the `<object>`. Instead, you need to use an inline `<style>` *within* the SVG file.
- Search engine optimization: SVG images referenced using `<object>` may not show up in image searches. One workaround to this problem is including an `<img>` source as a fallback within the `<object>`. However, this can introduce a double-loading side effect.



### INLINING WITH SVG MAPS

SVG can be powerful for icons,[184] offering a way to represent visualizations as a sprite without the quirky workarounds[185] that were needed for icon fonts. It has more granular CSS styling control than icon fonts (SVG `stroke` properties), better positioning control (no need to hack around pseudo-elements and CSS display), and SVGs are much more accessible.[186]

It is also possible to create a single SVG file that contains the different graphics you wish to use on a page, similar to image sprites.

184 https://smashed.by/iconfonts
185 https://smashed.by/accessibleicons
186 https://smashed.by/accessiblesvg

Each graphic (an icon, perhaps) has a unique ID allowing it to be referenced elsewhere in the HTML document. This SVG map can be dropped into a template for use on other pages across a site, which could make maintenance a little simpler than strictly relying on inlining alone. Remember to hide this SVG map from the user (via CSS, for example) so the map itself does not get rendered:

```
<svg class="svg-assets" xmlns="http://www.w3.org/2000/
svg" role="presentation"
    aria-hidden="true">
  <defs>
    <svg id="arrow-left" xmlns="http://www.
w3.org/2000/svg" version="1.1" class="triangle"
width='100' height='100'>
      <polygon points="0,50 100,0 100,100"/>
    </svg>
    <svg id="arrow-right" xmlns="http://www.
w3.org/2000/svg" version="1.1" class="triangle"
width='100' height='100'>
      <polygon points="100,50 0,100 0,0"/>
    </svg>
  </defs>
</svg>
```

When you then want to refer to a specific graphic or icon, you can reference it by ID. In the example below, we are using the “arrow-right” graphic defined in our SVG map:

```
<a href="/blog/">
  Visit Blog
  <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0
18 18" class="my-icon"
       role="presentation" aria-hidden="true">
    <use xlink:href="#arrow-right"></use>
  </svg>
</a>
```

SVG maps can also be referenced separately, linking directly to the SVG file rather than inlining it on the page.

```
<a href="/blog/">
  Visit Blog
  <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0
18 18" class="my-icon"
       role="presentation" aria-hidden="true">
    <use xlink:href="/assets/icons/navigation-icons.
svg#arrow-right"></use>
  </svg>
</a>
```

Tools like svg-sprite[187] and IcoMoon[188] can automate combining SVGs into sprites that can be used via a CSS sprite, symbol sprite, or stacked sprite. Una Kravetz has published a great practical write-up[189] on how to use gulp-svg-sprite in an SVG sprite workflow. Sara Soueidan also covers making the transition from icon fonts to SVG[190] on her blog.

187 https://smashed.by/svgsprite
188 https://icomoon.io/
189 https://smashed.by/spriteworkflow
190 https://smashed.by/iconfontstosvg

SVG

## Optimizing SVGs

Keeping SVGs lean means stripping out anything unnecessary. SVG files created with design tools usually contain a large quantity of redundant information (metadata, comments, hidden layers, and so forth). This content can often be safely removed or converted to a more minimal form without altering the look of the final image.

There are some useful principles to apply to any SVG to keep your files as lean as possible:

- Remove metadata that should be safe to drop:
    - `<?xml … ?>`: is the version of XML used.
    - `<!-- Comments -->`: Suggest the file was tool-generated.
- Individual `id`s. Unless they are being targeted directly, you do not need to define an ID on each element inside your file.
- Instead of paths, use predefined SVG shapes like `<rect>`, `<circle>`, `<ellipse>`, `<line>`, and `<polygon>`. Using predefined shapes decreases the amount of markup

needed to produce an image, meaning less code for the browser to parse and rasterize. Reducing SVG complexity helps a browser display it more quickly.

- If you must use paths, try to reduce your curves, and simplify and combine paths where you can. Illustrator's Simplify tool[191] is adept at removing superfluous points in even complex artworks while smoothing out irregularities.
- Minify and then compress your SVG files with gzip or Brotli[192] (a compression algorithm developed by Google that works best for text compression). SVGs are really just text assets expressed in XML and should be minified and compressed to improve performance.
- Avoid any Photoshop or Illustrator effects. They can end up converted into large raster images.
- Avoid using groups (<g> indicates groups). If you can't, try to simplify them. Ideally, everything in your image should be in a single top-level layer, removing the need for groups when distributing SVGs on the web.
- Delete invisible layers.
- Double-check for any embedded raster images that aren't SVG-friendly.

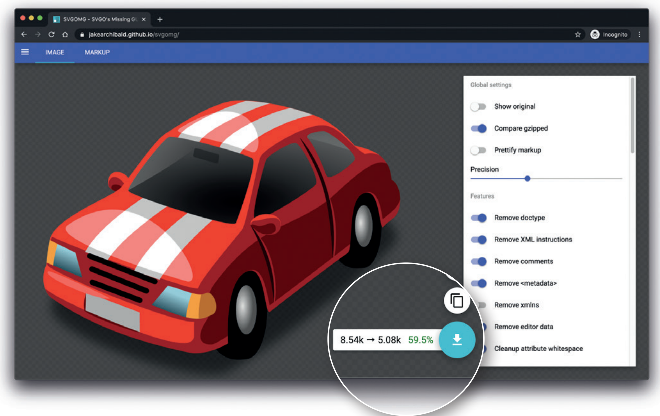


191 https://smashed.by/simplify
192 https://smashed.by/brotliwiki

## SVG Optimizer (SVGO)

Perhaps the most important thing to do is to use a tool to optimize your SVGs. SVGOMG,[193] by Jake Archibald, is a super handy web-based GUI for SVGO,[194] a Node.js-based tool for optimizing SVGs. It allows you to select and combine different optimizations and offers a live preview of the outputted markup. If you use Sketch, you can also use the SVGO Compressor plug-in[195] when exporting to shrink the file size.



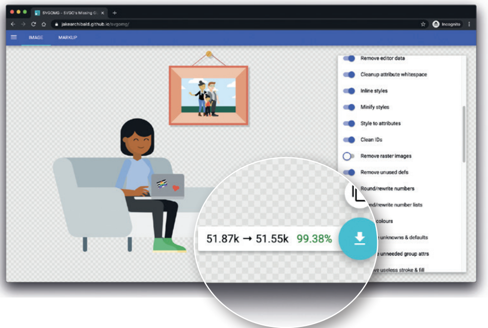



*SVGOMG further optimizing the size of an existing SVG image.*

Below is an illustration that I exported to SVG from Adobe Illustrator. The file size on my disk was 76 KB. SVGOMG shows it could be 51.87 KB if gzipped and 51.55 KB after additional clean-up and minification is applied.

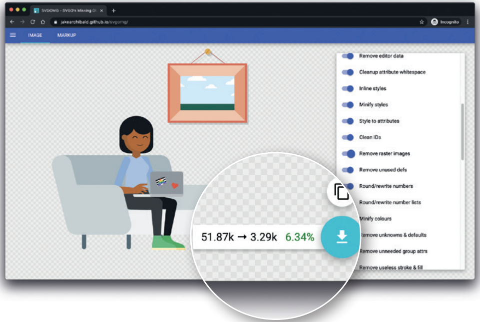
---

193 https://smashed.by/svgomg
194 https://smashed.by/svgo
195 https://smashed.by/svgocompressor






SVGs of this complexity are often a little smaller than this, so I began to play around with some of the optional optimizations in SVGOMG. This brought my attention to the “Remove raster images” option. I didn’t think that I had any embedded images in this illustration, but see what happens with this option on:

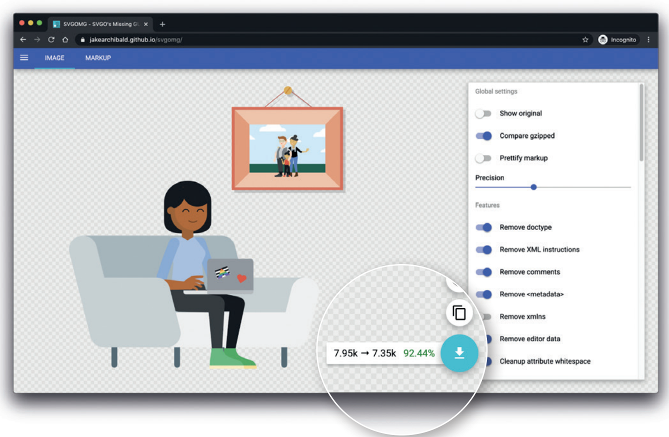



Lo and behold, the SVG size has been reduced from 51.87 KB to just 3.29 KB. As it turns out, the illustration of the family

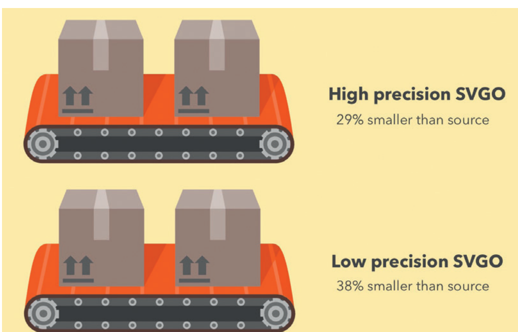
in the picture frame was a raster image – not a vector – and this was bloating up our file size. With this new information, I went back to Illustrator and made sure to redo this part of the illustration as a vector. (The family is flipped to the left this time around to distinguish it from the earlier image.)

And … presto! Our complete SVG is now only 7.35 KB in size. Amazing!





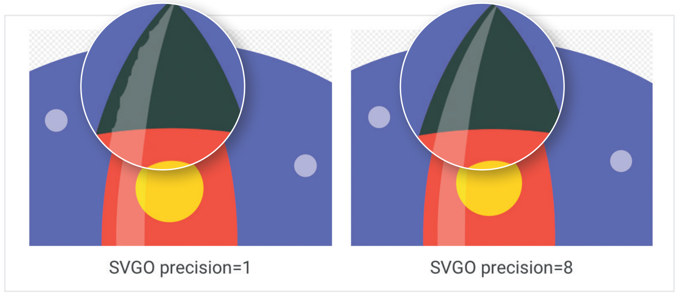

SVGO can reduce file size by lowering the precision of numbers in your definitions. Each digit after a point adds a byte and this is why changing the precision (number of digits) can heavily influence file size.




*SVGO precision reduction can sometimes have a positive impact on size.*



Be very careful when changing precision, however, as it can influence how your shapes look. It's important to note that while SVGO does well in the example above without oversimplifying paths and shapes, there are plenty of cases where the result doesn't end up looking great.




*Where SVGO can go wrong: oversimplifying paths and artwork.*

Observe how the light strip on the rocket is distorted at a lower precision.

SVG

### USING SVGO AT THE COMMAND LINE

If you prefer CLIs over GUIs, you can install SVGO as a global npm CLI:[196]

```
npm i -g svgo
```

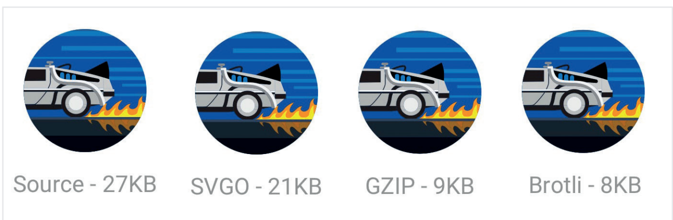
You can then run SVGO against a local SVG file as follows:

```
svgo input.svg -o output.svg
```

It supports all the options you might expect, including adjusting floating point precision:

```
svgo input.svg --precision=1 -o output.svg
```

See the SVGO documentation[197] for a list of supported options.




*Before and after running an image through SVGO, with further compression.*

196 https://smashed.by/npmcli
197 https://smashed.by/svgodoc

Also, don't forget to gzip your SVG assets[198] or serve them using Brotli. As they're text-based, they'll compress really well (around 50% of the original sources).

When Google shipped a new logo back in 2015, we announced[199] that the smallest version of it was only 305 bytes in size. There are lots of advanced SVG tricks[200] you can use to trim this logo down even further – all the way to 146 bytes! Suffice to say, whether it's through tools or manual clean-up, there's always a little more you can shave off your SVGs.

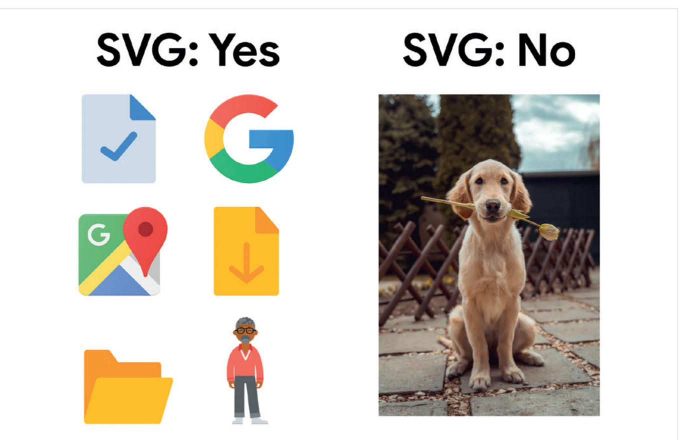
*The smallest version of the new Google logo was only 305 bytes in size.*

198 https://smashed.by/optimizingsvg
199 https://smashed.by/logo
200 https://smashed.by/svgtricks

Understanding when and how to leverage SVGs on the web can lead to sharper images with a smaller file size. Keep in mind that they're ideal for simpler graphics rather than detailed photography and are very flexible, being styleable with CSS or interactive with JavaScript.

SVGs are lightweight, scalable, have great tooling support and can be great for web performance compared to their raster counterparts. While learning how to work with this vector format can take some time, it's definitely worth the investment.






*SVG is optimal for simpler graphics such as illustrations, logos, and icons. It's not intended to render detailed photographic content. While you can try this, the size and memory use costs are likely to be far higher than using a raster format which is far better suited to this type of image. (Source: Richard Brutyo, Unsplash[201])*

201 https://smashed.by/brutyo

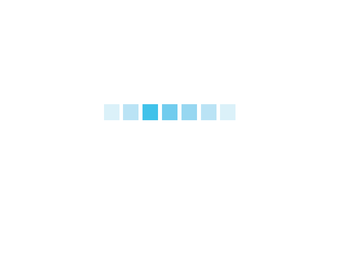
## Further Reading

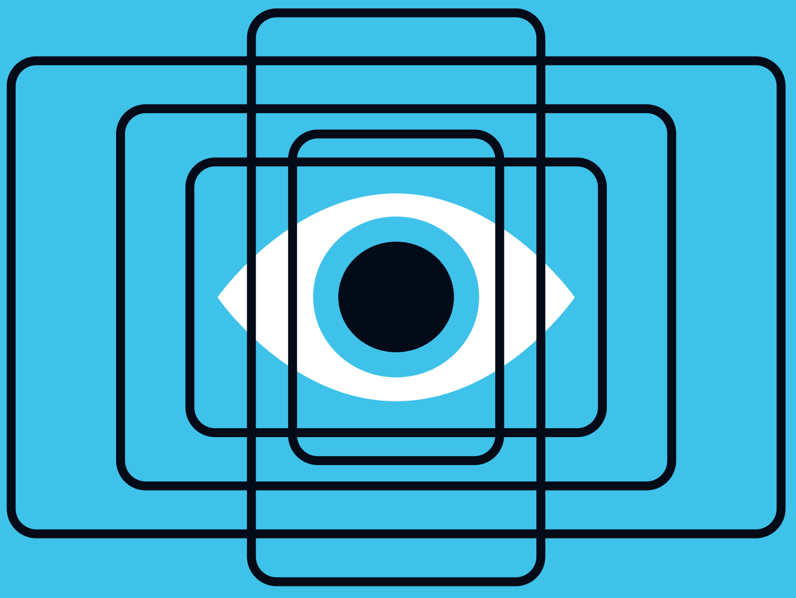
Sara Soueidan's "Tips for Optimising SVG Delivery for the Web"[202] and Chris Coyier's articles "Using SVG"[203] and "Tools for Optimizing SVG",[204] and his *Practical SVG* book[205] are excellent.

I've also found Andreas Larsen's "Optimizing SVG" posts (part 1[206] and part 2[207]) enlightening. "Preparing and Exporting SVG Icons in Sketch"[208] by Anthony Collurafici was also a great read.



202 https://smashed.by/svgdelivery
203 https://smashed.by/usingsvg
204 https://smashed.by/optimizationtools
205 https://smashed.by/practicalsvg
206 https://smashed.by/optimizingforweb1
207 https://smashed.by/optimizingforweb2
208 https://smashed.by/exportingsvg





Part Three

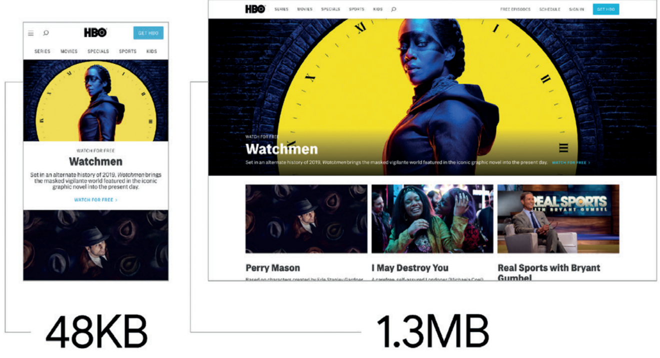
# Images in Browsers

Chapter 11 Responsive Images ...... 223
Chapter 12 Progressive Rendering Techniques.............. 238
Chapter 13 Caching Image Assets ... 256
Chapter 14 Lazy-Loading Images .....290
Chapter 15 Replacing Animated GIFs. . 314
Chapter 16 Image Content Delivery Networks........ 337

CHAPTER 11

# Responsive Images

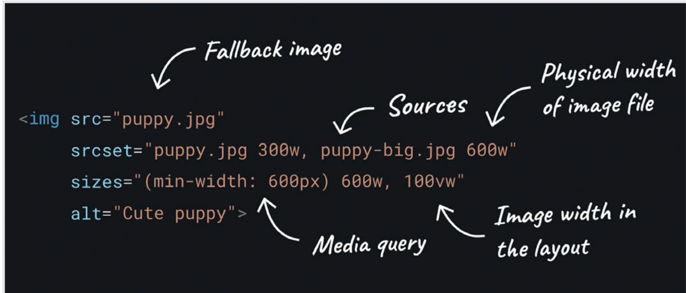
Using responsive images is the practice of serving multiple versions of the same image so the browser can choose the version that works best for the user's device. It's a key part of delivering a fully responsive web design.




*Using responsive images, HBO is able to deliver close-up high-resolution artwork to Retina devices while delivering smaller file sizes to mobile without sacrificing art direction.*

With responsive images, we can supply the browser with a variety of image resources depending on display density, size of the image element, or a number of other factors. For example, on high-resolution (2x) displays, high-resolution graphics can deliver sharpness.

In this chapter, we will cover a number of techniques for defining responsive images. We will build off the powerful `<img>` element, which can already download, decode, and render content in modern browsers across a range of different formats. Here's a sneak preview:




*The `<img>` tag can define multiple image sources of different sizes and resolutions, selecting the best fit for different situations.*

## Defining Multiple Image Sources

The `srcset` attribute enhances the `<img>` element, allowing us to supply a variety of different image sources for different device characteristics. `srcset` lets the browser select the best image; a 2x image on a 2x display, for example.

```
<img src="puppy.jpg" srcset="puppy@2x.jpg 2x"
alt="Cute puppy" />
```

When the browser encounters `srcset`, it parses the comma-separated list of images and conditions before making any image requests. Only the most appropriate image is then fetched and rendered.

```
<img src="puppy@1x.jpg" alt="Cute puppy"
     srcset="puppy@1x.jpg 1x,
             puppy@2x.jpg 2x" />
```

In the example above, we use `srcset` to handle different pixel densities. If the display is standard resolution, the browser will use the `1x` image and if the display is a `2x` high-DPI screen, we'll use the `2x`.

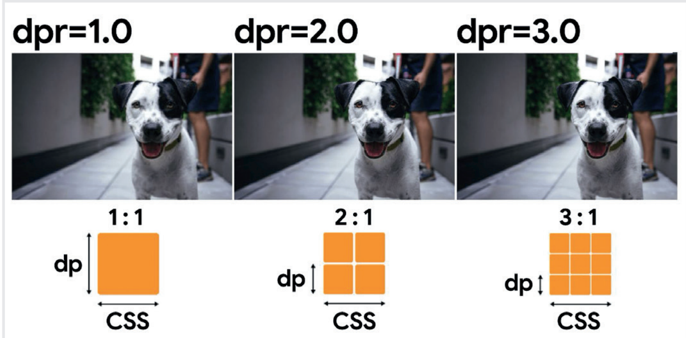
All modern browsers support `srcset`. Older, legacy browsers will simply use the default image specified by the `src` attribute. This is one reason it's key to include a 1x image that can be loaded for any device, as seen above.

When using the `x` descriptor (e.g. 1x), users will always get the same image on devices with a similar device pixel ratio, regardless of it being a large-screen laptop or a smartphone with the same device pixel ratio.

While accessibility is not the focus of this book, including the `alt` attribute on images is fundamental for those using screen readers or for any users who have images switched off.

### DEVICE PIXEL RATIO (DPR)

The device pixel ratio (DPR, and also called CSS pixel ratio) describes how a pixel in CSS translates to physical pixels on a hardware screen. High-resolution screens use more physical pixels to represent CSS pixels for more sharp visuals. To prevent blurry images on these displays, a larger image size should be loaded.

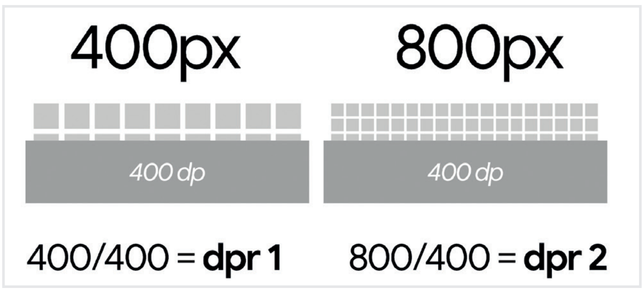



*A diagram showing device pixel ratio at 1x, 2x, and 3x, comparing device pixels to CSS pixels. Image quality appears to sharpen as DPR increases.*

Let's say we want to serve an image at 400 pixels wide:

```
<img src="puppy-400px.jpg" alt="Cute puppy"
     srcset="puppy@dpr-1.jpg 1x,
             puppy@dpr-2.jpg 2x,
             puppy@dpr-3.jpg 3x" />
```

This will deliver the best resolution for each device based on its device pixel ratio. A 400×300 image at DPR=2 (2x) will actually be an 800×600 pixel image. DPR=3 (3x) will be an image 1,200×900 pixels in size.




*Device pixel ratio is the ratio of the image width to the viewport width. It can be calculated by looking at the image width/viewport width. In the case of our* puppy-400px.jpg *image, 400/400 = DPR 1, 800/400 = DPR 2, 1200/400 = DPR 3, and so on.*

This gives you full resolution for devices that support it, without delivering larger file sizes than necessary to devices where it won't improve the user experience.

# BE RESPONSIVE

Use responsive images (<picture> element, <img srcset>) to deliver users the best-sized image based on device capabilities. This can massively save on bandwidth.

> Many sites track the DPR for popular devices, including material.io[1] and mydevice.io.[2]

### WIDTH DESCRIPTOR

When it comes to selecting which image to download, the browser needs to be aware of the dimensions of each image, but it can't strictly know this without downloading each image to check. This is where the width descriptor (`w`) comes in.

```
<img src="small.jpg" alt="Cute puppy"
     srcset="small.jpg 500w,
             med.jpg 1000w,
             large.jpg 1500w" />
```

The `w` descriptor tells the browser the width of each image in pixels. This allows the browser to select the right image for retrieval, based on characteristics like the screen's pixel density and viewport size.

1 https://smashed.by/materialio
2 https://smashed.by/mydeviceio

Use either width descriptors **or** pixel densities on all of your sources. Avoid setting both in the same `srcset` as this is considered invalid.

As a recap:

By width

```
<img src="cat.jpg"
  srcset="cat-240.jpg 240w,
          cat-480.jpg 480w
          cat-960.jpg 960w>
```

Width descriptor
w ≈ px

By density

```
<img src="cat.jpg"
  srcset="cat-1x.jpg 1x,
          cat-2x.jpg 2x,
          cat-3x.jpg 3x">
```

window.devicePixelRatio
Defines the relationship between device pixels and CSS pixels for a particular device.

*The* `srcset` *attribute accepts a width descriptor* ***or*** *a device pixel ratio (the number of device pixels per CSS pixel it is related to). For a DPR of 1,* cat-1x.jpg *is used; when DPR is 2,* cat-2x.jpg *is used; and for a DPR of 3,* cat-3x.jpg *is used.*

- Responsive images can serve different widths of an image **or** different densities of an image.
- Density refers to the device pixel ratio, or pixel density, of the device that the image is intended for.
- Older CRT monitors and pre-Retina screens may have a pixel density of 1, while Retina displays have a pixel density of 2.

## Specifying Sizes

The `sizes` attribute enhances the `<img>` element by providing the sizes of the element a `srcset` is attached to, which lets the browser use the most appropriate image. In this example, we inform the browser that the image will be displayed at 60% of the viewport width `(sizes="60vw")`.

```
<img src="small.jpg" alt="Cute puppy"
     srcset="small.jpg 600w,
             medium.jpg 1000w,
             large.jpg 1500w" sizes="60vw" />
```

If the width of the browser is 1,024 pixels wide, the image will be rendered at 512 pixels. The browser would select *small.jpg* as it is the smallest image that is still larger than the viewport width.

## Specifying Alternative Versions of an Image for Different Display Scenarios

The `<picture>` and `<source>` elements allow us to specify alternative sources for the same imagery. This can be used to deliver different image formats or for art direction (which we'll cover shortly). In this example, the browser will load the image in the first `<source>` element it can understand. If the

browser is unable to read the files specified in any `<source>` elements, the default image `src` will be loaded instead.

```
<picture>
    <source srcset="puppy.webp" type="image/webp">
    <source srcset="puppy.jpg" type="image/jpeg">
    <img src="puppy.jpg" alt="Cute puppy">
</picture>
```

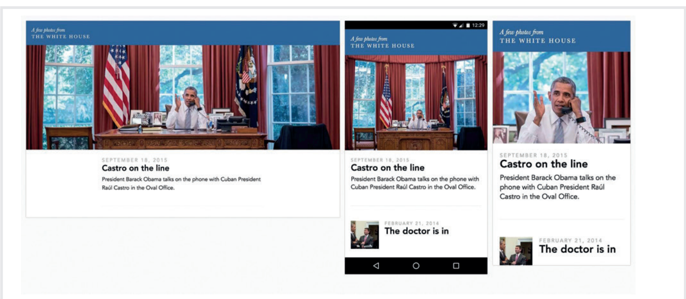
Here, the first `<source>` element includes a WebP image and the second and default `<img>` sources contain a JPEG as a fallback.

### ART DIRECTION

Although shipping an image at the best resolution to users is important, some sites also need to think about delivering the best artwork to account for device constraints. If a user is on a smaller screen, you might want to crop the image or zoom in on the subject to make the best use of available space.

SUCCESS STORY ■ "Instagram embeds allow third-party sites to display Instagram content on their own site. As a result of serving multiple image sizes, Instagram was able to reduce image transfer size by 20% for their Instagram embeds."

—Instagram (Nov, 2018)




*Responsive art direction in action: adapting to show more or less of an image in a cropped manner depending on the device.*

The `<picture>` element can be used for art direction. `<source>` can accept an optional `media` attribute that can take a media query. When this media query is triggered, the image in the `srcset` is loaded.

Only a single image will be loaded at a time. In the following example, the browser would load: *puppy-large.png* when the window width is 700 pixels or larger; *puppy-medium.png* when the window width is at least 512 pixels, but no more than 700 pixels; and *puppy-small.png* when the window is less than 512 pixels wide.

```
<picture>
    <source media="(min-width: 700px)" srcset="puppy-large.png">
    <source media="(min-width: 512px)" srcset="puppy-medium.png">
    <img src="puppy-small.png" alt="Cute puppy">
</picture>
```

Should the browser support neither `<picture>` nor `<source>`, the fallback image specified in `<img src>` would be loaded instead (*puppy-small.png*).

> If you would like to avoid writing image selection logic yourself, you may be interested in client hints:[3] a set of HTTP request header fields enabling automated delivery of optimized assets, like negotiation of image DPR resolution. At the time of writing, this is a feature only available in Chromium-based browsers (Chrome, Edge, Opera, and so on.).

## Bringing It All Together

Here is a more complex example that combines the different responsive image techniques discussed so far. In this example, we:

- Combine `<img srcset>` and media queries to specify images for different viewports.
- Use `<picture>`, `<source>` and `srcset` to provide different images for different pixel densities.

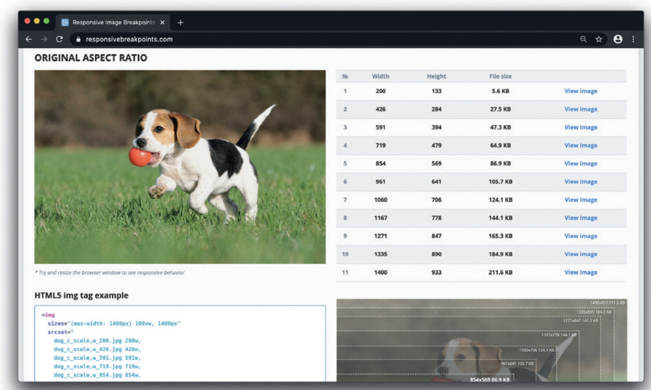
3 https://smashed.by/clienthints

```
<picture>
  <source media="(min-width: 1000px)" srcset="puppy_large_1x.jpg 1x, puppy_large_2x.jpg 2x">
  <source media="(min-width: 500px)" srcset="puppy_med_1x.jpg 1x, puppy_med_2x.jpg 2x">
  <img src="puppy_small.jpg" alt="Cute puppy">
</picture>
```

# Tools

## RESPONSIVE BREAKPOINTS

Would you like to create responsive images fast? Try Responsive Breakpoints.[4] This free tool determines the



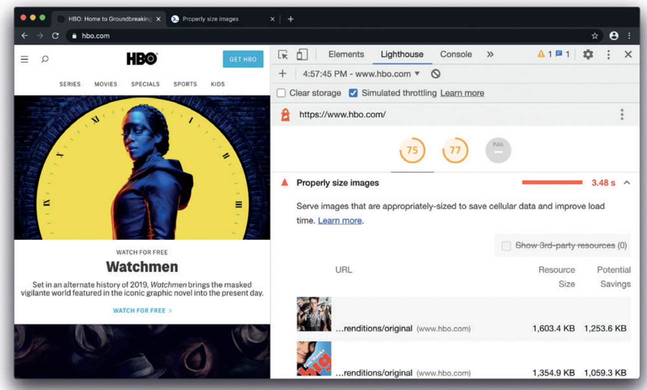

*The Responsive Breakpoints tool suggests both the optimal breakpoints for an image, as well as generating* `<img>` *HTML with* `srcset` *and* `sizes` *already filled out.*

4 https://responsivebreakpoints.com

optimal image breakpoints, generates `<img srcset>` code for you (complete with width descriptors) and creates a *.zip* of responsive images.

## LIGHTHOUSE IN CHROME DEVTOOLS

Lighthouse[5] can highlight images on your page that aren't appropriately sized, along with the potential savings. Responsive images are a great way to address this problem.




*The Lighthouse panel in Chrome DevTools highlighting that images on the page could be delivered more optimally if they were appropriately sized for mobile.*

Aim to avoid serving images larger than the version rendered on the user's screen. Anything larger can result in wasted bytes that increase page load time.

5 https://smashed.by/lighthouse

### IMAGEMAGICK CLI

The ImageMagick CLI[6] tool is helpful for image processing and includes two commands: `convert` and `mogrify`, which is similar to `convert`, but can process multiple images in place. Both commands support the `-resize` (`-r`) argument.

Image resizing (for responsive image generation, for example) can take a number of forms:

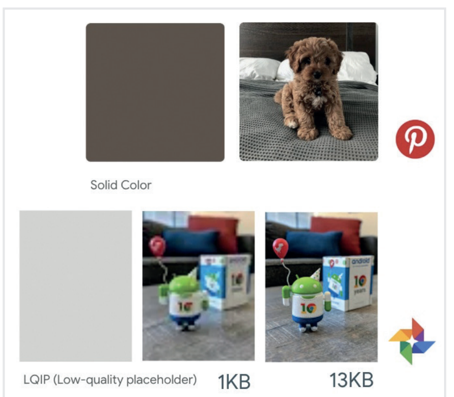
```
mogrify -resize 50% *.jpg # resize images to 50%
mogrify -resize 768x432 *.jpg # resize and keep
original aspect ratio
mogrify -r 768x432! *.jpg # resize and enforce exact
dimensions
convert source.jpg -r 768x432 output.jpg
```

6 https://smashed.by/imagemagick

CHAPTER 12

# Progressive Rendering Techniques

Short page-load time is critical for a positive user experience. Perceived load times can be shorter than the actual load time, if the user senses that the site is responsive and feels active while browsing it. The principle underlying the different techniques used to improve the perceived performance is to give users something as soon as possible rather than making them wait.

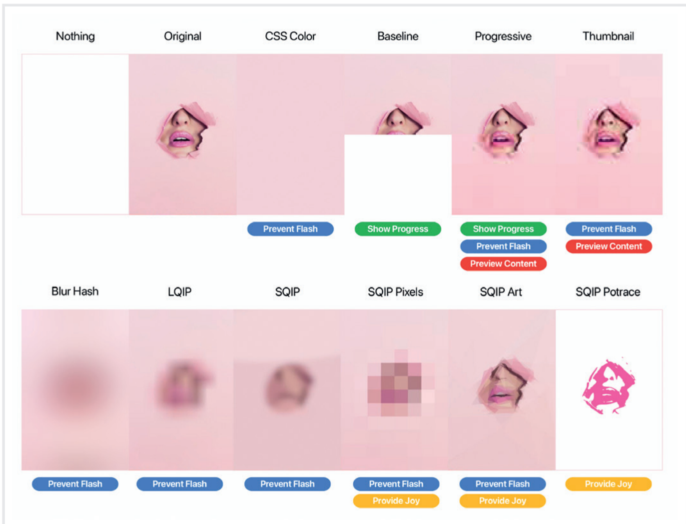



*Placeholders used for progressive image loading on Pinterest and Google Photos.*

In chapter 7 we covered progressive JPEGs. But there are many other progressive image loading techniques that can

shorten perceived load time. These techniques usually use a placeholder with a relevant substitute image shown to the user while the actual image loads. Such techniques are used on a range of large, image-heavy sites, including Pinterest and Google Photos.

This section looks at different ways of progressively loading images to improve performance.




*A visual comparison of progressive image rendering techniques by Gunther Brunner.*[7] *"Prevent Flash" includes techniques to prevent an unpleasant flash of invisible content. "Show Progress" highlights that the user knows something is happening. "Preview Content" shows low-quality partials so the user can tell if it's useful. "Provide Joy" does something interesting to keep the user engaged while waiting.*

7 https://smashed.by/brunner

## Baseline versus Progressive JPEGs

As a reminder (but do read chapter 7!), JPEGs are the most common image format on the web. JPEGs come in two varieties, baseline and progressive, and both of these have been around since the 1990s. By default, most image software saves images as baseline JPEGs. Baseline JPEGs are interpreted and displayed line by line. On the other hand, progressive JPEGs consist of multiple layers or scans of the image interlaced together such that each layer reveals a better-quality image than the previous layer. The output of each scan is merged with any outputs rendered earlier and displayed as an image. The difference between baseline and progressive JPEGs is illustrated well with this example from Liquid Web.[8]

8 https://smashed.by/liquidweb

For baseline JPEG, it appears that the user is expected to wait for the image to load, while the progressive image load gives the user some definite output in the same amount of time.

A baseline JPEG can be converted to progressive using a simple command in jpegtran:

```
$ jpegtran -progressive in.jpg > out.jpg
```

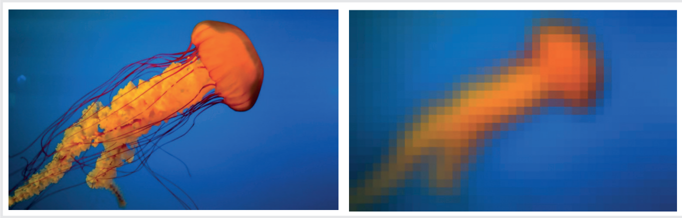
Because progressive JPEGs contain multiple scans it might be expected that they produce larger files, but that's not quite the case in practice. A study by Stoyan Stefanov in his *Book of Speed*[9] shows that images larger than 10 KB are more likely to be smaller when using the progressive JPEG format.

While most popular browsers are able to render progressive images, not all of them render the image progressively. IE 8.0 and below, Safari, and Opera render progressive images all at once, thus losing the expected advantage in terms of user experience. This is probably the reason why they are not as popular as some of the other techniques explained in this chapter.

9 https://smashed.by/bookofspeed

## Low-Quality Image Placeholders (LQIPs)

As the name suggests, LQIPs use low-quality images as placeholders to hold the user's interest while the actual high-quality, large-size images are loading. While with progressive JPEGs there is always a single image that gets progressively loaded, the LQIP technique relies on the creation of two versions of the same image. The low-resolution image file is automatically of lower size. The following example shows an actual image with its LQIP counterpart.



Here are the steps required to implement LQIPs:

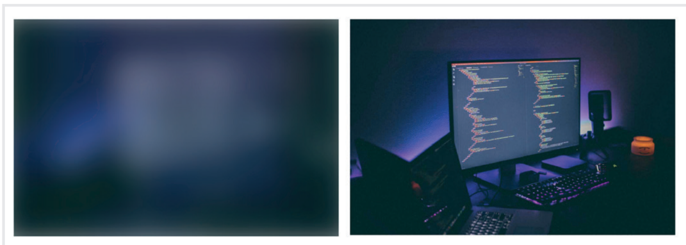
1. Create the low-quality images corresponding to every high-quality image you want to render progressively. Low-quality images can be generated in advance and saved on the server with all the other static content. In the case of dynamic content, they can also be generated at runtime using online tools: the Image and Video Manager[10] from Akamai and Cloudinary's image transformation[11] feature.

10 https://smashed.by/imagemanager
11 https://smashed.by/imagemanipulation

2. Load the LQIP images using the HTML <img> tag with src attribute. This will ensure that the page is loaded with the LQIP version of the images.

3. Swap the low-quality images with the actual images using JavaScript after the screen is loaded. The window.onload event can be used to call a JavaScript function that swaps the source paths of the LQIP images with the actual images, as shown in Carol Gumby's imgix blog post.[12]

LQIPs work best when they resemble the original image. Some developers tend to blur their LQIPs to such an extent that it provides no additional value in terms of user experience. When a LQIP is swapped with the actual image, the user registers the change, causing a blip in the user experience. This is the opposite of what we want.



*Blurry LQIP on Medium.*

12 https://smashed.by/lqip

Non-user-friendly LQIPs, like the example from Medium, seem to be the primary reason why this technique is sometimes criticized.[13] If you choose to use LQIPs, be sure your approach actually sets accurate expectations for what the final image will be.

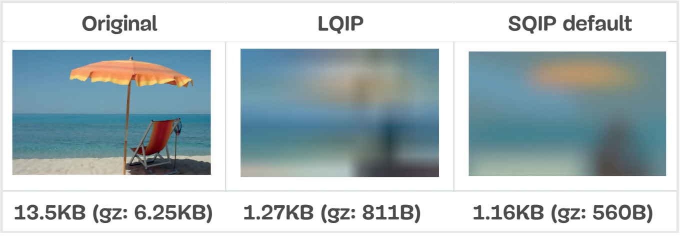
> In many cases developers use a low-resolution JPEG with a CSS blur filter. This can overwhelm the GPU and cause serious performance issues. The filters are often not applied once: they are re-applied on every composite. One way to work around this would be to blur in `<canvas>`, but you would need to be careful of main-thread CPU time.

## SQIP with SVG Images

SVG uses XML to display graphics or images (see chapter 10). As it is not pixel-based like JPEG or PNG, it offers higher scalability and versatility. JPEGs used as LQIPs tend to look very coarse and pixelated on increasing the compression. SVG is

13 https://smashed.by/placeholders

resolution-independent as it uses the shape and dimensions of vectors or shapes to generate the image. As such, scaling it in size does not make it look pixelated.




