6 Going further with images

This chapter covers

- Creating image sprites from multiple image files by using automated tools
- Reducing the file size of images without significantly degrading their visual quality
- Using the WebP image format from Google, and understanding how it compares to older formats
- Deferring the loading (lazy loading) of images that aren't in the viewport

In the preceding chapter, you learned the importance of optimal image delivery. This entailed using media queries to deliver images in CSS according to the user's device capabilities, as well as using new features in HTML to accomplish the same goal.

In this chapter, you'll go a step further in working with images. This entails reducing HTTP requests by combining images into sprites, reducing the size of raster and vector images through new compression methods, using Google's WebP image format, and understanding the benefits of lazy loading images.

6.1 Using image sprites

As a front-end developer, you're constantly looking for ways to improve your site's performance. With images representing a large portion of your page weight, it makes sense to want to tame these unruly critters and make them more manageable.

> **Warning: This section discusses an HTTP/2 antipattern!**
> Image sprites combine images to reduce HTTP requests, which is a form of concatenation. Although you *should* use image sprites on HTTP/1 to improve page-load times, you should *avoid* using them on HTTP/2. Check out chapter 11 for more information.

Surely you've noticed the iconography peppered throughout the sites you visit: images such as stars for ratings, social media icons, action icons that encourage the user to share content, and so on. Chances are good that these images are part of what's called an *image sprite*.

So, what *is* an image sprite? An image sprite is a collection of previously separate image files used throughout a website that have been assembled into one image file. These images are often global elements such as icons. Figure 6.1 shows an example image sprite.

Once created, an image sprite is referenced by the CSS `background-image` property, and manipulated by the `background-position` property to reveal only the relevant portion of that image within an element. The element's bounding box excludes the rest of the sprite from view, giving the impression that the element is displaying only a single image in the background.

The benefit is that you're taking what would normally be a larger number of images and reducing them to a single image. This results in more-efficient delivery of those resources and reduces the load time of the page by opening fewer connections to the web server.

In this section, you'll create an image sprite for a recipe website that has six SVG icons. Four are social media icons, and two are star icons used to represent recipe ratings. Using a command-line utility, you'll generate a sprite and the CSS necessary to use it. You'll then place this CSS into the project and replace all of the icons with the new sprite, to bring the request count of the page down from 25 to 20, and a load time of roughly 500 ms (using Chrome's Good 3G throttling profile) for these icons down to about 90 ms. After you're finished, you'll create a PNG fallback for older browsers that don't support SVG.

Figure 6.1 An image sprite of various social media icons

6.1.1 Getting up and running

Before you create the sprite, you need to download a utility to generate it. Then you'll download the website code with `git`. You can install the sprite generator with this command:

```
npm install -g svg-sprite
```

After this finishes installing, download the website code and run it on your local machine with the following commands in a folder of your choosing:

```
git clone https://github.com/webopt/ch6-sprites.git
cd ch6-sprites
npm install
node http.js
```

This launches a web server running the recipe website at http://localhost:8080.

> **Want to skip ahead?**
>
> If you want to skip ahead to see how the image sprite was generated and implemented in this section, you can enter `git checkout svg-sprite -f`. As always, be aware that you will lose any changes that you may have in your local repository.

Let's get started!

6.1.2 Generating the image sprite

In the img folder is a subfolder named icon-images. This contains the six separate SVG images that you'll combine into a sprite. Table 6.1 details these images.

Table 6.1 SVG icons in the recipe website that you'll combine into an image sprite

Image name	Image function	Image size (bytes)
icon_facebook.svg	Facebook icon	600
icon_google-plus.svg	Google Plus icon	938
icon_pinterest.svg	Pinterest icon	563
icon_star-off.svg	Rating star (inactive)	299
icon_star-on.svg	Rating star (active)	302
icon_twitter.svg	Twitter icon	759

These images represent six requests. By the end of this section, you'll have whittled this down to one. To generate the sprite, you'll use the `svg-sprite` command as follows from the root of the recipe website folder:

```
svg-sprite --css --css-render-less --css-dest=less
           --css-sprite=../img/icons.svg
           --css-layout=diagonal img/icon-images/*.svg
```

A lot is going on here, so let's go through each argument, diagrammed in figure 6.2.

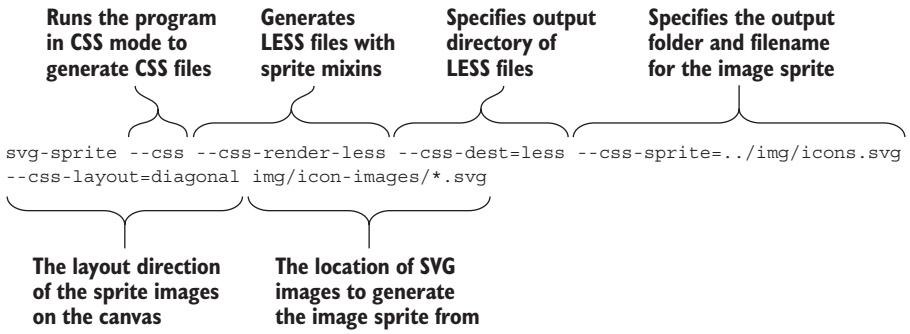


Figure 6.2 The anatomy of the `svg-sprite` command as used to generate an SVG sprite with LESS mixins

When this command finishes, the generated sprite will be in the img folder, and a new LESS file named sprite.less will be in the less folder. The generated sprite should look like figure 6.3.

