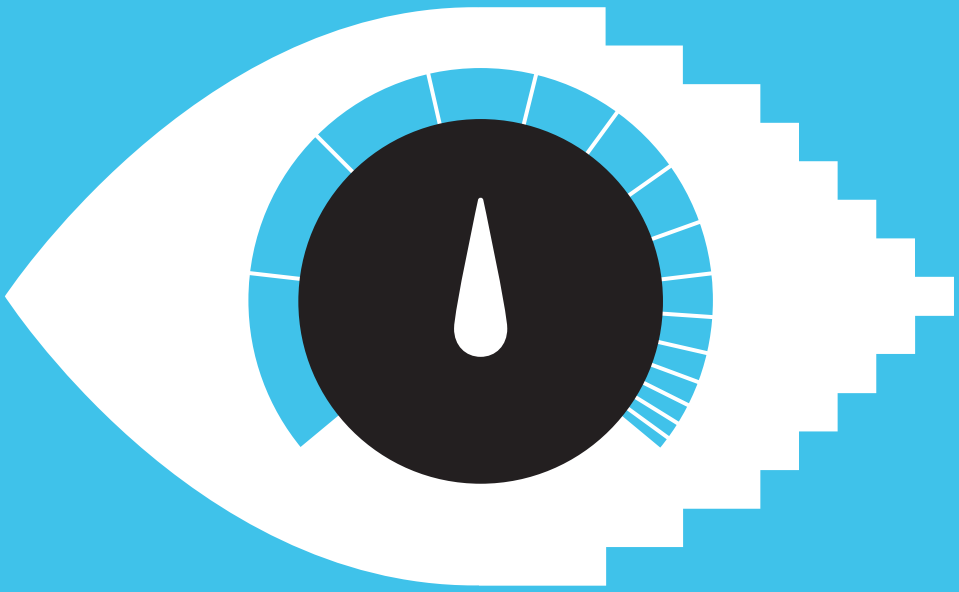


Image Optimization

by Addy Osmani

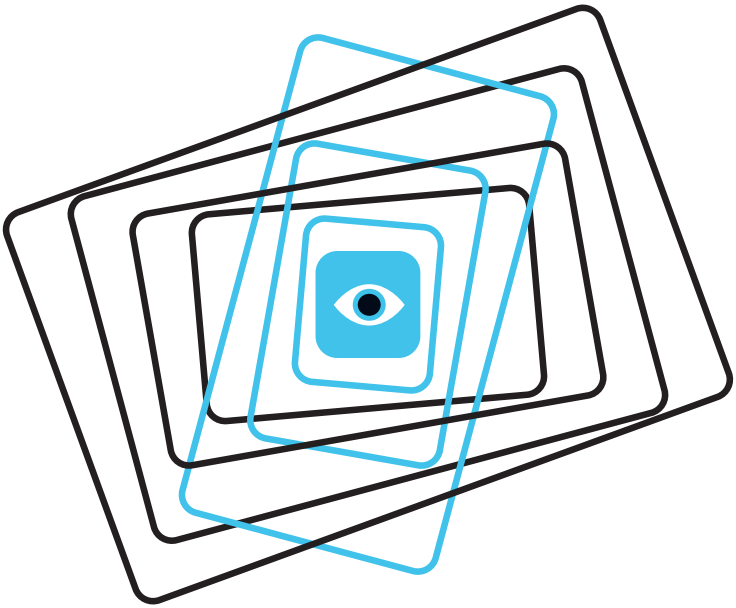


Image Optimization

by
Addy Osmani

Published 2021 by Smashing Media AG, Freiburg, Germany.

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

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

75%	The speed it takes to load the page
66%	How easy it is to find what I'm looking for
61%	How well the site fits my screen
58%	How simple the site is to use
25%	How attractive the site looks

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:

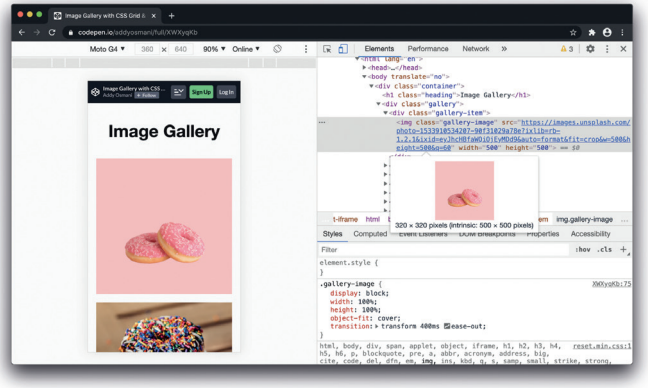
```
<img src="donut.jpg"
     alt="A delicious pink donut.">
```