*A comparison of an original image, a low-quality placeholder, and a SQIP version.*

SQIP[14] is a tool created by Tobias Baldauf for implementing LQIP with SVG images. The SQIP tool can be used to create SVG-based image placeholders which are 800 to 1,000 bytes in size, look smooth as compared with LQIP, and help to render images progressively.

The requirements and installation instructions for the tool are available on the project's GitHub page. The following command may be used to generate the placeholder SVG from an input JPEG file:

```
sqip -o output.svg input.jpg
```

14 https://smashed.by/sqip

SQIP provides options to control the number and type of primitive shapes used (triangle, circle, and so on), and options to control the Gaussian blur in the resultant SVG image. SQIP provides an improvement over basic LQIP in terms of byte size and image quality, and should provide a better user experience if the SQIP image bears closer resemblance to the original image.

> Make sure you measure the performance impact of SQIP in several browsers. There have been cases reported[15] where performance is good in one browser, but the GPU takes a hammering in another. These problems can be further exacerbated on mobile devices.

## Gradient Image Placeholders

Gradient image placeholders are the easiest to implement as image placeholders, as they do not require any additional images to be created, nor any extra requests to load first the low- and then the high-quality images. They can be implemented using simple CSS syntax for creating linear or radial

15 https://smashed.by/lenymo

gradients of any size: the `background: linear-gradient` and `background: radial-gradient` properties.

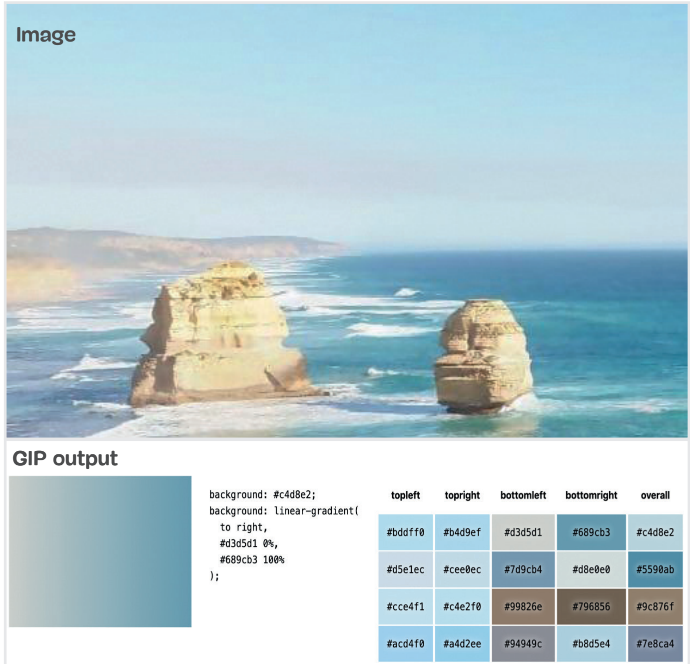
Most implementations of gradient image placeholders rely on identifying the two most prominent colors in an image and plotting a linear or radial gradient between them. These gradients are displayed as placeholders until the image is loaded.

The two most prominent colors could be determined manually for each image, but will have to be determined at runtime if there are multiple unknown images that might get displayed, as is the case with social media or e-commerce applications. Stoyan Stefanov has created a tool called GIP[16] which determines the four most dominant colors in each of the four quadrants of an image and then uses these to plot the gradient. GIP[17] can be incorporated in JavaScript to set the background and gradient for an image placeholder. The GIP function in the tool returns an object with three properties:

```
css: "background: #ab9f92; background: linear-
gradient(135deg, #cbc6c2 0%, #5d5347 100%)"
background: "#ab9f92"
gradient: "linear-gradient(135deg, #cbc6c2 0%, #5d5347
100%)"
```

16 https://smashed.by/gradient
17 https://smashed.by/gip

I used the demo for the GIP module[18] to generate the background gradient corresponding to the following image:




Of course, while GIP offers a low-fidelity solution as an image placeholder, it is completely based on color and gives no idea about the shapes composing the image. This might not be ideal when the designers or business expect users to get a clearer idea of what is to come.

18 https://smashed.by/gipdemo

A number of interesting, if experimental, alternative approaches exist outside of those covered so far. These include dynamic image tracing,[19] where a JavaScript library traces a low-res thumbnail of an image (< 5 KB), generates an SVG from it, and then animates this in, providing a line structure for what the final image may look like.

## Avoiding the display:none Trap

Older responsive image solutions have mistaken how browsers handle image requests when setting the CSS `display` property. This can cause significantly more images to be requested than you might expect, and is another reason `<picture>` and `<img srcset>` are preferred for loading responsive images.

Have you ever written a media query that sets an image to `display:none` at certain breakpoints?

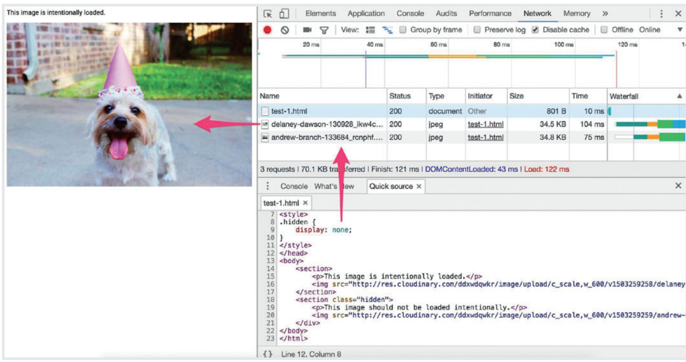
```
<img src="img.jpg">
<style>
@media (max-width: 640px) {
```

19 https://smashed.by/tracing

```
    img {
      display: none;
    }
}
</style>
```

Or toggled what images are hidden using a `display:none` class?

```
<style>
.hidden {
  display: none;
}
</style>
<img src="img.jpg">
<img src="img-hidden.jpg" class="hidden">
```




*Images hidden with* `display:none` *still get fetched.*

A quick check against the Chrome DevTools Network panel will verify that images hidden using these approaches still get fetched, even when we expect them not to be. This behavior is actually correct per the embedded resources spec.

Does `display:none` avoid triggering a request for an image `src`?

```
<div style="display:none"><img src="img.jpg"></div>
```

No. The image specified will still get requested. A library cannot rely on `display:none` here as the image will be requested before JavaScript can alter the `src`.

Does `display:none` avoid triggering a request for a `background:url()`?

```
<div style="display: none">
  <div style="background: url(img.jpg)"></div>
</div>
```

Yes. CSS backgrounds aren't fetched as soon as an element is parsed. Calculating CSS styles for children of elements with `display:none` would be less useful as they don't impact rendering of the document. Background images on child elements are neither calculated nor downloaded.

Jake Archibald's Request Quest[20] has an excellent quiz on the pitfalls of using `display:none` for your responsive images loading. When in doubt about how specific browsers handle image request loading, pop open their developer tools and verify for yourself.

## Choosing a Progressive Image Rendering Strategy

For any approach to providing image placeholders, the devil is in the details. Performance is not always about byte size here; in some cases, while SQIP size can be smaller than a blurred JPEG in LQIP, the computation cost on a client machine can be higher.

Because the web lacks an `img:loading` CSS pseudo-selector, you have to try to balance the cost of enabling LQIP/SQIP (JavaScript) with the cost of executing the approach (LQIP/SQIP running in the browser).

When deciding on your approach, here are a few questions you should think about:

- Does the placeholder offer tangible value? Is it a good approximation of the final image? If not, consider whether the real performance cost is worth the improvement to perceived performance.

20 https://smashed.by/quest

- How much of the placeholder can be computed ahead of time (for example, during a build step)?

- How much of the placeholder can't be done ahead of time? If you're using a CSS blur filter on a low-res JPEG, that computation is done in the browser. The browser has to process SVG placeholders too, so it's important to measure the performance implications of your implementation regardless of whether you are using LQIP or SQIP.

- How much of the placeholder can be inlined (in Base64, for example)? Can you avoid going back out to the network for both the low-res placeholder and the high-res final image? Some implementations go as far as to race the queue of low-res versus high-res images, and this can add lots of complexity to the equation. Again, the answer to this question can be found by measuring the performance of each scenario.

- How many images in the viewport need the placeholder? This matters because the aggregate cost of LQIP against SQIP may influence your decision.

- Do you need to worry about variable network availability? Most LQIP and SQIP implementations rely on JavaScript to make the switch from low- to high-res images when the final image is available. A JavaScript-heavy implementation may delay the user seeing any images

at all if it takes a while before the lines responsible for this are actually transferred and executed.

- Will your specific users react better to a blurry image preview (JPEG LQIP) or a preview with a sharper silhouette or shapes (SQIP)? (Glenn McComb has published an interesting comparison of different placeholders.)[21]

- Is this a short-term or long-term project? Consider your project timeline and whether you need to add this optimization now or if you can wait for solutions like JPEG XL (see chapter 19) that attempt to incorporate low-quality image placeholders as part of the format.

The effort it takes to set up LQIP and SQIP is roughly the same, and both have a potential performance cost. To choose the right strategy, consider how each option fits your project, and make sure you measure the performance impact in multiple browsers.

Having covered the different solutions available for progressive images, we can say that just like any other UX design problem, there isn't a one-size-fits-all solution available in this case either. Some users would like to see a substitute image before the original image loads, while others may get irritated by the switch.

21 https://smashed.by/placeholdercomp

A 2014 study by Tammy Everts, "Progressive Image Rendering: Good or Evil?"[22] tried to gauge user reactions to different techniques but failed to prove anything conclusive. However, it did highlight that users could be sensitive to how images render. So images need to be rendered as quickly, clearly, and simply as possible. It should also be noted that, since the progressive layer has to be decoded and rendered separately, it may take some processing power away from the normal page rendering process.

It is important to remember that these techniques were designed to keep users with slow internet connections engaged. Every website should cater to such users. The solution implemented should meet the needs of this subset of users, and the expected number of targeted users on a slow connection should be able to justify the additional cost of implementing a specific image-loading solution. Simple gradients would probably serve the purpose if the images are not dynamic, while LQIP and SQIP could be used where a more sophisticated approach is required. The goal should be to enhance the user experience and to ensure that the progression from the low-quality substitute to the original image is subtle and smooth.

22 https://smashed.by/progrendering

CHAPTER 13

# Optimizing Network Requests with Caching and Preloading

## HTTP Caching

Downloading files such as images or videos over the network can be slow and costly. Large files may require several round trips between browser and server to fetch them in full. Similarly, the loading of web pages can be delayed if critical resources, such as hero images, are still waiting on the network. Ideally, we should avoid keeping users waiting or paying a cost on their data plan as each extra network request can be a waste of money.

HTTP caching enables browsers to store a copy of a downloaded resource and serve it back when a page requests it again. When the HTTP cache has a requested resource, it intercepts the request and returns the copy on the file system rather than refetching it from the network. This reduces the load on servers, which don't always need to serve the same resources to users who have previously visited the page or site. It also optimizes performance as it's often quicker to read a resource from the local cache than fetching it from

the original server. When done correctly, HTTP caching can be a powerful tool for ensuring resources are cached until they change, rather than any longer than this.

HTTP caching allows us to:

- Control which network responses can be cached.
- Configure for how long responses can be cached (using `max-age` and Cache-Control or Expires).
- Customize the validators used for checking if responses are stale (such as ETag or Last-Modified).
- Perform a forced revalidation if needed.

HTTP caching in all modern web browsers is a widely agreed specification, making it easy to incorporate in web applications. Your application will significantly benefit from appropriate use of these requirements, optimizing response times, and reducing server load. Inaccurate caching, though, may cause users to see out-of-date content and bugs that are difficult to debug.

### REQUEST AND RESPONSE HEADERS

There are two things a browser needs to know in order to cache a file in the HTTP cache: how long it's permitted to

cache this file, and how to determine whether this file's content is fresh. When your browser receives a response from a network, it often indicates via headers if the resource can be cached and for how long, and the age of the file.

The behavior of the HTTP cache is controlled by request and response headers. Web developers should control both the code for our sites (request headers)[23] and the web server part (response headers).[24] There are a few primary HTTP headers that are effective in caching.

### Last-Modified

The Last-Modified header[25] uses a date and time strategy to decide if a file has changed. It looks at when the origin believes the file was last modified and is a good validator for checking if a file received or stored is the same. It's less accurate than the ETag header, which is content-based.

```
Last-Modified: Mon, 20 Jul 2020 11:43:22 GMT
```

### Cache-Control

Cache-Control[26] keeps instructions for caching in both requests and responses. Servers can return a Cache-Control to state how and for what length of time the browser and other caches should store responses.

23 https://smashed.by/requestheader
24 https://smashed.by/responseheader
25 https://smashed.by/lastmodified
26 https://smashed.by/cachecontrol

```
Cache-Control: public, max-age=600
```

To get started with Cache-Control, here are a few pointers:

- For resources that should be stored for a fixed period of time:
  `Cache-Control: max-age`
  (good for assets that are versioned).
- For resources that need to be revalidated each time they are used:
  `Cache-Control: no-cache.`
- For resources that should never be cached:
  `Cache-Control: no-store.`

### ETag

ETag[27] is an HTTP response header that identifies a specific version of a file. When browsers find a cached response that has expired, they can send a small token (often a hash of the file's content) to the server to validate if the file has been modified.

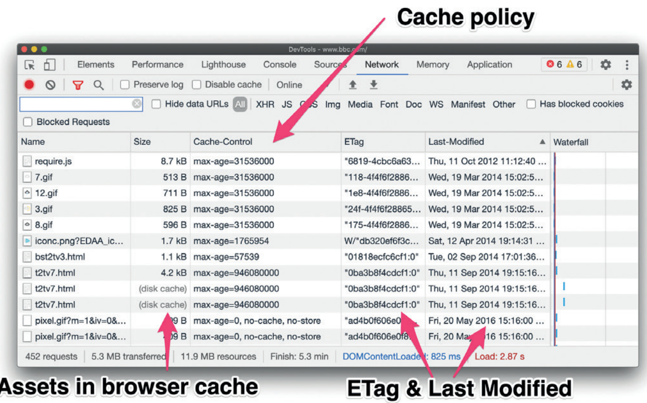
```
ETag: "v456.2.01"
Cache-Control: max-age=600
```

27 https://smashed.by/etag

Should the server return the same token, the file is the same and there is no need to download it again. ETags allow caches to be much more efficient as servers don't need to send a full response if content hasn't changed.

### CACHE DURATION

How long you cache a resource depends heavily on the sensitivity of what you are caching. Versioned images or JavaScript can be cached for a long period of time, while resources that are not versioned likely need a shorter cache time so that users are guaranteed to get a fresh version.




*The Chrome DevTools Network panel can optionally show the values of different caching headers, including Cache-Control, Last-Modified, ETag, and whether the file is already in the HTTP (disk) cache.*

There are often many different layers[28] in the modern tech stack where a site may leverage caching. These include the web browser (HTTP cache, service worker cache), CDNs and image CDNs (cache close to users on the edge), caching proxies that can sit in front of the site, and, finally, intermediate caches such as those between your site and database.

### Freshness

When a file gets downloaded and stored in the HTTP cache, one could imagine it being servable from there forever. However, as the user's file system has finite storage that can change over time, browsers also have finite storage and may need to periodically purge items from the cache to free up space. This is known as cache eviction.

Servers need to inform browsers of an expiration time for their resources. Prior to this expiration, a file is considered fresh and up to date; after this time, the file is stale or out of date.

This stale file isn't evicted immediately from the cache, but when the browser checks the cache for a stale file, it sends this request with an If-None-Match header to ensure it's up to date. Servers will return a 304 Not Modified[29] header but won't send the body to reduce bandwidth consumption.

---

28 https://smashed.by/fourcaches
29 https://smashed.by/status304

Freshness is based on many different headers. With the `Cache-Control: max-age=N` header, the length of freshness is `N`. If the header is missing, it's checked to see if an Expires header is in place. If it is, the freshness lifetime is its value minus the Date header value.

Should an origin server not configure a freshness preference using Cache-Control/Expires, it's possible other heuristics may be used.

### stale-while-revalidate (SWR)

The `stale-while-revalidate`[30] HTTP Cache-Control directive sets a grace period in which browsers can use an out-of-date (stale) asset while checking on a new version. This hides network and server latency from clients.

```
Cache-Control: max-age=31536000, stale-while-
revalidate=86400
```

This Cache-Control header states the amount of time in seconds a file should be cached for (`max-age=31536000` – this file should be good for a year). After a year, you have one day to keep serving this stale asset, while it is asynchronously revalidated in the background (`stale-while-revalidate =86400` – one day in seconds).

30 https://smashed.by/revalidate

With this directive, you can balance delivering on immediacy – serving content that is cached straightaway – and freshness – making sure that updates to the cached content are used next time. In browsers that don't support `stale-while-revalidate`, it will be ignored and `max-age` values will be used instead.

### Using URL Versioning for Long-Lived Caching

URL versioning (also referred to as "revving" or "hashing") is a helpful way to ensure you invalidate cached responses. Add `Cache-Control: max-age=31536000` to responses for versioned resources if they include contents you don't believe will ever change – this value represents a full year. Static image resources like a logo, illustration, or UI element are good candidates for such resources.

When you set a value like this, it informs the browser that it can directly use the cached response in the HTTP cache if the URL attempts to be loaded, without ever needing to make a request to the server at all. Automating file versioning[31] can be achieved with build tooling like webpack.

### Revalidation for Non-Versioned URLs

It's not possible to entirely skip the network using HTTP caching. To optimize caching for URLs that are not versioned you can use one of the following Cache-Control values:

31 https://smashed.by/versioning

- `public:` any cache can store these responses.
- `private:` intermediate caches can't cache the file, but browsers can.
- `no-store:` both the browser and intermediate caches such as CDNs should never store a version of the file.
- `no-cache:` the browser needs to revalidate with the server each time before a cached version of the URL can be used.

### Vary Header

It's important to understand if the file being requested is cached. This may seem straightforward, but often a URL on its own isn't sufficient to tell. Particular requests (let's say *index.html*) could be modified specifically for mobile users. To solve this problem, browsers give each cached file a cache key (a unique identifier). This cache key is just the URL of the resource by default, but we can add other details to it using the Vary header.[32]

```
Vary: Accept-Encoding
```

A Vary header informs the browser to add the value of request header values to the unique cache key. A pretty common example of this is done using compression,

32 https://smashed.by/vary

where `Vary: Accept-Encoding`[33] will create different entries in the cache for different Accept-Encoding values (e.g. `gzip` or `br` for Brotli). `Vary:Accept-Encoding, User-Agent` is also a popular directive that informs the browser to vary cached entries by both Accept-Encoding and User-Agent string values.

### IMPROVING CACHE HIT RATIOS

When the cache is able to fulfill a request for a file rather than needing to go back to the network to retrieve it, this is referred to as a "cache hit." If a user navigates to a page with a hero image of a dancing bear, the browser may request this image from the origin's image CDN. A cache hit from the browser's perspective would be this file being readable from the local HTTP cache.

Similarly, the image CDN may view a cache hit as there already being a copy of the dancing bear image in its storage that can be quickly sent back to the browser, perhaps requiring that the image be fetched from the origin instead.

A "cache miss" happens when the cache does not have a copy of the requested file. For an image CDN, if the dancing bear image is not in its storage cache, the request would need to defer back to the origin server to serve the image instead. The image CDN would then cache the image when

33 https://smashed.by/understandingvary

the origin responds to the request so future requests for it would result in a cache hit.

The cache hit ratio measures how many requests for content a cache can successfully fulfill compared to the number of requests it receives.

*The formula for cache hit ratio (Source: Cloudflare)*[34]

When dealing with images, a number of criteria can impact cache hit ratio:

- **Serving images with a shorter cache lifetime.** Image assets that are unlikely to change often can be cached with a much longer duration, in particular if URL versioning is being used should you need to perform an update. Too short a cache lifetime on such resources can decrease the chances of getting a cache hit.
- **Serving images on different URLs.** It's possible that you have an or custom image CDN that serves the exact same content but from different URLs. This can cause

34 https://smashed.by/hitratio

content to be downloaded and stored several times. For example, *foo.com/logo.jpg?w=500,h=100* and *foo.com/logo.jpg?h=100,w=500* are technically two different URLs, which will result in different requests being made to the server for the same resource.

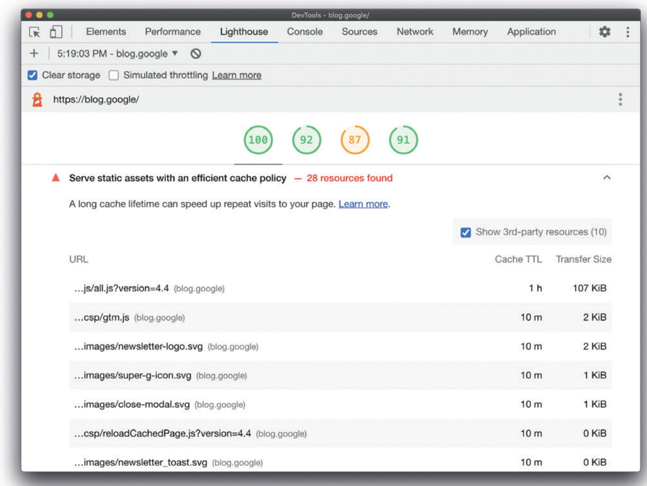
- **Serving images on a URL with a timestamp.** Sometimes site owners will append a timestamp to URLs to force the latest version of a resource to be delivered (for cache busting via *foo.com/logo.jpg?t=1365253323*, for example). This changes the URL being cached and can mean that while users will get the latest version, they may never benefit from caching as the URL being requested is always considered different. A better approach here would be to rely on revving or file versioning in the URL string.

- **Serving a broad range of responsive image variations.** It's possible you, your build process, or design team have a large number of image variations used to target different DPR values. If leveraging a CDN to serve these images, users may be the first ones to request a particular DPR before it is cached at the CDN layer. This may be fine in practice, but it's worth being aware of.

### IDENTIFYING CACHE OPPORTUNITIES

Lighthouse[35] in Chrome DevTools highlights opportunities to improve web pages. It includes an "efficient cache policy" audit,[36] which can evaluate if a page can benefit from a stronger set of caching rules.

The audit works by comparing the value of the Last-Modified header (how old the content is) with the time to live (TTL) of the cache and estimating the likelihood of a file being served from the cache.




*Lighthouse presenting a set of caching recommendations for specific requests.*

35 https://smashed.by/lighthouse
36 https://smashed.by/efficientcache

## Offline Caching with Service Workers

Next up are service workers.[37] Service workers and the HTTP cache serve the same purpose, but service workers provide more fine-grained control over what files get cached and how caching is completed.

Service workers are a type of web worker[38] that afford web pages greater control over network requests. They are effectively JavaScript files that run a lot like a proxy server, enabling you to modify both requests and responses, cache these requests to optimize performance, and enable offline access to content that has been cached.

The primary benefits of using service workers are:

- Enabling offline support in your web page or web application.
- Optimizing page load performance (e.g. from caching hero and article images).
- Accessing advanced browser features (e.g. push notifications).

There are two primarys that service workers use to make pages work offline. These are the Fetch standard[39] (a standardized way to download content from the network) and

37 https://smashed.by/serviceworkerapi
38 https://smashed.by/webworkers
39 https://smashed.by/fetchapi

the Cache[40] (a storage mechanism for page data that is both persistent and independent from the HTTP cache).

If you are building a progressive web application[41] (a web app that is installable) or just want to add offline caching to your page, service workers can enable you to cache both static and dynamic URLs, including images, videos, and other kinds of media. Similar to the HTTP cache, caching files with service workers and the Cache can make content load quicker under many kinds of network conditions.

Static image assets can include favicons, SVG icons, hero images, illustrations, and other images that are not likely to change regularly. Dynamic image assets may be those that may not be known easily ahead of time (for example, those returned from an external) or which may be user-uploaded content that can't be known at build time.

> Service workers will only run on HTTPS. As they can intercept and modify network responses, it's important that browsers limit possible "man-in-the-middle" attacks. It's possible to use services like Let's Encrypt[42] to secure SSL certificates for your site.

40 https://smashed.by/cacheapi
41 https://smashed.by/pwa
42 https://letsencrypt.org/

## IMPROVED PERFORMANCE FROM OFFLINE CACHING

When you cache files locally, pages can load more quickly. Service workers are able to cache files locally in the browser and retrieve them without needing to go back to the network. This can enable a quicker experience for users even if they're on a fast network connection.

At the core of a modern offline experience in the browser is your service worker. It allows you to select when files should be cached and when content should be fetched from the cache instead of going back to the network. One way to handle offline caching for static resources such as images is the following:

1. Register a service worker. A first visit to your page triggers the service worker installation flow.

2. On installation, cache the static resources for the page or site. Typically this is the core set of HTML, CSS, JavaScript, and images the page needs to be opened.

3. Set up the service worker to listen for file fetches. When a file is being fetched, the service worker will try to find it in the local cache (via the Cache) before going back to the network to download it.

4. Your service worker should cache copies of files that need to be fetched from the network so in future they can be retrieved from the local cache instead.

There are many other recipes for caching approaches available in Google's Offline Cookbook[43] and Workbox Recipes.[44]

To register a service worker, first check for browser support and then register the service worker. The first time a user visits your page, the service worker will be installed and will activate.

*index.html*

```
<script>
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('service-worker.
js')
   .then(registration => {
     console.log('Registered:', registration);
   })
   .catch(error => {
     console.log('Registration failed: ', error);
   });
}
</script>
```

43 https://smashed.by/offlinecookbook
44 https://smashed.by/workboxrecipes

While there are many possible strategies for caching the static assets in a site, a common one is when the service worker installs. When users navigate to your site for the first time, these static files can be cached so they can be retrieved from the cache for future visits. The code below defines a name of the cache and a list of what static files need to be cached. An install event listener is created which runs code once the service worker completes installation. The install listener here opens the cache and stores the static assets for us.

*service-worker.js*

```
const CACHE_NAME = 'static-cache';
const urlsToCache = [
  '.',
  'index.html',
  'logo.svg',
  'styles/main.css'
];
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(CACHE_NAME)
    .then(cache => cache.addAll(urlsToCache))
  );
});
```

## OFFLINE CACHING MADE EASIER WITH A JAVASCRIPT LIBRARY

Workbox[45] is a set of JavaScript libraries by Google that helps you author and manage service workers and caching with the Cache. It includes robust support for a number of best-practice patterns and documented offline caching recipes to ease your path to adding offline support with examples from large production sites.

One example of what you can achieve with Workbox is to create an image cache. The code below allows your service worker to respond to requests for images with a cache-first strategy. This means that once they're in the cache, a user doesn't need to make another request for them. By default, the approach caches at most 60 images, each for 30 days:

```
import { registerRoute } from 'workbox-routing';
import { CacheFirst } from 'workbox-strategies';
import { CacheableResponsePlugin } from 'workbox-
cacheable-response';
import { ExpirationPlugin } from 'workbox-expiration';
const cacheName = 'images';
const matchCallback = ({ request }) => request.
destination === 'image';
const maxAgeSeconds = 30 * 24 * 60 * 60;
```

45 https://smashed.by/workbox

```
const maxEntries = 60;
registerRoute(
  matchCallback,
  new CacheFirst({
    cacheName,
    plugins: [
      new CacheableResponsePlugin({
        statuses: [0, 200],
      }),
      new ExpirationPlugin({
        maxEntries,
        maxAgeSeconds,
      }),
    ],
  }),
);
```

Recent versions of Workbox come with a `workbox-recipes`[46] package: a set of standardized reusable recipes to quickly set up routing and caching with service workers for common scenarios. Here's how we can use `workbox-recipes` to accomplish the same functionality as the more verbose above example:

```
import { imageCache } from 'workbox-recipes';
imageCache();
```

46 https://smashed.by/imagecache

For more guidance on how to use Workbox and service workers to set up caching and routing in your application, check out the official Workbox documentation.[47]

Effectively leveraging the HTTP cache and service workers can be a powerful way to ensure that browsers and other intermediate layers (like CDNs) work together to deliver content as efficiently as possible to users. The benefits to performance from caching (even if just for first-party content you own yourself) can be significant thanks to reductions in round-trip times and avoiding repeating requests to the network for files.

There is a lot of nuance to caching and it's worth investing time to get a deeper understanding of the different HTTP headers that represent freshness and allow us to validate entries we wish to cache. It's worthwhile auditing your site to understand opportunities to better cache resources, and tools such as Lighthouse are a good first step in getting you on your journey to effectively using browser caches.

47 https://smashed.by/workboxdoc

## Image Spriting

As we're discussing how to optimize network requests, let's also discuss image spriting.[48]

Spriting is an optimization which combines a number of images (sprites) into one "sprite sheet" image. The term was

```
.flags-canada, .flags-mexico, .flags-usa {
  background-image: url('../images/flags.png');
  background-repeat: no-repeat;
}

.flags-canada {
  height: 128px;
  background-position: -5px -5px;
}

.flags-usa {
  height: 135px;
  background-position: -5px -143px;
}
```

*A single sprite sheet composed of three flag images. By setting the same* `background-image` *on each flag class via CSS, we can then adjust the background position and dimensions to "crop" to a specific flag [Source: CSS Tricks]*[49]

48 https://smashed.by/spriting
49 https://smashed.by/csssprite

popularized in the video games industry back in the 1970s where it was originally used to composite several graphics into a single image and only display a portion of the image at a time. On older gaming hardware, this was more efficient for memory and storage.

When used in a web page, each "sprite" sets a `background-image` URL that points to the sprite sheet image and leverages offsets in CSS to display the correct image while hiding the rest of the sheet from view.

Image sprites (or CSS sprites) are supported by all browsers, and have been a popular way to reduce the number of images a page loads by combining them into a single larger image.

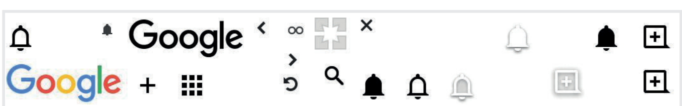



*Image sprites are still used in some production sites, including the Google homepage.*

Under HTTP/1.x, some developers used spriting to reduce HTTP requests. This came with a number of benefits, but care was needed as you quickly ran into challenges with cache invalidation – changes to any small part of an image sprite would invalidate the entire image in a user's cache.

Today, spriting could be considered an HTTP/2 anti-pattern.[50] With HTTP/2, it may be best to load individual images[51] since multiple requests within a single connection are now possible. Measure to evaluate whether this is the case for your network setup.

## Preloading

Preloading[52] allows you to inform the browser about critical resources you want to load as soon as possible, before they're discovered in HTML.

If you are optimizing largest contentful paint[53] (LCP – see chapter 22), preload can be a game-changer for boosting how soon a browser downloads late-discovered resources loaded via JavaScript, or that are delayed by other resource fetches taking precedence. Preload is supported[54] in Chrome, Edge, Safari and Firefox 84+.

```
<link rel="preload" as="image" href="hero-image.jpg">
```

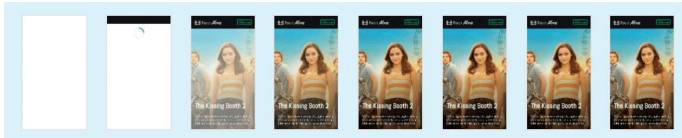
The markup version of preloading (`<link rel="preload">`) is a declarative fetch allowing you to force the browser to make a request for a resource without blocking the docu-

50 https://smashed.by/antipattern
51 https://smashed.by/individualimages
52 https://smashed.by/preloading
53 https://smashed.by/lcp
54 https://smashed.by/caniusepreload

ment's `onload` event. Browsers will more quickly download resources that might otherwise not be discovered until later in the document parsing process.

> Any resources you preload should be chosen with care so you don't accidentally cause network contention with other important files in the critical rendering path of the page.

Preloading can substantially improve LCP especially if you need critical images (like hero images) to be prioritized over the loading of other images on a page. While browsers will try their best to prioritize the loading of images in the visible viewport, <link rel="preload"> offers page authors some further control over this process.




*A hero image is the primary photo you see at the top of a web page. It may be the largest contentful element visible in the viewport, such as the movie poster in the filmstrip above.*

Preloading might be able to substantially speed up image display if you currently have:

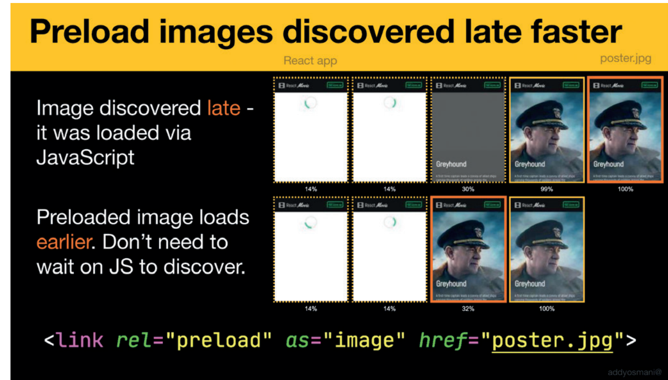
- hero images that rely on JavaScript to load them:
  - a React, Vue, or Angular component loading `<img>` tags client-side
  - client-side rendered HTML responsible for loading images
- background hero images in CSS (these are discovered really late)
- hero images that rely on JavaScript and a network fetch to load (e.g. require a JSON fetch from an to discover images)
- use a webpack loader to load in images

The key idea here is to avoid the browser having to wait for the script before beginning to load the image, as this could heavily delay when users can actually see it.

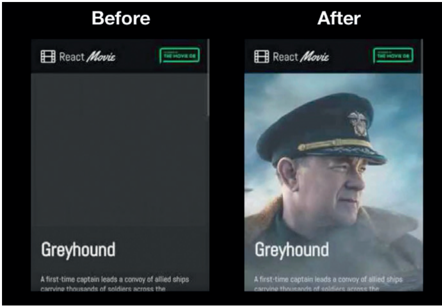
Overleaf is a WebPageTest[55] filmstrip from loading a React-based movie browser. The app uses client-side rendering (*app.js*) and also relies on a fetch to an to return a JSON feed of images (*movies.json*). This means the browser may need to

55 https://webpagetest.org/

process *app.js* before it starts fetching movies.json and can discover our hero image (*poster.jpg*).




*A filmstrip of a React app that highlights how a key poster image was discovered and loaded late. With preload, we can help the browser discover the image far sooner. This helps us reach largest contentful paint (the orange frame) one second sooner on 4G.*




*Using preload on the hero image, we are able to prioritize fetching content in the optimal order, rendering useful image content (right) instead of the empty gray background in the original version (left).*

The largest contentful paint[56] metric measures the render time of the largest image or text block visible within the viewport.

### CHOOSING WHAT TO PRELOAD

`<link rel="preload">` can be used in a few different ways to optimize the loading of late-discovered images. Preload a hero image so it's discovered before the time JavaScript outputs an `<img>`:

```
<link rel="preload" as="image" href="poster.jpg">
```

Now that browser support for formats like WebP and AVIF has improved, you might also like to know that such images can also be preloaded:

```
<link rel="preload" as="image" href="poster.webp"
type="image/webp">
```

56 https://smashed.by/lcp

> Preloading a specific format won't also preload the fallback image for browsers that don't support that format, but it will optimize loading for the ones that do.

Preload a responsive image[57] so the correct source is discovered sooner:

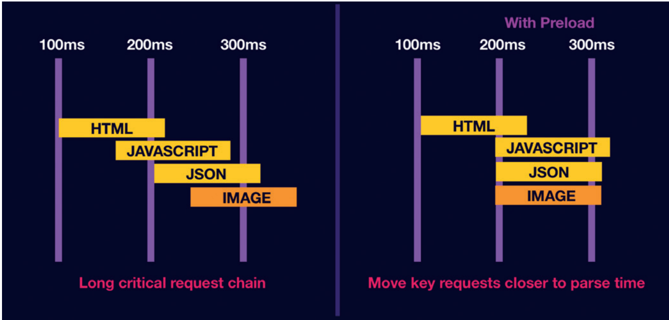
```
<link rel="preload" as="image"
     href="poster.jpg"
     imagesrcset="
        poster_400px.jpg 400w,
        poster_800px.jpg 800w,
        poster_1600px.jpg 1600w"
    imagesizes="50vw">
```

Preload the JSON as `fetch` so it's discovered before JavaScript requests it:

```
<link rel="preload" as="fetch" href="movies.json">
```

In the case of the React movies app mentioned earlier, *movies.json* requires a cross-origin fetch, which you can get working with preload if you set the `crossorigin` attribute on the `link` element:

57 https://smashed.by/responsiveimages




*When a preload is declared in markup, resources can be fetched before the HTML parser has even reached the element (e.g* `<img>`, `<script>`*) where the resource is defined. As such, preload shifts resource fetches much closer to parse time.*

```
<link rel="preload" as="fetch" href="foo.com/api/
movies.json" crossorigin>
```

For bonus points, you can also preconnect[58] to the origin that this `fetch` is going to connect to:

```
<link rel="preconnect" href="https://foo.com/"
crossorigin>
```

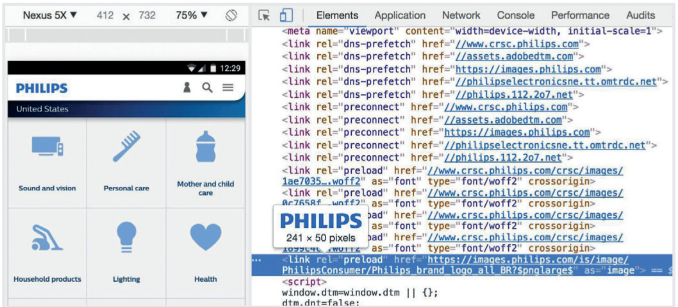
Preload the JavaScript to shorten the time it takes to discover from HTML:

```
<link rel="preload" as="fetch" href="app.js">
```

58 https://smashed.by/preconnect

> Don't overuse preload – used without care, it can regress metrics like first contentful paint. Reserve it for critical resources the browser's preload scanner can't find quickly.

Sites like Philips,[59] Flipkart,[60] and Xerox[61] use `<link rel="preload">` to preload their main logo assets (often used early in the document). Kayak[62] also uses preload to ensure the hero image for the header is loaded as soon as possible.




*Philips uses* `<link rel="preload">` *to preload its logo.*

## LINK PRELOAD HEADER

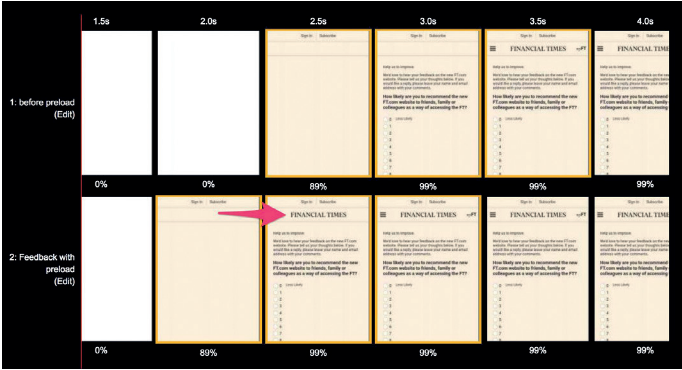
A preload link can be specified using either an HTML tag or an HTTP Link header.[63] In either case, a preload link directs the browser to begin loading a resource into the memory

59 https://www.usa.philips.com/
60 https://www.flipkart.com/
61 https://www.xerox.com/
62 https://kayak.com/
63 https://smashed.by/linkheader

cache, indicating that the page expects with high confidence to use the resource and doesn't want to wait for the preload scanner or the parser to discover it. A Link preload header for images would look like this:

```
Link: <https://example.com/poster.jpg>;
   rel=preload; as=image
```

When the Financial Times introduced a Link preload header to its site, one second was shaved off[64] the time it took to display the masthead image.




*The FT using preload. Displayed are the WebPageTest before and after traces showing improvements. Bottom: with the Link header[65] to preload; top: without.*

Similarly, Wikipedia improved time-to-logo performance[66] with the Link preload header.

64 https://smashed.by/onesecond
65 https://smashed.by/ftheader
66 https://smashed.by/timetologo

### WHEN NOT TO PRELOAD

Only preload image assets that are important and needed early on. If images aren't critical to your user experience, focus your early loading efforts on other content instead. By prioritizing image requests, you might end up pushing other important resources further down the queue.

> To learn more about preloading, see my "Preload, Prefetch, and Priorities in Chrome"[67] and "Preload: What Is It Good For?"[68] by Yoav Weiss.