Image sprites can be used for more than icons (although this is a common use of the technique). You can sprite other elements such as nonrepeatable backgrounds, button images, or other imagery that's not content-specific. With this step completed, you'll continue by updating the recipe website's CSS to use the generated sprite.

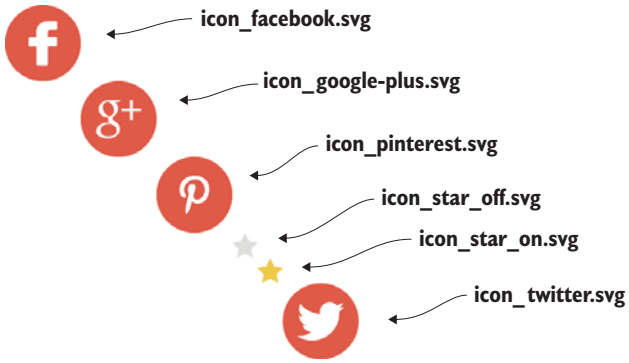


Figure 6.3 The newly generated image sprite with annotations showing the names of the standalone files prior to being added to the sprite

6.1.3 Using the generated sprite

The next step is to include the generated sprite.less file into main.less, both of which are in the less folder. At the beginning of main.less, add this line:

```
@import "sprite.less";
```

This adds LESS mixins for the sprite to the CSS. From here, you can replace the individual image icon references with these LESS mixins to put the sprite to use. You need to make six changes. What you'll do is search for the string `.svg` in your text editor, and replace each `background-image` reference with the corresponding LESS mixins shown in table 6.2.

Table 6.2 Icon images and the LESS mixins needed to replace them

Image name	LESS mixin
icon_facebook.svg	`.svg-icon_facebook;`
icon_google-plus.svg	`.svg-icon_google-plus;`
icon_pinterest.svg	`.svg-icon_pinterest;`
icon_star-off.svg	`.svg-icon_star-off;`
icon_star-on.svg	`.svg-icon_star-on;`
icon_twitter.svg	`.svg-icon_twitter;`

To help you with this, I'll walk you through replacing one of the images with the new sprite. Take the Facebook icon image in the first row of table 6.2, and do the following:

1. Search for icon_facebook.svg in global_small.less in your text editor, and replace the line it appears on with the `.svg-icon_facebook` mixin. In this example, the line you're replacing looks like this:

   ```
   background-image: url("../img/icon-images/icon_facebook.svg");
   ```

   Replace this line with the `.svg-icon_facebook` mixin:

   ```
   .svg-icon_facebook;
   ```

2. Compile the LESS files. On UNIX-like systems, run less.sh. On Windows, run less.bat.

After completing these steps, reload the page. You'll notice that the Facebook icon looks the same as it did which is exactly what you want. Now look to see that the image sprite is in use by inspecting the image element in the browser's developer tools, and check its CSS.

At this point, it's a matter of repeating this process for the rest of the images listed in table 6.2. When finished, you'll have reduced the total number of requests from 25 to 20, and the load time for the sprite assets down from approximately 500 ms to about 90 ms.

Although the gains aren't as pronounced in this instance with just six icons, the positive effects on performance scale up as the number of images added to the sprite increase. A good example of image sprites at work is with Facebook, which uses image sprites to serve many images, such as the icons used through the site, button images, backgrounds, and so forth. If all of these icons were served separately, performance could be diminished.

6.1.4 Considerations for image sprites

So far, you've learned how to create sprites by using an SVG sprite generator. But you should keep some considerations in mind when creating them.

As I said earlier in this section, sprites are used to combine global visual elements of a page, such as iconography. Creating sprites for content-specific images isn't a fruitful endeavor. Content-specific images are usually relegated to the page they're relevant to, whereas global images appear on every page on the site. Creating sprites that contain content-specific imagery penalizes users by forcing them to download content for pages that may not use it. Figure 6.4 shows the recipe website for which