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">
```

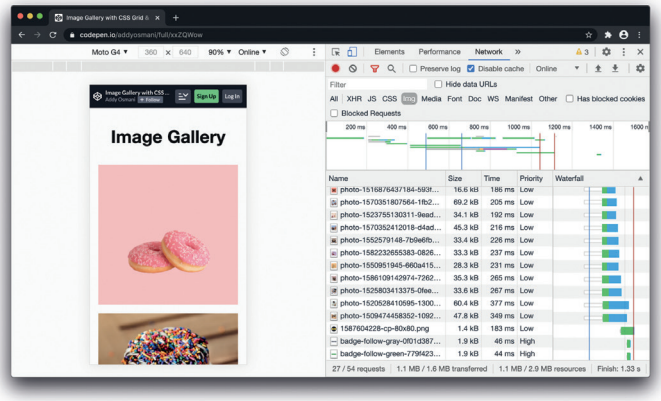
The `srcset` attribute defines the set of images the browser can select from, as well as the size of each image. Each image string is separated by a comma and includes: a source filename (`donut-400w.jpg`); a space; and the image's intrinsic width specified in pixels (`400w`), or a pixel density descriptor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



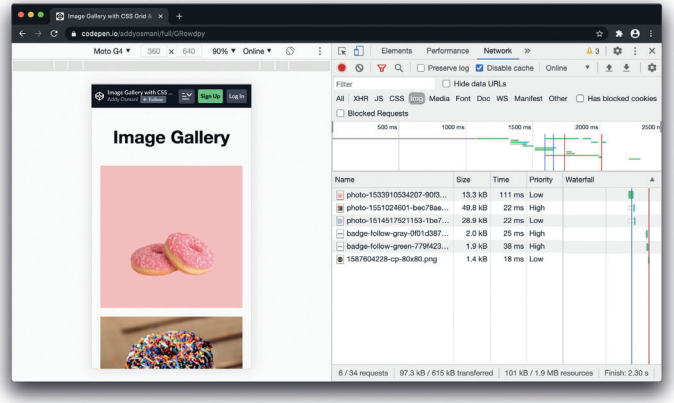
An image gallery eagerly loading all the images it needs up front, as shown in the Chrome DevTools ***Network*** *panel. 1.1 MB of images have been downloaded, despite only a small number being visible when the user first lands on the page.*

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.

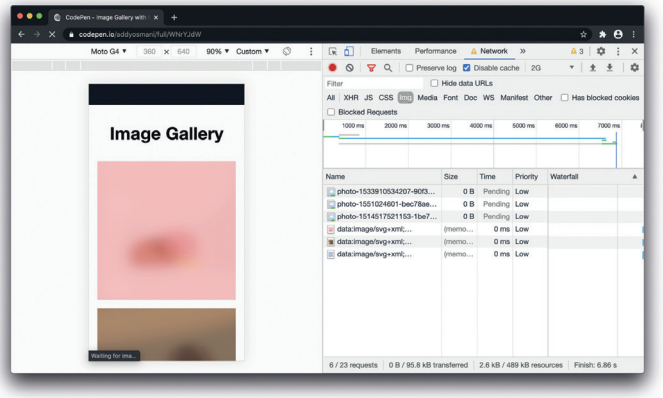
Placeholders are often inline, Base64-encoded data URLs which are low-quality image placeholders (LQIP) or SVG image placeholders (SQIP). This allows users to get a very quick preview of the image, even on slow network connections, before the sharper final image loads in to replace it.

```
<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([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.

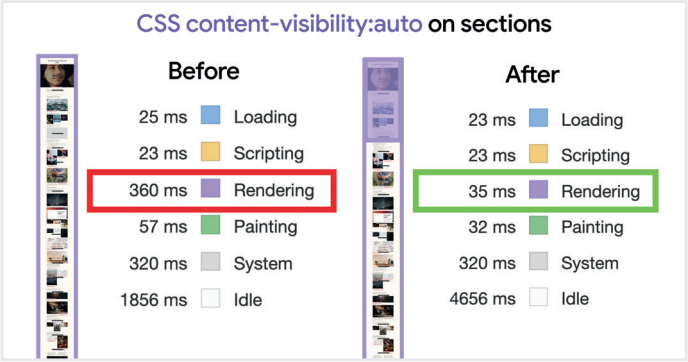
Lazy-Render Offscreen Content

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:

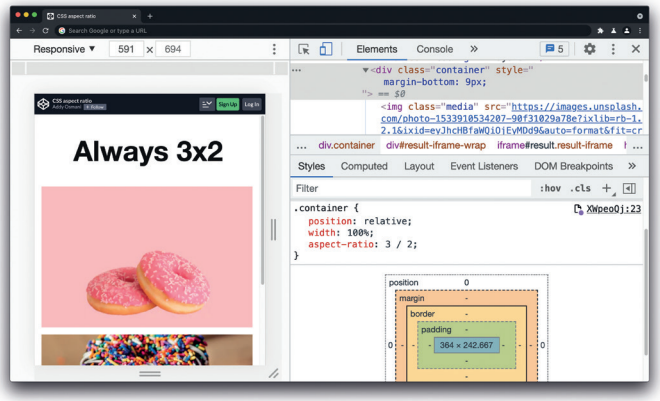
```
<div class='container'>
     <img style='position: absolute; top:0;'>
</div>
```

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.

Index

SMASHING MAGAZINE

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

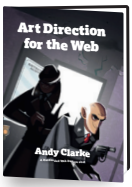
by Heydon Pickering

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.
SMASHING
MAGAZINE

"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.

SMASHING MAGAZINE

9 783945 749944