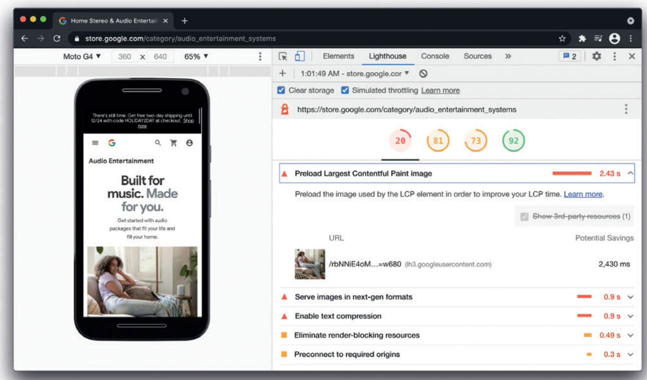
### TOOLING FOR PRELOAD

The Lighthouse panel in Chrome DevTools can suggest opportunities to preload largest contentful paint images that could be discovered late. This includes images that might be loaded as CSS background images.

The "Opportunities" section of the Lighthouse report also has a more general "preload key requests"[69] audit, which highlights requests in your critical request chain that could be good candidates to preload.

---

67 https://smashed.by/priorities
68 https://smashed.by/preloadbenefits
69 https://smashed.by/keyrequests




*The "Preload Largest Contentful Paint image" audit in Lighthouse highlighting that a hero image could be preloaded so that it loads sooner.*

Preload helps ensure critical hero images and resources get shown to users as soon as possible. It's an important web performance feature that gives developers more control over the loading sequence for files in a page, with browser support continuing to improve.

To discover if there are opportunities preload could make a difference to your app, try out Lighthouse[70] or PageSpeed Insights,[71] which have an audit for preloads.[72]

70 https://smashed.by/lighthouse
71 https://smashed.by/pagespeedinsights
72 https://smashed.by/preloadaudit

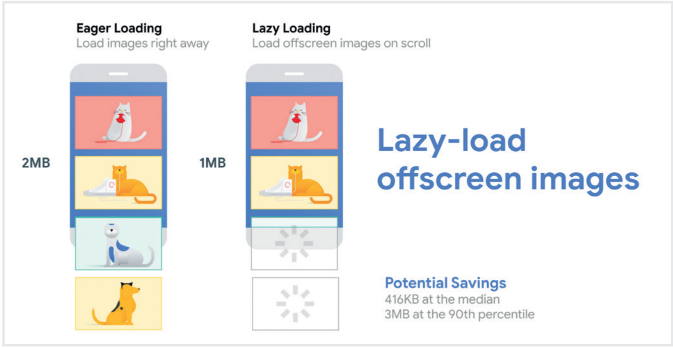
CHAPTER 14

# Lazy-Loading Offscreen Images

*With special thanks to Houssein Djirdeh and Mathias Bynens for their contributions to native image lazy-loading guidance for web developers.*

Web pages often contain a large number of images, contributing to page bloat, data costs, and how fast a page can load. Many of these images are offscreen; to see them, a user would have to scroll.

Lazy loading is a web performance pattern that delays loading offscreen images until a user needs to see them. One way to achieve lazy loading is as a user scrolls. This technique complements the data savings you see with a good image compression strategy.




*The difference between eager and lazy loading: loading images only when users will see them can improve performance.*

Images that must appear in the viewport when the web page first appears (what used to be known as "above the fold") are loaded straight away.

The images which fall offscreen, however, are not yet visible to the user. They do not have to be immediately loaded into the browser. They can be loaded later – or lazy-loaded – if and when the user scrolls down and it becomes necessary to show them.

Lazy loading is sometimes combined with a placeholder-based solution, such as a placeholder containing a color, a placeholder image, or low-resolution preview that is displayed while the original is being lazy-loaded.

## Benefits of Lazy Loading

This lazy way of loading images only if and when necessary has many benefits:

- **Reduced data consumption**: Because you don't assume users need every image fetched ahead of time, you only load the minimal number of resources. This is always a good thing, especially on mobile devices with more restrictive data plans.

# LAZY-LOAD IMAGES

Are there non-critical images you just have to keep?
Consider lazy-loading them to speed up page loads.

- **Reduced battery consumption**: Less workload for the user's browser, saving battery life.
- **Improved download speed**: Decreasing your overall page load time on an image-heavy website from several seconds to almost nothing is a tremendous boost to user experience. In fact, it could be the difference between a user staying around to enjoy your site and just another bounce statistic.

But like all tools, lazy loading should be wielded with the appropriate care and attention.

Avoid lazy-loading images that should appear in the viewport immediately. Use it for long lists of images: product shots, for example, or lists of user avatars. Don't use it for the main page hero image. Lazy-loading critical images can make loading slower, technically and as perceived by users. It can kill the browser's preloader and progressive loading, and the JavaScript can create extra work for the browser.

Be careful not to lazy-load images when users scroll. If you wait until the user scrolls they are likely to see placeholders and may eventually get images, if they haven't already scrolled past them. One recommendation would be to start lazy loading after the critical images are displayed, loading all of the images independent of user interaction.

Be mindful, too, of the cost of JavaScript, particularly on low-end devices. When implementing lazy loading choose

lightweight options that have low execution times and low battery impact.

## Implementing Lazy Loading

Without native browser support, there are two ways to defer the loading of offscreen images:

- using the Intersection Observer[73]
- using `scroll`, `resize`, or `orientationchange` event handlers[74]

Either option can let developers include lazy-loading functionality, and many developers have built third-party libraries to provide abstractions that are even easier to use.

## Lazy-Loading with Intersection Observer

Most libraries that implement lazy loading rely on the Intersection Observer[75] to track when a particular element enters or exits the viewport. Dean Hume[76] provides an im-

73 https://smashed.by/intersectionobserver
74 https://smashed.by/eventhandlers
75 https://smashed.by/ioapi
76 https://smashed.by/hume

plementation[77] for using Intersection Observer to lazy-load images on web pages and also describes how to combine lazy-loading based on Intersection Observer with SVG-based SQIP images.[78]

There are a number of techniques and plug-ins available for lazy loading. I recommend lazysizes[79] by Alexander Farkas because of its decent performance, features, its optional integration with Intersection Observer,[80] and support for plug-ins. Once you have imported it to your website, all HTML elements with `class="lazyload"` will be lazy-loaded. This can also be used with the LQIP technique:

```
<img alt="100%x200"
src="low-quality.jpg"
data-src="original.jpg"
class="lazyload" />
```

## Lazysizes

Lazysizes is a JavaScript library. It requires no configuration. Download the minified *.js* file and include it in your web page.

77 https://smashed.by/implementation
78 https://smashed.by/lazyloadingsqip
79 https://smashed.by/lazysizes
80 https://smashed.by/intersectionobserver

Here is some example code taken from the README file:

> Add the class `lazyload` to your images/iframes in conjunction with a `data-src` and/or `data-srcset` attribute.
>
> Optionally you can also add a `src` attribute with a low quality image:

```
<!-- non-responsive: -->
<img data-src="image.jpg" class="lazyload" />

<!-- responsive example with automatic sizes
calculation: -->
<img
    data-sizes="auto"
    data-src="image2.jpg"
    data-srcset="image1.jpg 300w,
    image2.jpg 600w,
    image3.jpg 900w" class="lazyload" />

<!-- iframe example -->

<iframe frameborder="0"
    class="lazyload"
    allowfullscreen=""
    data-src="//www.youtube.com/embed/ZfV-aYdU4uE">
</iframe>
```

Lazysizes features include:

- Automatically detects visibility changes on current and future `lazyload` elements.
- Includes standard responsive image support (`<picture>` and `<srcset>`).
- Adds automatic `sizes` calculation and alias names for media queries feature.
- Can be used with hundreds of images/iframes on CSS and JS-heavy pages or web apps.
- Extendable: supports plug-ins.
- Lightweight but mature solution.
- SEO improved: does not hide images/assets from crawlers.

### MORE LAZY LOADING OPTIONS

Lazysizes is not your only option. Other lazy loading libraries include: Lazy-Load XT,[81] BLazy.js[82] (or [Be]Lazy), Unveil,[83] and yall.js (Yet Another Lazy-Loader).[84]

81 https://smashed.by/lazyloadxt
82 https://smashed.by/blazy
83 https://smashed.by/unveil
84 https://smashed.by/yalljs

### JAVASCRIPT LAZY LOADING: CAVEATS

Screen readers, some search bots, and any users with JavaScript disabled will not be able to view images lazy-loaded with JavaScript. We can work around this with a `<noscript>` fallback.

Scroll listeners, used for determining when to load a lazy-loaded image, can have an adverse impact on browser scrolling performance. They can cause the browser to re-draw many times, slowing the process to a crawl. However, smart lazy-loading libraries will use throttling to mitigate this. One possible solution is Intersection Observer, which is supported by lazysizes.

Lazy-loading images is a widespread pattern for reducing bandwidth, decreasing costs, and improving user experience. Until recently, lazy-loading images could only be implemented using JavaScript libraries. But from 2019, browsers started to support this capability natively.[85]

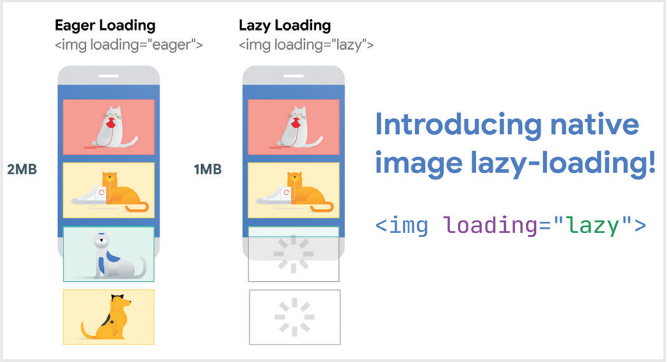
## Native Lazy Loading

In this section, we'll look at the `loading` attribute which brings native `<img>` lazy-loading to the web! For the curious, here's what the syntax looks like:

85 https://smashed.by/nativelazy

```
<img src="celebration.jpg" loading="lazy" alt="..." />
```

Historically, to limit the impact offscreen images have on page load times, developers have needed to use a JavaScript library (like lazysizes)[86] to defer fetching these images until a user scrolls near them.




*Native image lazy-loading landed in Chromium-based browsers (Chrome, Edge, Opera, etc.) and Firefox in 2019 and 2020.*

What if the *browser* could avoid loading these offscreen images for you? This would help content in the viewport load quicker, reduce overall network data usage, and reduce memory usage on lower-end devices. I'm happy to share that this is now possible with the `loading` attribute for images.

86 https://smashed.by/lazysizes

## The loading Attribute

Today, browsers already load images at different priority levels depending on where they're located with respect to the device viewport. Images below the viewport are loaded with a lower priority, but they're still fetched as soon as possible.

In modern browsers, you can use the `loading` attribute on the `<img>` element to completely defer the loading of offscreen images that can be reached by scrolling:

```
<img src="image.png" loading="lazy" alt="…"
width="200" height="200">
```

Here are the supported values for the `loading` attribute:

- `auto`: Default lazy-loading behavior of the browser, effectively the same as not including the attribute.
- `lazy`: Defer loading of the resource until it reaches a calculated distance from the viewport.
- `eager`: Load the resource immediately, regardless of where it's located on the page.

> Although available in Chromium, the `auto` value is not mentioned in the specification.[87] Since it may be subject to change, I recommend not to use it until it gets included.

## Browser Compatibility

`<img loading="lazy">` is supported by most popular Chromium-powered browsers (Chrome, Edge, Opera), Firefox,[88] and the implementation for WebKit (Safari) is in progress.[89] Browsers that do not support the `loading` attribute simply ignore it without side effects.

If Lite mode[90] is enabled on Chrome for Android, Chromium automatically lazy-loads any images that are well suited to being deferred. This is primarily aimed at users who are conscious about data savings (see chapter 21).

Here is the support data[91] for major browsers:

87 https://smashed.by/attributes
88 https://smashed.by/firefox75
89 https://smashed.by/webkit
90 https://smashed.by/litemode
91 https://smashed.by/supportdata

| BROWSER | LAZY LOADING SUPPORT |
|---|---|
| Chrome | 76 |
| Edge | 79 |
| Firefox | 75 |
| Internet Explorer | Not supported |
| Opera | 64 |
| Safari | Under active development |

Background images in CSS cannot take advantage of the `loading` attribute.

### DISTANCE-FROM-VIEWPORT THRESHOLDS

All images that are immediately viewable without scrolling load normally. Those that are below the device viewport are only fetched when the user scrolls near them.

Chromium's implementation of native lazy loading tries to ensure that offscreen images are loaded early enough so that they have finished loading once the user scrolls near to them. By fetching nearby images before they become visible in the viewport, we maximize the chance they are already loaded by the time they become visible.

Compared to JavaScript lazy-loading libraries, the thresholds for fetching images that scroll into view may be considered conservative. Chromium is looking at better aligning these thresholds with developer expectations.

> Experiments conducted using Chrome on Android suggest that on 4G 97.5% of offscreen lazy-loaded images were fully loaded within 10 ms of becoming visible. Even on slow 2G networks, 92.6% of such images were fully loaded within 10 ms. This means native lazy loading offers a stable experience regarding the visibility of elements that are scrolled into view.

The distance threshold is not fixed and varies depending on several factors:

- the type of image resource being fetched
- whether Lite mode[92] is enabled on Chrome for Android
- the effective connection type[93]

You can find the default values for the different effective connection types in the Chromium source.[94] These num-

92 https://smashed.by/litemode
93 https://smashed.by/networkinformation
94 https://smashed.by/chromiumsource

bers, and even the approach of fetching only when a certain distance from the viewport is reached, may change in the near future as the Chrome team improves heuristics to determine when to begin loading.

> In Chrome 77+, you can experiment with these different thresholds by throttling the network[95] in DevTools.
> In the meantime, you will need to override the effective connection type of the browser using the *chrome://flags/#force-effective-connection-type* flag.

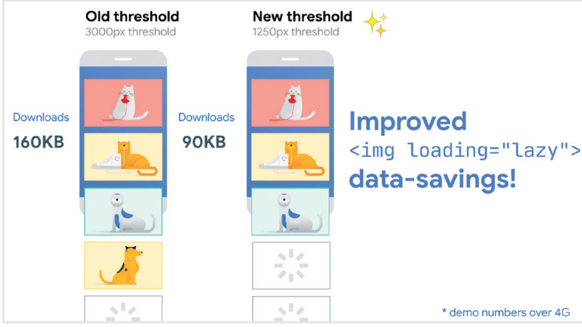
#### IMPROVED DATA-SAVINGS AND DISTANCE-FROM-VIEWPORT THRESHOLDS

As of July 2020, Chrome has made significant improvements to align the native image lazy-loading distance-from-viewport thresholds to better meet developer expectations.

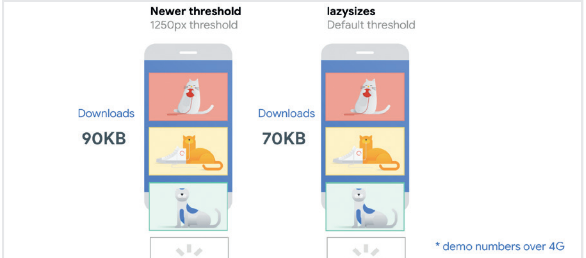
On fast connections (4G and up), Chrome's distance-from-viewport thresholds reduced from 3,000px to 1,250px; on slower connections (3G and below), the threshold reduced from 4,000px to 2,500px. This change achieves two things:

95 https://smashed.by/networkthrottle

- `<img loading="lazy">` behaves more like the experience offered by JavaScript lazy-loading libraries.
- The new distance-from-viewport thresholds still allow us to guarantee images have probably loaded by the time a user has scrolled to them.




*The new and improved thresholds for native image lazy loading in Chrome, reducing the distance-from-viewport thresholds for fast connections from 3,000px down to 1,250px.*




*The new distance-from-viewport thresholds in Chrome, loading 90 KB of images natively compared with lazysizes loading in 70 KB under the same network conditions.*

> To ensure Chrome users on recent versions also benefit from the new thresholds, Google has backported these changes so that Chrome 79–85 inclusive also use them. Please keep this in mind if attempting to compare data savings from older versions of Chrome to newer ones.

These values are hard-coded and can't be changed through the. However, they may change in the future as browsers experiment with different threshold distances and variables.

## Include Image Dimensions

While the browser loads an image, it does not immediately know its dimensions unless they are explicitly specified. To enable the browser to reserve sufficient space on a page for images, it is recommended that all `<img>` tags include both `width` and `height` attributes. Without dimensions specified, layout shifts[96] can occur, which are more noticeable on pages that take some time to load.

```
<img src="image.png" loading="lazy" alt="…"
width="200" height="200">
```

96 https://smashed.by/cls

Alternatively, specify their values in an inline style:

```
<img src="image.png" loading="lazy" alt="…"
style="height:200px; width:200px;">
```

The best practice of setting dimensions applies to `<img>` tags regardless of how they're to be loaded. With lazy loading, this can become more relevant. Setting `width` and `height` on images in modern browsers also allows browsers to infer their intrinsic size. Of course, images will still load if their dimensions are not included, but specifying them decreases the chance of layout shift.[97] If you are unable to include dimensions for your images, lazy-loading them can be a trade-off between saving network resources and potentially being more at risk of layout shift.

While native lazy loading in Chromium is implemented in a way such that images are likely to be loaded once they are visible, there is still a small chance that they might not be. In this case, missing `width` and `height` attributes on such images increase their impact on cumulative layout shift.

A demo from Mathias Bynens shows how the `loading` attribute works with 100 pictures.[98]

97 https://smashed.by/specify
98 https://smashed.by/loadingdemo

Images defined using the `<picture>` element can also be lazy-loaded:

```
<picture>
  <source media="(min-width: 800px)" srcset="large.jpg
1x, larger.jpg 2x">
  <img src="photo.jpg" loading="lazy">
</picture>
```

Although a browser will decide which image to load from any of the `<source>` elements, the `loading` attribute only needs to be included to the fallback `<img>` element.

## No Lazy-Loading in the First Visible Viewport

You should avoid setting `loading="lazy"` for any images that are in the first visible viewport.

It is recommended to add `loading="lazy"` only to images positioned below the viewport, if possible. Images that are eagerly loaded can be fetched right away; the browser has to wait until it calculates where lazy-loaded images are positioned on the page, which relies on the Intersection-Observer being available. It is safer to avoid lazy-loading above-the-fold images, as browsers such as Chrome won't include them in the preload scanner.

> In Chromium, the impact of images in the initial viewport being marked with `loading="lazy"` on “largest contentful paint” (a Core Web Vital: see chapter 22) is fairly small, with a regression of <1% at the 75th and 99th percentiles compared with eagerly loaded images.

Generally, any images within the viewport should be loaded eagerly using the browser’s defaults. You do not need to specify `loading="eager"` for this to be the case for in-viewport images.

```
<!-- visible in the viewport -->
<img src="product-1.jpg" alt="..." width="200"
height="200">
<img src="product-2.jpg" alt="..." width="200"
height="200">
<img src="product-3.jpg" alt="..." width="200"
height="200">

<!-- offscreen images -->
<img src="product-4.jpg" loading="lazy" alt="..."
width="200" height="200">
<img src="product-5.jpg" loading="lazy" alt="..."
width="200" height="200">
<img src="product-6.jpg" loading="lazy" alt="..."
width="200" height="200">
```

Only images below the viewport can load lazily; images in the viewport but not immediately visible – behind a carousel, for example, or hidden by CSS for certain screen sizes – will load normally.

## Third-Party Libraries and Scripts and Native Lazy Loading

The `loading` attribute should not affect code that currently lazy-loads your assets in any way, but there are a few important things to consider:

1. If your custom lazy-loader attempts to load images or frames sooner than when Chrome loads them normally – that is, at a distance greater than the load-in distance threshold – they are still deferred, and will load based on normal browser behavior.

2. If your custom lazy-loader uses a shorter distance to determine when to load a particular image than the browser, then the behavior will conform to your custom settings.

One of the important reasons to continue to use a third-party library alongside `loading="lazy"` is to provide a polyfill for browsers that do not yet support the attribute.

### HANDLING BROWSERS WITHOUT SUPPORT FOR LAZY LOADING

Create a polyfill or use a third-party library to lazy-load images on your site. The loading property can be used to detect if the feature is supported in the browser:

```
if ('loading' in HTMLImageElement.prototype) {
  // supported in browser
} else {
  // fetch polyfill/third-party library
}
```

For example, lazysizes[99] is a popular JavaScript lazy-loading library. You can detect support for the `loading` attribute to load lazysizes as a fallback library only when `loading` isn't supported. This works as follows:

- Replace `<img src>` with `<img data-src>` to avoid an eager load in unsupported browsers. If the `loading` attribute is supported, swap `data-src` for `src`.
- If `loading` is not supported, load a fallback (lazysizes) and initiate it. As per lazysizes docs, you use the `lazyload` class as a way to indicate to lazysizes which images to lazy-load.

99 https://smashed.by/lazysizes

```
<!-- Let's load this in-viewport image normally -->
<img src="hero.jpg" alt="…">

<!-- Let's lazy-load the rest of these images -->
<img data-src="unicorn.jpg" alt="…" loading="lazy"
class="lazyload">
<img data-src="cats.jpg" alt="…" loading="lazy"
class="lazyload">
<img data-src="dogs.jpg" alt="…" loading="lazy"
class="lazyload">

<script>
  if ('loading' in HTMLImageElement.prototype) {
    const images = document.
querySelectorAll('img[loading="lazy"]');
    images.forEach(img => {
      img.src = img.dataset.src;
    });
  } else {
    // Dynamically import the LazySizes library
    const script = document.createElement('script');
    script.src =
      'https://cdnjs.cloudflare.com/ajax/libs/
lazysizes/5.1.2/lazysizes.min.js';
    document.body.appendChild(script);
  }
</script>
```

A demo showing this pattern is available.[100] Try it out in a browser like Firefox or Safari to see the fallback in action.

100 https://smashed.by/firebaselazy

> The lazysizes library also provides a native loading plug-in[101] that uses native lazy loading when available but falls back to the library's custom functionality when needed.

## Impact of Native Lazy-Loading

Andy Potts, a senior software engineer at the BBC, added the `loading` attribute[102] to images on one of its internal sites, thereby decreasing load time on a fast network by around 50% (reduced from about 1 second to less than 0.5 seconds) and saving up to 40 requests to the server.

Similarly, by adding `loading="lazy"` to all its images, TheyWorkForYou (a non-political site taking open data from UK parliamentary proceedings and making it easily available and understandable) cut down total page load for one of its pages by about 90%.[103] Your mileage may vary, of course, but given the ease of testing the `loading` attribute on your pages, I'd certainly recommend giving it a go! Baking in native support for lazy-loading images can make it significantly easier for you to improve the performance of your web pages.

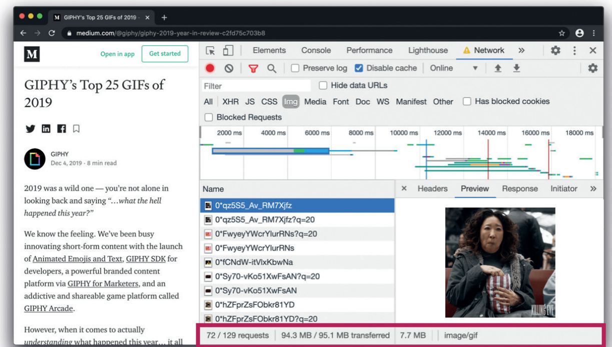
101 https://smashed.by/nativeplugin
102 https://smashed.by/bbcloading
103 https://smashed.by/90perc

CHAPTER 15

# Replacing Animated GIFs

*This chapter includes contributions from performance experts Jeremy Wagner and Houssein Djirdeh, who also strongly prefer videos to animated GIFs! :)*

Have you ever read an article with an animated GIF and inspected it in DevTools only to learn that the GIF was really a video? There's a great reason for that. Animated GIFs can be huge. It's common for GIFs from popular memes, movies, and TV shows to be several megabytes in size, depending on the quality and length. If you're aiming to improve the loading performance of your pages, animated GIFs aren't very compatible with that goal. But this is an area of loading performance where, without a lot of work, you can get significant gains without a loss of content quality.

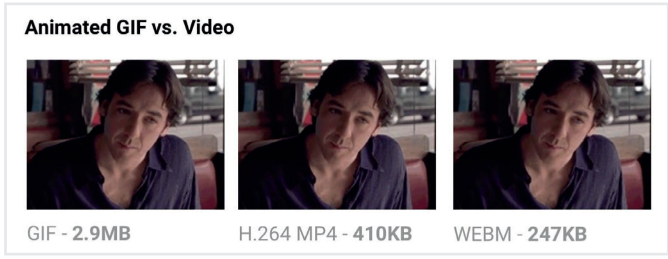



*An animated GIF from the TV show "Killing Eve." It was viewed over 144 million times in 2019 and is almost 8 MB. That's a lot of wasted bandwidth!*

In this chapter, you'll learn to use the same techniques that popular image CDN and GIF hosting sites employ to keep their bandwidth bills as low as possible, by converting those large GIFs into lean, fast-to-load video files. You'll also learn how to embed these videos into web pages so they behave like GIFs do. We'll take a brief look at decoding performance for GIFs and video and, before you know it, you'll be on your way to shaving megabytes off your GIF-heavy pages!

## The Problem with Animated GIFs

Although animated GIFs can be found everywhere from news articles to social media sites, the format was never supposed to be heavily used for video storage or animation. The GIF89a specification[104] states: "The Graphics Interchange Format is not intended as a platform for animation."




*Animated GIF vs. video: a comparison of file sizes at roughly equivalent quality for different formats.*

104 https://smashed.by/gif89a

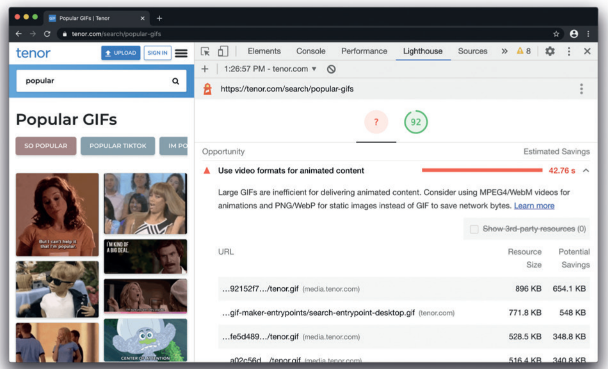
Animated GIFs often waste a significant amount of bandwidth. The number of colors, number of frames, and their dimensions[105] all influence their file sizes. They take longer to load, typically include fewer colors compared with video, and typically offer a subpar user experience. Switching to video offers the largest savings.

Why are animated GIFs so much larger than video files? Animated GIFs store each frame as a lossless GIF image. You read that right – lossless. The degraded quality we often experience is due to GIFs limited 256-color palette. The format has other deficiencies, such as not considering neighbor frames for compression, unlike video codecs like H.264 and H.265. An MP4 video generally stores each key frame as a lossy JPEG, discarding some of the original data to achieve superior compression.

## Measure First

You can use Lighthouse to check your page for animated GIFs that can be converted to videos. In Chrome DevTools, select the **Lighthouse** tab and check the **Performance** checkbox. Then run Lighthouse and check the report. If you have any GIFs that could be better served as videos, you should see a "Use video formats for animated content" opportunity.

105 https://smashed.by/gifdimensions




*A Lighthouse report for popular GIF-sharing site Tenor. The most popular animated GIFs during this particular week were pretty large!*

## Converting Animated GIFs to Video

There are a number of ways to convert animated GIFs to video. The tool I recommend is FFmpeg,[106] and that's what I'll use in the examples throughout this chapter.

### FFMPEG QUICK START: MP4 AND WEBM

To use FFmpeg to convert *animation.gif* to an MP4 video, run the following command in your console:

```
ffmpeg -i animation.gif -b:v 0 -crf 25 -f mp4 -vcodec
libx264 -pix_fmt yuv420p animation.mp4
```

106 https://www.ffmpeg.org/

This tells FFmpeg to take *animation.gif* as the input, signified by the `-i` flag, and to convert it to a video called *animation.mp4*. Your mileage may vary depending on the input, but typically the output should be significantly smaller now!

While MP4 has been around since 1999, WebM[107] is a relatively new file format released in 2010. WebM videos are much smaller than MP4 videos, but not all browsers support WebM so it makes sense to generate both. To convert *animation.gif* to a WebM video, run:

```
ffmpeg -i animation.gif -c vp9 -b:v 0 -crf 41 my-animation.webm
```

The cost savings between an animated GIF and a video can be significant. In an example where *animation.gif* is 3.7 MB, compare it to the MP4 which is 551 KB or the WebM, which is 341 KB.

```
$ ls -lh
total 4.5M
-rw-r--r-- 1 app app 3.7M May 26 00:02 cat-herd.gif
-rw-r--r-- 1 app app 551K May 31 17:45 cat-herd.mp4
-rw-r--r-- 1 app app 341K May 31 17:44 cat-herd.webm
```

*Comparing file sizes of MP4 and WebM conversions of an animated GIF.*

107 https://www.webmproject.org/

If you're interested in diving into a real example, Rob Dodson has a good codelab on replacing GIFs with video.[108] Next, let's take a look at a more detailed version of this workflow with more nuance.

### FFMPEG WORKFLOW

How you install FFmpeg will differ based on the operating system you use.

- For macOS, you can install via Homebrew[109] or compile it yourself.[110]
- For Windows, use Chocolatey.[111]
- For Linux, check if your preferred distro's package manager (e.g. apt-get or yum) has a package available.

For WebM support, you might want to make sure whatever FFmpeg build you install is compiled with libvpx.[112]

Once FFmpeg is installed, pick a GIF to convert and you'll be ready to roll. For the purposes of this guide, we will use a GIF,[113] which is just shy of 14 MB. To start off, let's try our hand at converting it to MPEG-4!

108 https://smashed.by/replacinggif
109 https://smashed.by/ffmpegbrew
110 https://smashed.by/ffmpegself
111 https://smashed.by/chocolatey
112 https://smashed.by/libvpx
113 https://smashed.by/jazzgif

## CONVERTING GIF TO MPEG-4

When you embed videos on a page, you will want to have an MPEG-4 version as MPEG-4 enjoys the broadest support[114] of all video formats across browsers. To get started, open a terminal window, go to the directory containing your test GIF, and try this command:

```
ffmpeg -i input.gif output.mp4
```

This is the most straightforward syntax for converting a GIF to MPEG-4. The `-i` flag specifies the input, after which we specify an output file. This command takes our test GIF of 14,024 KB and reduces it to a reasonably high-quality MPEG-4 video weighing in at 867 KB. That's a reduction of 93.8%. Not bad, but maybe you're curious to see if we can go a little further.

It turns out FFmpeg is very configurable, and we can use this to our advantage to fine-tune the video output by employing an encoding mode called constant rate factor[115] (CRF). CRF is great when video quality is a high priority.

```
ffmpeg -i input.gif -b:v 0 -crf 25 output.mp4
```

114 https://smashed.by/mpeg4support
115 https://smashed.by/crf

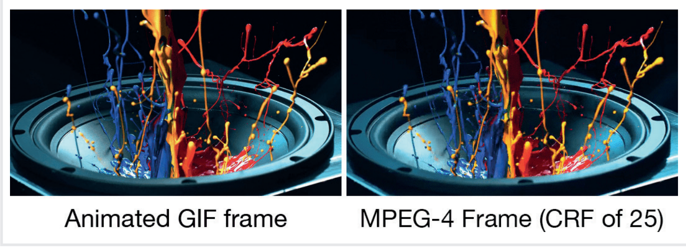
This command is similar to the one before it, but with two key differences: The `-b:v` flag normally would limit the output bit rate,[116] but when we want to use CRF mode, it must be set to 0. The `-crf` flag accepts a value between 0 and 51. Lower values yield higher quality (and larger) videos, whereas higher values do the opposite.

Using our test GIF, this command outputs an MPEG-4 video 687 KB in size. That's an improvement on the first compression of roughly 20%! If you want even smaller file sizes, you could specify a higher CRF value, but be aware that higher values will yield lower quality videos, so always check the encoder's output to ensure you're happy with the results.

> Video processing tools use CRF values as indicators of desired video quality. Depending on the underlying codec, the specific compression strategies will be different for the same CRF values (similar to the quality index used in image processing tools). The quality of two videos is not directly comparable based on their CRF values.

116 https://smashed.by/limitbitrate

These commands yield a large reduction in file size over GIF, which in turn will substantially improve initial page load time and reduce data usage. While the visual quality of the video is somewhat lower than the source GIF, the reduction in file size is a reasonable trade-off to make.




*Visual comparison of an animated GIF frame against an MPEG-4 frame from a video encoded with a CRF of 25.*

While the illustration is no substitute for a comprehensive visual comparison, the MPEG-4 is certainly sufficient as an animated GIF replacement. It also pays to remember that your users likely won't have a reference to the GIF source. Always adhere to your project's standards for media quality, but be willing to make trade-offs for performance where appropriate.

While MPEG-4 has wide support and is certainly suitable as a replacement for animated GIF, we can go a bit further by generating an additional WebM version.

### CONVERTING GIF TO WEBM

While MPEG-4 has been around in some form since at least 1999[117] and continues to see development, WebM is a relative newcomer having been initially released in 2010.[118]

While browser support for WebM[119] isn't as wide as MPEG-4, it's still very good. Because the `<video>` element[120] allows you to specify multiple `<source>` elements,[121] you can state a preference for a WebM source that many browsers can use while falling back to an MPEG-4 source that all other browsers can understand.

Try converting your test GIF to WebM with FFmpeg using this command:

```
ffmpeg -i input.gif -c vp9 -b:v 0 -crf 41 output.webm
```

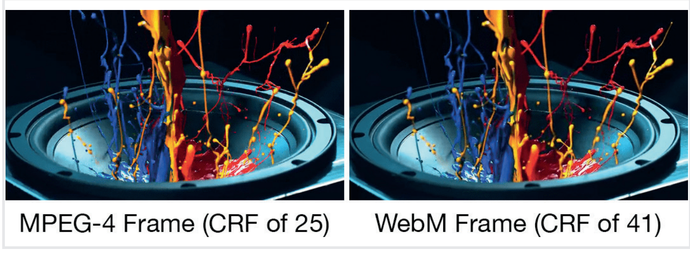
You'll notice this method is pretty similar to the previous GIF-to-MPEG-4 conversion command using CRF mode, but there are two key differences:

- The codec we specify in the `-c` flag is VP9, which is the successor to the VP8 codec used by the WebM format. If this fails for you, replace VP9 with VP8.

117 https://smashed.by/mpeg4parts
118 https://smashed.by/libvpxrelease
119 https://smashed.by/webmsupport
120 https://smashed.by/videoelement
121 https://smashed.by/multiplesources

- Because CRF values don't yield equivalent results across formats, we need to adjust it so our WebM output is visually similar to the MPEG-4 output. A `-crf` value of 41 is used in this example to achieve reasonably comparable quality to the MPEG-4 version while still outputting a smaller file.

In this example, the WebM version was roughly 66 KB smaller than the MPEG-4 at 611 KB. Its visual quality is reasonably similar to the MPEG-4 version too.




*Visual comparison of an MPEG-4 frame encoded with a CRF value of 25 versus a WebM frame encoded with a CRF value of 41.*

Owing to how the VP8 and VP9 codecs encode video, compression artifacts in WebM may affect the quality of the result in ways different from MPEG-4. As always, inspect the encoder output and experiment with flags (time permitting) to find the best result for your application.

> If you like tinkering, consider trying out two-pass encoding[122] to see if the results are more to your liking!

Now that we know how to convert GIFs to both MPEG-4 and WebM, let's learn how to replace those animated GIF `<img>` elements with `<video>`!

## Replacing Animated GIF <img> Elements with <video>

Unfortunately, using a video as an animated GIF replacement is not quite as straightforward as dropping an image URL into an `<img>` element. Using `<video>` is a bit more complex, but not too difficult. We'll walk through how to do this step by step and explain everything, but if you just want to see the code, a CodePen demo is available.[123]

### GETTING THE BEHAVIORS RIGHT

Animated GIFs have three key traits:

1. They play automatically.

122 https://smashed.by/twopass
123 https://smashed.by/gifvideo

2. They loop continuously (usually, though it is possible to prevent looping).

3. They're silent.

The only true advantage of using animated GIF over video is convenience. We don't have to be explicit in defining these traits when we embed GIFs. They just behave the way we expect them to. When we want to use video in place of GIFs, however, we have to explicitly tell the `<video>` element to autoplay, loop continuously, and be silent. Let's start by writing a `<video>` element like so:

```
<video autoplay loop muted playsinline></video>
```

The attributes in this example are pretty self-explanatory. A `<video>` element using these attributes will play automatically, loop endlessly, play no audio, and play inline (that is, not fullscreen). In other words, all the hallmark behaviors we expect of animated GIFs.

> If faithful emulation of animated GIF behavior isn't crucial to your application, you could take a more conservative approach by allowing users to initiate

playback instead of autoplaying. If you go this route, remove the `autoplay` attribute, and consider specifying a placeholder image via the `poster` attribute.[124] Additionally, use the `controls` attribute[125] to allow the user to control playback, and add the `preload` attribute[126] to control how the browser preloads video content.

There's more to this than simply emulating GIF behavior, though. Some of these attributes are required for autoplay to even work. For example, the `muted` attribute must be present[127] for videos to autoplay, even if they don't contain an audio track. On iOS, the `playsinline` attribute is required for autoplay to work[128] as well.

### SPECIFY YOUR <SOURCE>S

All that's left to do is specify your video sources. The `<video>` element requires one or more `<source>` child elements pointing to different video files the browser can choose from, depending on format support:

```
<video autoplay loop muted playsinline>
```

124 https://smashed.by/attrposter
125 https://smashed.by/attrcontrols
126 https://smashed.by/attrpreload
127 https://smashed.by/autoplay
128 https://smashed.by/iospolicies

```
  <source src="oneDoesNotSimply.webm" type="video
/webm">
  <source src="oneDoesNotSimply.mp4" type="video/mp4">
</video>
```

> Browsers don't speculate about which `<source>` is optimal, so the order of `<source>`s matters. For example, if you specify an MPEG-4 video first and the browser supports WebM, browsers will skip the WebM `<source>` and use the MPEG-4 instead. If you'd prefer a WebM `<source>` to be used first, specify it first!

Now that we know how to convert GIFs to video and how to use those videos as GIF replacements, let's see how each of these solutions performs in the browser.

## Performance of Video versus Animated GIF

Though smaller resources are preferable, file size isn't everything. We also need to pay attention to how a media resource performs after it has been downloaded, because media assets must be decoded before playback.

GIFs (and other animated image formats) are suboptimal because an image decode is incurred for every frame in the image, which can contribute to jank. This makes sense, because each frame in a GIF is simply another image. Let's see how this looks in the Performance panel in Chrome's DevTools for a page where the only content is an `<img>` element pointing to an animated GIF.

*The Performance panel in Chrome's developer tools showing browser activity as an animated GIF plays.*

As you can see in the figure, image decodes occur on the rasterizer threads as each new frame of the GIF is decoded. Now let's look at a comparison table of total CPU time for GIF versus MPEG-4 and WebM videos:

| FORMAT | CPU TIME |
| --- | --- |
| GIF | 2,668 ms |
| MPEG-4 | 1,994 ms |
| WebM | 2,330 ms |

# AUTOMATE

Image optimization could be automated using tools or a CDN. It's easy to forget, best practices change, and content that doesn't go through a build pipeline can easily slip.

These figures were gathered in Chrome's tracing utility (record your own Chrome traces at *chrome://tracing*) over a period of approximately 6.5 seconds for each format. As you can see, GIF takes the most CPU time, and less CPU time occurs for both videos, particularly MPEG-4. This is good stuff! It means that videos generally use less CPU time than animated GIF, which is a welcome performance enhancement beyond simply reducing file size.

It should be mentioned, however, that some CPUs and GPUs offer hardware-accelerated encoding/decoding of video (for example, Quick Sync Video[129] on Intel CPUs). Many processors can handle encoding and decoding for MPEG-4, but WebM codecs such as VP8 and VP9 have only recently started to benefit from hardware-accelerated encoding/decoding on newer CPUs. A Kaby Lake Intel processor was used in these tests, meaning that video decoding was hardware assisted.

> Hardware-accelerated decoding sometimes has color accuracy issues in video playback. Software decoding offers more consistency because bugs can be fixed quicker and rolled out to all users.

129 https://smashed.by/quicksync

## Potential Pitfalls

You've heard enough about the advantages of using video instead of animated GIF, but I would be remiss in my responsibility if I didn't also point out some of the potential pitfalls. Here's a couple for your consideration.