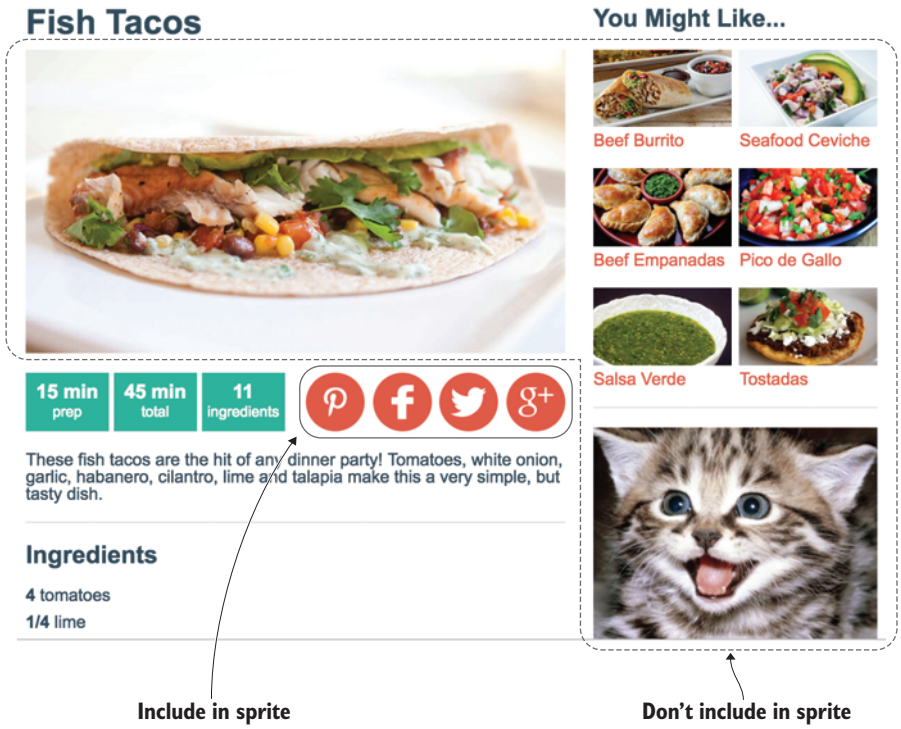


Figure 6.4 An overview of images on the recipe website that are or aren't candidates for inclusion in an image sprite. Iconography is marked for inclusion, whereas imagery such as recipe images and ads isn't.

you created the image sprite in this section, and annotates the images that are suitable for inclusion in sprites.

You shouldn't be too strict as far as deciding what constitutes *global* imagery. Some iconography may not be used on all pages, but these kinds of images should still be included, because they add little to the sprite's file size. Loading them up front speeds the loading of subsequent pages, because the images will be in the browser cache by the time they're used. Each scenario is unique, so do an inventory of images and create a sprite that best fits your website.

6.1.5 Falling back to raster image sprites with Grumpicon

Earlier in this section, you created an SVG sprite by using a command-line utility. Most of the time when you're creating sprites, the images you're working with (icons and so forth) are good candidates for SVG, because they tend to comport with the capabilities of the format.

Although most browsers have SVG support, it may be necessary to specify a fallback to a more traditional format that's more widely supported. That's where Grumpicon comes in handy.

Grumpicon is a web-based tool that accepts SVG files and generates a PNG version of the sprite with fallback options. What you're interested in is converting your icons.svg sprite to a PNG version for older browsers. To get started, you'll switch to a new branch of code with the following command:

```
git checkout -f png-fallback
```

After the new code is downloaded to your computer, head over to http://grumpicon.com and upload icons.svg from the img folder. You can do this by browsing to the file on your computer, or by dragging and dropping the file on the Grumpicon beast shown in figure 6.5.

After you upload icons.svg, a zip file automatically begins downloading. Open this file and go to the png folder inside it. A file named icons.png is there. Copy this file to the img folder of the recipe website.

From here, you'll tweak the LESS mixin in sprite.less to fall back to this file in the event that SVGs aren't supported. The way you achieve this fallback is by using a cascade of `background-image` declarations. Open sprite.less in your text editor and find the `.svg-common()` mixin on line 1. Change the content of this mixin to what you see in this listing.

Figure 6.5 SVG files can be converted to PNG by dragging and dropping SVG files on the Grumpicon beast (or by browsing to them).

Listing 6.1 Fallback to PNG for browsers without SVG support

```
.svg-common(){
    background: url("../img/icons.png") no-repeat;
    background: none, url("../img/icons.svg");
}
```

The fallback to the PNG version is specified first.

A multiple background image reference with an SVG fallback

This code does two things: In the first `background-image` declaration, you specify your fallback to `icons.png`. On the next line, you specify multiple backgrounds, the last of which is your SVG image sprite. After you've made the change to incorporate the fallback image, recompile your LESS files by running `less.sh` (or `less.bat` for Windows machines).

This fallback works because older browsers will read the first `background` property and apply it to the page. When an older browser attempts to interpret the second `background` property, it will fail because older browsers can't parse multiple backgrounds. These less-capable browsers will then default to the initial reference to icons.png. More-capable browsers will pick up and use icons.svg just fine, and ignore the icons.png file. This works because the browser will anticipate that icons.png isn't needed because it's overridden by the reference to icons.svg, and the browser will choose to not download the PNG file.

Now that you understand image sprites, their benefits, and how to create them for all browsers, you can learn how to reduce the file size of your images.

6.2 Reducing images

Imagine a client who has a website with a recipe collections page that's heavy with image content. This client has noticed that on all devices, this page takes a long time to load, even though the images are responsive. These pages are strong when it comes to driving traffic to other pages on the site, but the client knows that if you can cut the load time of this page, you may be able to coax the client's more-impatient users further into the site.

> **Reduce the size of your images automatically!**
> This section teaches you how to write Node scripts that will perform bulk optimizations on images in an example project. If you're interested in automating the techniques taught in this section, check out chapter 12 on using gulp.

That's where *image reduction* comes in. This process reduces the file size of images without significantly degrading their visual quality. Many image-editing programs don't produce output that's optimal for the web. A good example is Photoshop's ironically named Save for Web dialog box. Although Save for Web has useful presets and options, it doesn't quite compare to what's possible with modern image-reduction algorithms.

To get started, you need to download the client's recipe website and get it running on your machine with these commands:

```
git clone https://github.com/webopt/ch6-image-reduction.git
cd ch6-image-reduction
npm install
node http.js
```

Next, browse to http://localhost:8080 and you should see the client's recipe website, as shown in figure 6.6.

With the website up and running, you'll optimize JPEG images by using a Node program called `imagemin`. Then, you'll go further and learn how to use it to optimize PNG images. Finally, you'll learn how to use the `svgo` Node program to optimize an SVG image.

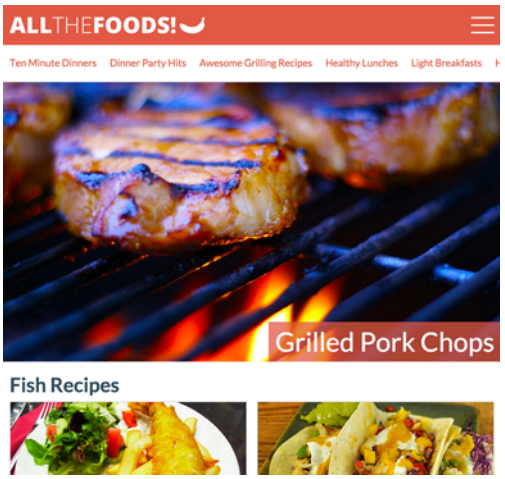


Figure 6.6 The client's recipe website as it appears in the tablet breakpoint

6.2.1 *Reducing raster images with imagemin*

The tool of choice for optimizing images on this site is `imagemin`, which is a generalized image-optimization module written in Node. It's capable of optimizing all types of images used on the web. In this section, you'll write a small Node program that uses `imagemin` to optimize all the JPEG images in the recipe website's img folder.

> **Mind your role**
>
> Maybe this goes without saying, but whether you apply these techniques to the sites you work on depends on the responsibilities that come with your role. If you play the part of both designer and developer, you have more latitude in the choices you can make. But if your organization's roles are more granular, and your responsibilities are relegated to development, loop in the project's designer on optimizations you want to make.

Unlike the work you've done up to this point, this tool requires you to write a little bit of JavaScript for Node, rather than running command-line tools installed by `npm`. Don't worry, though! The process is simple, and I'll walk you through it step-by-step.

OPTIMIZING JPEG IMAGES

Before you start, let's take an inventory of the project's images. If you peek in the site's img folder, you'll see 34 recipe images ending with either -1x.jpg or -2x.jpg. The comes out to 17 pairs of JPEGs. You might have correctly guessed that these signify

images intended for standard DPI and high DPI screens, respectively. This page uses the `srcset` attribute you learned about in chapter 5 to deliver images according to a device's capabilities.

You don't have a single baseline for how much of your page weight is represented by images, but rather two. It changes based on the type of screen and the device resolution. Table 6.3 lists screen DPI, total image payload, and the corresponding load time on the Good 3G network throttling profile in Chrome's network utility for the website.

Table 6.3 Screen DPI as it relates to the size of images and the total load time of the page

Screen DPI	Image payload	Load time
High	2089 KB	11.5 seconds
Standard	732 KB	4.38 seconds

Although devices with standard screens load the site in a somewhat expedient fashion, high DPI devices are definitely suffering. Let's give them a shot in the arm with our pal `imagemin`. To install `imagemin` for this project, run the following commands:

```
npm install imagemin imagemin-jpeg-recompress
mkdir optimg
```

The first command installs two packages: the `imagemin` module, and a JPEG optimization plugin for `imagemin` named `jpeg-recompress`. The second command creates a new folder named optimg that the `imagemin` code will write the optimized images to. After these are installed, you'll write a small Node program that does the work. In the root folder of the website, create a new file named reduce.js and add the following contents.

Listing 6.2 Using `imagemin` to optimize all JPEGs in a folder

```
var imagemin = require("imagemin"),
    jpegRecompress = require("imagemin-jpeg-recompress");

imagemin(["img/*.jpg"], "optimg", {
    plugins: [
        jpegRecompress({
            accurate: true,
            max: 70
        })
    ]
});
```

Import the imagemin Node package.

Import the jpeg-recompress plugin for imagemin.

Tell jpeg-recompress to favor accuracy over speed.

Create an imagemin object that reads and optimizes JPEGs and writes the output.

Set the maximum JPEG quality of the output image.

The preceding code is simple: The `require` statements import the `imagemin` modules. You use these modules to create an `imagemin` object that processes all of the JPEGs in

img and writes the optimized output to `optimg`. When you're finished, save reduce.js and run it with Node:

```
node reduce.js
```

This can take a little time. On my laptop, this command took about 10 to 15 seconds. Setting the `accurate` flag in listing 6.2 to `false` can cut processing time if the script is taking too long.