### EMBEDDING VIDEO IS NOT AS CONVENIENT AS EMBEDDING A GIF

Nothing is more convenient than slapping a GIF in an `<img>` element and moving on with your life. It's a simple one-liner that just works, and that's huge for the developer experience.

However, your experience as a developer isn't the only one that matters. Users matter more. On the bright side, using video in the `<img>` element[130] is possible in Safari, so an easier solution for using videos as GIF replacements may be on the way. It's just not an approach you can currently depend on in all browsers.

### ENCODING YOUR OWN VIDEOS CAN TAKE TIME

As developers, we want to save time. When it comes to something as subjective as the notion of media quality, however, it can be difficult to come up with an automated process that provides the best results for all scenarios.

130 https://smashed.by/gifevolution

The safest thing to do is analyze the encoder output for each video, and ensure the results are adequate. This may only be a reasonable solution for projects with few video resources. For larger projects with many videos, you may want to go with a conservative encoding strategy that emphasizes quality over file size. The good news is that this strategy will still yield great results, substantially improve loading performance, and reduce data usage for all users.

Additionally, converting all your GIFs to video takes time – time you might not have. In this case, you might consider a cloud-based media hosting service such as Cloudinary,[131] which does the work for you. Nadav Soferman's article on Cloudinary's blog[132] explains how their service can transcode GIF to video for you.

### DATA SAVER MODE

On Chrome for Android, autoplaying video can be disallowed when Data Saver[133] is enabled, even if you follow this guide's instructions to the letter. If you're a web developer, and you're struggling to figure out why videos aren't autoplaying on your Android device, disable Data Saver to see if that fixes the issue for you.

To cover edge cases such as these, you should consider setting the `poster` attribute so the `<video>` element's space is

131 https://cloudinary.com/
132 https://smashed.by/cloudinaryblog
133 https://smashed.by/chromelite

populated with some meaningful content in the event Data Saver is on (or really any possible scenario where autoplay could be disallowed). Another possible approach could be to set the `controls` attribute conditionally based on the presence of the Save-Data header,[134] which is a header Data Saver sends when it's active.

> If you must use animated GIFs, be sure to optimize them as much as possible. Gifsicle[135] is a tool that can strip metadata and unused palette entries, and minimize what changes between frames. It also supports lossy encoding with the `--lossy` flag, which can shave off between 60 and 65% of size. For more information on GIF optimization, checkout *The Book of GIF*[136] by Rigor.

When you use video instead of animated GIF, you're doing your users a big favor by reducing the amount of data you send to them, as well as potentially reducing system

134 https://smashed.by/savedataheader
135 https://smashed.by/gifsicle
136 https://smashed.by/bookgif

resource usage. Ditching animated GIFs is worth serious consideration, especially if they feature prominently in your content. In a time where performance is more important than ever, yet many performance improvement strategies require a significant investment of time, transitioning your GIFs to video is a proportionally small effort when compared to the massive improvement it can have on loading performance.

CHAPTER 16

# Image Content Delivery Networks

*With special thanks to Colin Bendell and Katie Hempenius for their valuable input.*

We try to optimize the content and behavior of our web pages to ensure they're delivered in a timely and efficient manner. For sites with large traffic and a global reach, basic optimizations at build time are usually not enough. Teams need to handle known static assets ahead of time (a logo, for example, or icons) as well as dynamic assets, such as user-uploaded images.

Large sites often rely on a content delivery network (CDN): a network of distributed servers that deliver web content based on the geographic locations of users. Content from the server nearest to the user's location is likely to be delivered faster with minimum hops.

CDNs not only improve website load times but also provide advanced compression and caching services for static content, and improve site reliability due to their inherent load-balancing quality.

## What Is an Image CDN?

Image CDNs are specialized services for delivering images, animations, and videos that augment the normal CDN offering. Image CDNs work by providing you with an for accessing your images and, more importantly, manipulating them. Image CDNs can be a service you manage yourself (via self-hosting) or that you leave to a third party.

...cat.jpg?q=50&w=300&h=200

Set quality level to 50 | Resize to 300x200

**Popular Image CDNs**

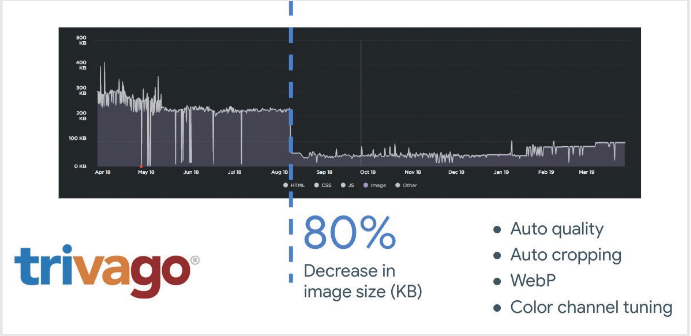
Akamai Image Manager | Cloudinary | Imgix | Thumbor (self-hosted)

*An image CDN can often apply optimizations in a simple query string.*

They often include image-processing functionality like compression, resizing, and delivering progressive or responsive images as required. Most image CDNs also support multiple formats so you can apply your chosen image optimization technique and image format just by referencing an image that will be served by the image CDN.

## Why Businesses Use Image CDNs

Businesses often use a third-party image CDN service as they find it can be more efficient for their engineers to focus on their core product rather than building and maintaining yet another pipeline.




*Trivago, a popular travel site in Europe, switched to using Cloudinary for its image CDN needs and found that image size decreased by 80%.*

While these results are astounding, they are not out of the ordinary. When we spoke to brands who switched to image CDNs,[137] we found that many experienced image savings of between 40 and 80%. Part of this is that image CDNs offer a level of specialization and optimization difficult to achieve independently.

137 https://smashed.by/cdnswitch

*In Google's Speed at Scale talk,*[138] *brands that switched to image CDNs often found they experienced image savings from 40 to 80%. Image CDNs offer a level of specialization and optimization that is often difficult to achieve on your own.*

We know that images are the single largest component of most sites, so this can often translate into significant savings in overall page size. The next section looks at how to define a long-term strategy for optimizing your images with an image CDN.

## Defining a Long-Term Image Optimization Strategy

### WHEN YOU SHOULD USE AN IMAGE CDN

An image CDN solves the basic issues around optimizing images and applies the best practices in real time, while

138 https://smashed.by/savings

also providing a safety net for security and performance. However, many situations might not require image delivery through image CDNs, and a local image optimization and delivery pipeline may be sufficient.

A requirement for an image CDN depends on the following factors:

- How is the image being used on the website?
- What is the source of the image?
- What is the impact of using a local pipeline?

Images, videos, and animations may be classified by how they are being used on the website, including:

- Images used in UI (rounded corners, logos).
- Images used for marketing (landing pages, product listing pages).
- Creative content (hero images, product detail pages, news pages).
- Images uploaded by users (as part of a CMS, to social networks, etc.).

These images usually come from different sources. For example, images used in the UI would most likely be part

of the code repository, and a local pipeline would be sufficient to deliver them. However, images may be created and uploaded by other users who provide content to the website. These could be editors who upload content in a CMS, or vendors on e-commerce sites who provide product details.

With users uploading and referencing content outside the codebase, there are other risks and expectations that need to be evaluated. The primary risks revolve around service-level agreements, performance budgets, and security risks from media injection attacks. An image CDN should be able to provide automation for these images while addressing the above risks and expectations.

SUCCESS STORY ■ “[Adopting Thumbor] allowed [Wikipedia] to reduce [its] thumb-nailing cluster's size when it was deployed.”
—Gilles Dubuc, Wikipedia, November 2018

To decide if an image CDN would be the ideal solution to a specific scenario, it is necessary to evaluate if using a local pipeline has turned image competency into a dependency for the website. Consider the following metrics:

**Effort to build and maintain**: This would indicate the effort required to build and maintain an image optimization pipeline. Effort exceeding 40 hours per year (about 45 minutes per week) implies that the image pipeline

may have become the main driver for product updates and support requests.

**Disaster recovery**: This would verify how long it would take to restore the image/video catalogue and all the pipeline variations, and what would be the cost to business during this visual outage. A "mean time to recovery" over two hours would indicate that an image CDN may be needed.

Answers to the questions above would help determine when an image CDN would effectively solve issues caused by a local pipeline.

## COST–BENEFIT ANALYSIS AND ROI CALCULATION FOR IMAGE CDNS

In order to shift to an image CDN, engineers may be required to show to their stakeholders the added value to be gained by using image CDNs. From a business perspective this would involve monetizing the advantages and comparing them with the investment required to use an image CDN. This could involve quantification and comparison of various metrics.

### 1. Image Delivery

One of the biggest advantages of any CDN would be the ability to speed up delivery due to the logical compression in the network path from client to server. Hence, the performance improvement in image delivery is one of the criteria

that needs to be measured to determine the ROI of image CDNs. Improvement in performance is an important requirement to improve the user experience and attract more users. This can be measured using different metrics such as:

- session length
- pixels or bytes delivered per session
- user engagement, which can be measured using standard KPIs like conversions and checkouts depending on the type of website

### 2. Engineering Effort

An engineering team might not always be up to date with all the intricacies involved in maintaining an image optimization pipeline like support for new formats, edge cases, and so on. Moreover, implementing such support may require time for both development and testing, thereby delaying the delivery of the final product. The cost of developing and maintaining the solution in house would involve hiring costs for image optimization experts, data scientists, and so on. These costs in the aggregate could be comparable to the cost of using an image CDN over the lifetime of value they provide.

An image CDN has the resources to do the required research at scale for all its customers and implement the best choice automatically in many cases. For example, when

Cloudinary introduced support for WebP,[139] it did so in a manner transparent to both the site owner as well as users. If WebP is supported on the user's browser, it is automatically served by Cloudinary; otherwise they serve the same image in some other format such as JPEG.

*A Cloudinary image CDN URL which uses* f_auto*, allowing Cloudinary to analyze the image content and select the best format to deliver to the browser. In Chrome, for example, Cloudinary delivers a WebP image using the above* .jpg *URL, while older browsers receive a JPEG instead.*

For clients that support WebP, Cloudinary can additionally check if specific WebP features, like transparency and grayscale, are supported. A client using a libwebp version earlier than 0.6 for encoding/decoding may not support some of these features. In such cases, Cloudinary takes appropriate actions to polyfill the delivered image.

139 https://smashed.by/webpcdn

### 3. Content Presentation

Some sites could explore different ideas to present content, like converting animations to videos, auto-cropping videos, auto transparency, and automatic alt attribute injection. Some of these features might not be absolute requirements but nice-to-haves. In such cases, site owners may not want to invest time and effort to incorporate these features but rather turn them on or off as required. This can be easily provided by image CDNs. To convert this into a measurable metric, owners would need to assign a priority to the features required and evaluate the cost of implementing them.

### 4. Risk Management

Risk management could involve creating an action plan for risk mitigation that addresses issues around disaster recovery (DR) and data recovery, security, anti-virus protection, and content that's not safe for work (NSFW). We will examine how image CDNs address data recovery in the following sections. Image CDNs can also automatically provide other security features like virus scans for uploaded files, and removal of offensive, unethical, and otherwise NSFW images.

For example, Cloudinary provides automatic image moderation[140] by removing adult or inappropriate photos using its WebPurify add-on service. Rejected images are automatically removed from the site. Cloudinary is also contributing

140 https://smashed.by/moderation

to limit the spread of terrorist content[141] on the web. It is a member of the Global Internet Forum to Counter Terrorism[142] (GIFCT) and has access to the Facebook anti-terrorism database containing unique digital fingerprints of media content. Cloudinary scans uploaded content against this dataset before accepting it.

## Implementing an Image Optimization Pipeline

The image and video landscapes are ever changing, and the learning associated with them is never ending. New formats and optimizations are constantly emerging, and new expectations are being set for marketing and creative divisions of organizations. Stakeholders may often ask for implementation of trending ideas such as transparent videos, wide gamut, high frame rate, cinemagraphs, paper quality illustrations, 3D spinning, AR, and VR.

To add to the complexity, users can access websites from a variety of device and browser combinations. The first requirement for implementing an image optimization pipeline would be to understand the breadth of variations that the website will eventually support, and the mechanisms required to address these.

141 https://smashed.by/curbingcontent
142 https://gifct.org/

Images required in the UI elements can be included as part of regular builds, but if the images are uploaded on the go by users, then sites need to implement a set of machinery that ensures the uploaded images remain at their highest quality while they are being prepared for delivery on the web. Preparation for web could involve various image transforms affecting size, format, and quality.

Sites also need to ensure that the instrumentation to establish a baseline and monitor the impact of changes to the image optimization pipeline is in place. This would monitor the following aspects.

- **Performance impact**: This can be monitored using software like WebPageTest[143] or Lighthouse,[144] or tracked based on monitoring real users.

- **Operations impact**: This would track factors like uptime, completion rates, time to first byte (TTFB), cache-hits, etc.

- **Risk impact**: This would involve instrumentation for monitoring new common vulnerabilities and exceptions (CVE),[145] fingerprint reporting (for NSFW or illegal content), and content quality (source pixel volume, blurriness, color profiles, etc.).

143 https://www.webpagetest.org/
144 https://smashed.by/lighthouse
145 https://smashed.by/sve

- **Accessibility**: This would ensure the implementation of factors affecting the accessibility of the images like color ambiguity, color blindness indexes, `alt` and description text.

Finally, it is of utmost importance that the image pipeline connects to the marketing and brand stakeholders or the creative experts. They should act as visual gatekeepers and track visual parity, ideally as a part of the instrumentation. In the absence of a QA team or stakeholders who can review images visually, metrics to track visual parity can also be provided by structural dissimilarity tools.

### HANDLING BACKUPS WITH IMAGE CDNS

Most sites that deliver content with images would rely on a combination of content management system (CMS) and/or digital asset management (DAM) software to manage content, images, and videos. CMS systems like WordPress, Magento, or Netlify allow you to upload content through their platforms into a storage bucket. Templates and transforms may be applied after uploading the content and before or after saving it to this bucket.

The original content and the variations may or may not be saved depending on the CMS or DAM being used. In this situation, addressing backup and retrieval becomes the development team's responsibility. The team could either extract

and back up the WordPress stack periodically, or develop a real-time mirroring system to back up the content. Image CDNs provide three possible solutions to address backup and retrieval.

- Lazy ingestion, where the CDN pulls the data from the source. In this case there is a certain lag between the master and the replicated data.

- Upload to CDN and replicate to cloud storage at the same time.

- DIY replication withs, which could be implemented by uploading to the CDN first, followed by hooks to replicate out asynchronously. This is the recommended method because it allows for audit trails and DR strategies beyond those provided by the CDN.

A DR strategy could also be affected by other factors, such as the way image CDNs store images. Some image CDNs use path-based parameters to force transformations (for format, crop, quality, and so on) and these could affect the way images are stored. The actual DR recovery plan should take this into consideration. If a team is using a load-balancing server like Nginx, then a point to consider could be whether it works directly with the replicated images and without the image CDN, or if it needs additional logic to translate query parameters to corresponding image paths.

### HANDLING DSSIM WITH IMAGE CDNS

As a reminder (see chapter 2), the structural similarity (SSIM) index helps to predict the perceived quality of images. Structural dissimilarity (DSSIM) can be derived from SSIM. These indices help to measure the similarity between two images: the original and the transformed image.

While some users might want to find out and use the DSSIM themselves, others may leave it up to the image CDN to make decisions based on the DSSIM. Image CDNs use DSSIM for automatic quality selection so that the best quality is used for images.

Automatic quality selection may also calibrate images on a use-case basis in addition to the perceptual quality metric like DSSIM. Cloudinary uses the following factors to decide on the compression format and degree of compression to be used:

- Illustration: indicates if the image looks like an illustration with hard lines.
- Color ambiguity: used to judge if the image will be problematic for a person with color blindness. It reports the volume of red-green or blue-green colors that are ambiguous in clustering.

- Source quality: indicates if the image source may be already heavily precompressed, or suffers from other aesthetic problems before applying any optimizations.
- Grayscale.
- Focus.

Perceptual quality also depends on formats in addition to the factors above. A quality setting of 80 for JPEG may not be the same as a quality of 80 for WebP. Perceptual quality does not map linearly between formats and is affected by different variables like target quality, pixel volume, color volume and illustration-ness. Remember that there are differences between image quality measurement units used by different tools: an image quality of 40 in Photoshop's "Save for Web" feature, for example, corresponds to 80 with libjpeg-turbo and GIMP quality 93.[146]

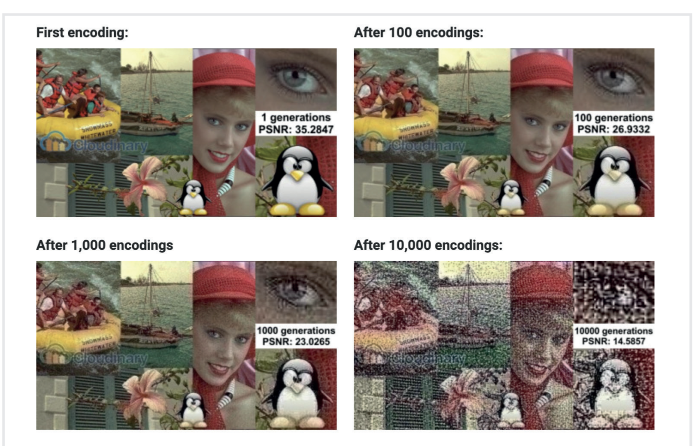
Not all image CDNs use this wide range of classifiers for quality selection: some may just classify images based on the source format of the original image. There is scope to add additional intelligence in this area in the future. AI generation artifacts and semantic content could also be considered. For example, this could involve making quality selection more sensitive to changes in the skin coloring of

146 https://smashed.by/gimp93

the person being photographed but less sensitive to changes on the mountains in the background.

## RECOMPRESSION ISSUES

Reoptimizing images from a semi-optimized source using an image CDN is similar to applying the photocopier effect.[147] In some cases, reapplying lossy transforms is avoidable.




*A demonstration of the photocopier effect, which can happen after repeatedly applying lossy compression transforms to the same image. Shown are the results after 1, 100, 1,000, and 10,000 encodings.*

For simple transforms (cropping, for example), there is no need to dequantize and requantize the image. This can

147 https://smashed.by/photocopier

limit the loss due to transform. It can also work for responsive images. However, a transform that involves changing the quality factor will dequantize and requantize the image.

Using perceptual qualities can help to minimize the generation loss from recompression. The metric is calculated against the inflated form of the image, which includes the original generation loss. SSIM-based calculations are sensitive to the amplification from these recompressed images. So, the amount of data loss from regeneration is minimized through this process.

However, the photocopier effect cannot be avoided completely. Once there is a loss of quality it is impossible to ascertain if the new image is as good as the original. But it will certainly be as good as the semi-optimized image with a little bit of additional loss. Tools such as jpeg2png[148] have attempted to reverse some of the quantization artifacts by intelligently filling the missing information and creating a smoother picture.

In this context, it is recommended to always attempt to start with images of the highest possible quality. Image CDNs can incorporate tools that help identify incoming images of low quality and optimize accordingly.

148 https://smashed.by/jpeg2png

## RESPONSIVE WEB DESIGN

A responsive image is expected to adapt in response to different environmental conditions like the device it is being viewed on, the viewport size, bandwidth, and so on. Adaptations can include, but are not limited to, changing the dimensions of the image, cropping the image, or automatically selecting a different source image based on the viewport size (see chapter 11 for more details). The next generation of responsive web design and responsive images needs to make way for art direction[149] and smart cropping features. This could involve cropping an image or using an altogether different image to ensure that the main subject of the image is always in the picture.

In addition to generating resolution-based responsive images corresponding to any image, CDNs have also explored automatic art direction[150] by which an image's context is intelligently identified and focused on when making cropping decisions. It uses edge detection, face detection, and visual uniqueness to generate a heat map of the most important aspects captured in the image.

## LEGAL AND LICENSING ISSUES

Specific legal and licensing issues need to be considered when it comes to automatic image compression because the image could be altered by the compression process. These issues can be categorized as follows.

149 https://smashed.by/artdirection
150 https://smashed.by/automaticartdirection

### Ownership

This category of legal concerns is related to visual material related to trademarks and brand identities; for example, celebrity endorsements and logos. Examples of legal concerns might include whether a brand logo is displayed exactly as intended, or ensuring that a celebrity's identifiable face is not photoshopped into an image without their endorsement. Logos usually need to be displayed with strict Pantone colors and should not suffer from color shifting. Similarly, skin tones for celebrity faces should not drift, particularly for people of color. This has implications not just for the photographer, but also for the image algorithm that might shift colors up or down when normalizing. This scenario provides a strong case for the need for SSIM-based scoring when selecting the right quality factor in JPEG. One of the three main variables in SSIM is contrast, which will help flag shifts in skin tones.

### Distribution Rights

This involves ensuring that you have the digital rights to the content you manipulate. The technological implications include ensuring that you have mechanisms to track where you are showing images, how you mitigate hotlinking, and removing content you no longer have rights to show. Image CDNs should be able to address access control to prevent these issues; for example: setting time-based access control to match the legal contracts; preventing hotlinking; and authorizing image requests so that only consumers on your web page can see the images, thereby deterring others from

referencing your content. There is a risk of images being copied when we send them. Complying with the distribution requirements can help in mitigating the risk.

#### Transmission

These issues deal with the technology or medium used to transmit and distribute images and if there are patent implications for using these technologies and transmitting them in this medium. It is an issue relevant in the transmission of formats like Advanced Video Coding (H.264), High Efficiency Video Coding (H.265), and High Efficiency Image File Format (HEIF) where licensing is involved. Image CDNs can license these patents directly or indirectly so that customers don't have to worry about related legal issues. Cloudinary has a licensing agreement with Nokia for HEIF so generating HEIC/HEIF content (see chapter 17) is not a legal burden on customers.

## Evaluating Your Need for an Image-Processing CDN

To achieve optimal page load times, you need to optimize your image loading. So far we have covered the thought process of managing an image optimization pipeline with and without an image CDN. This optimization calls for a responsive image strategy and can benefit from on-server image compression, auto-picking the best format, and

responsive resizing. What matters is that you deliver the correctly sized image to the proper device in the proper resolution as fast as possible.

Because of the complexity and ever-evolving nature of image manipulation, doing this is not as easy as one might think, and image CDNs can save you a lot of time and trouble. Industry expert Chris Gmyr[151] even goes as far as saying:

> *If your product is not image manipulation, then don't do this yourself. Services like Cloudinary* [or imgix and others] *do this much more efficiently and much better than you will, so use them. And if you're worried about the cost, think about how much it'll cost you in development and upkeep, as well as hosting, storage, and delivery costs.*

Still, the argument for cost and convenience of image CDNs varies on a case-by-case basis. An image-heavy site with a lot of traffic could cost hundreds of US dollars a month in CDN fees. There is also a certain level of prerequisite knowledge and programming skill required to get the most out of these services. If you are not doing anything too fancy, you're probably not going to have any trouble. But if you're not comfortable working with image processing tools ors, then you are looking at a relatively difficult learning curve. (For example, to accommodate the CDN server locations, you will

151 https://smashed.by/gmyr

need to change some URLs in your local links. Before diving in, do the right due diligence.)

### CDN PERFORMANCE

CDN delivery performance is mostly about improved latency[152] and speed.

Latency always increases somewhat for completely uncached images. But once an image is cached and distributed among the network servers, the fact that a global CDN can find the shortest hop to the user, added to the byte savings of a properly processed image, almost always mitigates latency issues when compared to poorly processed images or solitary servers trying to reach across the planet.

The reduced latency increases download speed, which affects page load time, and this is one of the most important metrics for both user experience and conversion.

As mentioned, the advantages of using an image CDN are not the same for every project. To help, here's a checklist of some of the key factors to consider when evaluating the need for an image CDN. If you answer mostly yes, your website is a good candidate for using one.

152 https://smashed.by/latency

1. Do you need sophisticated image and video capabilities, like advanced image processing, support for the latest image formats, etc?
2. Will you need to hire or train engineers to implement all the requirements with respect to image optimization?
3. Are images frequently uploaded to your website by vendors or users?
4. Will uploaded images be delivered to different types of devices and browsers across the world?
5. Is the effort required to build and maintain the image pipeline greater than 40 hours per year?
6. Will it take more than two hours to restore media content in the event of an outage?
7. Will you be building instrumentation to measure and optimize performance impact due to images?
8. Do you need features to improve the accessibility of images? (e.g. support for color blindness, automatic `alt`-text injection, etc.)
9. Will you be required to automatically select the quality of images to be delivered based on their structural similarity to originals?

10. Are you required to moderate uploaded content for illegal or inappropriate images and videos?
11. Are there legal or licensing issues to be addressed with respect to image compression, transmission, distribution?

To answer some of these questions, it would also be a good idea to have an inventory of the different types of images and image functionality required on the website. The following table shows a sample.

| IMAGE TYPE | IMAGE PROCESSING REQUIRED | LICENSED FORMATS REQUIRED | ACCESSIBILITY FEATURES REQUIRED | IMAGE MODERATION REQUIRED |
|---|---|---|---|---|
| UI chrome images | N | N | Y | N |
| Landing page images | Y | Y | N | X |
| Product images on detail page | Y | Y | Y | Y |
| Blog images | Y | X | Y | Y |

If you are currently serving your own images (or planning to) and decide to consider CDNs, here are some options:

- Cloudinary:[153] a paid service which offers a free tier.
- Imgix:[154] a paid service which offers a free trial.
- Thumbor:[155] an open-source software alternative.

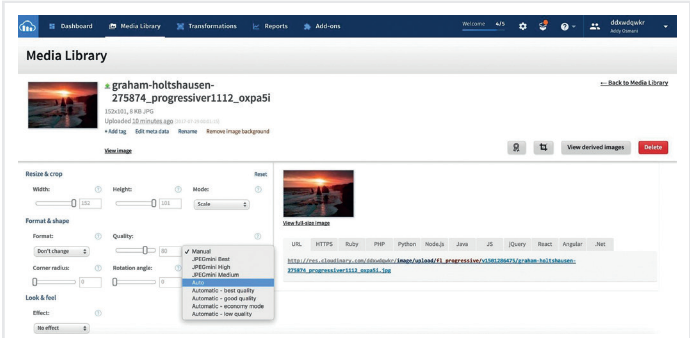
### CLOUDINARY AND IMGIX

Unless you are the owner of a network of servers like Cloudinary and imgix, their first huge advantage over rolling your own solution is that they use a distributed global network system to bring a copy of your images closer to your users. It's also far easier for a CDN to future-proof your image loading strategy as trends change – doing this on your own requires maintenance, tracking browser support for emerging formats, and following the image compression community.

Both services provides so developers can access the CDNs programmatically and automate their processing. Client libraries, framework plug-ins, and documentation are also available, with some features restricted by price.

153 http://cloudinary.com/
154 https://www.imgix.com/
155 https://smashed.by/thumbor

For static images, both Cloudinary and imgix offer a range of image manipulation methods, and both support primary functions such as compression, resizing, cropping, and thumbnail creation.




*Cloudinary's Media Library. By default, Cloudinary encodes non-progressive JPEGs. To opt in to generating them, check the "Progressive" option in "More options," or pass the* fl_progressive *flag.*

Cloudinary lists seven broad image transformation[156] categories, with a total of 48 subcategories within them. Imgix advertises over 100 image processing operations.[157]

There are so many uncontrolled variables in image manipulation that a head-to-head performance comparison between

156 https://smashed.by/transformation
157 https://smashed.by/operations

the two services is difficult. So much depends on how much you need to process the image – which takes a variable amount of time – and what size and resolution are required for the final output, which affects speed and download time. The most important factor for you might ultimately be cost. If you're determined to set up your own image CDN, in the following section we'll take a close look at Thumbor and the optimization features it can provide.

SUCCESS STORY ■ "Our average page weight went from 4 MB to 400 KB after implementing [optimization with an image CDN]. [...] Best of all, we have seen a 100% increase in conversions and a 20% decrease in bounces"
—Furnspace (May, 2018)

# Using Thumbor to Roll Your Own Image CDN

Thumbor[158] is an open-source, smart image optimization service that can be used as a self-hosted image CDN. Like any other image CDN[159] it includes features to crop, resize, flip, and transform images or apply filters on demand. It supports multiple image compression formats like JPEG, PNG, WebP, and GIF.

If you already have a background in how image CDNs can be used to improve image load performance (you read the previous chapter, right?), dive straight in to this guide. We will explore the image optimization features included in Thumbor and available options for hosting it.

## API Basics

The Thumbor consists of smart URLs that can be used to specify the original image URL and the transformation required. Let's illustrate this with an example. For an original image URL like this one:

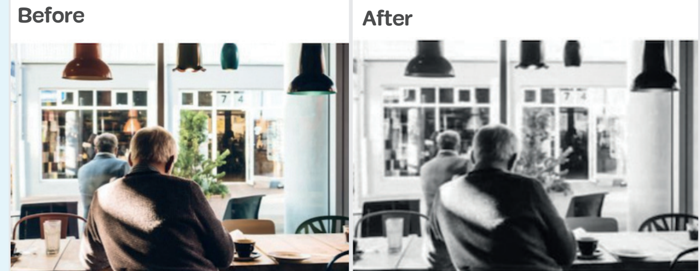
```
<img src="<your-server>/500x300.jpg" alt="original"/>
```

158 http://thumbor.org/
159 https://smashed.by/imagecdns

The following smart URL can be used to apply a Thumbor transformation that would display the image in grayscale and resize it:

```
<img src="<thumbor-server>/unsafe/300x200/filters:g
rayscale():format(jpeg)/<your-server>/500x300.jpg"
alt="thumbnail"/>
```

Here's what it looks like before and after applying the transforms:




In general, a Thumbor URL can be written with this formula:

```
http://<thumbor-server>/hmac/trim/AxB:CxD/fit-in/-
Ex-F/HALIGN/VALIGN/smart/filters:filtername(arguments)
:filtername(arguments)/<image-uri>
```

It consists of the following components:

| URL COMPONENT | DESCRIPTION |
|---|---|
| <thumbor-server> | The URL and port number of the server where the Thumbor service is hosted. |
| hmac | A 28-character security key or authentication code to prevent URL tampering. In the absence of a security key, the word “unsafe” may be used in this part of the URL. |
| trim | Removes surrounding space in images using top-left pixel color unless specified otherwise. |
| crop (AxB:CxD) | Used to manually crop the image at left-top point AxB and right-bottom point CxD. |
| size (-Ex-F) | Specifies the desired size of the image (300×200 in the previous example, with E as width in pixels, F as height). This is optional and can be skipped to use the original size of the image. The - (minus) sign can be used to indicate the direction in which the image should be flipped. |

| | |
|---|---|
| `fit-in` | Specifies that the generated image should not be cropped and should just fit into the dimensions specified in the size component of the URL. |
| `HALIGN` | Indicates the horizontal alignment of the crop. Possible values are `left`, `right` or `center`. |
| `VALIGN` | Indicates the vertical alignment of the crop. Possible values are `top`, `middle` or `bottom`. |
| `smart` | Requests smart detection of focal points. |
| `filters: filtername (arguments)` | Specifies the desired filters (`grayscale()` and `format(jpeg)` in the example above). Filters may be omitted completely or multiple filters may be specified as a colon-separated list. |
| `<image-uri>` | This is the original image URL. |

Details about each of these components are available in Thumbor's documentation.[160] Now we'll explore some of the key features supported in Thumbor, and illustrate how they can be used through examples.

160 https://smashed.by/thumbordocs

# Features

## IMAGE COMPRESSION FORMATS

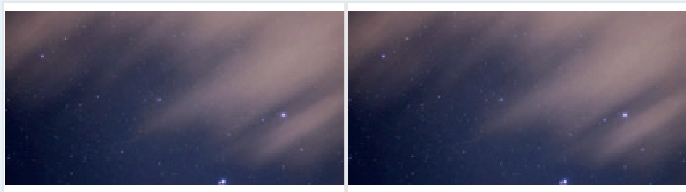
Thumbor supports WebP, JPEG, PNG, and GIF formats and `format()`[161] may be used in the filters section of the URL to convert images to any of these formats. The following example shows how an image may be converted from JPEG to WebP.

Source file: *<your-server>/images/local-test-file.jpeg*

Transform using the `format()` filter:

```
<thumbor-server>/unsafe/740x380/
filters:format(webp)/<your-server>/images/local-test-
file.jpeg
```

Output:



161 https://smashed.by/thumborformat

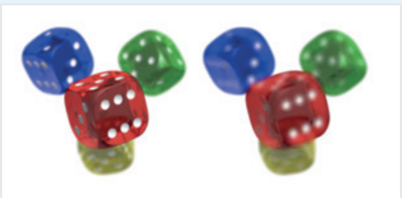
## TRANSFORM WITH FILTERS

A list of all the filters Thumbor supports[162] can be found in the documentation. We will see how these can be used or combined in the following examples.

### Convert to WebP and Blur

Source file: *<your-server>/images/dice.png*
Transform using the `format()` and `blur()`[163] filters together. Here we apply a radius of 4 and sigma of 3 in `blur()`.

```
<thumbor-server>/unsafe/filters:blur(4,3):format(web
p)/<your-server>/images/dice.png" alt="Blurred dice
converted to WebP."
```



*The WebP image output by Thumbor from the JPEG original shows the blur filter applied in the smart URL.*

162 https://smashed.by/thumborfilters
163 https://smashed.by/thumborblur

### Convert to JPEG and Add a Fill color

Source file: *<your-server>/images/dice.png*

Transform using the `fill()`[164] and `format()` filters together. The hex code for the color to be filled is specified here in the `fill()` function as `C0C0C0` and the 1 indicates that we want to fill the transparent areas of the image as well.

```
<thumbor-server>/unsafe/filters:fill(C0C0C0,1):format(
jpeg)/<your-server>/images/dice.png" alt="Filled dice
converted to JPEG."
```

*As well as converting from PNG to JPEG, the background color has been filled.*

## OPTIMIZATION

Thumbor can also perform on-the-fly optimizations on images without transforming them. You can enable these by configuring Thumbor to use any of the available plug-ins

164 https://smashed.by/thumborfill

like MOZJPEG, pngquant, jpegtran, and gifv. The following examples illustrate how this can be done.

### Optimizing a JPEG Using Thumbor's MozJPEG encoder

Source file: *<your-server>/images/500x300.jpg*
Use Thumbor to render it without any transform:

```
<thumbor-server>/unsafe/<your-server>/images/500x300.
jpg" alt="An image with MozJPEG encoding."
```

For an image of comparable quality, the file size goes down from 33.03 KB to 25.75 KB.

### Optimizing an Animated GIF MP4 Video Using Thumbor's GIFv Optimizer

Source file: *<your-server>/images/local-test-file.gif*

Thumbor's gifv optimizer uses FFmpeg (see chapter 15) to convert GIFs to video for reduced file size:

```
<thumbor-server>/unsafe/740x380/
filters:gifv(mp4)/<your-server>/images/local-test-
file.gif
```

*Automating the conversion of a large animated GIF to a video (x264 MPEG4)*

## CONFIGURATION

When self-hosting Thumbor installations, you can customize the configuration file. Default configurations are available as a Python script and can be changed by writing them out to a commented text file using the command below:

```
thumbor-config > ./thumbor.conf
```

Required default configurations may be edited. Some of the generally relevant configurations are shown below.

1. Minimum and maximum height and width: allow users to set the lowest and highest dimensions for generated images. The available configuration parameters are as follows:

```
MIN_WIDTH = 1
MIN_HEIGHT = 1
MAX_WIDTH = 1200
MAX_HEIGHT = 800
```

2. Quality: allows users to set the quality at which JPEG images will be generated, with a default value of 80.

3. Maximum age: defines the number of seconds after which the image will expire from the browser cache. For example, here's how to set `MAX_AGE` to enable cache for 24 hours:

```
MAX_AGE = 24 * 60 * 60
```

4. `AUTO_WEBP`: tells Thumbor to always send WebP images if the request indicates that the browser supports the format.

```
AUTO_WEBP = True
```

5. `SECURITY_KEY`: specifies the security key that Thumbor can use to sign secure URLs.

Details about all configurable parameters[165] can be found in Thumbor documentation.

### HOSTING

Thumbor can be run locally in the development or production environment, or hosted in the cloud through major service providers including Google Cloud Platform. When self-hosting, users also need to address other requirements such as:

- Configuration: updating configuration file as per required defaults.
- Load balancing: configuring a static IP address and global load-balancer.
- Security: ensuring security by enabling the security key feature.
- Monitoring: monitoring the application using Supervisor or other solutions.
- Logging: capturing event logs and metrics.

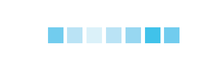
The maintenance overhead due to these requirements should be considered when hosting an image CDN like Thumbor.

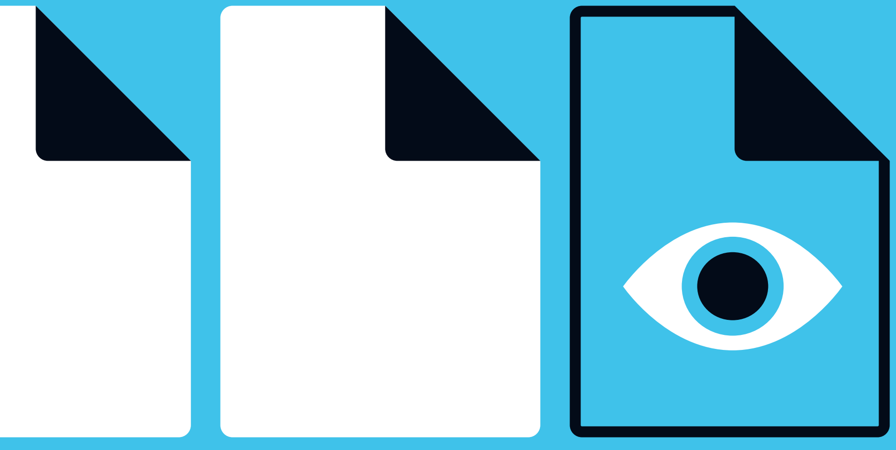
165 https://smashed.by/thumborconfig

Thumbor provides a rich set of features for image optimization and transformation, and this guide highlights some of them. It also supports intelligent cropping and resizing functionality and other features that are well documented.[166] Another recommended starting point for your Thumbor journey is the web.dev Thumbor guide.[167]

Thumbor can help to optimize the image delivery pipeline without relying on a third-party CDN infrastructure provider. While there would be additional maintenance required, installation and operational procedures are similar to any other application and can be easily addressed in most cases.

166 https://smashed.by/thumborfeatures
167 https://smashed.by/thumborguide





## Part Four

# New Image Formats

CHAPTER 17 ■ HEIF and HEIC .......... 382

CHAPTER 18 ■ AVIF .................... 398

CHAPTER 19 ■ JPEG XL ................ 416

CHAPTER 20 ■ Comparing New Image Formats .......... 433

## New and Emerging Image Formats for the Future of the Web

While JPEG is the most widely used and supported standard for image compression, the industry has been on a constant lookout for a new format that will be widely accepted and supported by all the major browsers across mobile and desktop. There are several reasons for wanting to switch to a new file format.

- Despite innovations in recent years like the MozJPEG optimizer and Guetzli (see chapter 7), popular JPEG compression techniques are generally quite old. With the increased use of images in e-commerce and social media sites, there is a need for rendering good-quality images at a high speed without stressing CPUs. This implies a need for better compression techniques that can further reduce the size of images without affecting image quality, thereby reducing the bandwidth needed to render quality images.

- JPEG supports a maximum file size of 65,535×65,535 pixels. Newer cameras and photography equipment are already exceeding this limit.

- JPEG compression is not lossless, although solutions like lossless JPEG have existed for some time (1993). Increasing the compression of the image usually means the picture will look pixelated and a little blurry.

- JPEG images have a long header which cannot be compressed. This makes them unsuitable for use in LQIPs, which are used to display low quality images while the actual image is still loading.

- JPEG images are not fully responsive by design, meaning you cannot always render variants of the same image based on the user's screen size and display.

- JPEG does not support transparency or transparent backgrounds.

- JPEG does not support new photo technology like bursts of shots, panoramas, live photos and 3D scene data.

- JPEG does not support adding stickers or overlays (HEIF, AVIF and JPEG XL can do this).

JPEG 2000 and JPEG XR were introduced to replace JPEG but were never able to secure broad adoption. JPEG 2000 is supported only in Safari and on iOS, while JPEG XR is supported only in Edge and IE.

The WebP format was developed by Google to overcome the limitations of JPEG: namely, support for lossless compression and transparency. Even though it achieves

15–20% better compression compared with JPEG, it has failed to gain maximal adoption. (See chapter 9 for a full discussion of WebP.)