After the program finishes, you'll see that the optimg folder is populated with optimized images. Let's look at the unoptimized and optimized output of chicken-tacos-2x.jpg in each folder. Figure 6.7 depicts the two side by side.

Figure 6.7 A comparison of the unoptimized (left) and optimized versions of chicken-tacos-2x.jpg. The optimized version is about 55% smaller, but the visual differences are virtually imperceptible.

With the JPEGs optimized, you need to point index.html to them. To do this, you can copy and paste images from the optimg folder to the img folder, and choose to overwrite for all conflicts. This method involves no changes to index.html. If you want to preserve the unoptimized files, you can change the `<img>` tag references to point to files in the optimg folder.

After you open the page and check its load time in Chrome's Network tab, you'll notice that the load speed of the page should improve drastically, while none of the images should look much, if at all, different from their unoptimized versions.

Because of your `imagemin` script, you've achieved a 59% reduction of file sizes for images in both categories. The performance improvements are noted in figure 6.8, which represent about a 50% reduction in page-load times.

With these results, you've achieved far beyond your client's expectations. All but the most impatient users ought to be satisfied with the site's improved performance, which offers them unimpeded access to more recipes through this content portal.

It's possible to further tweak the output of this program by adding and tweaking `imagemin-jpeg-recompress` options in reduce.js. All options are documented at the `imagemin-jpeg-recompress` `npm` package page at www.npmjs.com/package/imagemin-jpeg-recompress. Be aware that aggressive optimizations can cause noticeable degradation

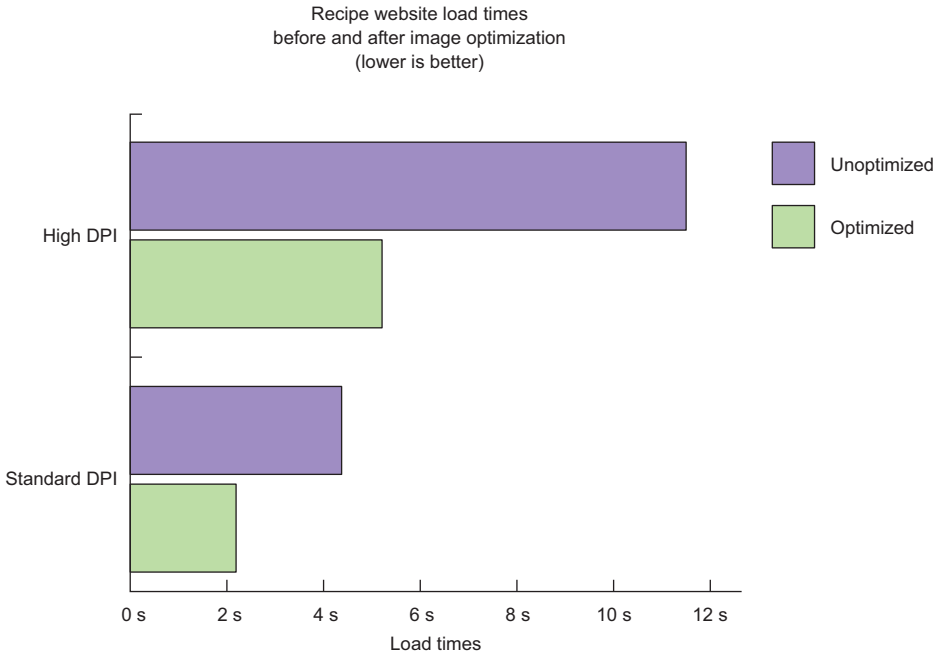


Figure 6.8 Website load times before and after the optimization of images for the recipe website using the Good 3G networking throttling profile in Google Chrome

of images, so always compare the optimized output with the unoptimized input to ensure that the results are up to your standards.

Additionally, `imagemin-jpeg-recompress` isn't the only JPEG optimization library out there. You can learn more about many others at www.npmjs.com/browse/keyword/imageminplugin.

Optimizing PNG images

Optimization of PNGs with `imagemin` is largely the same as optimizing JPEGs, but it's beneficial to get hands-on with optimizing these image types as well. To get started, you'll pull down a new branch of code by entering this command:

```
git checkout -f pngopt
```

Other than bringing in the optimized JPEGs from earlier in this section, the only thing this command changes is that the site logo is swapped out from an SVG to a PNG image set. Two PNG files are added, logo.png for standard DPI screens and logo-2x.png for high DPI screens, which are 4.81 KB and 8.83 KB, respectively. To start optimizing, you'll need to download the `imagemin-optipng` plugin with this command:

```
npm install imagemin-optipng
```

After installation finishes, open reduce.js and change it to the content in this listing.

Listing 6.3 Using `imagemin` to optimize PNGs

```
var imagemin = require("imagemin"),
    optipng = require("imagemin-optipng");

imagemin(["img/*.png"], "optimg", {
    plugins: [optipng()]
});
```

Imports the optipng plugin for imagemin.

Imports the optipng plugin, processes PNGs in the img directory, and writes the output to the optimg folder.