To be widely adopted and supported by all the major players in the mobile and browser world, a new image format has not only to overcome the limitations of JPEG, but also be royalty-free. Currently, the contenders for the place of JPEG in the future are: HEIF, AVIF, and JPEG XL.

The following chapters explore each new format in detail and provide insight into their potential adoption.

CHAPTER 17

# HEIF and HEIC

The high-efficiency image file format (HEIF) made its consumer debut on Apple's iPhones in iOS 11, offering smaller file sizes and higher image quality than JPEG. It achieves this using more advanced compression methods and is based on the high efficiency video coding (HEVC) format. HEIF includes features like transparency and 16-bit color, which are a nice upgrade over JPEG's 8-bit color. HEIF has very limited browser and platform compatibility, but on iPhones and iPads HEIF images are converted back to JPEGs when sharing. While other image formats covered in this part of the book may offer broader compatibility, HEIF is worth being familiar with as you may have users wishing to work with this format on iOS.



## HEIF: The High Efficiency Image File Format

In 2017 anyone who upgraded to iOS 11 from an older version, most likely noticed the new ability in iPhones to capture "Live Photos." These live photos capture a series of frames over three seconds rather than a single frame.

When you transfer these photos over a messaging app or email, they are sent as still JPEG image files since JPEGs cannot store multiple frames. If you were to transfer these images to a laptop or desktop computer running macOS High Sierra (10.13) or above, you would notice the images still work as live photos and the files have an *.heic* extension. This is the new image file format that was introduced with iOS 11 and has been used to store photos on iPhones and iPads since then.

To understand this new format better, we will try to discuss everything about the HEIF format from which HEIC is derived. We will also see how it fares as compared to JPEG and other image formats in terms of image optimization, support available, and the possible advantages and disadvantages of using HEIF (or HEIC) on web pages.



### WHAT HEIF IS

HEIF stands for high efficiency image format and it is a container format: a wrapper that can contain a variety of different types of data compressed by standardized coders. Containers handle packaging, transport, and presentation but don't specify what codec the container format uses. MPEG-4 is a container often represented by the *.mp4* file type.

While the current standard allows HEIF to contain HEVC, Advanced Video Coding (AVC), or JPEG-encoded bitstreams, iOS uses HEVC encoding. HEVC – also known as H.265 – was developed by the Moving Picture Experts Group (MPEG) as a successor to AVC and offers a 25% to 50% better compression for the same video quality. Thus, HEVC-encoded images contained in HEIF files use the same advanced compression methods to achieve image files which are half the size of JPEG files for the same or better picture quality.

HEVC, however, is patented technology, and using it for encoding or decoding photos requires a license.[1]

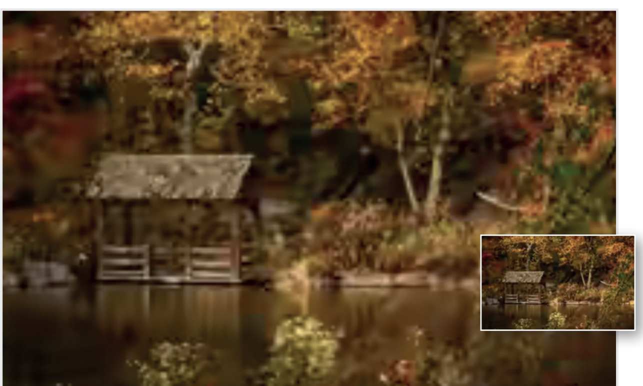
The technical specification for HEIF was finalized in 2015 and the HEIC format was released on iOS in 2017. HEIF files can store the following types of data, illustrated in examples from Nokia Technologies.[2]



1 https://smashed.by/hevc
2 https://smashed.by/nokia

### Still Images

Each HEIF file includes a high-quality image, its thumbnail, and their related metadata (e.g. EXIF).

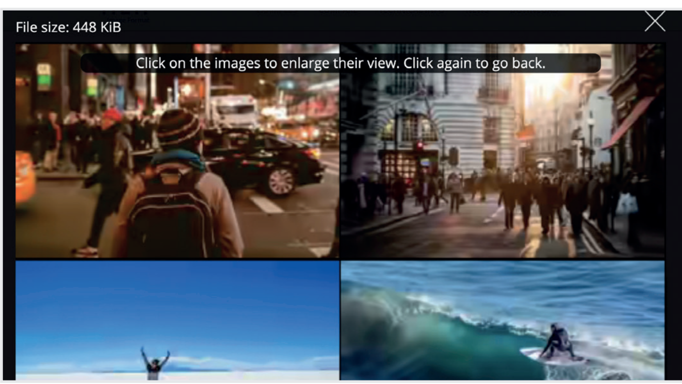


*Thumbnail image.*

*High-res image.*

### Image Collections

Each file contains a group of images, their thumbnails, and related metadata.

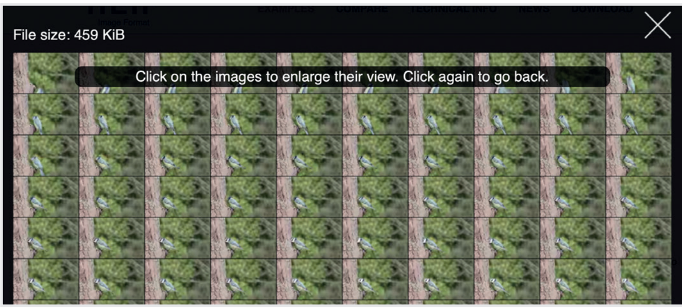



*An example image collection within one HEIF file.*

HEIF
HEIC

### Image Sequences

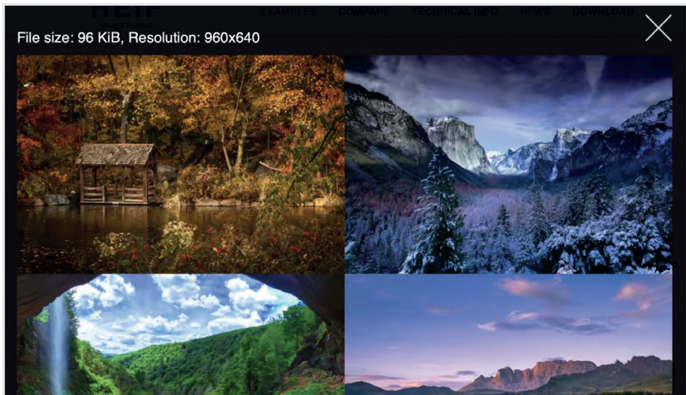
HEIF allows image bursts or cinemagraphs that are time-related, along with their associated properties and thumbnails. Since these images are very similar in nature, the compounded file size of all frames is much smaller than expected. Different algorithms may be used to reduce the overall size of the image sequence based on predicted similarities.




*A burst of images of a perching bird, all stored in one HEIF file.*

## Image Derivations

These store editing instructions separately in the image file along with the original image(s). Derivations can include rotation, overlay, and grid view (as seen in the example). These are stored alongside the original image so that it is never lost.



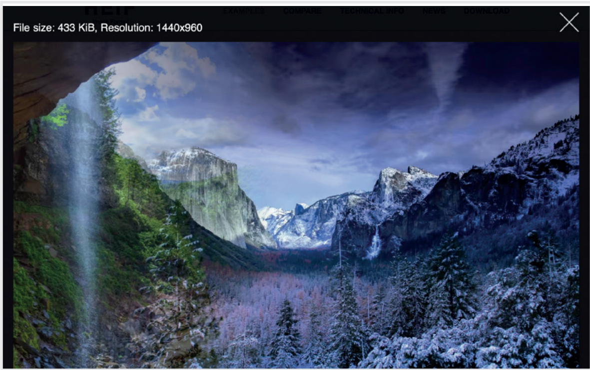

*The information to create the grid view is stored in the HEIF file alongside the images.*

## Auxiliary Image Items

Image data which complements another image item can be stored in the HEIF file, such as a depth map or alpha plane. The example shown includes a derived image that is composed of two overlays and one of the images has a gradient alpha mask that provides gradual transparency.

*Original image.*






*The image with the mask derived from auxiliary image data in the HEIF file.*

### Image Metadata

EXIF, Extensible Metadata Platform (XMP), or any other metadata that may accompany the image can be contained in an HEIF file.

## OTHER FEATURES OF HEIF

As noted, HEIF is a media container format. It is not itself an image or video encoder. Hence, the quality of the visual media depends highly on the proper usage of the visual media encoder (e.g. HEVC). Thus, the HEIF standard might be easily extended in the future to other visual media codecs.

HEIF has many powerful features, some of which are currently not present in other image file formats like JPEG, PNG, or GIF, such as:

- Encapsulation of images encoded using HEVC, scalable HEVC (SHVC), multiview HEVC (MV-HEVC), AVC, or JPEG, which may be extended to other codecs in the future.
- Encapsulation of image sequences with audio encoded using HEVC, SHVC, MV-HEVC, or AVC.
- Support for efficient storage of image bursts and cinemagraphs.
- The use of widely adopted ISO base media file format (ISO BMFF) for storage.

- Support for both lossy and lossless image data storage.
- Support for transparency and 16-bit colors allowing for pictures with a wider range of colors.
- Support for image editing without altering the original image. The editing information is stored as metadata that can be undone to revert to the original image.
- Easier distribution of still images, image collections, and related metadata.

*This image compares the JPEG and HEIF versions of an image with respect to their quality and sizes. Left: 156 KB JPEG; right: 135 KB HEVC. (Source: 500px[3])*



3 https://smashed.by/nokia

## HEIC: A Subset of HEIF

HEIC (high efficiency image container) is the file format for image files which use the HEIF standard and are encoded using HEVC. Apple uses the HEIC container format to store both still images and live photos encoded with HEVC. Since HEIC files sizes are much smaller than JPEG, photos will use less of the limited storage available on iPhones and iPads. With the increasing quality of images, sizes of JPEG image files were growing, and space for storing a growing library of photos was always an issue on these devices.

With the implementation of HEIC format for images, Apple hopes that users will be able to easily store a large number of photos on their devices. Features like Live Photos and advanced editing also improve the overall photography experience using iPhone and iPad.



Using the features supported in HEIC files, the inbuilt photo editor on iPhone and iPad is able to offer distinct editing features:

- Ability to revert to original photos even after you have saved the changes because editing instructions are saved separately.

- Remove effects like background blur or depth control which you applied to your photos when clicking them.

- Edit Live Photos with options to turn off the audio and select a particular key photo from the series of frames which will be used when sharing the photo as a still image.

Both HEIC and AV1 (see chapter 18) support a smaller pixel volume as they are based on a video standard. To overcome this, the format containers support tiling to allow for very large megapixel files, as is highlighted in the table below from Cloudinary:[4]

| FORMAT | MAXIMUM IMAGE DIMENSIONS (IN A SINGLE CODE STREAM) | MAXIMUM BIT DEPTH, MAXIMUM NUMBER OF CHANNELS |
|---|---|---|
| JPEG | 4,294 megapixels (65,535 × 65,535) | 8-bit, three channels (or four for CMYK) |
| PNG | Theoretically 4 megapixels ($10^{18}$) but no way to efficiently decode crops | 16-bit, four channels (RGBA) |
| WebP | 268 megapixels (16,383 × 16,383) | 8-bit, four channels (RGBA) |
| HEIC | 35 megapixels (8,192 × 4,320) | 16-bit, three channels (alpha or depth as separate image) |



4 https://smashed.by/universality

| | | |
|---|---|---|
| AVIF | 9 megapixels (3,840 × 2,160) | 12-bit, three channels (alpha or depth as separate image) |
| JPEG XL | 1,152,921,502,459 megapixels (1,073,741,823 × 1,073,741,824) | 24-bit (integer) or 32-bit (float), up to 4,100 channels |

## Transfer and Conversion

So, time to get real – can we use HEIC and HEIF cross-platform today? The short answer is… yes and no. While Apple devices (iOS, macOS) have strong native support for viewing these formats, and popular image sharing services are starting to improve their support for reading the format, the tools for converting to HEIC and HEIF could be better.

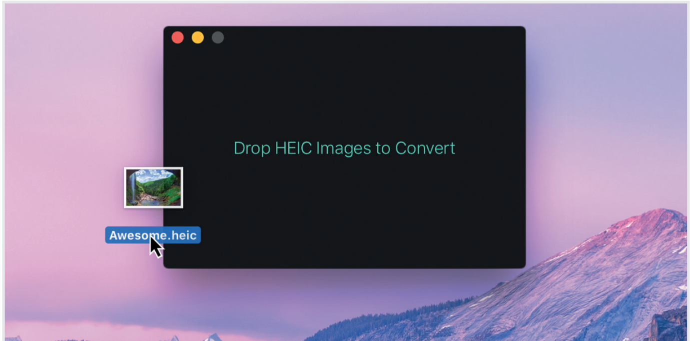


### IOS AND MACOS

iOS automatically converts HEIC photos to JPEG when sharing them on other platforms, so a naive user may not notice what format the photo is in. HEIC (and HEIF) files may be viewed, edited, or saved as JPEG using the Preview software on macOS 10.13+. HEIC Converter[5] by Sindre Sorhus is a

5 https://smashed.by/HEICconverter

macOS app that also makes it easy to quickly convert images in the HEIC format to JPEG or PNG.




*Converting HEIC images using HEIC Converter.*

#### ANDROID



Android started supporting HEIF photos with Android 9 Pie,[6] released in August 2018, to improve compression of pictures and reduce the amount of storage needed. It is available for Google Pixel devices. Phones running Android Pie cannot create HEIF files yet, due to the special hardware and licenses required for this. However, Android Pie supports compression of images using HEIC[7] and viewing them. Android app developers can use this format in their apps[8] targeted for Android Pie and above.

6 https://smashed.by/pie
7 https://smashed.by/androidsupport
8 https://smashed.by/media

### WINDOWS

Microsoft has released HEIF Image Extensions[9] for Windows 10 which enables reading and writing of HEIF files. A third-party image viewer or conversion software such as CopyTrans HEIC[10] is required to view the original HEIF/HEIC on older versions of Windows. The iMobie HEIC Converter,[11] another free online tool, provides batch conversion of photos from HEIC to JPEG.

### THIRD-PARTY TOOLS AND SERVICES

Google Photos supports photos uploaded in HEIF and HEIC formats, and these can be easily viewed in the web and native app versions of the service. Photos can be downloaded as HEIC or JPEGs on other devices with Google Photos using the **Save as**... function.

Releases of Adobe Lightroom and Camera Raw[12] since June 2018 support HEIC image files on capable operating systems. Similarly, HEIC is also supported in other image editing software like GIMP and Pixelmator. There are other third-party tools that allow conversion of existing JPEGs to HEIF and HEIC files: jpgtoheif.com[13] provides steps to convert files using FFmpeg; and a sample implementation of HEIF is provided by Nokia Tech.[14]



9 https://smashed.by/heifextensions
10 https://smashed.by/copytrans
11 https://smashed.by/imobie
12 https://smashed.by/heiclightroom
13 http://jpgtoheif.com/
14 https://smashed.by/nokiaheif

## HEIF and HEIC for Web Developers

The HEIF and HEIC image formats are not supported in the HTML `<img>` element, and no browsers (including Safari) support HEIF or HEIC images at the moment. As such, if an application allows users to upload these images, developers need to take special care to be able to display these images on all browsers. All browsers support H.264 playback and a JavaScript parser could be used to pass the frame data to the H.264 decoder to read the image file as video.

The libheif[15] library, an encoder and decoder for HEIF files, may be used in browsers by compiling it to JavaScript using Emscripten.[16] The demo[17] for this library allows users to upload and view HEIF and HEIC files. Alternatively, Cloudinary also supports HEIF and HEIC images[18] and can be used to embed images on web pages.

HEIF
HEIC

While there are many benefits to using HEIF and HEIC, the fact remains that HEVC is patent-encumbered and, hence, difficult to implement and support. Apple managed to incorporate it in its ecosystem, but since Android and Windows operating systems are used by a variety of device manufac-

15 https://smashed.by/libheif
16 https://emscripten.org/
17 https://smashed.by/libheifdemo
18 https://smashed.by/cloudinarysupport

turers, the image format is yet to be universally adopted by them. Since browsers do not support the format, developers have to take additional steps to display the format.

Even two years after its launch and speculations that it would replace JPEG, the format has yet to find solid ground in web and mobile applications. Meanwhile, JPEG still rules and we wait for a format that will be as widely accepted in the future as JPEG is today.

Next we'll discuss a potential competitor, AVIF, which also uses the HEIF specification with AV1 encoding.

CHAPTER 18

# AVIF

The AV1 image file format (AVIF) is an open-source image format for storing still and animated images that supports very efficient lossy and lossless compression modes. You can think of it as a royalty-free cousin of HEIF. Images can be up to ten times smaller than JPEGs of similar visual quality.

AVIF is the image version of the popular AV1 video format, which is compatible with high dynamic range (HDR) imaging, allowing images to reproduce a broader range of luminosity than is possible with standard digital imaging techniques. AVIF also supports ten bits of color depth and monochrome channels.



AVIF compresses much better than most popular formats on the web today (JPEG, WebP, JPEG 2000, and so on), with some tests showing it offers a 50% saving in file size compared to JPEG and 20% compared to WebP. Below is a high-level breakdown from Daniel Aleksandersen's Ctrl blog,[19] comparing AVIF to JPEG and WebP images of the same visual quality:

19 https://smashed.by/webpvsavif

File Size Comparison

50% smaller than .jpeg .avif

30% smaller than .jpeg .webp

.jpeg

*Daniel Aleksandersen's file-size comparison of JPEG, webP and AVIF images of the same visual quality.*

AVIF aims to produce high-quality compressed images that lose as little quality as possible during compression. These images can support transparency (for UI elements, similar to alpha PNG), lossy or lossless compression, HDR color (think better color bit depth and brightness), wide color gamut, and can support a sequence of animated frames (giving us a lighter, high-quality version of animated GIFs).

## Context



In February 2019, the Alliance for Open Media[20] (AOMedia) officially released the first specification for the AV1 image file format.[21] AOMedia itself is a non-profit organization supported by industry heavyweights such as Mozilla, Google, Microsoft, Netflix, and others.

20 https://aomedia.org/
21 https://smashed.by/av1spec

The AV1 standard was developed as a new open-source video coding format which is both state of the art and royalty free. Additionally, the AV1 format is free of patent licensing requirements, reducing the friction for tools to adopt it. It claims to offer 20% better data compression than HEVC (high efficiency video coding, also known as H.265 – see chapter 17) and 50% better than AVC (advanced video coding, or H.264).

Even though new image file formats have been introduced in the past, because of the intrinsic support available for AVIF from the beginning, it is highly likely that adoption of AVIF across web and mobile devices will be quicker than that of its predecessors. Early adopters of AVIF are likely to benefit more from the advantages provided by this new file format.

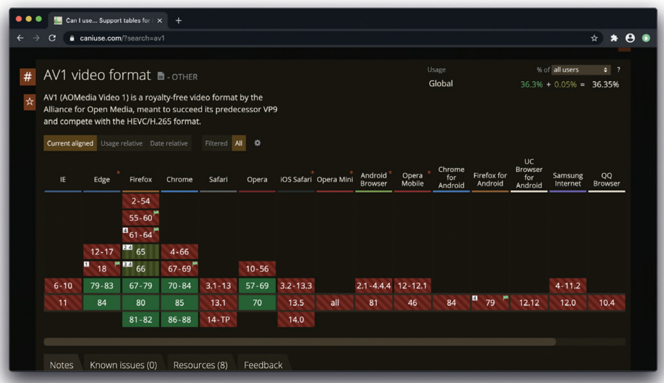


Some image compression experts consider AVIF a compelling new format that could get near universal acceptance and replace JPEG over the years. This is mainly because AVIF may offer up to 50% better compression compared to JPEG for similar image quality.

Let's cover the basics of AVIF and why we need it. I will also provide a list of resources available to developers who want to start using AVIF immediately.

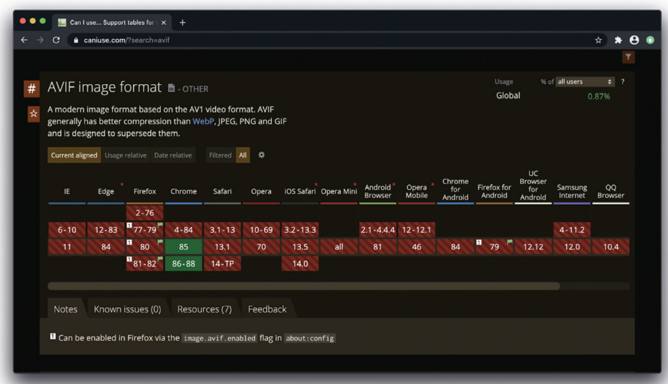
## Browser Support for AVIF

The AV1 video format is already supported by Chromium-based browsers and Firefox.




*Browser support for the AV1 video format. (Source: caniuse.com)*

At the time of writing, support for AVIF can be found in a growing number of browsers, with Chrome 85+ able to display the format and Firefox 77+ support activated via a flag. While AVIF images have relatively limited support, they can be used as a progressive enhancement.




*Browser support for the AVIF image format. (Source: caniuse.com)*

## AVIF AS A PROGRESSIVE ENHANCEMENT

While AVIF is not supported in all browsers at the time of writing, it can still be used in HTML via the `<picture>` element. As `<picture>` allows browsers to skip images they do not recognize, images can be included in our order of preference, and the browser will select the first one it supports.



```
<picture>
    <source srcset="img/photo.avif" type="image/avif">
    <source srcset="img/photo.webp" type="image/webp">
    <img src="img/photo.jpg" alt="Description"
width="360" height="240">
</picture>
```

## Features of AVIF

The AVIF specification provides a method for using AV1 compression for still images or image sequences. AVIF allows the storage of an AV1 bitstream in a HEIF file compatible with ISO BMFF. This is similar to the HEIF and HEIC formats mentioned in chapter 17, except that it uses the royalty-free AV1 encoding instead of HEVC. AVIF claims the following advantages over JPEG:

- High-quality images with up to 50% more effective compression than JPEG.
- Features for supporting animations, live photos, and more through multilayer images stored in image sequences.
- Better support for graphical elements, logos, and infographics where JPEG had limitations.
- Better lossless compression than JPEG.
- Support for high dynamic range (HDR) and wide color gamut (WCG) images with a better span of bright and dark tones.
- Support for monochrome images as well as multichannel images.

The upper end of the 50% savings may only apply to large images with high pixel density (over two megapixels). This is unsurprising as there tend to be more duplicate pixels in large canvases meaning compression algorithms can be more successful at optimizing them. Keep in mind that this upper end may not be indicative of the savings you will see. Your savings will also be influenced by the type of imagery being compressed. If the content is photographic, it may compress well. If it contains text overlays on top of photographic content, however, these savings might well be lost, as you will likely have to increase the effective quality levels to avoid the visible subsampling blur on the text.

## AVIF Performance

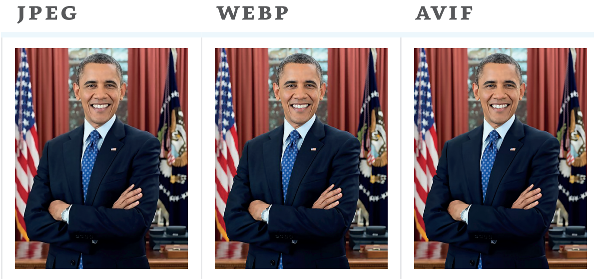
Many of the major players in the industry have been quick to announce support for AVIF files. Netflix published the first AVIF images in December 2018. VLC media player has recently added support for AVIF files. Microsoft added support for AVIF in Windows 10 (version 1903) in May 2019, which includes support for AVIF in File Explorer and Microsoft Paint. Mozilla is working to support AVIF in Firefox, and it's expected that Apple and Google will soon build support for AVIF files in their browsers.

At the moment, however, most of this is still either speculation or work in progress. Web and image performance analysts are trying various hacks to embed AVIF images in their web pages and test the overall performance of AVIF images compared with other formats in terms of compression ratio, image quality, and decoding speeds while rendering. Here's a summary of their findings.

1. In an analysis published in Web Performance Calendar[22] at the end of 2018, AV1 demonstrated a 50% gain in image compression over JPEG, and a 26% gain over WebP when comparing AV1, WebP, and JPEG images at a similar quality to a lossless PNG.

22 https://smashed.by/webperfcal

Following are the test images published by Performance Calendar that were used for the above comparison:




The following comparison uses the DSSIM scores to judge the image quality and shows the gain in compression in each case:

| IMAGE TYPE | DSSIM | IMAGE DATA WEIGHT | GAIN COMPARED WITH JPEG |
|---|---|---|---|
| **JPEG** | 0.005833 | 49,314 | 0 % |
| **WebP** | 0.006170 | 36,426 | 26.1 % |
| **AV1** | 0.005782 | 23,796 | 51.7 % |

AVIF

2. On the downside, the same analyst also observed that decoding an AVIF file with the decoders available at present may require 10 to 15 times more processing power and time as compared to WebP. With more efficient decoders, the processing time could be improved. Eventually, web developers who want to

use AVIF images will have to consider if the gain due to compression is enough to offset the additional processing time required.

3. Kornel Lesiński of ImageOptim confirmed in his talk at 2018's performance.now() conference[23] that AVIF provides optimization gains of about 50% compared with JPEG.

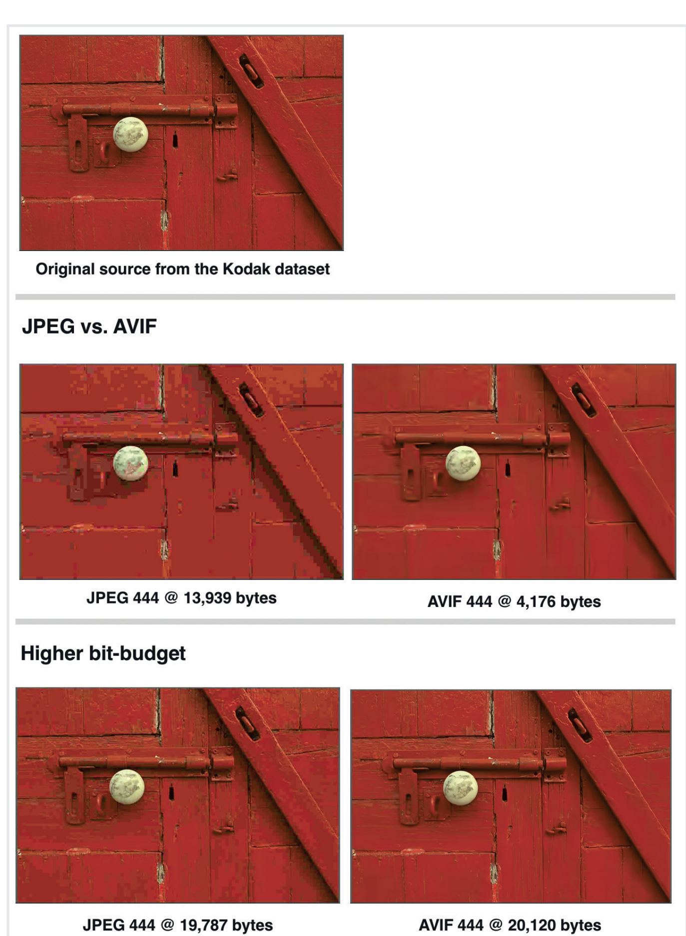
4. Various independent analysts have published the results of a comparison between different formats[24] in a Google spreadsheet.

### NETFLIX AVIF EXPLORATIONS

In early 2020, Netflix shared details of its work on improvements to HDR images[25] for the UI and aims to use AVIF for encoding these images.

From Netflix's AVIF comparison tests, we can see an image from the Kodak dataset (second row) with JPEG subsampling 4:4:4 at 13,939 bytes and AVIF 4:4:4 at 4,176 bytes. Notice how the JPEG has more blocky artifacts around the slanted edge and more visible color distortions. In contrast, the AVIF looks clearer, despite being 66% smaller than the JPEG. This is impressive given the compression factor of 282×.

23 https://smashed.by/kornelperfnow
24 https://smashed.by/formatcomparison
25 https://smashed.by/avifnextgen




*A Netflix image comparison of AVIF and JPEG comparing subsampling at 4:4:4.*

In the third row is the same comparison under a higher bit-budget. The JPEG 4:4:4 clocks in at 19,787 bytes versus the AVIF 4:4:4 at 20,120 bytes.

Notice how the JPEG still appears to show blocky artifacts around the slanted edge, but the AVIF encode looks very close to the original source.

## Developer Resources

In addition to the AVIF specification (and current working draft)[26] on GitHub, and in the absence of browser support, there are multiple resources being published on how to use AVIF in development. This section summarizes the various available resources and techniques suitable for different use cases.

### AVIF IMAGE SAMPLES

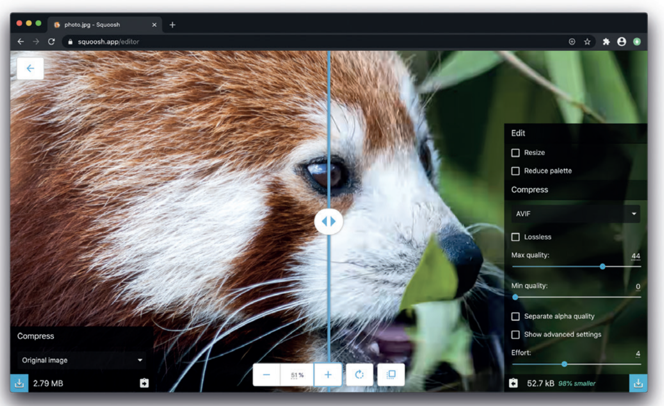
Both Netflix[27] and Microsoft[28] have published AVIF file samples that can be used for initial testing.

26 https://smashed.by/avifworkingdraft
27 https://smashed.by/netflixsample
28 https://smashed.by/microsoftsample

## ENCODING AND DECODING AVIF IMAGES

Several open-source projects demonstrate different methods to encode/decode AVIF files:

- AOMedia has produced the reference software libaom[29] for AV1 encoding and decoding.
- libavif[30] is a portable C implementation of the AVIF specification based on libaom, that can encode and decode AVIF images.
- Squoosh,[31] a web app that lets you use different image compressors, now also supports AVIF, making it relatively straightforward to convert and create *.avif* files online.




*Using Squoosh to compare and create AVIF files.*

29 https://smashed.by/aom
30 https://smashed.by/libavif
31 https://squoosh.app

- go-avif[32] implements an AVIF encoder for Go using libaom. As shown below, it can be used to convert other image files to AVIF using different settings:

```
# Encode jpeg to avif with default settings
avif -e cat.jpg -o kitty.avif
# Encode PNG with slowest speed and quality 15
avif -e dog.png -o doggy.avif --best -q 15
# Fastest encoding
avif -e pig.png -o piggy.avif --fast
```

- The MP4Box[33] application from the GPAC multimedia open-source project can create AVIF files from frames in an AV1 video stream by specifying the timestamp of the frame. MP4Box.js is a decoder based on MP4Box with support for AVIF files. An AVIF file may be inspected online using the MP4Box.js demo.[34]

- AVIF.js[35] provides support for displaying AVIF files in a browser. It repacks the AVIF image as a single-frame AV1 video and decodes it using the native AV1 decoder, which is already supported in the latest versions of Chrome and Firefox. The AV1 video embedded in a MP4 video file can be decoded and displayed using a standard `<video>` tag. Demo files[36] for this are available.



32 https://github.com/Kagami/go-avif
33 https://smashed.by/mp4box
34 https://smashed.by/mp4boxdemo
35 https://smashed.by/avifjs
36 https://smashed.by/avifjsdemo

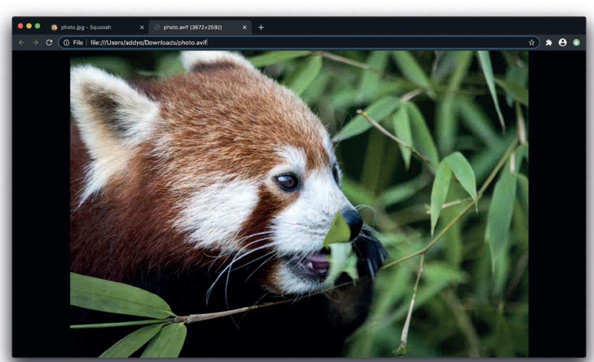
- The latest version of the FFmpeg[37] can be used in combination with libaom to convert other image file types using AV1 compression and stored in an MKV container. The following is the snippet of code used to generate the AVIF image shown earlier:

```
$ ffmpeg -loglevel panic -i 400px-President_
Barack_Obama.png -c:v libaom-av1 -crf 41 -b:v 0
-strict experimental -vf format=yuv420p 400px-
President_Barack_Obama.av1.mkv
$ ffmpeg -loglevel panic -i 400px-President_
Barack_Obama.av1.mkv 400px-President_Barack_Obama.
av1.mkv.png
```

### VIEWERS AND EDITORS

Browsers, like Chrome, that support AVIF allow you to open *.avif* files on supported platforms. Here, I've opened up the AVIF output of Squoosh without the need for a separate viewer.





*Viewing an AVIF in Chrome.*

37 https://smashed.by/ffmep

In time, as support for AVIF improves, my hope is that we'll start to see better native support outside of the browser in macOS's Finder and QuickLook, as well as in Windows File Explorer. Windows File Explorer in Windows 10 (build 18305 or above) can edit metadata fields from AVIF files and rotate these files. The Paint app in the same build can also be used to view and edit AVIF files. AVIF files can be viewed in VLC media player or Windows Media Player in Windows 10 (build 18317 and above).

The sentiment around AVIF is relatively positive at this point in time, and we will need to wait and see just how much industry adoption it gets over the long term. The main concerns right now seem to be to get sufficient tooling ready to easily start incorporating the AVIF format at every stage in the life cycle of an image. For AVIF to become the next JPEG, tooling would be required at the following stages:

1. Ability to save to AVIF in all sources where images are generated: cameras and other photography equipment, including DSLRs and point-and-shoot cameras on mobile devices.

2. Ability to save as AVIF in tools used for generating images for infographics, flowcharts, icons, and similar.

3. Ability to save screenshots as AVIF images on all major operating systems on desktop and mobile devices alike. Windows, macOS, Linux, iOS, and Android should support AVIF images at the minimum.

4. Support in commonly used image editors (Photoshop et al.) and the default image editors provided by different OS vendors.

5. Ability to export to AVIF in image conversion software.

6. Support in HTML `<img src>` to embed AVIF source files.

7. Browser support across all commonly used browsers to understand AVIF requests and decode and render the corresponding images.

Work has already begun in most of the areas above, and with time, tools are expected to be widely available. While the future of AVIF looks promising at this early stage, much depends on the availability of better tooling from the major vendors of operating systems, browsers, and other image processing software. Apple has not yet made any announcements on how and when it will start supporting AVIF images, and the web community seems to be waiting in anticipation for some indication from Apple.

While performance analysts have accepted the higher compression rate and corresponding data savings that could be gained by using AVIF, they hope for better performance when encoding and decoding the images in the future. In the long run, we may see AVIF used for many kinds of images on the web, with JPEG XL (see chapter 19) used for large images that need progressive rendering support or to be fully lossless. While AVIF is still rolling out to browsers in 2021, it's worth keeping an eye on it as tooling, browser, and image CDN support for AVIF improves.

CHAPTER 19

# JPEG XL

*With thanks to Jon Sneyers, co-chair of the JPEG XL ad-hoc group for his input to this chapter.*

JPEG XL[38] is an advanced image format aiming to deliver very significant compression benefits over JPEG. Looking specifically at compression performance, JPEG XL's lossless JPEG transcoding reduces JPEG file sizes by 16–22%.[39] Starting from pixels, JPEG XL is visually lossless at approximately half the bit rate required by JPEG. For lossy encoding, JPEG XL is up to 60% smaller than JPEG[40] for the same visual fidelity.[41]

The JPEG XL codec comprises a broad range of up-to-date features that include optimization for responsive web environments, and a number of decoding methods (parallel, progressive, and partial). Layers, thumbnails, and animation are also supported, with different options available for blending frames.



JPEG XL was designed with three primary criteria in mind:

- Output must be of high-fidelity compared to the source image.
- Encoding and decoding speed must be competitive.

38 https://smashed.by/jpegxl
39 https://smashed.by/whitepaper
40 https://smashed.by/visualfidelity
41 https://smashed.by/jxl

- Compression ratios are high (generally 20:1 to 50:1).

Building on this set of criteria, JPEG XL is designed to be a universal and future-proof image codec whose features also include:

- Lossless JPEG transcoding.
- Parallelization in the encoder and decoder.
- Support for additional channels like alpha, depth, spot colors, and potentially other spectrums used in scientific research.
- High bit depth, wide gamut, and HDR.
- Progressive decoding (for both image resolution and precision). For low-quality placeholders (LQIP), a discrete cosine (DC) component representing the average (arithmetic mean) of the entire image can be encoded as a subimage which itself can be progressively encoded.
- Region of interest decoding
- Support for any kind of image content across photography and synthetic imagery. This includes: photographs, illustrations, screenshots, rendered images, document scans, medical imaging, game graphics, and UI elements.

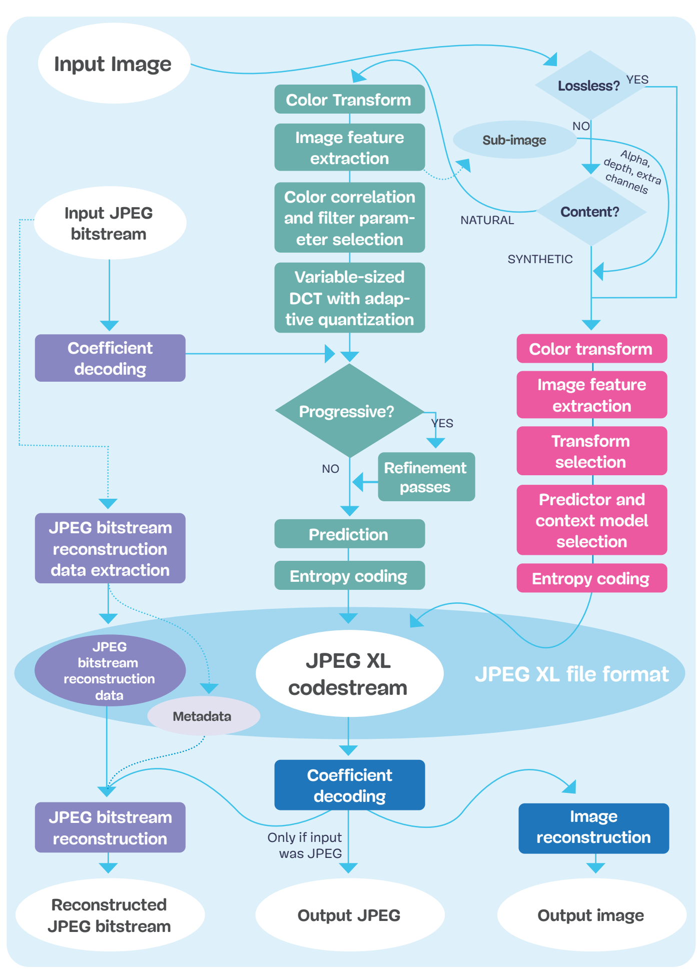
- Coverage of the quality spectrum from very low bit rates all the way to lossless. JPEG XL has smoother quality degradation across a wide range of bit rates.
- Support for optimizing screen content. Images with non-photographic or repetitive elements such as font glyphs can be encoded separately as a "sprite sheet" subimage that can be applied as patches to the main image. This is useful for screenshots.
- Various trade-offs between encoding and decoding speed and compression density.

## History

JPEG XL has been free, royalty-free, and open-source software from its inception. In early 2018 the JPEG committee declared its intention to establish a new image standard that would combine considerably more efficient compression with a wide variety of online use cases. JPEG XL inherits some of the best properties of efforts to offer new image formats over the years, such as strong focus on preserving detail and texture (from Google's PIK),[42] being responsive by default (from Cloudinary's free universal image format (FUIF)),[43] and from both of these efforts has legacy-friendliness as a strong cornerstone. This makes it smooth to transition from existing file formats, like JPEG and PNG, to JPEG XL.



42 https://smashed.by/pik
43 https://smashed.by/fuif




*An overview of the JPEG XL encoder architecture. The three supported modes are: lossless transcoding of JPEG input (purple); lossy encoding of photographs with an emphasis on human visual perception (green); and mathematically lossless encoding (pink). (Source: JPEG XL Whitepaper)*[44]



44 https://smashed.by/whitepaper

JPEG XL is built on years of research into the formats that came before it. One of them is FLIF (free lossless image format), a format created by Jon Sneyers in 2015 that eventually evolved into FUIF. FLIF supports lossless as well as progressive image rendering. Cloudinary supports dynamic conversion of any image format to FLIF, such that FLIF could be served to a browser that supports it, and PNG or WebP to browsers that don't.