This code works similarly to the JPEG optimizer, except you're processing PNG files instead. To test it, run reduce.js with `node`, like so:

```
node reduce.js
```

You should then see the optimized PNG files in the optimg directory. Figure 6.9 shows the file sizes before and after the optimizer runs.

Because of your optimizations, the file sizes of logo.png and logo-2x.png are reduced by about 33% to 37%, respectively. The visual quality of the images didn't suffer in the process.

It's possible to coax more out of this program by using the `optimizationLevel` option in the `imagemin-optipng` plugin. This option is the only one available in this plugin, and accepts an integer from `0` to `7`. Higher values *can* further reduce file sizes. After a certain point, though, this will fail to yield better results. The default value for

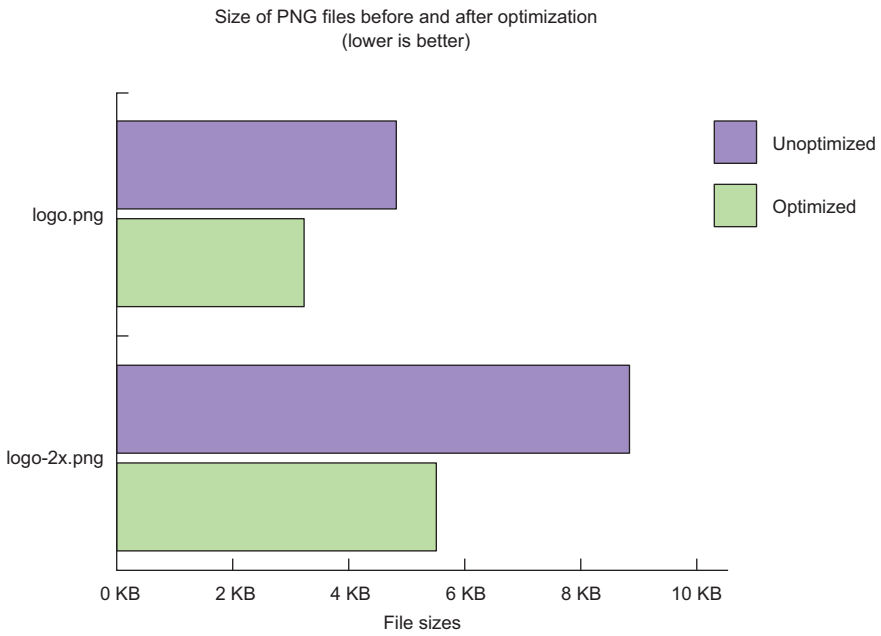


Figure 6.9 A comparison of the logo.png and logo-2x.png files before and after optimization

optimizationLevel is 2, and even when raised to the maximum value of 7 in this instance, no more gains were realized than with the default. Different images may yield different results, so experiment to see what's possible.

Other PNG optimization plugins are available at www.npmjs.com/browse/keyword/imageminplugin.

6.2.2 Optimizing SVG images

The mechanism behind optimizing SVG images is a bit different than it is with raster images. The reason behind this is that whereas raster images are binary files, SVG files are text files—so optimizations such as minification and server compression can be used on them.

The client is happy with the work you've done and has let you off the hook. Your colleague, sensing that you have time to kill, emails you about an SVG logo she's designed for her client, a timber and pulp company named Weekly Timber. She has designed a great logo, but it's 40 KB. She'd like to see if anything can be done to trim it down.

Fortunately for you, a command-line tool in Node named svgo is also available as an imagemin plugin. Because you're optimizing one file, the command-line tool will be more convenient than writing a JavaScript routine. To install svgo on your system, use npm:

```
npm install -g svgo
```

This installs svgo globally on your system so that you can use it anywhere. Then you need to grab the SVG at http://jlwagner.net/webopt/ch06/weekly-timber.svg. After it's downloaded, go to the folder it resides in at your terminal and try the following command:

```
svgo -o weekly-timber-opt.svg weekly-timber.svg
```

The format of this command is simple. It starts with the -o parameter, which is the name of the file that svgo will write the optimized output to. After that is the name of the unoptimized SVG file. When you run this command, you'll receive this output:

```
39.998 KiB - 28.4% = 28.656 KiB
```

Not too bad! Turns out svgo's default behavior optimizes a lot by simplifying the SVG's content as well as minifying it. This yields about a 28% savings compared to the original. Let's see how this impacts image quality by opening the optimized and unoptimized versions in the browser, and comparing the two. You can see this comparison in figure 6.10.

The image quality is virtually unaffected. You can open both files in a program such as Photoshop or Illustrator and compare. If you don't have imaging software, you can open SVGs in any modern browser. You shouldn't notice much, if any, difference between the two. If you do notice any differences, they should be minor. Aggressively

Unoptimized (39.99 KB) Optimized (28.66 KB)

Figure 6.10 The Weekly Timber logo before (left) and after optimization with `svgo` using the default options

optimized SVG images are characterized by a lack of fine details, particularly in the quality of Bézier curves.