JPEG XL aims to provide significantly smaller image file sizes at subjectively equivalent quality, and reversible encoding of existing JPEG files. It has yet to have broad browser support, but once this improves it should be a great option for most photo (lossy/lossless) or non-photo (lossless) use cases.



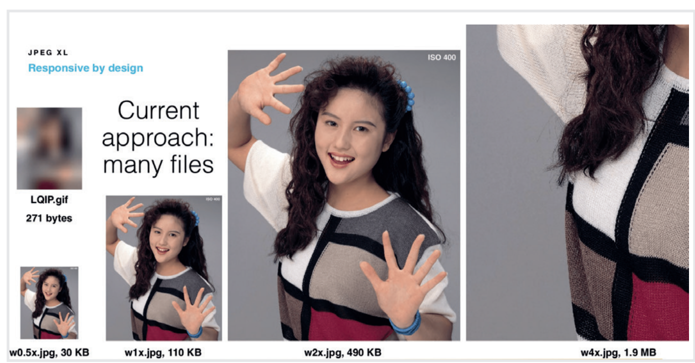
## Responsive Design

JPEG XL was conceived with responsive web design in mind. Because people use many different kinds of devices to visit websites, responsive images and web design adapt

users' experiences to best fit their screens whether phone, tablet, or desktop. But outputting and serving responsive images remains cumbersome.

An image format designed with responsivity in mind could allow it to handle image quality and decoding speeds more efficiently. This enables JPEG XL to render imagery well on a wide range of devices. Its "modular" mode allows JPEG XL to recover subresolution images using a Haar-like integral transform.

It can also compress images more effectively through a meta-adaptive context model which evolved from FLIF. This model uses a decision tree known in advance by the encoder and which can be heavily tuned to adapt based on the content of the image. The "varDCT" mode has several features aimed at responsive delivery: a 1:8 preview is always available quickly, and further progressive scans for 1:4 and 1:2 can be added; scans do not have to go from top to bottom (they can, for example, also go "middle out"), and "saliency scans" are possible (first scanning details in a face, for example, and later in the background); and the 1:8 preview can recursively be encoded progressively. It can take existing JPEG images and encode them in a more progressive way using the varDCT mode (covered later in this chapter in the "Transcoding Legacy JPEGs" section).




*For responsive web design a set of individual images has to be generated.*

Above is a visualization of the current approach to responsive images from Jon Sneyers' excellent JPEG XL talk,[45] where a series of images at different resolutions must be generated and served to devices of different screen sizes. Below is how JPEG XL handles responsive images, where a single file can offer multiple subresolutions efficiently.

*JPEG XL can serve the same image file responsively.*

JPEG XL

45 https://smashed.by/jpegxltalk

## Transcoding Legacy JPEGs

JPEG XL has strong support for features to help users transition from legacy JPEGs. Existing JPEGs can be losslessly transcoded to JPEG XL, while potentially reducing their size significantly. It has a light, lossless conversion process for going back to JPEG, ensuring compatibility with existing clients, such as older phones and browsers that might only support JPEGs. This unlocks a nice serving story as servers can store a single JPEG XL file that will serve both JPEG *and* JPEG XL users.

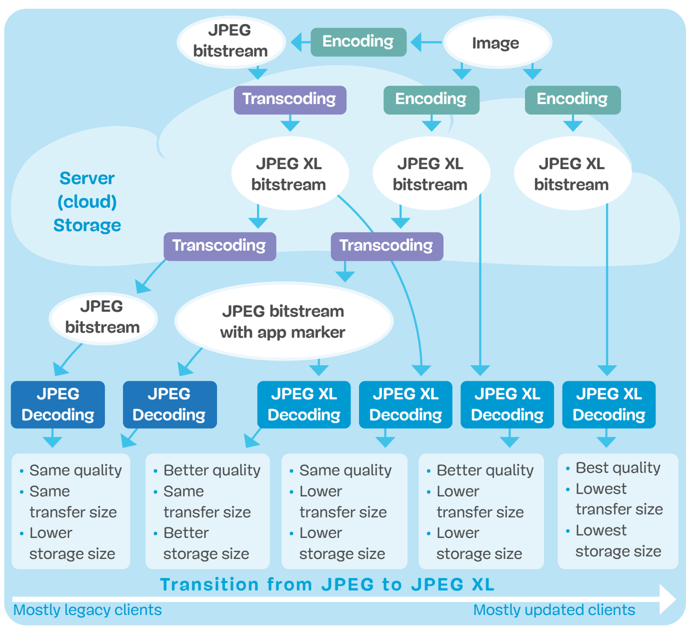
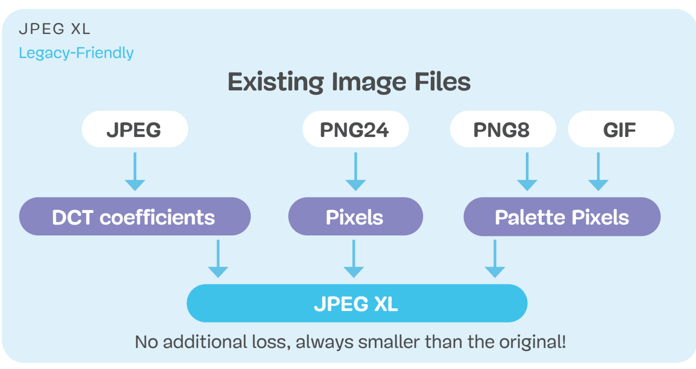



*Diagram visualizing the cloud serving the transition from JPEG to JPEG XL. (adapted from a diagram by Jon Sneyers)*

The JPEG XL recompression format attempts to remedy many of JPEG's limitations. This mode is inspired by Brunsli,[46] Google's lossless JPEG repacking library that allows for up to a 22% decrease in file size while allowing original JPEG source images to be recovered byte-for-byte. JPEG XL also allows for parallel and efficient cropped decode and additional compression of ICC color profiles.

Additional benefits include the ability to add an alpha channel or overlays to existing JPEGs. JPEG XL supports DCT with variable block sizes (2×2 to 256×256) and adaptive quantization (for JPEG recompression, all the blocks just happen to be 8×8 and the quantization is constant throughout the image). This mode uses a Butteraugli-driven,[47] perceptually optimizing encoder. (We covered Butteraugli – a tool for measuring perceived differences between images – in chapter 2.)




*JPEG XL's transcoding avoids pixels to remain lossless while reducing file sizes.*

JPEG XL

46 https://smashed.by/brunsli
47 https://smashed.by/butteraugligit

Typically, when you transcode images in an old format with a new encoder, it introduces generational loss. You begin with something lossy and then you make it more lossy leading to the accumulation of artifacts. JPEG XL directly[48] encodes JPEGs without going into the pixel domain by encoding DCT coefficients (that is, an exact representation of what the JPEG contains). This is lossless as you aren't introducing additional loss to the process and can still produce a smaller image than the original.

## High-Quality Imaging

JPEG XL aims to meet the requirements of professional photographers that demand high-quality imaging. It has a color-managed processing pipeline with full 32-bit per channel precision, enabling support for wide color gamut or high dynamic range imagery. Psychovisual modelling plug-ins allow it to reach high compression efficiency at visually lossless quality. The use cases for JPEG XL range from photo galleries to cloud-stored images, 360-degree images, image bursts, and more.

The authors of JPEG XL feel other image codec efforts have failed to deliver on psychovisual performance for a few reasons. For example, they focus on trying to offer the best performance at low bit-rates, yet don't extend their

48 https://smashed.by/directencoding

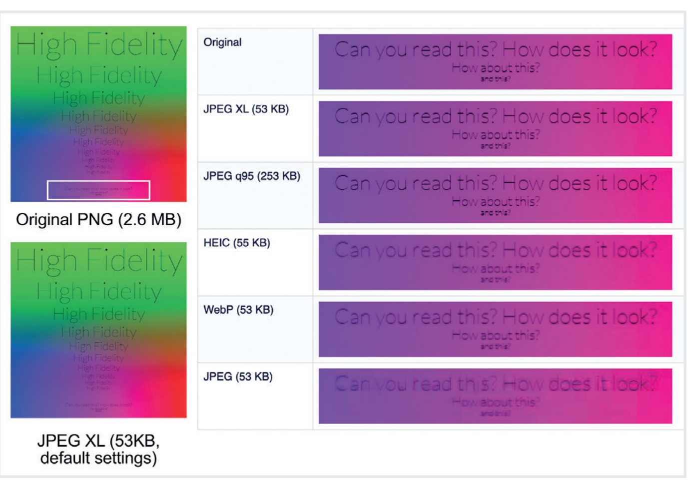
approach to higher bit-rates without a loss of efficiency. This can be seen in image codecs which have been derived from successful video-encoding research. Video codecs tend to focus on low bit rates as they have a lot of pixels to encode. You typically only see a single frame for between 16 and 42 milliseconds. Video frames have a lot of smoothing and distilling, which is great for video but not so great for high-quality still images.

In order to give JPEG XL a strong chance of getting this right, the community behind it collected and analyzed real-world usage data on JPEGs. Through independently run experiments, they studied how this could map well to image quality and bit-rate settings in JPEG XL. One of the outcomes of this focus has been targeting higher image bit-rates than other approaches.

JPEG XL has several other benefits. It offers a true lossless mechanism that is more efficient than PNG or JPEG 2000. This is useful for illustrations, and archival and other scientific or forensic applications. JPEG XL also supports many more channels, which will enable new color spaces to emerge that might not be based on YCbCr or CIELAB or CIE XYZ (all were attempts to match the color-receptor cells in the human eye). This gives JPEG XL the opportunity to be future-friendly. It also means that scientific research that needs to capture the electromagnetic spectrum outside of the visual spectrum can use this format to create images.

## ADAPTIVE QUANTIZATION

The JPEG format we currently use applies the same quality in every region. It limits the choice of quantization to a single matrix per channel for the entire image. This results in the same quantity of quantization in all regions of an image, even though certain places are more detailed and might benefit from improved quantization. In JPEG XL, quality can be different in different regions: it's adjusted automatically based on perceptual metrics.




*By default, the JPEG XL reference encoder produces a well-compressed image that often can't be distinguished from the original. Notice how JPEG XL preserves text quality better than the larger JPEG at quality=95, which includes noise around the letters. At a similar rate of compression, WebP HEIC, and JPEG all look less sharp than the JPEG XL. (Source: "How JPEG XL Compares to Other Image Codecs")*[49]



49 https://smashed.by/universality

JPEG XL has some tricks that allow it to lower the number of artifacts found in more complex parts of an image. It can achieve this without modifying the number of bits used in other areas of the image. JPEG XL achieves this through a global quantization matrix which can be locally scaled; encoders can avoid big variations in image artifacts by aiming for a more uniform quantity of loss across the image. That is, if you combine this with a measure of acceptable loss.

### GROUP SPLITTING

You may have an image with a large dimension (such as larger than 256px in any axis). When you encode such large images with JPEG XL, they get split into subrectangles of 256×256 pixels. Each subrectangle is independently encoded and the bitstream position indices of them are stored. This allows decoders to seek to the beginning of each of them. Decoders can process each subrectangle in parallel (for speed) and decode specific areas of large images. If something goes wrong

TRIVIA ■ JPEG XT defines extensions to the 1992 JPEG specification. For extensions to have pixel-perfect rendering on top of original JPEG, the specification had to clarify the old 1992 spec and libjpeg-turbo was chosen as its reference implementation (based on popularity).

(perhaps if a prior rectangle fails because of data corruption), decoding can also be restarted.

### ENTROPY CODING

When encoding images with legacy JPEG, developers have the option to use Huffman coding or arithmetic coding. Both options are used in lossless data compression as an entropy coder. There are, however, some gotchas to look out for when using either coding approach as they can be both inefficient and lead to slow image decoding speeds. JPEG XL can be up to 30 times faster through usage of asymmetric numeral systems[50] (ANS), a pretty recent entropy coder that can help achieve image compression ratios comparable to arithmetic coding with the benefit of being faster to decode.

### ADAPTIVE PREDICTOR

Next, let's discuss compression ratios. In legacy JPEG, you may enjoy significant savings from a primitive prediction mode. This mode subtracts the DC coefficient value for the previously encoded block (the one to the left of the current block) from the DC coefficient of the current block. Unfortunately, this doesn't factor in how bi-dimensional images can be.



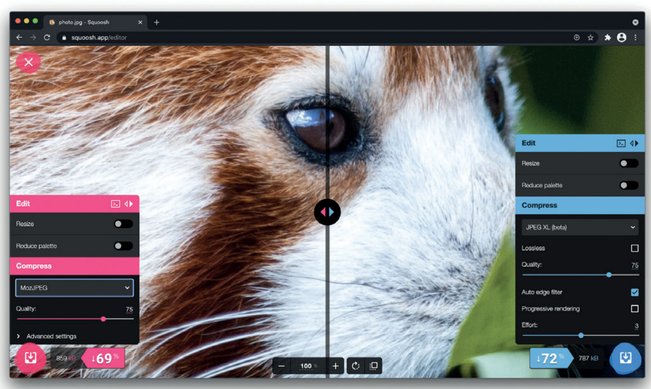
JPEG XL tackles this by using a more advanced predictor that is both bi-dimensional and adaptive, selecting any of

50 https://smashed.by/ans

fourteen different prediction modes where the predictor mode can depend on the meta-adaptive context. Both DC encoding and lossless encoding gets done using this predictor. JPEG XL is visually lossless at about half the bit rate required by JPEG.[51]

## Tools

Popular online image compression tool Squoosh[52] supports JPEG XL with support for customizing lossy/lossless, automatic edge-filter and progressive rendering features:




*Squoosh allows you to compress images using several new modern image formats including JPEG XL and AVIF.*

51 https://smashed.by/whitepaper
52 https://squoosh.app

I'm excited by the goals JPEG XL is striving for. I like that it has a good interoperability with legacy JPEG, shrinking the size of existing images without a perceivable loss of quality and creating new images that could be transcoded back. This gradual upgrade path benefits both image hosts and users.

While a number of new image formats are being explored, JPEG XL looks promising as a direction that could reduce user-experienced latency and simplify image serving. I'd certainly recommend keeping an eye on it.

To learn more about JPEG XL, check out Jon Sneyers' talk,[53] his JPEG XL articles[54] and the official project site.[55]

Similar to AVIF and WebP, support for JPEG XL can be added to a page as a progressive enhancement using the `<picture>` element. Browsers that don't yet support it will select the first format in the list they support, be it AVIF, WebP or otherwise:

```
<picture>
    <source srcset="photo.jxl" type="image/jxl">
    <source srcset="photo.avif" type="image/avif">
```

53 https://smashed.by/jpegxltalk
54 https://smashed.by/universality
55 https://smashed.by/jpegxl

```
    <source srcset="photo.webp" type="image/webp">
    <source srcset="photo.jpg" type="image/jpeg">
    <img src="photo.jpg" alt="Description">
</picture>
```

## JPEG XL Changes Image Compression Recommendations

Once JPEG XL ships in a browser, it should be a great option for most photo (lossy/lossless) or non-photo (lossless) use cases – probably better than WebP. Legacy friendliness is a plus.

AVIF (see chapter 18) may be better than JPEG XL if you need to go to low bit rates, such as caring more about bandwidth than image fidelity. At those bit rates I can imagine it looking crisper than JPEG XL.

Until we have JPEG XL, I'd focus more on AVIF or WebP. WebP is lossless and a good option for lossy compared to MozJPEG, while also being strong for non-photo images. If analytics suggest most users can be served AVIF (great for most use cases) or WebP (and you care less about wide gamut or text overlays, where chroma subsampling workarounds are required), these are good contenders. If not, a MozJPEG/OxiPNG fallback seems reasonable.

CHAPTER 20

# Comparing New Image File Formats

While the new image formats support roughly the same sets of capabilities, the strength of each differentiates between them. The following tables aim to offer a summary of some of the more important features and how well each format handles various image types.

## High-Level Comparison

| FEATURES | HEIC | AVIF | JPEG XL |
|---|---|---|---|
| Lossless compression | Approximate | Approximate | Yes |
| Bursts and sequences | Yes | Yes | Yes (intra-only) |
| Stickers and overlays | Yes | Yes | Yes |
| Maximum bit depth | 16-bit | 12-bit | 24-bit |
| Maximum number of channels | 3 (alpha or depth as separate image) | 3 (alpha or depth as separate image) | up to 4,100 |
| Maximum image dimensions (in a single code stream) | 35 MP (8,192 × 4,320) | 9 MP (3,840 × 2,160) | 1.1tn MP (1,073,741,823 × 1,073,741,824) 1,152,921 terapixels |

The grading system below (0–3 stars) reflects opinions from book contributors based on first-hand experience. As always, I encourage readers to also test and determine their own opinions on grading as well.

### PHOTOGRAPHS (CREATED FROM PHOTO SENSORS)

| | HEIC | AVIF | JPEG XL |
|---|---|---|---|
| Simple photographs | ★★★ | ★★★ | ★★★ |
| Photos with text | ★ | ★★ | ★★★ |
| Isolated subject with matte background (typical for product shots) | ★★★ | ★★★ | ★★★ |
| Medical and scientific photographs (such as X-rays) | ✖ | ✖ | ★★★ |

COMPARE

## GRAPHICS (COMPUTER-GENERATED)

| | HEIC | AVIF | JPEG XL |
|---|---|---|---|
| Illustrations, diagrams, maps and charts | ★★ | ★★ | ★★★ |
| Logos and icons | ★ | ★ | ★★ |
| Cartoons and hand-drawn illustrations | ★ | ★★ | ★★★ |
| Background images | ★★ | ★★ | ★★★ |
| Screenshots | ★ | ★ | ★★★ |
| Prints | ✖ | ✖ | ★★★ |

## ANIMATIONS

| | HEIC | AVIF | JPEG XL |
|---|---|---|---|
| User-generated memes | ★★★ | ★★★ | ★ |
| Cinemagraphs and live photos | ★★★ | ★★★ | ✖ |
| Animated educational illustrations | ★★ | ★★ | ★★★ |

## Lower-Level Comparison

Next is a more expansive comparison of image codec and format qualities using a 0–5 star grading system. This lower-level comparison is particularly helpful if you have a deeper knowledge of image compression and would like to compare a broader set of trade-offs between codecs. Special thanks to Cloudinary's Jon Sneyers for sharing his insights.[56]

### COMPRESSION (PHOTOGRAPHIC IMAGES)

✔ supported ✖ not supported ★ okay ★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| Overall | ★★ | ✖ | ★★ | ★★★ | ★★★ | ★★★★ | ★★★★ | ★★★★ | Reduce bandwidth and storage for digital photos. |
| Low fidelity | ★ | ✖ | ★★ | ★★★ | ★★ | ★★★★ | ★★★★ | ★★★ | On really slow networks, low-quality but very small image files might be useful. |
| Medium fidelity | ★★ | ✖ | ★★ | ★★★ | ★★★ | ★★★★ | ★★★ | ★★★★ | On typical networks, a good trade-off between fidelity and density is useful. |
| High fidelity | ★★ | ★ | ★ | ★ | ★★★ | ★★★ | ★★ | ★★★★ | On sufficiently fast networks, high fidelity is desired. |
| Lossless | ✖ | ★ | ★★ | ★★★ | ★★★ | ★★ | ★★ | ★★★★ | In image editing workflows, lossless is needed. |

COMPARE

COMPARE

56 https://smashed.by/responsivebydesign

## COMPRESSION (NON-PHOTOGRAPHIC IMAGES)

✔ supported ✖ not supported ★ okay ★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★ | ★★ | ★★★ | ★★★ | ★ | ★★★ | ★★ | ★★★★ | Reduce bandwidth and storage for other types of digital images. |
| **Lossy** (non-photographic) | ★ | ★★ | ★★★ | ★★★ | ★ | ★★★★ | ★★ | ★★★★ | For web delivery of non-photographic images, lossy compression is good. |
| **Lossless** (non-photographic) | ✖ | ★★★ | ★★★★ | ★★★★ | ★ | ★ | ★ | ★★★★ | In image editing workflows, lossless is needed. |
| **Mixed photographic and non-photographic** | ★ | ★ | ★★ | ★★ | ★ | ★★★ | ★★ | ★★★★ | Memes, promotion images, etc. are a combination of photo and non-photo. |

## SPEED

✔ supported ✖ not supported ★ okay ★★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★★★★ | ★★★ | ★★ | ★★ | ★★ | ★ | ★★ | ★★★★ | If it's too slow, it becomes less practical. |
| **Single-core encode speed** | ★★★★ | ★★ | ★★★ | ★★★ | ★★★ | ★ | ★★ | ★★★★ | Compression speed matters for editing workflows and latency of on-the-fly encoding. |
| **Single-core decode speed** | ★★★ ★★ | ★★★ ★★ | ★★★ | ★★★★ | ★★★ | ★★ | ★★ | ★★★★ | If it decodes slowly, even if bandwidth is reduced by superior compression, the overall time to render suffers. |
| **Parallelizable** | ★ | ★ | ★ | ★★★ | ★★★ | ★★★ | ★★★ | ★★★★ | To what extent can encoders/decoders benefit from multiple CPU cores to speed things up? |



COMPARE

## LIMITS

✔ supported ✖ not supported ★ okay ★★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★★ | ★★★ | ★ | ★★ | ★★★★★ | ★★★ | ★★ | ★★★★★ | What are the limitations of the codec/format? |
| **Supports animation** | ✖ (✔ MJPEG) | ✔ (APNG) | ✔ | ✔ | ✖ (✔ MJP2) | ✔ | ✔ | ✔ | Not very relevant for most use cases – usually a video codec is a better choice. |
| **Maximum image dimensions** | 65,535 × 65,535 | 2,147,483,647 × 2,147,483,647 | 16,383 × 16,383 | 16,383 × 16,383 | 4,294,967,295 × 4,294,967,295 | 8,193 × 4,320 (or grid with potential boundary artifacts) | 8,193 × 4,320 (or grid with potential boundary artifacts) | 1,073,741,823 × 1,073,741,823 | What's the largest width and height in pixels the codec supports? |
| **Efficient cropped decode** | ✖ | ✖ | ✖ | ✖ | ✔ | ✖ | ✖ | ✔ | For huge images (e.g. gigapixels), can a crop (region of interest) be decoded without having to decode all or most of the compressed image? |
| **Image pyramids** | ✖ | ✖ | ✖ | ✖ | ✔ | ✖ | ✖ | ✔ | For huge images, can a lower-resolution version of the image be decoded without having to decode the entire image? |
| **Precision (max bit depth)** | 8 | 16 | 7.5 (8 for lossless) | 10 | 38 | 10 | 10 | 32 | For SDR image delivery, 8 bits are enough. For HDR and/or (very) wide gamut image delivery, 10 or 12 bits are needed. For editing workflows, more precision is needed. |
| **Can do (lossy) 4:4:4** | ✔ | ✔ | ✖ | ✔ | ✔ | ✔ | ✖ | ✔ | Obligatory chroma subsampling puts a limit on high-fidelity encoding, since it makes compression artifacts unavoidable on some kinds of images (e.g. colored text gets blurred). |
| **Can do 4:2:0 chroma subsampling** | ✔ | ✖ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | |
| **Can do 4:2:2 chroma subsampling** | ✔ | ✖ | ✖ | ✖ | ✔ | ✖ | ✖ | ✔ | |
| **Wide gamut** | ✔ | ✔ | not really | ✔ | ✔ | ✔ | ✔ | ✔ | Can the codec accurately reproduce wide gamut color (e.g. P3, ProPhoto or Rec.2020)? |
| **HDR** | ✖ | ✔ | ✖ | ✔ | ✔ | ✔ | ✔ | ✔ | High dynamic range is already common on television screens; is the codec ready for this? |
| **Supports JUMBF extensions** | ✔ | ✖ | ✖ | ✖ | ✔ | ✖ ? | ✖ ? | ✔ | The JPEG universal metadata box format (JUMBF) is a generic extension mechanism for features like 360 images, privacy, and security metadata, etc. |

COMPARE

COMPARE

## DELIVERY FEATURES

✔ supported ✘ not supported ★ okay ★★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★★★ | ★★★ | ★ | ★★ | ★★★★ | ★ | ★ | ★★★★★ | How suitable is this codec for (web) delivery of images? |
| **Progressive decoding** | ★★★ | ★ | ✘ | ✘ | ★★★★ | ✘ | ✘ | ★★★★ | Progressive decoding allows getting a reasonable preview of the image (which gets gradually refined) when only part of the image data has been transferred |
| **Progressive with alpha** | ✘ | ✔ | ✘ | ✘ | ✔ | ✘ | ✘ | ✔ | Do the progressive previews include the alpha channel? |
| **Separate/ redundant preview image** | ✔ (via EXIF) | ✔ (via EXIF) | ✘ | ✔ | ✘ | ✔ | ✔ | ✔ | Can you embed a (redundant) preview image in the format? |
| **Preview image "for free"** | ✔ | ✘ | ✘ | ✘ | ✔ | ✘ | ✘ | ✔ | Can a decoder produce a good preview image (e.g. a 1:8 resolution downscale) from the first ~10% of the image data? |
| **LQIP "for free"** | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✔ | Can a decoder produce a low-quality image placeholder (e.g. a 1:64 down-scale) from the first 1-2 kilobytes of image data? |
| **"Responsive by design"** | ✘ | ✘ | ✘ | ✘ | ✔ | ✘ | ✘ | ✔ | Can an image file be truncated to get a 1:2 or 1:4 version of the image? Can the truncation offset be derived from the image header? |
| **Low format overhead** | ✘ | ✔ | ✔ | ✔ | ✘ | ✘ | ✘ | ✔ | For small images, the overhead of headers and other obligatory bitstream elements can be significant, e.g. several hundred bytes. |
| **Perceptual encoder** | ★★ | n/a | ★ | ? | ★ | ★ | ★ | ★★★★★ | Does an encoder exist that encodes based on a perceptual target (e.g. multiples of just-noticeable-difference) instead of based on technical parameters (e.g. amount of quantization) |
| **Compressed ICC profile** | ✘ | ★★ | ✘ | ✘ | ★ | ★ | ★ | ★★★ | ICC profiles are used to represent the color space of an image. Uncompressed ICC profiles add unnecessary overhead. |



COMPARE

## NON-RGB COMPONENTS

✓ supported ✗ not supported ★ okay ★★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★ | ★★ | ★★ | ★★ | ★★★ | ★★★ | ★★★ | ★★★★★ | Support for extra channels besides the RGB color image itself. |
| **Maximum number of channels** | 4 | 4 | 4 | 4 | 16,384 | 5 | 5 | 4,099 | e.g. satellite imagery can use 12 channels. |
| **Alpha transparency** | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | Useful for blending (overlaying) an image over a background, or to represent non-rectangular images. |
| **Depth map** | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | Contains information about the distance from the camera (useful to separate foreground and background). |
| **Thermal map** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | Infrared cameras (already deployed in some phones) can create images representing estimated temperature. |
| **CMYK** | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | Commonly used in the printing industry. |
| **Spot colors** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | Used in the printing industry. |

COMPARE
COMPARE

## AUTHORING FEATURES

✓ supported ✗ not supported ★ okay ★★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★ | ★ | ★ | ★ | ★ | ★★ | ★★ | ★★★★★ | How suitable is this codec in an authoring/editing workflow? |
| **Overlays** | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | Can an image consist of multiple layers that are overlaid (like in GIMP or Photoshop)? |
| **Named layers** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | Can the name of the layers be stored? |
| **Selection masks/multiple alpha** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | Can selection masks (or additional alpha channels) be stored? |
| **Lossless floating point** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | Can the codec encode lossless floating point, like e.g. OpenEXR, TIFF and PSD? |
| **Fast weakly compressed lossless** | ✗ | ✓ | ✓ (within limits) | ✓ (within limits) | ✗ | ✗ | ✗ | ✓ | Is there a way to very quickly save an image (with weaker compression)? |
| **Compressed EXIF/XMP metadata** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | Does the format support compressed metadata? |
| **Generation loss resilience** | ★★★ | n/a | ★ | ★ | ★★ | ★★ | ★★ | ★★★ | How resilient is the codec to degradation due to repeated saving of an image (e.g. memes)? |

COMPARE

COMPARE

## TRANSITIONAL FEATURES

✓ supported ✗ not supported ★ okay ★★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★ | ★ | ★★ | ★★ | ★ | ★★ | ★★ | ★★★★★ | Are there any features aimed at easing the transition from the existing codecs to the new codec? |
| **Lossless JPEG recompression** | n/a | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | Existing JPEG files can be transcoded (to a smaller file) without introducing additional loss |
| **Can it replace PNG?** | ✗ | n/a | ✓ (unless 16-bit) | ✓ (unless 16-bit) | ✗ | not really | not really | ✓ | Existing PNG files can be converted to a smaller file |
| **Can it replace GIF?** | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | Existing GIF files can be converted to a smaller file |

## ROYALTY-FREE

✓ supported ✗ not supported

| JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | Is the codec patent-encumbered (royalties need to be paid to use the codec) or not? |

COMPARE

COMPARE

## SOFTWARE SUPPORT

✓ supported ✗ not supported ★ okay ★★★★★ excellent

| | JPEG | PNG | WEBP | WEBP 2 | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL | WHAT IS THIS? WHY CARE ABOUT IT? |
|---|---|---|---|---|---|---|---|---|---|
| **Overall** | ★★★★★ | ★★★★★ | ★★★ | ✗ | ★★ | ★★ | ★ | ✗ | Current codec adoption or availability of software support. |
| **Desktop Chrome/ Opera** | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | 73% of desktop browser market share. |
| **Mobile Chrome/ Opera** | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | 65% of mobile browser market share. |
| **Safari** | ✓ | ✓ | ✓ (MacOS 11, ±) | ✗ | ✓ | ✗ | ✓ | ✗ | 17% of total browser market share (24% of mobile). |
| **Firefox** | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ (behind flag) | ✓ | ✗ | 8% of desktop browser market share |
| **Edge** | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | 6% of desktop browser market share. |
| **Internet Explorer** | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | 2% of desktop browser market share. |
| **Samsung Internet/ UC** | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | 9% of mobile browser market share. |
| **Android** | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ (planned) | 73% of mobile apps market share. |
| **iOS** | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | 27% of mobile apps market share. |
| **Image-Magick** | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ (will happen) | Popular cross-platform FOSS batch image processing/conversion library/ utilities. |
| **GIMP** | ✓ | ✓ | ✓ | ✗ | ✓ (read only) | ✗ | ✓ | ✗ (planned) | GNU Image Manipulation Program, popular FOSS image editor. |
| **Adobe Photoshop** | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ (Mac only) | ✗ | Popular proprietary image editor. |
| **Apple Preview** | ✓ | ✓ | ✓ (MacOS 11) | ✗ | ✓ | ✗ | ✓ | ✗ | Built-in image viewer of macOS. |
| **Good FOSS encoder and decoder** | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | Does a free and open source encoder/ decoder exist that produces state-of-the-art compression results? |

COMPARE

COMPARE

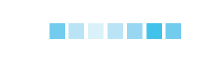
## SUMMARY

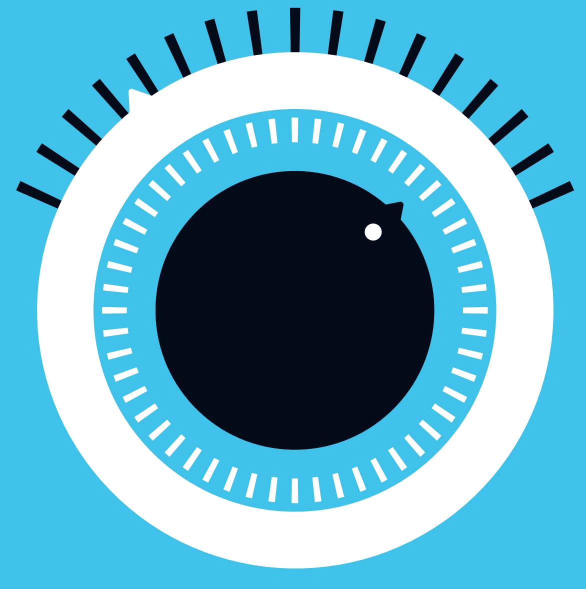
✓ supported ✗ not supported ★ okay ★★★★ excellent

| | JPEG | PNG | WEBP | JPEG 2000 (KAKADU) | AVIF | HEIC | JPEG XL |
|---|---|---|---|---|---|---|---|
| **Compression** (photo) | ★★☆☆ | ☆☆☆☆ | ★★☆☆ | ★★★☆ | ★★★★ | ★★★★ | ★★★★ |
| Low fidelity | ★☆☆☆ | ☆☆☆☆ | ★★☆☆ | ★★☆☆ | ★★★★ | ★★★★ | ★★★☆ |
| Medium fidelity | ★★☆☆ | ☆☆☆☆ | ★★☆☆ | ★★★☆ | ★★★★ | ★★★☆ | ★★★★ |
| High fidelity | ★★☆☆ | ★☆☆☆ | ☆☆☆☆ | ★★★☆ | ★★★☆ | ★★☆☆ | ★★★★ |
| Lossless | ☆☆☆☆ | ★☆☆☆ | ★★☆☆ | ★★★☆ | ★★☆☆ | ★★☆☆ | ★★★★ |
| **Compression** (other images) | ★☆☆☆ | ★★☆☆ | ☆☆☆☆ | ★☆☆☆ | ★★★☆ | ★★☆☆ | ★★★★ |
| Lossy non-photographic | ★☆☆☆ | ★★☆☆ | ★★★☆ | ★☆☆☆ | ★★★★ | ★★☆☆ | ★★★★ |
| Lossless non-photographic | ☆☆☆☆ | ★★★☆ | ★★★★ | ★☆☆☆ | ★★☆☆ | ★☆☆☆ | ★★★★ |
| Mixed photo / nonphoto | ★☆☆☆ | ★☆☆☆ | ★★☆☆ | ★☆☆☆ | ★★★★ | ★★☆☆ | ★★★★ |
| **Speed** | ★★★★ | ★★★☆ | ☆☆☆☆ | ★★☆☆ | ★★☆☆ | ★★☆☆ | ★★★★ |
| Single-core encode speed | ★★★★ | ★★☆☆ | ★★★☆ | ★★★☆ | ★☆☆☆ | ★★☆☆ | ★★★★ |
| Single-core decode speed | ★★★★ | ★★★★ | ★★★★ | ★★★☆ | ★★☆☆ | ★★☆☆ | ★★★★ |
| Parallelizable | ★☆☆☆ | ★☆☆☆ | ★☆☆☆ | ★★★☆ | ★★★☆ | ★★★☆ | ★★★★ |
| **Limits** | ★★☆☆ | ★★★☆ | ★☆☆☆ | ★★★★ | ★★★☆ | ★★★☆ | ★★★★ |
| Maximum image dimensions | 65,535 × 65,535 | 2,147,483,647 × 2,147,483,647 | 16,383 × 16,383 | 4,294,967,295 × 4,294,967,295 | 8,193 × 4,320 | 8,193 × 4,320 | 1,073,741,823 × 1,073,741,823 |
| Precision (max bit depth) | 8 | 16 | 8 | 38 | 10 | 10 | 32 |
| Can do (lossy) 4:4:4 | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ |
| Wide gamut / HDR | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Maximum number of channels | 4 | 4 | 4 | 16,384 | 5 | 5 | 4,099 |
| **Features** | ★☆☆☆ | ★★☆☆ | ★☆☆☆ | ★★★☆ | ★★★☆ | ★★★☆ | ★★★★ |
| Supports animation | ✗ (✓ MJPEG) | ✓ (APNG) | ✓ | ✗ (✓ MJP2) | ✓ | ✓ | ✓ |
| Progressive decoding | ★★★☆ | ★☆☆☆ | ✗ | ★★★★ | ✗ | ✗ | ★★★★ |
| Alpha transparency | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Depth map | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Overlays | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Authoring workflow suitability | ★☆☆☆ | ★★☆☆ | ★☆☆☆ | ★★☆☆ | ★☆☆☆ | ★☆☆☆ | ★★★★ |
| Generation loss resilience | ★★★☆ | n/a | ★☆☆☆ | ★★☆☆ | ★★☆☆ | ★★☆☆ | ★★★☆ |
| Lossless JPEG recompression | n/a | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| **Royalty-free?** | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |

COMPARE

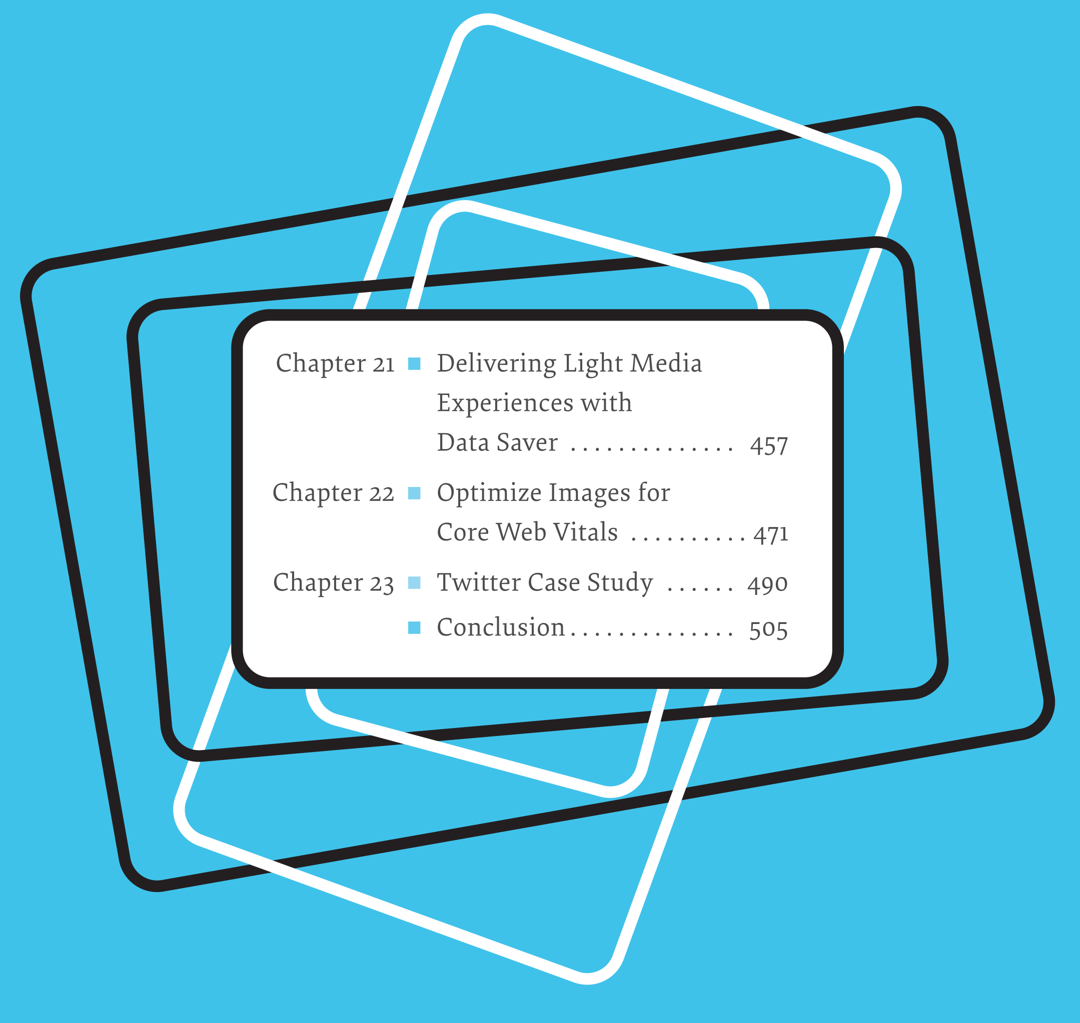
COMPARE





## Part Five

# Further Optimization


Chapter 21 ■ Delivering Light Media Experiences with Data Saver . . . . . . . . . . . . . . 457
Chapter 22 ■ Optimize Images for Core Web Vitals . . . . . . . . . . 471
Chapter 23 ■ Twitter Case Study . . . . . . 490
■ Conclusion . . . . . . . . . . . . . . 505

CHAPTER 21

# Delivering Light Media Experiences with Data Saver

Browsing the web with poor connectivity can be a frustrating, slow, and expensive experience, especially with data affordability being a huge problem in many countries. Users currently have to adjust their behavior (not watch videos, for example, or use the lightest available version of pages) to make the most of their data plans.

Browsers with Data Saver features give users a chance to explicitly tell us that they want to use a site without consuming so much data. The Save-Data header can be supplied by any supported browser back to a site when a user requests an experience that processes less data. At the time of writing, this is primarily Chrome, Edge, Opera and other Chromium-based browsers.

Given how heavy modern sites can be, reducing page weight when this signal is present can lead to a better user experience. The following code checks the value of the Save-Data client hint request header:

```
// Check if `Save-Data` header is set to a value of
"on"
const isDataSaverEnabled() {
    if (strtolower($_SERVER['HTTP_SAVE_DATA']) ===
'on') {
        // `Save-Data` is on!
        return true;
    }
    return false;
}
```

When a Data Saver feature is on, a browser could send the Save-Data header and not do anything to improve the user experience directly itself. Historically, this has not been the case as some browsers have used proxy services to rewrite pages so they can be served much faster without developers needing to do anything.

With most of the web now on HTTPS, the Data Saver landscape has changed to focus more on optimizing when or if resources load, rather than heavily rewriting pages.

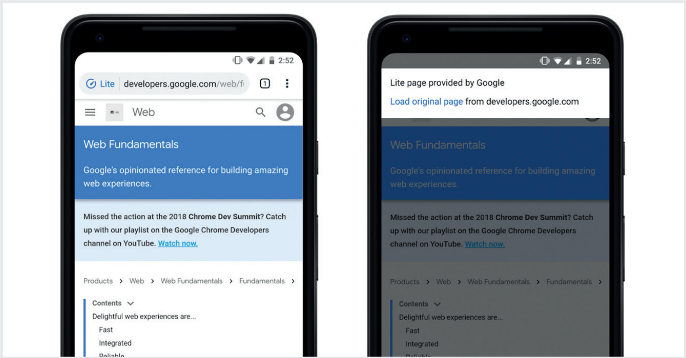
Chrome on Android's Lite mode is one such Data Saver feature that helps by automatically optimizing web pages to make them load faster. On web pages that are expected to load slowly, Chrome may modify loading behavior to

provide a faster page load, serving a "Lite" page instead. Lite page modifications can take many forms, including but not limited to:

- Applying interventions to improve page load speed (e.g. automatic lazy loading or deferring the execution of costly resources like JavaScript).
- Proxying pages (served over HTTP), applying server-side optimizations to improve how quickly they load.
- Adding Save-Data HTTP headers.

The first two optimizations are only implemented when the loading experience would be painful. They are applied when the network's effective connection type is poor (2G, for example) or when Chrome estimates the page load will take more than five seconds to reach "first contentful paint" (a Core Web Vital: see chapter 22) given current network conditions and device capabilities.

To indicate when a page has been optimized, Chrome displays an icon in the address bar. Tapping the icon allows users to load the original page. Should a user frequently choose the original, Chrome disables Lite mode on a per-site or per-user basis.




*The Lite mode icon in Chrome's address bar can be tapped to allow users to load the original page. (Source: "Chrome Lite Pages – For a faster, leaner loading experience"[1] by Ben Greenstein and Nancy Gao.)*

Frequent recourse to Data Saver can indicate a user's preference for limiting their data usage. Perhaps their data plan is restrictive, their connection speeds are often slow, or maybe they'd just like pages to load a little more quickly.

## Adaptive Loading with Data Saver

When a user has Data Saver on, developers can adapt how they serve a light, low-fidelity version of their pages. Some use cases for adaptive loading include:

- serving low-quality images and videos
- avoiding loading components requiring heavy JavaScript

1 https://smashed.by/chromelitepages

- throttling the frame rate of animations
- avoiding computationally heavy operations
- blocking third-party JavaScript
- disabling background images
- removing web fonts
- disabling service workers, prefetching, and precaching
- disabling tracking and third-party scripts

Adaptive loading allows you to serve a core experience to all users and progressively add high-end features and resources for users without Lite mode on. Let's look at two popular patterns in more detail.

### ADAPTIVE MEDIA LOADING

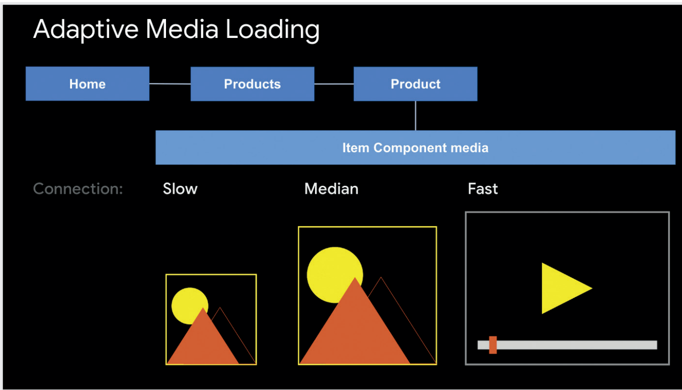
Adaptive media loading aims to serve low-quality images and videos to reduce bandwidth and memory consumption.

For sites relying heavily on media, this could mean:

- A photo gallery site might deliver lower resolution previews, or use a less code-heavy carousel mechanism.

- A search site could reduce the quality of images in search results or disable images altogether.
- An e-commerce site could replace large product videos with static images or 3D models (via WebXR) instead if they are lighter in weight.
- A news site could rely less on rich images for article hero images, and serve lo-fi versions or disable.

While conditionally serving different resources based on device capabilities is a good strategy, sometimes it can be even better to not serve a resource at all. Adaptive loading unlocks this choice.

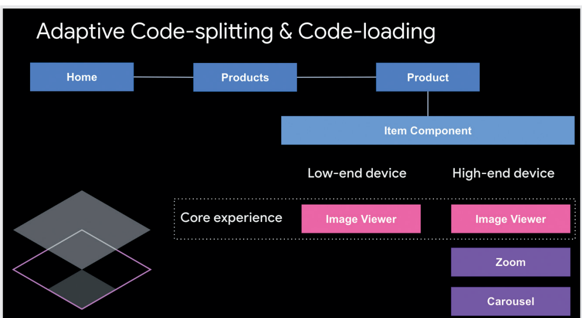



*In an e-commerce site with a product page, adaptive media loading could allow the page to deliver a small image on slow network connections, a medium-sized image on median connections, and a high-resolution video of the product on fast connections.*

## ADAPTIVE CODE LOADING

Adaptive code loading is all about shipping a light, core interactive experience to all users and progressively adding high-end features on top – device-awareness like this takes progressive enhancement one step further. JavaScript is costly in two main ways: the time it takes to download it, and the time it takes to process it. On slow networks and low-end devices, both of these costs can delay how soon a page is ready.

When it comes to images, for the lightest experience a site could disable features that require more JavaScript, like image carousels and instead only serve one hero image. On high-end devices, we can conditionally load more highly interactive components or run more computationally heavy operations, while not sending these scripts to slower devices.




*An example of adaptive code splitting and loading for an e-commerce site.*

# CLEAN & COMPRESS

Shrink your images by removing unnecessary metadata and compressing them to reduce the file size

As shown in the illustration on page 463, an e-commerce product page could have a product item component used to render details about what is being sold. With adaptive code loading, a low-end device could receive the core experience: an image viewer with product details. A higher-end device could be served the core experience plus a component for zooming into the product image, and a product image carousel. Users on both types of device are still delivered a useful experience, but the higher-end device gets something more enhanced as it can handle the extra functionality well.

## Detecting Data Saver Mode

Let's discuss `navigator.connection.saveData`, which allows us to determine if a user has switched on Data Saver in Chromium-based browsers.

The Network Information API provides details of the quality of a user's network connection in JavaScript. It is exposed as `navigator.connection` and includes values such as `connection.effectiveType` (`3g`, `4g`, and so on). This can be used to switch between delivering high-quality and low-quality resources based on the user's effective connection quality.

In Chromium-based browsers (Chrome, Edge, and so on), the `connection` object is also home to the Boolean

`saveData`. If `true`, a user likely has turned on their browser's Data Saver mode, meaning we can conditionally deliver data-saving strategies. Keep in mind that not all browsers support the Network Information API, so check for the existence of the `connection` object before using it.

```
if ('connection' in navigator) {
    if (navigator.connection.saveData === true) {
        // Implement data saving approach
    }
}
```

This can also be logged to your analytics and real user monitoring (RUM) to get a sense of the percentage of your users who have a Data Saver feature turned on. Below is an example[2] of checking for the Save-Data value from inside a service worker. It can be combined with other signals, such as checking for whether an effective connection was slower than typical 3G or had low RAM.

```
if (
    // Save-Data is on
    fetchEvent.request.headers.get('save-data')
 // bandwidth or RTT is slower than a typical 3G
connection
    || (navigator.confirmWebWideTrackingException.
effectiveType.match(/2g/))
    // we have less than ~1GB of RAM
```

2 https://smashed.by/fastapps

```
    || (navigator.deviceMemory < 1)
)
```

Furthermore, you can check for the Save-Data client hint request header[3] to assess if Data Saver mode is on. This lets you conditionally deliver the lighter experience to your users. When Data Saver is on, the browser appends the Save-Data request header to all outgoing requests (both HTTP and HTTPS). At the moment, only one `on-` token is output in the header (`Save-Data: on`); this is likely to be expanded as time goes on to reveal further user preferences.

```
// Check the `Save-Data` header exists and is set to a
value of "on".
if (isset($_SERVER["HTTP_SAVE_DATA"]) && strtolower($_
SERVER["HTTP_SAVE_DATA"]) === "on") {
  // `Save-Data` was detected
  $saveData = true;
}
```

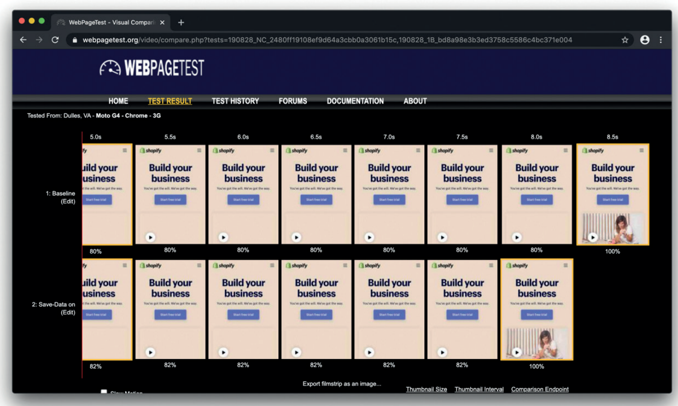
Web developers can choose to opt out of Data Saver transformations by including the `Cache-Control: no-transform` directive in the header of the original page's main HTML response. `no-transform` indicates to browsers that no transformations should be made to resources. Chrome respects this directive to disable Lite mode.

3 https://smashed.by/savedataheader

## Production Examples

### SHOPIFY

Popular sites are becoming Save-Data-aware. Shopify saw a 13% reduction in page weight as a result of this change, with data showing that 20% of requests from users in India and Brazil contain a value for the Save-Data header.

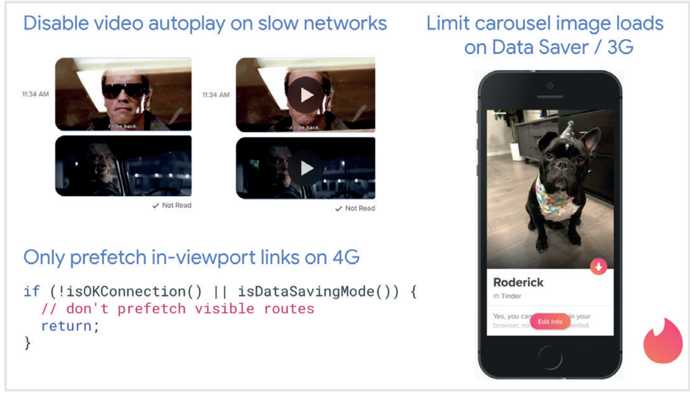



*Shopify serves 1x images instead of 2x when the Save-Data header is present. Tests show there shouldn't be a perceivable difference because of the oversizing of many of the images.*

### TINDER

Tinder uses checks for Save-Data in production to keep the user experience fast for everyone. On slow networks

or when Data Saver is enabled, video autoplay is disabled, route prefetching is limited, and loading the next image in the carousel is restricted so they load just one at a time. Tinder has seen great improvements in the average swipe count for each user on Tinder Lite (for example, 7% more swipes in Indonesia).




*Tinder Online limits image loads when Data Saver is enabled so that they only load one at a time. This improves performance for users who are data-conscious.*

Thankfully, there is not much complexity in the Save-Data header: it is either on or off. We carry the responsibility of delivering appropriately light experiences to users who have the feature enabled.

It is worth keeping in mind, however, that people may use Data Saver in many different ways. While some might use it to keep the pages they view as light and quick to load as possible, others could be cautious if they worry pages will lose some important functionality. It's a good idea to assume users prefer the full experience until they provide a signal they would prefer a lighter version.

SUCCESS STORY ■ "After adding webp support, we saw a 30% improvement in page load time on WebP supported browsers. We also saved 15 gb per million image requests!"

—The Tribune (Nov, 2018)

We can choose to allow the browser to automatically try delivering more lightweight pages if Data Saver is on, or use Data Saver as a signal to further customize the optimized experience they get when in a Lite mode.

CHAPTER 22

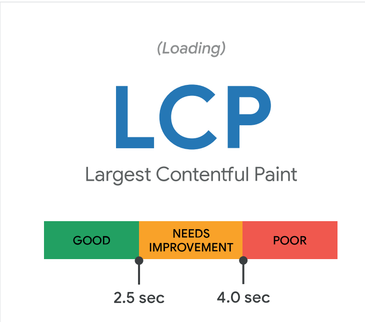
# Optimize Images for Core Web Vitals

A high-quality user experience is critical to the success of your site in the long term. Over the years, Google has developed several metrics and tools to help identify opportunities to improve user experience, helping millions of sites along the way. At the same time, having so many metrics can sometimes create its own challenges. Which metrics should you focus on if unsure where to start? Which ones help you optimize for user-centric outcomes?

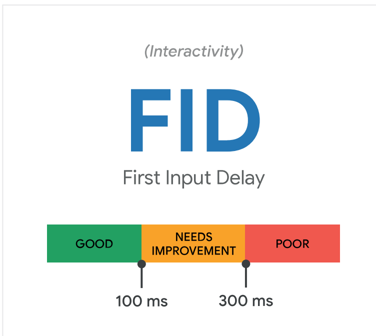
That's where Core Web Vitals[4] come in. Core Web Vitals is an initiative from Google to offer developers unified guidance on page quality signals. There are many ways to measure if a user experience is high quality, but Core Web Vitals focuses on a set of metrics considered critical for all experiences on the web. This set of metrics aims to evolve over time with a predictable cadence.

The Core Web Vitals look at three key aspects of user experience. There's the page loading experience, interaction readiness, and the visual stability of the page. Let's take a quick look at the three metrics that correspond to these quality signals and their respective thresholds:

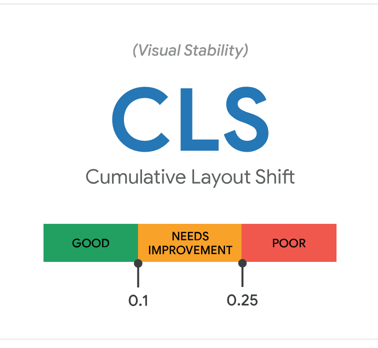
4 https://smashed.by/vitals




*Largest contentful paint*[5] *threshold recommendations.*




*First input delay*[6] *threshold recommendations.*




*Cumulative layout shift*[7] *threshold recommendations. (Source: https://web.dev/vitals/)*

5 https://smashed.by/lcp
6 https://smashed.by/fid
7 https://smashed.by/cls

- Largest contentful paint (LCP) measures loading performance. To provide a good user experience, LCP should occur within 2.5 seconds of when the page first starts loading.

- First input delay (FID) measures interactivity. To provide a good user experience, pages should have a FID of less than 100 milliseconds.

- Cumulative layout shift (CLS) measures visual stability. To provide a good user experience, pages should maintain a CLS of less than 0.1.

Google's set of developer tools that support the Core Web Vitals[8] consider a page reaching these targets at the 75th percentile[9] of page loads (on both mobile and desktop) as a pass. For the latest details, consult the official documentation.[10]

During 2020, Google announced[11] that it would incorporate the Core Web Vitals alongside existing signals for page experience into a Google Search ranking change. Interest from the web community led to a median 70% increase in developers using tools like Lighthouse and PageSpeed Insights, and several using Search Console to identify how they could improve. Now is a great time to think about optimizing for Core Web Vitals, as these signals can be influenced by how you load resources like images.

---

8 https://smashed.by/vitalstools
9 https://smashed.by/corewebvitals
10 https://smashed.by/learnwebvitals
11 https://smashed.by/pageexperience

Let's dive into how to optimize image loading to minimize their impact on the Core Web Vitals. These will primarily be cumulative layout shift and largest contentful paint.

## Optimize Largest Contentful Paint

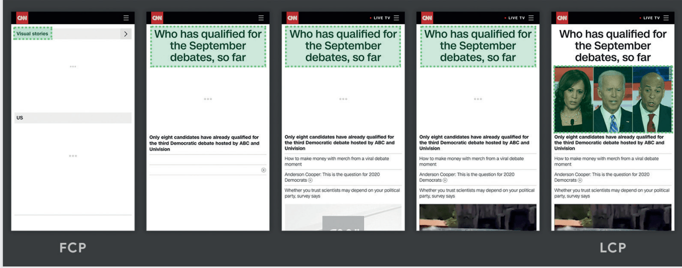
When browsing the web, you can find yourself asking why it sometimes takes so long for the main content to appear. Developers haven't had a reliable metric which correlates well with the visual rendering experience encountered by their users.

While some of the existing metrics like first contentful paint[12] look at initial rendering, they don't assess the importance of what is being painted. This means they might miss times when the user is still waiting for a page to be useful. Largest contentful paint (LCP)[13] aims to address this gap, better correlating with user experience and being easier for developers to reason about.

*Largest contentful paint (LCP) measures a point during page load when a page's main content has likely loaded*

12 https://smashed.by/timing
13 https://smashed.by/firstcontentfulpaint

LCP captures the speed of delivering the largest contentful element to the screen – that is, the main content a user might look at. This main content could be a hero image (such as an article header or product image) or it could be a block-level element that includes text, like the paragraph in a news article.




*A filmstrip highlighting in green the element considered the largest contentful paint element as a page load progresses.*

Good LCP values are 2.5 seconds; poor values are greater than 4.0 seconds; and anything in between needs improvement.

### ELEMENTS CONSIDERED FOR LCP

At the time of writing, the elements[14] taken into account for largest contentful paint are:

- `<img>` elements
- `<image>` elements inside an `<svg>` element

14 https://smashed.by/lcp

- `<video>` elements (the `poster` image is used)
- elements with a background image loaded via the `url()` function (as opposed to a CSS gradient)
- Block-level elements containing text nodes or other inline-level text elements children

LCP looks at the largest element as a way to approximate what the main content on the page is. As the size of elements can change during page load, LCP uses the size of the first paint of an element to determine which is largest. For pictorial elements like images, this first paint is the one after the image is fully loaded.

This use of initial size can affect pages where the images move. A good example of this happens with animated image carousels. Carousel images that are initially not in the viewport and slide in may see LCP consider their painted size when added to the DOM (zero) but this could change.

To learn more about how browsers report largest contentful paint, check out the guidance in Philip Walton's article[15] at web.dev.

15 https://smashed.by/lcp

### IMPROVING A POOR LCP SCORE

The most common causes of a poor LCP score are:

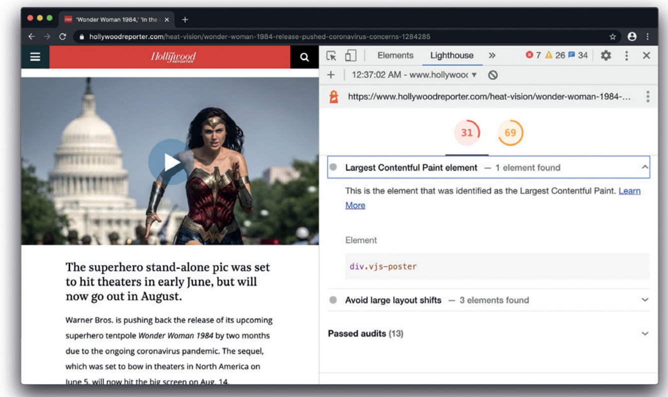
- slow resource load times
- slow server response times
- render-blocking JavaScript and CSS
- content loading delayed by client-side rendering

Images can certainly have slow loading times. They're frequently the largest contentful element immediately visible to users on many websites. News articles and blog posts often have large hero images, while product pages on retail sites may have an image of a product as their LCP element. It's important for these images to load and render as quickly as possible. When images are the largest page element, improve LCP by:

- compressing images
- using responsive images where possible
- serving images in modern formats
- considering if the image really adds value: if not, remove it.

## IDENTIFYING THE LARGEST CONTENTFUL PAINT ELEMENT

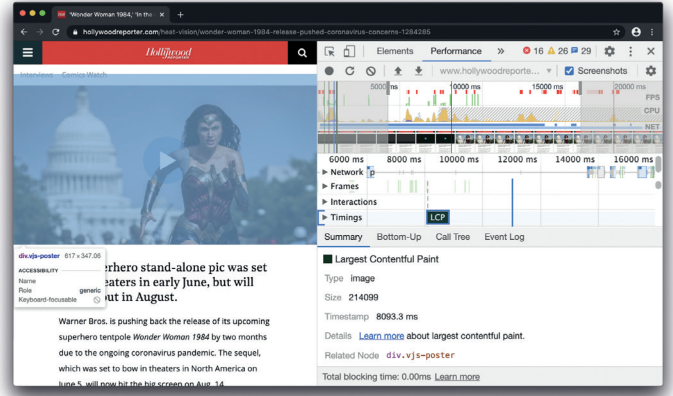
The Lighthouse[16] panel in the Chrome DevTools can run an audit to discover the largest contentful element. You will also find this audit in the Lighthouse section of PageSpeed Insights.[17]




*The largest contentful paint element audit in the Lighthouse panel of Chrome's DevTools.*

LCP can also be measured in the Chrome DevTools Performance panel. When you perform a recording, the Timings section will include LCP. If you click on an LCP record and hover over its related node, you'll see which element was the largest contentful paint element. In this case, we can see that it's the hero image for the page.

16 https://smashed.by/lighthouse
17 https://smashed.by/pagespeedinsights




*Using the Performance panel in Chrome's DevTools to identify the largest contentful paint element.*

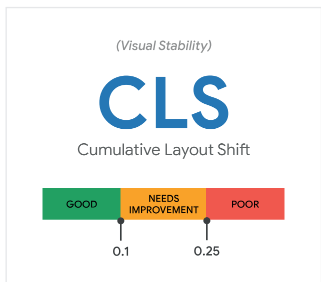
## Optimizing for Cumulative Layout Shift from Images

"I was about to click that! Why did it move?"

Layout shifts can be distracting. Imagine you've started reading an article when all of a sudden elements shift around the page, throwing you off and requiring you to find your place again. This is very common on the web, including when reading the news, or trying to click those "Search" or "Add to Cart" buttons. Such experiences are visually jarring and frustrating. They're often caused when visible

elements are forced to move because another element was suddenly added to the page or resized.

Cumulative layout shift (CLS)[18] is a user-centric experience metric measuring the instability of content by summing shift scores across layout shifts that don't occur within 500ms of user input. It looks at how much visible content shifted in the viewport as well as the distance the affected elements were shifted.




*Cumulative layout shift (CLS) is a metric that measures the visual stability of a page.*

Good CLS values are under 0.1; poor values are greater than 0.25; and anything in between needs improvement.

The most common causes of a poor CLS score are:

- images without dimensions
- ads, embeds, and iframes without dimensions
- dynamically injected content

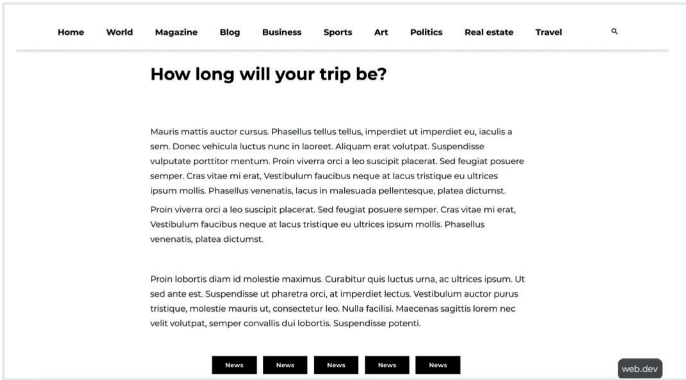
18 https://smashed.by/cls

- web fonts causing a flash of invisible text (FOIT) or unstyled text (FOUT)
- actions waiting for a network response before updating DOM

Let's look at the most common causes of poor CLS score and how to address them in more detail.

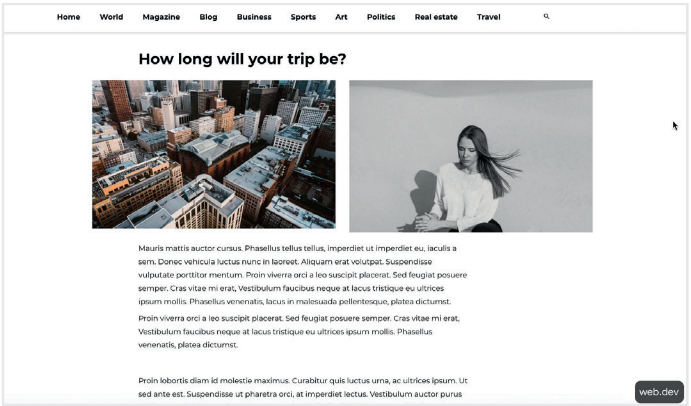
### IMAGES MISSING DIMENSIONS

Always include `width` and `height` attributes on your images and video elements. Similarly, reserve the required space with CSS aspect ratio boxes.[19] This approach ensures that browsers can allocate the correct amount of space in the document while images load.

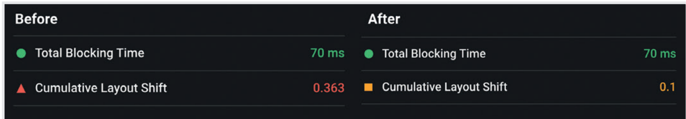



*Images without width and height specified.*

19 https://smashed.by/ratioboxes




*Images with width and height specified.*




*Lighthouse report showing the before and after impact to cumulative layout shift of setting dimensions on images.*

### History

In the early days of the web, developers would add `width` and `height` attributes to their `<img>` tags to ensure sufficient space was allocated on the page before the browser started fetching images. This would minimize reflow and redrawing.

```
<img src="puppy.jpg" width="640" height="360"
alt="Puppy with balloons.">
```

You may notice that `width` and `height` above do not include units. These pixel dimensions would ensure a 640 × 360 pixel area would be reserved. The image would stretch to fit this space, regardless of its true dimensions.

When responsive web design[20] became mainstream, developers began to omit width and height, and started using CSS to resize images instead:

```
img {
  width: 100%; /* or max-width: 100%; */
  height: auto;
}
```

A downside to this approach was that space could only be allocated to an image once it began to download and the browser could determine its dimensions. As images loaded in, the page would reflow as each image appeared on screen. It became common for text to suddenly pop down the screen – not a great user experience at all.

This is where aspect ratio comes in. The aspect ratio of an image is the ratio of its width to its height. It's common to see this expressed as two numbers separated by a colon (for example, 16:9 or 4:3). For an *x:y* aspect ratio, the image is *x* units wide and *y* units high.

20 https://smashed.by/rwdguidelines

This means if we know one of the dimensions, the other can be determined. For a 16:9 aspect ratio:

- If *puppy.jpg* has a height of 360px, its width is $360 \times (16 \div 9) = 640$px
- If *puppy.jpg* is 640px wide, its height is $640 \times (9 \div 16) = 360$px

Knowing the aspect ratio allows the browser to calculate and reserve sufficient space for the height and associated area.

### Modern Best Practice

Modern browsers set the default aspect ratio of an image based on its `width` and `height` attributes, so it's valuable to set them to prevent layout shifts. Thanks to the CSS Working Group, developers just need to set `width` and `height` as normal:

```
<!-- set a 640:360 i.e a 16:9 - aspect ratio -->
<img src="puppy.jpg" width="640" height="360"
alt="Puppy with balloons.">
```

... and the user agent style sheets[21] of all browsers add an intrinsic aspect ratio[22] based on those attributes of the element:

```
img {
    aspect-ratio: attr(width) / attr(height);
}
```

21 https://smashed.by/useragent
22 https://smashed.by/intrinsic

This calculates an aspect ratio based on the `width` and `height` attributes before the image has loaded. It provides this information at the very start of layout calculation. As soon as an image is told to be a certain width (for example `width:100%`), the aspect ratio is used to calculate the height.

> If you're having a hard time understanding aspect ratio, *aspectratiocalculator.com* is available to help.[23]

These image aspect ratio changes have shipped in Firefox and Chromium, and are coming to WebKit (Safari). For a fantastic deep-dive into aspect ratio with further thinking around responsive images, see "Jank-free page loading with media aspect ratios"[24] by Craig Buckler.

If your image is in a container, you can use CSS to resize the image to the width of this container. We set `height: auto;` to avoid the image height being a fixed value (for example, 360px).

```
img {
  height: auto;
  width: 100%;
}
```

23 https://smashed.by/ratiocalc
24 https://smashed.by/jankfree

### Responsive Images

When working with responsive images,[25] `srcset` delineates the images you allow the browser to select between and what size each image is. To ensure `<img> width` and `height` attributes can be set, each image should use the same aspect ratio.

```
<img width="1000" height="1000"
       src="puppy-1000.jpg"
    srcset="puppy-1000.jpg 1000w,
            puppy-2000.jpg 2000w,
            puppy-3000.jpg 3000w"
          alt="Puppy with balloons"/>
```

What about art direction?[26] Pages may wish to include a cropped shot of an image on narrow viewports with the full image displayed on desktop.

```
<picture>
  <source media="(max-width: 799px)" srcset="puppy-
480w-cropped.jpg">
  <source media="(min-width: 800px)" srcset="puppy-
800w.jpg">
  <img src="puppy-800w.jpg" alt="Puppy with balloons">
</picture>
```

It's very possible these images could have different aspect ratios. Browser makers are still evaluating the most

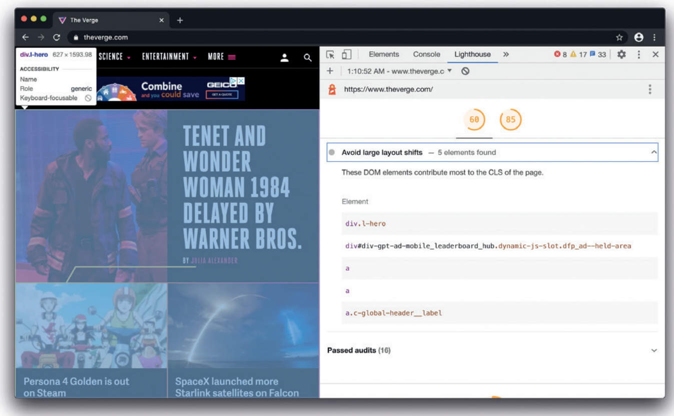
---

25 https://smashed.by/servingresponsive
26 https://smashed.by/responsiveartdirection

efficient solution here, including if dimensions should be specified on all sources. Until a direction is chosen, redrawing is still possible here.

### IDENTIFYING ELEMENTS THAT SHIFTED

There are a number of tools available to measure and debug CLS. Lighthouse[27] includes support for measuring CLS in a lab setting on your desktop. The Lighthouse "Avoid large layout shifts" audit also highlights the DOM elements contributing most to the CLS of the page.

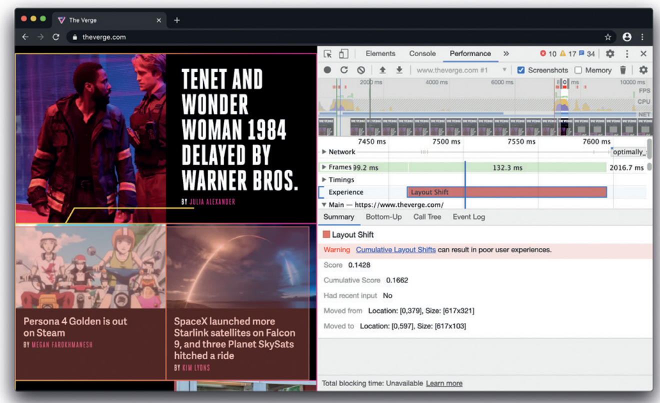



*The Lighthouse report includes an "Avoid large layout shifts" audit.*

You can hover over any of these DOM elements to highlight them, or click to view them in the Elements panel.

27 https://smashed.by/lighthouse

The Chrome DevTools Performance panel also has an Experience section[28] that can help you discover unexpected layout shifts. This is helpful for finding and fixing visual instability issues on your page.




*A layout shift being highlighted in the Experience section of Chrome DevTools. Clicking it expands a summary that includes further detail about what shifted.*

Select a Layout Shift to view its details in the Summary tab. To visualize where the shift itself occurred, hover over the Moved from and Moved to fields.

To measure CLS from the perspective of your users, you can also use the Chrome User Experience Report[29] or measure the metric yourself[30] via your analytics provider and real user monitoring (RUM).

28 https://smashed.by/devtoolscls
29 https://smashed.by/chromeux
30 https://web.devlc

Hopefully these tips will help keep your pages just a little less shifty.

Ensuring images load quickly while not causing layout shifts to the page will help ensure you're delivering a great user experience.

Although the Core Web Vitals metrics measure three important facets of user experience, there are other aspects Google intends to expand on, including new metrics to measure smoothness, and support better delivery of instant and privacy-preserving experiences on the web.

Google aims to update Core Web Vitals annually and provide developers with updates on potential candidates for new metrics and the motivation behind them as time goes on. To keep up to date, follow web.dev for further updates.

CHAPTER 23

# Case Study: Twitter's Image Pipeline

*With special thanks to Nolan O'Brien for his years of work on Twitter's image optimization pipelines.*

Twitter has fast become a platform for sharing news, views, and more with the world. In some cases people are more likely to consume the latest news from around the world as short tweets rather than full-length articles online or through other media.

Tweets are often accompanied by images to illustrate, amuse, and increase user engagement. There are around 330 million monthly active users around the world who consume, create, and share information on Twitter and 80% of them are on mobile.

It's important for Twitter that users get the most out of the images they upload or see on their timelines. That is why it places so much emphasis on a strong image optimization process. This article focuses on the different steps that Twitter has taken to load images faster while ensuring they are as impactful as intended.

## Progressive JPEGs

Progressive JPEGs (see chapter 7) consist of multiple interlaced layers of the image. The layers are rendered one at a time and merged with the previously rendered layers to display the image. Displaying images as progressive JPEGs improves the perceived image-loading performance.

All JPEG images uploaded to Twitter are transcoded to progressive JPEGs, with an 85% quality setting if the image has a higher quality. To achieve this transformation Twitter uses libjpeg-turbo[31] with a few customizations. Additionally, even though WebP is a supported upload format, all WebP images uploaded are served as progressive JPEGs at 85% quality. A WebPageTest report for Google's page on Twitter[32] shows that all JPEGs loaded were progressive JPEGs.

Twitter evaluated the perceived latency (time to first scan and overall load time) with different image formats like progressive JPEG, WebP, and JPEG 2000. The goal was to achieve low file sizes for images of comparable quality with acceptable transcode and decode times. They found that, overall, progressive JPEGs were better at addressing these requirements when compared with WebP and JPEG 2000.

31 https://libjpeg-turbo.org/
32 https://smashed.by/vitals

> Twitter's research[33] into user-perceived latency involved conducting a number of experiments. This was a long churn of trying out formats of different settings until they settled on formats that were promising. A feature switch allowed them to put 50% of users on a current control format and 50% on the treatment format. This required full user base splits to remove bias from CDN hydration of competing formats. Even a 60/40 split could incur bias to the higher allotted format. Remember to A/B test properly, kids!

Users with fast internet connections may not perceive a difference between progressive JPEG and baseline JPEG, but those with slow connections see images on screen much faster. Twitter achieved a 10-fold speed-up after implementing progressive JPEGs with respect to the time it took a user to go from a blank screen to something with content. This resulted in an increase in the number of tweets consumed. Users in India and Indonesia experienced a 90-fold improvement for a progressive first scan of low-quality JPEG when compared to PNG.

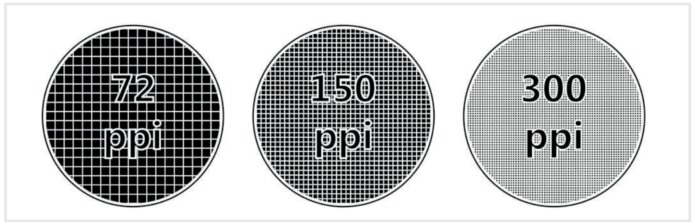
33 https://smashed.by/twitterperf

## Image Pixel Density

If you work with images, it's important to know about resolution, pixels per inch and dots per inch.

Resolution refers to the number of pixels on a device in each dimension (width × height) that it is possible to display on a screen: a device with a resolution of "2,880×1,800" is 2,880 pixels wide and 1,800 pixels high.

Pixels per inch (PPI) or pixel density represents a measurement of sharpness on a display screen. It's the number of pixels per inch on the display, or how much detail is in an image based on the concentration of pixels. Higher pixel density means greater sharpness when viewing images on the device.




*A lower PPI results in less detail and a pixelated image. A higher PPI results in more detail and a sharper image.*

Dots per inch (DPI) represents the resolution of a printer. Printers produce images by throwing out small dots, and

the quality of dots per inch affects the amount of detail and quality you see in the print. As a good rule of thumb, PPI is for digital images and DPI is for print.

Equipped with this knowledge, we know that displays with a higher pixel density show sharper images. When Apple unveiled its Retina display, Steve Jobs[34] mentioned that with a pixel density of 300 PPI the human eye could no longer distinguish between individual pixels for a device held 10 to 12 inches from the eye. Later, some experts[35] agreed that this claim was correct for a person with average eyesight or 20/20 vision. This also implies that an average human eye cannot distinguish pixel densities beyond this.

For an average-sized phone display the pixel density of 300 ppi corresponds to 2×2 pixels per dot, also known as a 2x scale screen. However, phone manufacturers, including Apple, are now adopting screens with ultra-high resolutions of 458 PPI (iPhone XS) and 570 PPI (Samsung Galaxy S9) which corresponds to a 3x or more than 3x resolution.

With these ultra-high-resolution screens, the number of pixels per inch required for an image is much higher, leading to increased data usage and load latency.

The following illustration shows the difference between these resolution types.

34 https://smashed.by/retina
35 https://smashed.by/experts

*A visual comparison of increasingly high pixel density values. Higher pixel density can result in sharper images; however, beyond a certain pixel density the human eye can't distinguish meaningful differences.*

Twitter realized that it is unnecessary to serve images with a pixel density higher than the Retina density of 2x since a human eye cannot perceive that level of detail. Twitter started capping image density to 2x for all images served on the timeline for iOS, Android, and mobile web clients. For screens with resolutions higher than 2x, Twitter now calculates the variant and size of the image to load it as if it were for a 2x resolution.

*Above are the uncapped and capped versions of the same image provided on the Twitter engineering blog.*[36] *Twitter observed that there is no perceivable difference between the two images, yet it achieved a 38% saving on data and 32% on latency for this particular image.*

36 https://smashed.by/twittereng

> Latency refers to the total duration time to load. On a 3G connection, the time it took to load an entire image reduced by 32% (whether preview or full size). On faster connections, the delta was less for preview images but effectively the same for full size images.

This change is not applicable to images viewed in the gallery or full-screen images where users may pinch-zoom. The full image will always be loaded in these scenarios.

The change is especially beneficial for users with the latest high-end devices and ultra-high resolution screens. On these devices, images now load roughly 33% faster with the amount of data used reduced by one third.

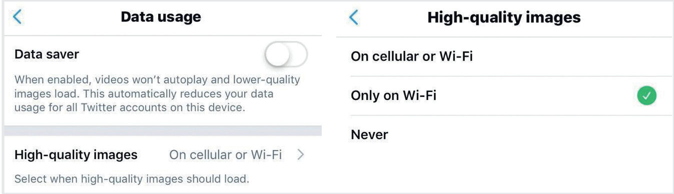
## Data Saver Mode

Across all applicable platforms, Twitter is designed to minimize data usage. In October 2018, Twitter introduced data saver mode for images on its native iOS and Android apps. This feature had previously been available on Twitter Lite,[37] Twitter's progressive web app. Once data saver

---

37 https://smashed.by/twitterlite

mode is enabled, images in tweets are presented as a small blurred preview on Twitter Lite and as a lower quality image on iOS and Android. The data saver mode can be enabled in **Settings > Data Usage** on both iOS and Android apps as shown. Even when data saver is not enabled, users can choose to load high-quality images only on Wi-Fi to conserve cellular data, which is expensive in many countries.