`svgo` is a powerful program with a lot of options. Maybe we should dive in to see if you can further optimize this image. Type `svgo -h` to see additional options. One that sticks out is the `-p` argument, which you can use to control the precision of floating-point numbers. Try setting this value to `1`, and see what the output looks like with this command:

```
svgo -p 1 -o weekly-timber-opt.svg weekly-timber.svg
```

When this command runs, you should see the following output:

```
39.998 KiB - 53.9% = 18.42 KiB
```

This yields another 25%! Let's not get too hasty in declaring victory, though. You should observe the output to see whether any anomalies have been introduced. Figure 6.11 compares the output of the original unoptimized image and your further optimized version.

You can observe *some* noticeable differences by inspecting these images more closely, but they're still relatively minor. You *can* go too far, though. Figure 6.12 provides a closer look at what happens when you remove all precision from the same unoptimized SVG.

Unoptimized (39.99 KB) Optimized (18.42 KB)

Figure 6.11 The Weekly Timber logo before (left) and after (right) optimizing even further by reducing decimal precision with `svgo` to a value of 1

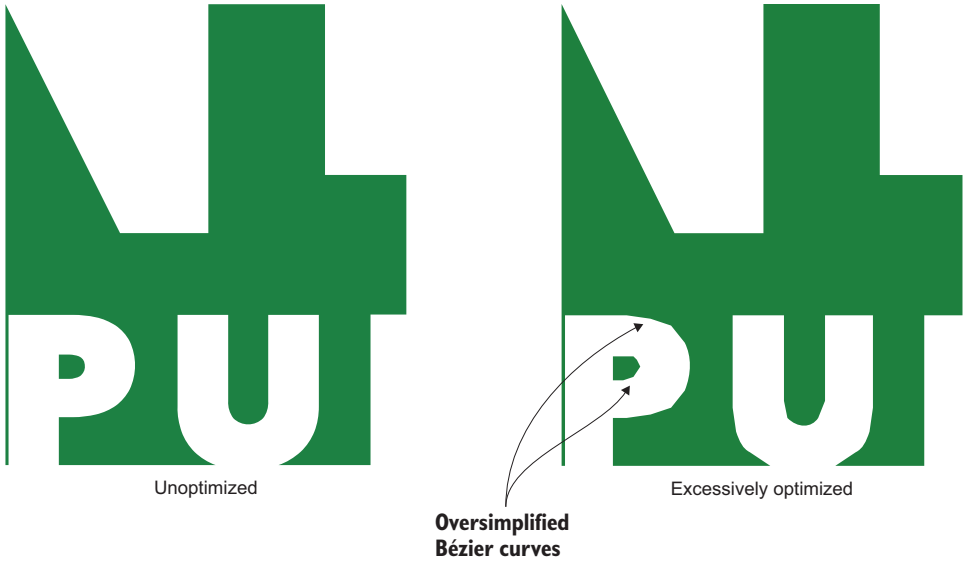


Figure 6.12 An examination of the unoptimized logo.svg (left) compared to an overoptimized version . All precision is stripped from the SVG shapes, resulting in a loss of fidelity, especially with Bézier curves.

This version of the SVG weighs in at 10.81 KB, which is the smallest yet, but the lower file size isn't worth the degradation. Optimizing too aggressively has drawbacks, so always, *always* ensure that the results are satisfactory to you, and *especially* to your client! You can usually identify any detrimental effects of SVG optimization by zooming in on the artwork and comparing it to the unoptimized version.

With image optimization techniques for all types of web images under your belt, you're ready to discover the usefulness of Google's WebP image format.

6.3 Encoding images with WebP

Since the early days of the commercial internet, the only available options for raster images were the JPG, GIF, and PNG formats. Little has changed in this landscape as far as new formats until somewhat recently, when Google introduced WebP.

Your recipe website client has caught wind of this new image format and wonders whether any gains can be made by using it. The client wants you to check out what gains could be made on the recipe collections page in particular.

Luckily for you, the `imagemin` program you've been using has a plugin for converting images to the WebP format, appropriately named `imagemin-webp`. Using this plugin involves using the same pattern as you've used before, so this won't be anything new for you at all.

Unlike other image formats, WebP can be encoded in both lossy and lossless formats. In this section, you'll use the `imagemin-webp` plugin to encode both lossy and

lossless WebP images. You'll also use the `<picture>` element to provide a fallback for WebP-incapable browsers.

6.3.1 Encoding lossy WebP images with imagemin

Encoding lossy WebP images is easy with `imagmein`. It's the same pattern you used before to optimize JPEGs, except in this case, you're using it to convert JPEGs into WebP images. To get started, switch over to a new branch of the recipe client's website with this command:

```
git checkout -f webp
```

When this command finishes, you'll need to install `imagemin` and the `imagemin-webp` plugin:

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

Now you'll write the WebP image-conversion code, which is like the other `imagemin` programs you've written. Create a file called reduce-webp.js and enter the following code into it.

Listing 6.4 Encoding JPEG images into lossy WebP with `imagemin`

```
var imagemin = require("imagemin"),
    webp = require("imagemin-webp");

imagemin(["img/*.jpg"], "optimg", {
    plugins: [webp({
        quality: 40
    })]
});
```

Include the imagemin-webp plugin.