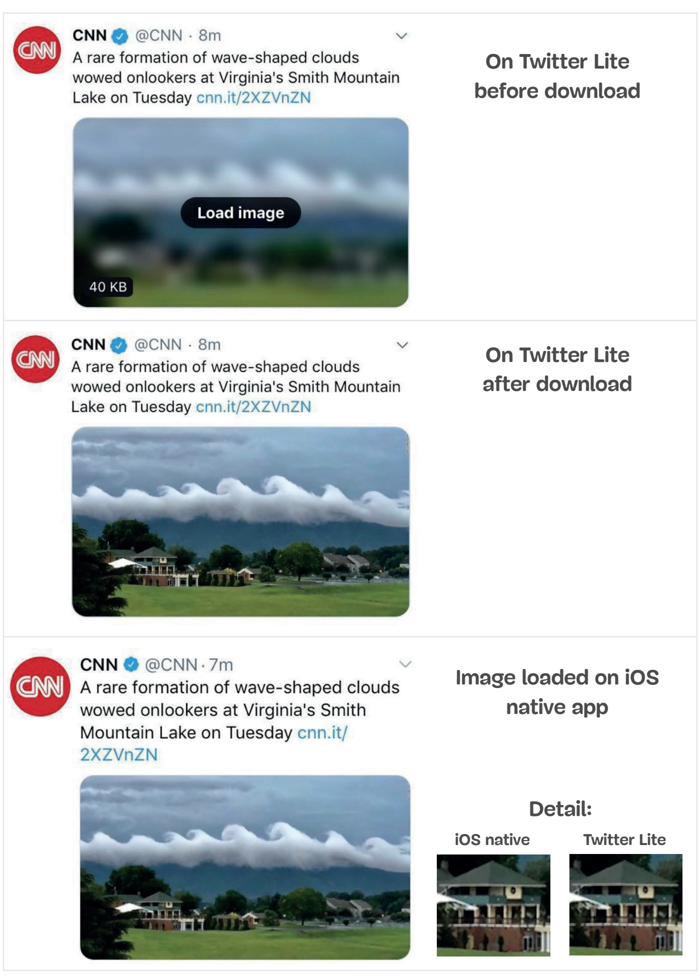



*Users in the Twitter native apps on iOS and Android can toggle on "Data saver" mode but also control when high-quality images should be loaded (e.g only when on Wi-Fi)*

On Twitter Lite, the image initially shown (1, overleaf) is a highly optimized 64×64 pixel image (a 1,000-byte JPEG) that is blurred using a technique similar to low-quality image placeholders (LQIP: see chapter 12). The actual image is loaded only when you tap on the preview (2).

On iOS and Android, the preview image is not blurred, but it has a slightly lower resolution than normal (3). Tapping to open the image will just expand that preview image and you

need to long-press or select the **More** options, then choose **Load High Quality.**



With these improvements Twitter observed a 50% reduction in data usage from images on iOS and Android, and an 80% reduction on the Twitter website. Since videos do not play automatically in data saver mode, a 96% improvement in data usage was observed by disabling video autoplay. Such savings add up and are especially beneficial for users on limited data plans.

## PNG-8 Support for Digital Artwork

Another optimization Twitter explored took advantage of certain kinds of PNG artwork only needing a limited color palette. Before we get into that, let's see what palettes mean for PNGs. Per the PNG specification,[38] PNGs can be:

1. palette-based (with or without transparency)
2. grayscale (with or without transparency/alpha channel)
3. red/green/blue (RGB) (with or without transparency/alpha channel)

A color palette[39] is a one-dimensional array of color values. Using a palette, image data can be stored as a series of index values. This can significantly reduce the size of pixel data when only a limited number of colors needs to be represented.

38 https://smashed.by/pngspec
39 https://smashed.by/palettes

**Indexed colors**

**Palette**
24 bits per color

| 1 |
| --- |
| 2 |
| 3 |
| 4 |

+

**Pixel Data**
8 bits per pixel

| 1 | 1 | 1 | 1 | 1 | 1 |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 3 | 2 | 1 |
| 1 | 3 | 4 | 4 | 3 | 1 |
| 1 | 3 | 4 | 4 | 3 | 1 |
| 1 | 2 | 3 | 3 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

=

**Final Image**

*4-bit pixel data could be used to represent 16-color images. Such colors are often defined in a palette in the image file. Applications that render images read pixel values from a file and use it as an index into a palette. This is used to specify colored pixels on the device outputting the image.*

The number of bits used per pixel differs for each of the PNG types listed above. Palette-based images are supported in four pixel depths: 1, 2, 4, and 8 bits. These correspond to a maximum of 2, 4, 16, or 256 palette entries. Thus, a PNG-8 image (8 bits per pixel) is much larger than a PNG-4 image (4 bits per pixel). Similarly, an RGB TrueColor image which is supported in two depths (24 and 48 bits per pixel) is even larger.

People specializing in digital art would often share and advertise such PNG-24 and PNG-32 images on Twitter. When served to a global audience these large PNGs proved to be very expensive and slow for those browsing Twitter on slow

network connections. At the same time, artists did not like the idea of serving such images as lossy JPEGs.

Digital artists usually do not need more than 256 colors, as they wish to share high-quality versions of their artwork but not at full resolution so they can sell or license the original high-resolution images. This implies that pixel art needs to be high quality but not high resolution. Twitter recognized this use case when it made changes to its PNG image support[40] in February 2019.

*An example of digital pixel artwork shared on Twitter. This image does not require more than 256 colors, nor does it need to be high-resolution, but it does need to be presented to users in high quality.*

Images uploaded as PNG-8 and below are left untouched when presented on Twitter, including any transparency.

40 https://smashed.by/twitterpng

This is because PNG-8 images are almost equivalent to their JPEG counterparts with respect to performance. At the same time, compromise on image artifacts that result from JPEG compression can be avoided. This serves as an encouragement for artists who would have previously uploaded their images as large PNG-24 or PNG-32 files. They can now upload their images as PNG-8 and simultaneously reach a wider audience of people on slow connections. This leads to images that are four to eight times smaller in size overall, which is a huge win.

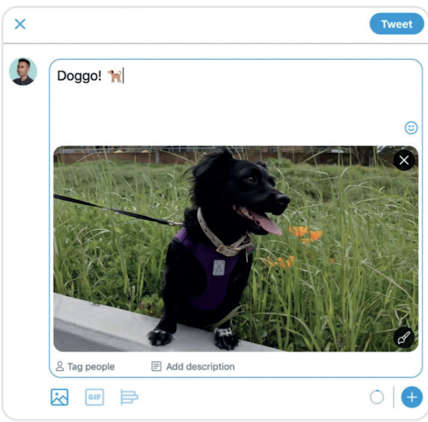
PNG-24, PNG-32, or higher are served as JPEG images on Twitter if the file size is considerably higher than the expected file size after converting to JPEG. This check offers a chance for image creators to sufficiently compress their PNG before upload so that it is more likely to be served as a lossless quality PNG. Images dominated by solid colors have a good chance of falling into this category.

## Client-Side Compression of Uploaded Images

The changes above were all targeted at enhancing the user experience when viewing images in the Twitter timeline. However, slow network connections also mean that it would take a long time to upload high-resolution or HDR images

taken using the latest smartphone cameras. To enhance the image upload experience, Twitter now checks if an image appears to be above a particular pixel/byte threshold. In such cases, it is drawn to a canvas and output at 85% quality JPEG to check for size reduction.

This often reduces phone-captured images in the range of 4 MB to around 500 KB with no discernible quality differences. Since image upload is especially slow over 2G and 3G, when such connections are detected Twitter rescales the image such that the image file does not exceed 150 KB. On its native iOS app, Twitter converts all images to JPEG at 85% quality before upload. Similarly, on Android, WebP images are posted unmodified, but all other images, including WebP images modified in the app, are converted to JPEG at 85% quality before upload.




*An image (of my dog Elvira) being uploaded to Twitter on a slow network connection. Twitter rescales the image and converts it to JPEG at 85% quality prior to uploading.*

Slow image uploads are frustrating, leading to users canceling the upload on many occasions. As such, Twitter measured the number of canceled uploads before and after the changes and found there was an overall reduction of 9.5% in the number of canceled photo uploads after these changes which is significant.

Twitter's investments in image performance have led to large improvements in its user experience. This is a great example of how research into image compression and serving techniques can benefit not just the web, but all platforms.

# Conclusion

Throughout this book, we've looked at ways to reduce image size through modern compression techniques with minimal impact to quality. The smaller in file size you can make your images, the better a network experience you can offer your users – especially on mobile.

Choosing an image optimization strategy will come down to the types of images you're serving to your users and what you decide is a reasonable set of evaluation criteria. It might be using SSIM or Butteraugli, or – if it's a small enough set of images – relying on human perception for what makes the most sense.

## Compress Images Efficiently

We should all be compressing our images efficiently.

At minimum: use ImageOptim.[41] It can significantly reduce the size of images while preserving visual quality. Windows and Linux alternatives[42] are also available.

More specifically: run your JPEGs through MozJPEG[43] (q=80 or lower is fine for web content) and consider progres-

41 https://imageoptim.com/
42 https://smashed.by/imageoptimversions
43 https://smashed.by/mozjpeg

# PERFORMANCE IS A JOURNEY

Lots of small changes can lead to big gains.

sive JPEG[44] support, PNGs through pngquant,[45] and SVGs through SVGO.[46] Explicitly strip out metadata (`--strip` for pngquant) to avoid bloat. Instead of crazy huge animated GIFs, deliver H.264[47] videos (or WebM[48] for Chrome, Firefox, and Opera). If you can't, at least use Giflossy.[49] If you need higher-than-web-average quality and you are OK with slow encoding times and using extra CPU cycles, try Guetzli.[50]

Some browsers advertise support for image formats via the Accept request header. You can use this to conditionally serve different formats; for example, lossy WebP[51] for Blink-based browsers like Chrome, and fallbacks like JPEG and PNG for other browsers.

## Automate Image Compression

Image optimization should be automated.

It's easy to forget, best practices change, and content that doesn't go through a build pipeline can easily slip. To automate: use imagemin,[52] libvips,[53] or one of many alternatives for your build process.

---

44 https://smashed.by/martians
45 https://pngquant.org/
46 https://smashed.by/svgo
47 https://smashed.by/h264
48 https://www.webmproject.org/
49 https://smashed.by/giflossy
50 https://smashed.by/guetzlisize
51 https://smashed.by/webp
52 https://smashed.by/imagemin
53 https://smashed.by/libvips

Most content delivery networks, like Akamai,[54] for example, and third-party solutions, like Cloudinary,[55] imgix,[56] Fastly's Image Optimizer,[57] or ImageOptim[58] offer comprehensive automated image optimization solutions.

The amount of time you'll spend reading blog posts and tweaking your configuration is probably worth the monthly fee for a service (Cloudinary has a free[59] tier). If you don't want to outsource this work because of cost or latency concerns, the open-source options mentioned above are solid. Projects like imageflow[60] or Thumbor [61] enable self-hosted alternatives.

## Do More

There's always more you can do.

Tools exist to generate and serve `srcset` breakpoints. Resource selection can be automated in Blink-based browsers with client hints,[62] and you can ship fewer bytes to users who opted into "data savings" in-browser by heeding the Save-Data[63] hint.

---

54 https://smashed.by/whyakamai
55 https://cloudinary.com
56 https://imgix.com
57 https://smashed.by/fastly
58 https://smashed.by/imageoptimapi
59 https://smashed.by/cloudinarypricing
60 https://smashed.by/imageflow
61 https://smashed.by/thumbor
62 https://smashed.by/resourceselection
63 https://smashed.by/savedata

Here are my closing recommendations.

If you can't invest in conditionally serving formats based on browser support:

- Guetzli and MOZJPEG's jpegtran are good optimizers for JPEG quality > 90.
- For the web, q=90 is wastefully high. You can get away with q=80, and on 2x displays even with q=50. Since Guetzli doesn't go that low, for the web you can MOZJPEG.
- Kornel Lesiński recently improved MOZJPEG's cjpeg command to add a tiny SRGB profile to help Chrome display natural color on wide-gamut displays.
- PNG pngquant + advpng has a pretty good speed/compression ratio.

If you can conditionally serve (using `<picture>`, the Accept header,[64] or Picturefill[65]):

- Serve WebP to browsers that support it.

64 https://smashed.by/acceptheader
65 https://smashed.by/picturefill

- Create WebP images from original 100% quality images. Otherwise you'll be giving browsers that support it worse-looking images with JPEG distortions and WebP distortions! If you compress uncompressed source images using WebP it'll have the less visible WebP distortions and can compress better too.

- The default settings used by the WebP team (-m 4 -q 75) are usually good for most cases where they optimize for speed/ratio.

- WebP also has a special mode for lossless (-m 6 -q 100), which can reduce a file to its smallest size by exploring all parameter combinations. It's an order of magnitude slower but is worth it for static assets.

- As a fallback, serve MozJPEG-compressed sources to other browsers.

Happy compressing!

# Index

7-zip. . . . . . . . . . . .158

accessibility . . . . . . xii, 206, 226, 348, 359

adaptive
- code loading . . . . 463, 465
- media loading . . . 461–462
- predictor . . . . . . 429
- quantization . . . . 424, 427

Adobe
- Illustrator. . . . . . 202, 213
- Lightroom . . . . . 395
- rgb color space. . . 71–73

Advanced Video Coding (AVC). . . 384

advdef . . . . . . . . . 165

AdvPNG . . . . . . . . . 158, 509

Akamai . . . . . . . . . 242, 508

Aleksandersen, Daniel . . . . . . . 398–399

aliasing. . . . . . . . . 161–162

Alliance for Open Media (aomedia) . . . . . 399

alpha transparency . 148, 152, 154

Amazon S3 . . . . . . 190

Android . . . . . . . . 78, 122, 187, 301, 303, 333, 394–395, 397, 414, 458, 495–497, 499, 503
- Android Pie. . . . . 394

Animated GIFs . . . . v, 174, 222, 399, 507
- replacing . . . . . . Chapter 15: 314–335

animations . . . . . . vi, xi, 337, 340, 345, 403, 435, 461

anti-aliasing. . . . . . 152, 157, 161–163

API . . . . . . . . . . . 364
- fetch . . . . . . . . . 269

Apple . . . . . . . . . . 131, 382, 391, 393, 397, 405, 414, 494

art direction. . . . . . xxii, 223, 231–233, 354, 486, 515

artifacts . . . . . . . . 43–44, 51, 128, 135, 161–162, 172, 178, 184, 196, 324, 351, 353, 407, 409, 425, 428, 502

aspect-ratio . . . . . . 42–44, 484

attributes
- decoding . . . . . . 37, 96
- poster . . . . . . . . 327, 333
- sizes . . . . . . . . . 32
- srcset. . . . . . . . . 33, 224, 230

Atwood, Jeff . . . . . . 155

Authoring Features . 447

av1 video codec . . . . 131

avc . . . . . . . . . . . 384, 389, 400

AVIF. . . . . . . . . . . Chapter 18: 398–415
v, xiii, xxiv, 131, 199, 283, 378, 380–381, 430–432

background image . 38, 476

Backups . . . . . . . . 348

Baldauf, Tobias . . . . 245

bandwidth. . . . . . . xix, 119, 194, 202, 261, 298, 314–316, 354, 379, 432, 461, 466
- savings . . . . . . . 45

Base64-encoded . . . 38

Bash. . . . . . . . . . . 186, 197

Bazel . . . . . . . . . . 50

British Broadcasting Company (BBC) . 313

Beamtic . . . . . . . . 148

Bendell, Colin . . . . ii, xvi, 130, 336

Better Portable Graphics (BPG). . . . . . . . 133

Blink . . . . . . . . . . 98, 188

blur filter, CSS . . . . 244, 253, 369

Book of Speed . . . . 241

Brunner, Gunther . . 239

Butteraugli . . . . . . 47, 49–51, 54–55, 133, 138–139, 142, 424, 505

Bynens, Matthias . . ii, 290, 307

C++ . . . . . . . . . . . .50

cache . . . . . . . . . . .98, 193, 256–258, 260–261, 263–274, 276, 278, 287, 373
- duration. . . . . . . .260
- enabler. . . . . . . .193
- hit ratios . . . . . . .265
- lifetime . . . . . . .266
- miss . . . . . . . . .265
- offline caching . .269-271, 274
- opportunities . . .268

Cache-Control . . . .257–260, 262–263, 467

Caching Image Assets Chapter 13: 256–289

Calibre . . . . . . . . .106–107

Camera Raw . . . . .395

Cascading Style Sheets (CSS) . . . . . . . .ix, x, xx, 33, 40, 43–44, 81, 92, 201, 204, 207–208, 210, 218, 226, 230, 246–247, 251–253, 271, 273, 277–278, 281, 297, 302, 310, 476–477, 483–485
- aspect ratio . . . . .481
- backgrounds . . . .38, 251, 288
- blur filter . . . . . .244, 253, 369
- dimensions. . . . .viii
- display. . . . . . . .208, 249
- pixel volume . . . .viii
- pseudo-selector . .252
- sprites . . . . . . . .278
- Working Group . .484

Chocolatey . . . . . .319

chroma . . . . . . . . .55, 57–58, 124–131, 159
- subsampling . . . .57, 61, 67, 87, 124–128, 130–131, 135, 178, 193–196, 432
- upsampling . . . .90
- values . . . . . . . .195

Chrome on Android 78, 458

Chrome DevTools ...
- audits panel . . . .46
- elements panel . .32, 487
- lighthouse panel .100, 236, 288, 478
- network panel . . .34-35, 188, 251, 260, 514
- performance panel 84, 93, 329, 478-479, 488, 514

CIELAB . . . . . . . . . 71, 129, 426

client-side . . . . . . . 94, 281, 477, 502

Clipping Path Zone. . 73

Cloudinary . . . . . . 46, 54–55, 59, 119, 142, 242, 333, 338, 343–346, 350, 356–357, 361–362, 392, 396, 418, 420, 437, 508

content management system (CMS) . . 340–341, 348

CodePen . . . . . . . . 325

color management
  Chapter 4: 68–79

color models . . . . . 68–69, 71, 129
  additive . . . . . . . 69
  cmyk . . . . . . . . . 69
  rgb . . . . . . . . . . 69, 71
  subtractive . . . . . 69

color profile . . . . . . 76–78, 172

color space . . . . . . 71–72, 74, 76, 78, 87, 89–90, 159, 175
  Adobe RGB . . . . . 71–73
  ProPhoto RGB . . . 72
  SRGB. . . . . . . . . 71–74, 77–78, 132, 143, 509

compression
  client-side. . . . . . 502–504
  modes . . . . . . . . 115, 398
  photographic images comparison. . . . 437
  non-photographic images comparison. . . . 438

Consistent Aspect-Ratio . . . 42

constant rate factor (CRF) . . . . . . . . 320–324

content delivery network (CDN) . . . . . . . xxiv, 192–193, 265–267, 276, 315, 415, 492
  conversion . . . . . 192
  Image CDN. . . . . Chapter 16: 337–379
  performance . . . . 358

CopyTrans. . . . . . . 395

Core Web Vitals . . . v, xxv, 456, Chapter 22: 471–489

CPU . . . . . . . . . . . xxiii, 80, 122, 141, 173, 244, 329, 331, 507
CSS Tricks . . . . . . . 277
cumulative layout shift (CLS). . . . . 307, 472–474, 479–480, 482, 487–488
cyclic redundancy code (CRC) . . . . 145
Data Saver
Chapter 21: 457–470
Data Saver mode. . . xxv, xxvi, 333, 465–467, 496–497, 499
decoding . . . . . . . . iv, xvii, 39, 42, 80–81, 83–85, 87, 89–99, 121, 131–132, 143, 178, 199, 315, 331, 344, 384, 405–406, 410, 415–418, 421, 429
asynchronous . . . 37, 95–97
attribute. . . . . . . 37, 96
decode() method . 95, 97–98, 416
developer-controlled 94
image performance
Chapter 6: 100–113
Deflate . . . . . . . . . 158, 165
Delivery Features comparison. . . . . 442–443, 452–453
Derivations . . . . . . 387
device pixel ratio. . . viii–ix, xxvi, 104, 225–227, 230
digital asset management (DAM) . . . . . . . 348
discrete cosine transform (DCT). . . . . . . . 53, 87–90, 177, 424–425
display:none . . . . . 249–252
distance threshold. . 303, 310
DOM tree . . . . . . . 82
dots per inch (DPI) . 493–494
Drasner, Sarah . . . . 79
DSSIM . . . . . . . . . 48, 54, 67, 350, 406
Edge. . . . . . . . . . . 78, 83, 96, 130–131, 151, 234, 261, 279, 299, 301, 333, 343, 354, 380, 407, 409, 457, 465
Efficient Compression Tool (ECT) . . . . . . . . 165–166

element
- img . . . . . . . . . . Chapter 1: 29–40
- picture . . . . . . . . 44, 189–190, 231–235, 249, 297, 308, 402, 431, 486, 509
- source . . . . . . . . 189–190, 231–234, 308, 323, 327–328
- video . . . . . . . . . 323, 325–328, 411, 476

Elements panel. . . . 32, 487

encoding. . . . . . . . xiv, xxi, 47, 51, 75, 85, 88, 104, 111, 132, 134, 137, 139, 171, 175, 177, 183, 194, 196, 320, 325, 331–334, 344, 371, 384, 397, 403, 407, 410–411, 415–420, 425, 429–430, 507

Entropy Coding . . . 429

Erdmann, Christoph . . . . . 128

ETag. . . . . . . . . . . 257–260

Everts, Tammy . . . . 255

EXIF. . . . . . . . . . . 117, 129, 172, 385, 389

extensible markup language (XML) 39, 136, 200, 207, 210–211, 244

Facebook. . . . . . . . 118, 120, 179, 346

fallback image . . . . 234, 284

Fetch api . . . . . . . . 269

FFmpeg . . . . . . . . 317–320, 323, 372, 396, 411–412

Figma. . . . . . . . . . 202

Firefox . . . . . . . . . 78, 96–97, 131, 188, 279, 299, 301, 312, 401, 405, 411, 485, 507

First input delay (FID) . . . . . . . . 472–473

free lossless image format (FLIF). . . . . . . . 53, 420–421

free universal image format (FUIF) . . . . . . . 418

freshness . . . . . . . 261–263, 276

Furnspace . . . . . . . 362

gamma correction. . 75

gamut . . . . . . . . . .57, 72–74, 78, 130–131, 178, 346, 399, 403, 417, 425, 432

Gao, Nancy . . . . . .460

Gaussian blur. . . . .246

Gifsicle . . . . . . . . .334

GIMP . . . . . . . . . .viii, 187, 351, 395

Giphy . . . . . . . . . .174

GitHub . . . . . . . . .139, 190, 245, 409, 411

Gmyr, Chris . . . . . .357

Google . . . . . . . . .xiii, xx, 49, 51, 131, 133, 138, 170, 175–177, 181, 187, 206, 211, 217, 238–239, 274, 278, 306, 339, 380, 394–395, 399, 405, 407, 418, 424, 491
- Core Web Vitals . .v, xxv, 456, Chapter 22: 471–489
- Cloud Platform . .374
- doodle . . . . . . . .157, 176
- Offline Cookbook 272
- Pixel . . . . . . . . .394
- Workbox Recipes 272

gradient image placeholders 246–247

graphics . . . . . . . .75, 81, 111, 133, 144, 147–148, 162–163, 223, 244, 278, 315, 417, 435
- graphics processing unit (GPU) . . . . . . .80, 244, 246
- portable network graphics (PNG) Chapter 8: 144–169
- raster. . . . . . . . .111, 162
- scalable vector graphics (SVG) Chapter 10: 200–219
- vector . . . . . . . .111, 162, 202, 518, 525

Greenstein, Ben . . .460

Group Splitting . . .428

Guetzli . . . . . . . . .51, 111, 134, 138–143, 379, 507, 509

gulp . . . . . . . . . . .123, 136, 140, 166–167, 184–185

Gumby . . . . . . . . .243

gzip . . . . . . . . . . .158, 211, 217, 265

Hansen, Patrick . . .153–154

hashing . . . . . . . . 263

high dynamic range (HDR) 130–131, 398–399, 403, 407, 417, 425, 502

High Efficiency Image File Format (HEIF) . . v, xxiv, 61, 64–65, 67, 131, 133, 356, 378, 403
- Chapter 17: 382–397
- HEIC . . . . . . . . . v, xiii, 356, 378, 382–385, 387, 389, 391–397, 403, 427
- HEIC Converter . . 393–395
- high efficiency video coding (HEVC) . . . . 133, 382, 384, 389–391, 396, 400, 403

Hidayat, Ariya . . . . 143

histogram . . . . . . . 159

.htaccess . . . . . . . . 190–192

Huffman coding algorithm . . . . . 88

International Color Consortium (ICC) 77–78, 172, 424
- color profiles. . . . 424

Image Formats
- Comparing . . . . Chapter 3: 54–67; Chapter 20: 433–453

ImageMagick. . . . . 47, 53, 122, 136, 187, 237

imagemin . . . . . . . 122–123, 136, 140, 167, 183–185, 507

ImageOptim . . . . . 43, 77, 135–136, 140, 164–165, 407, 505, 508

<img>. . . . . . . . . . iv, xxvi, 44, 98, 189, 202–204, 206–207, 224, 231–232, 235, 243, 281, 283, 285, 298, 300, 306–308, 325, 329, 332, 396, 475, 482, 486
- <img>element. . . Chapter 1: 29–40

imgix . . . . . . . . . . 243, 357, 361–362, 508

Instagram . . . . . . . 232

interlaced display . . 154

Intersection Observer ... 294–295, 298

iOS . . . . . . . . . . . .118, 120, 122, 130–131, 327, 380, 382–384, 393, 414, 495–497, 499, 503

ISO BMFF . . . . . . . .389, 403

jank . . . . . . . . . . . .91–92, 97–99, 329

JavaScript . . . . . . . .x, xx, xxiii, 81, 84, 94, 97, 201, 218, 243, 247, 249, 251–253, 260, 269, 271, 274, 279, 281, 283–285, 293, 295, 298–299, 303, 305, 311, 396, 459–461, 463, 465, 477
- libraries . . . . . . .xxiii, 274, 295, 298–299

Jetpack . . . . . . . . . .193

Jobs, Steve . . . . . . . .494

JPEG . . . . . . . . . . .chapter 7: 115–143
- 2000 . . . . . . . . . .xiii, 61, 63, 65, 67, 130, 133, 380, 398, 426, 491
- compression modes . . . . . . .115
- decoding . . . . . .131
- encoders . . . . . .58, 61, 89, 119, 134, 138, 142
- jpeg2png . . . . . .353
- standard. . . . . . .89
- XL. . . . . . . . . . . .v, xxiv, 132, 199, 254, 378, 380–381, Chapter 19: 416–432
- XR . . . . . . . . . . .xiii, 130, 380
- XT . . . . . . . . . . .428
- whitepaper . . . . .416

Kayser, Frédéric . . .126

KeyCDN . . . . . . . . .193

Kobes, Steve. . . . . .82–83, 91

Kodak. . . . . . . . . . .407

largest contentful paint (LCP) . . . .279, 282–283, 288–289, 309, 472–476, 478–479

Last-Modified . . . .257–258, 260, 268

latency . . . . . . . . . .92, 120, 122, 262, 358, 431, 491–492, 494–496, 508

lazy loading . . . . . . xii, xv, xxiii, 36, 290–291, 293–295, 297–298, 302–303, 305, 307, 310–311, 459
images, Chapter 11: 223–237

Lempel-Ziv-Welch algorithm . . . . . 158

Leptonica . . . . . . . 187

Lesinski, Kornel . . . 48, 142, 407

libaom . . . . . . . . . 410–411

libHEIF . . . . . . . . . 396

libjpeg-turbo . . . . . 58–61, 67, 119, 128, 135, 351, 428, 491

libvips . . . . . . . . . 53, 507

licensing . . . . . . . . 133, 354, 356, 360, 400

Life of a Pixel . . . . . 83, 91

Lighthouse panel . . 100, 236, 288, 478

Lightroom . . . . . . . 72, 395

limits . . . . . . . . . . 106, 427, 441, 469
comparison tables 440, 452

Linux . . . . . . . . . . 165, 319, 414, 505

Liquid Web . . . . . . 240

live photos . . . . . . . 380, 382–383, 391–392, 403

lossy files . . . . . . . 171

low quality image placeholders (LQIP) 38, 242–243, 245–246, 252–255, 295, 417, 497

luma values . . . . . . 195

LZ77 . . . . . . . . . . . 158

macOS . . . . . . . . . 31, 130, 164, 179–180, 182, 187, 319, 383, 393–394, 413–414

Magento . . . . . . . . 348

McAnlis, Colt . . . . . 160, 176

McComb, Glenn . . . 254

Median Cut . . . . . . 166

metadata . . . . . . . . 159, 164, 172, 183, 210, 334, 385–386, 389–390, 413, 507

Microsoft Paint . . . 405

MKV container . . . . 412

Moving Picture Experts Group (MPEG). . 384
mpeg-4 . . . . . . . 319–325, 328–329, 331, 383

Mozilla . . . . . . . . . 51, 135, 399, 405

mozjpeg . . . . . . . . 51–52, 58–61, 67, 111, 117, 119–120, 128, 133–138, 142–143, 186, 371, 379, 432, 505, 509

mp4box . . . . . . . . 411

natural pixels . . . . . viii

Navbharat Times . . 194

Netflix . . . . . . . . . 175, 399, 405, 407–409

Netlify . . . . . . . . . 348

Nginx. . . . . . . . . . 192, 349

Nine Degrees Below 78

Node.js . . . . . . . . . 48, 101, 183

Nokia . . . . . . . . . . 356, 384, 390, 396

Offscreen Content. . 40

opacity . . . . . . . . . 144, 148-149, 152-153

OpenGL . . . . . . . . 81

optimizing
Image Quality . . . Chapter 2: 43–53
JPEG . . . . . . . . . 134
largest contentful paint (LCP) . . . . . . . . 279
reoptimizing . . . . 352
SVG . . . . . . . . . 219
Thumbor . . . . . . 341, 361, 363–365, 367–375, 508

Optimus . . . . . . . . 193

optipng. . . . . . . . . 165–167

Orback, Vincent . . . 190, 192

padding-top hack . . 42–44

PageSpeed Insights . 46, 289, 473, 478

Parallelization . . . . 417

performance
AVIF . . . . . . . . . 405
budget . . . . . . . . 106-107
CDN . . . . . . . . . 358
impact . . . . . . . . xx, 347, 359
page load . . . . . . 40, 269
panel . . . . . . . . . 84, 93, 329, 478-479, 488, 514

perceived . . . . . . 39, 115, 117, 238, 252, 523

photographs . . . . . xxii, 115, 417, 419, 434

Photoshop . . . . . . . viii, 47, 72–73, 76, 123, 128, 144, 160, 187, 212, 351, 414

PIK . . . . . . . . . . . 418

Pingo . . . . . . . . . . 167

Pinterest . . . . . . . . 120–121, 238–239

pixel
- density. . . . . . . . viii, 33, 229–230, 493–495
- fitting . . . . . . . . 162–163
- hinting . . . . . . . 162
- Pixelmator . . . . . 187, 395
- Pixels per inch (ppi) 493

pjpeg . . . . . . . . . . 115, 118–119, 121

placeholder . . . . . . 38–40, 42, 93, 98–99, 239, 245, 247–248, 252–253, 291

Portable Network Graphics (PNG) . . . . . . . 144, 518
- Beamtic . . . . . . . 148
- Chapter 8: 144–169
- compression . . . . 158
- palette modes . . . 146
- png-8. . . . . . . . . 146–147, 150, 153, 156–157, 163, 499–502
- png-24 . . . . . . . . 146–148, 153–154, 156, 163, 500, 502
- png-32 . . . . . . . . 146, 148, 153–154, 156, 500, 502
- posterizer . . . . . . 166

PNGcrush . . . . . . . 164, 166–167

PNG Optimization Tools . . . . . . . . 164
- pngOptimizer . . . 168

PNGout . . . . . . . . 164, 168–169

PNGquant. . . . . . . 164, 168, 371, 507, 509

PNGwolf. . . . . . . . 169

polyfill . . . . . . . . . 310–311, 344

Portis, Eric. . . . . . . 142

Posterization . . . . . 160–161
Potts . . . . . . . . . . 313
Preloading. . . . . . . 256, 279–281, 284, 288
progressive . . . . . . v, 40, 98, 115–118, 120–124, 131, 135, 154–155, 222, 270, 293, 337, 362, 415–417, 420–421, 430–431, 496
  decoding . . . . . . 178, 417
  enhancement . . . 401–402, 431, 463
  image rendering . 155, 239, 252, 255
  jpeg. . . . . . . . . . xv, 115, 117-118, 123, 241, 491-492, 505
  png . . . . . . . . . . 155
  scans . . . . . . . . . 421
  rendering techniques Chapter 12: 238–255
ProPhoto RGB . . . . 71–74
Proxying pages. . . . 459
Peak signal-to-noise ratio (PSNR). . . . . . . 54
quality index . . . . . 44, 47, 321
quantization . . . . . 52–53, 57–58, 88, 135–136, 353, 424, 427–428
QuickLook. . . . . . . 180, 413
Quick Sync Video . . 331
RGB
  color model. . . . . 69, 71
  transparency. . . . 172
raster . . . . . . . . . . 83, 111, 144, 161–162, 201, 212–214, 218
  rasterization . . . . 82–85
real user monitoring (RUM) . . . . . . . 466, 488
recompression . . . . 52, 352–353, 424
responsive
  breakpoints. . . . . 235
  design . . . . . . . . xxii, 203, 420
  images. . . . . . . . Chapter 11: 223–237
  image techniques 234
Retina . . . . . . . . . ix, 223, 230, 494–495
  density. . . . . . . . 495
  display. . . . . . . . 494

revving . . . . . . . . .263, 267

Rigor . . . . . . . . . .334

Rogers, Philip. . . . .82–83, 91

royalty-free . . . . . .381, 398, 403, 418, 449
  comparison. . . . .448, 452

Safari . . . . . . . . . .36, 78, 97, 130–131, 241, 279, 301, 312, 332, 380, 396, 485

scalable vector graphics (SVG) Chapter 10: 200–219

screen pixels . . . . .viii

security . . . . . . . .53, 204, 340–341, 345, 373–374

sequences . . . . . . .131, 386, 389, 403

service workers . . .269–271, 274–276, 461

Shopee . . . . . . . . .98–99

Shopify. . . . . . . . .468

Sketch . . . . . . . . .43, 124, 144, 187, 202, 212, 219

Sneyers, Jon. . . . . .ii, 52, 416, 420, 422–423, 431, 437

SOASTA . . . . . . . .xx

Software Support . .
  comparison. . . . .450,452

speed . . . . . . . . . .
  comparison. . . . .438, 452

SpeedCurve . . . . . .106–107

spriting. . . . . . . . .277–279

SQIP . . . . . . . . . .38, 244–246, 252–255, 295

Squoosh . . . . . . . .43, 164, 181, 197, 410, 412, 430

srcset . . . . . . . . . .32–33, 36–38, 93, 189, 203, 224–225, 227, 229–236, 249, 297, 308, 402, 431–432, 486, 508

SRGB . . . . . . . . . .71–74, 77–78, 132, 143, 509

SSIM . . . . . . . . . .47–49, 51, 133, 137, 143, 350, 355, 505
  SSIMulacra. . . . .54–55, 59, 67

stale-while-revalidate (SWR) . . . . . . .262–263

Stefanov, Stoyan . . .241, 247

SVG
Chapter 10: 200–220
icons . . . . . . . . . .205–206, 219, 270
maps . . . . . . . . . .207, 209
svgo . . . . . . . . . . 186, 212, 214–216, 507
svgomg . . . . . . .212–213

Thumbor. . . . . . . . .341, 361, 363–365, 367–375, 508

TIFF. . . . . . . . . . . .53

Tinder . . . . . . . . . .469

transitional features
comparison. . . . .448, 452

transparency . . . . .55, 67, 144, 159, 163, 166, 168–170, 172–173, 184, 199, 344–345, 382, 388, 390, 499, 501
alpha . . . . . . . . . .148, 152, 154
support . . . . . . .131, 149, 168, 380, 399

Tribune. . . . . . . . .470

Tumblr. . . . . . . . . .174

Twitter . . . . . . . . . .v, xxv, xxvi, 93–94, 120, 174, 456

Image Pipeline . . Chapter 23: 490–504
Lite . . . . . . . . . . .93, 496–497

URL . . . . . . . . . . .39, 101, 180, 193, 205, 251, 263–264, 266–267, 278, 325, 344, 364–365, 368–369, 476

Vary header . . . . . .264

vector. . . . . . . . . .161–162, 200–202, 214, 218
graphics. . . . . . .111, 162, 202

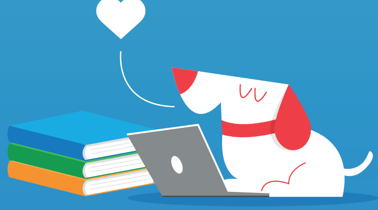
versioning. . . . . . .263, 266–267

video . . . . . . . . . .vi–xi, xv, 65, 75, 92, 125, 133, 171, 242, 278, 314–328, 331–335, 342, 346, 356, 359, 371–372, 382, 384, 389, 392, 396, 398, 400–401, 411, 426, 462, 476, 481
autoplay. . . . . . . 326-327, 469, 499
decoding . . . . . .131, 331
Quick Sync Video 331

VLC media player . . 405, 413
VoxMedia . . . . . . . 175
W3C. . . . . . . . . . . 202
Wagner, Jeremy . . . ii, 114, 186
Web Almanac. . . . . vi, ix, xii
WebM . . . . . . . . . 317–319, 322–325, 328–329, 331, 507
WebP . . . . . . . . . .
  browser support. . 179
  Chapter 9: 170–199
  encoding . . . . . . 175
  lossless . . . . . . . 172
  lossy files . . . . . . 171
  serving . . . . . . . 170, 175, 188, 192-193
WebPageTest . . . . . 45–46, 281, 287, 347, 491
Web Performance Calendar. . . . . . 405
WebPShop. . . . . . . 187
Website Speed Test . 46
WhatDoesMySiteCost.com . . . . . . . . . . . . x
wide color gamut (WCG) 131, 399, 425
width descriptor . . . 229–230
Wikipedia . . . . . . . 287, 341
Wiltzius . . . . . . . . 91
WordPress. . . . . . . 193, 348–349
Workbox Recipes . . 272
XnConvert. . . . . . . 182–183, 186
YCbCr. . . . . . . . . . 87, 90, 129, 178, 426
Yelp . . . . . . . . . . . 58, 120
zopfli . . . . . . . . . . 158, 169
ZopfliPNG. . . . . . . 165, 169

# Smashing Library

Expert authors & timely topics
for **Smashing Readers.**



smashed.by/library

## More Smashing Books

Crafted with care for you, and for the Web

**TypeScript in 50 Lessons**

by Stefan Baumgartner

**Inclusive Components**

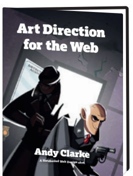
by Heydon Pickering

**The Ethical Design Handbook**

by Trine Falbe,
Martin Michael Frederiksen
and Kim Andersen

**Form Design Patterns**

by Adam Silver




**Art Direction for the Web**

by Andy Clarke

**Click!**
**How to Encourage Clicks Without Shady Tricks**

by Paul Boag

The world is a miracle. So are you.

**Thanks for being smashing.**

"An incredibly comprehensive overview of image optimization. This book will teach you everything you need to know about delivering effective and performant images on the web."

—*Katie Hempenius, Google*

"Optimizing image delivery is key to building high-performance web apps. This book explains everything developers should know about choosing the right image format, compressing image assets — and more!"

—*Mathias Bynens, Google*

"Images are the heart and soul of the web; they help create that emotional connection with humans. Yet, it is really easy to ruin that experience through slow loading or worse, over quantizing the pixels and distorting images. Understanding how images work is essential for every engineer; the last thing we want is to deal with open bugs from bad creative or performance experiences."

—*Colin Bendell, Shopify*

Addy Osmani is an engineering manager working on Google Chrome. His team focuses on speed, helping keep the web fast. Devoted to the open-source community, Addy's past open-source contributions include Lighthouse, Workbox, Yeoman, Critical, and TodoMVC.