Set the quality of the WebP encoder to 40 out of 100.

Run this script by typing `node reduce-webp.js`. After it runs, all the JPEGs in the img folder will be encoded to WebP and saved to the optimg folder. Next, you'll compare the quality of one of the optimized JPEGs from section 6.2.1 to the WebP output. Figure 6.13 shows the comparison.

Optimized JPEG (79.41 KB) WebP (67.67 KB)

Figure 6.13 A JPEG optimized by using imagemin's `jpeg-recompress` plugin (left) compared to a WebP image encoded from the unoptimized JPEG at a quality setting of 40.

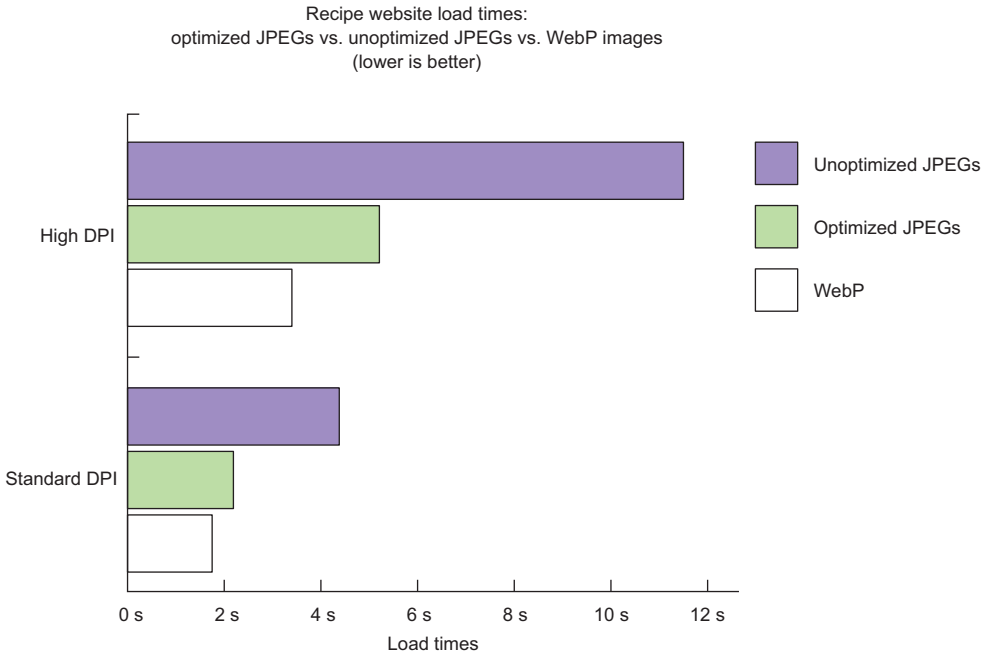


Figure 6.14 A comparison of load times on the recipe website of JPEG and WebP images on standard and high DPI screens. The WebP images offer better loading performance in comparison to both the optimized and unoptimized JPEG images.

There doesn't appear to be much in the way of huge differences. WebP has some drawbacks in that lower-quality settings can produce visual artifacts, but this is also true of JPEGs. After all images have been converted to WebP, switch all the references to JPEGs in index.html over to the WebP files. Using the Good 3G throttling profile in Chrome, let's compare the loading performance of the WebP files to the optimized and unoptimized JPEGs. Figure 6.14 shows this comparison.

These optimizations have yielded a 35% and 20% decrease in page-load time for high-DPI and standard DPI screens, respectively, when compared to load times for the optimized JPEGs. This definitely signifies that WebP is worth the effort, even if the support for it isn't universal.

But you haven't yet investigated the potential of WebP when it comes to lossless images. In the next section, you'll investigate how WebP stacks up against PNG files.

6.3.2 Encoding lossless WebP Images with imagemin

WebP also supports lossless encoding similar to the full-color PNG format, supporting 24-bit color with full transparency. In this short section, you'll convert the PNG version of the recipe website's logo to WebP. You need to tweak only a few parts of your reduce-webp.js script. Modified lines are annotated and in bold in the following listing.

Listing 6.5 Encoding PNG images into lossless WebP with `imagemin`

```
var imagemin = require("imagemin"),
    webp = require("imagemin-webp");

imagemin(["img/*.png"], "optimg", {
    plugins: [webp({
        lossless: true
    })]
});
```

Change first argument in imagemin call.

Replace options with the lossless option and set it to true.

All you've changed here is the file wildcard in the first argument in the `imagemin` call to point to PNG files in the img folder, and replaced the options in the `webp` object with the `lossless: true` option, which tells `imagemin` to losslessly encode WebP images. After making the changes, rerun the script.

When the script finishes converting the PNG images, they'll be placed into the optimg folder. Figure 6.15 compares the file sizes of the lossless WebP files to the optimized PNG files from section 6.2.1 and the original unoptimized PNG files.

Without sacrificing the visual quality of the logo, you've managed to further optimize these images by about 40% and 33% for logo.png and logo-2x.png, respectively. Next you'll learn how to instruct browsers that don't support WebP to gracefully fall back to images they *can* support by using the `<picture>` element.

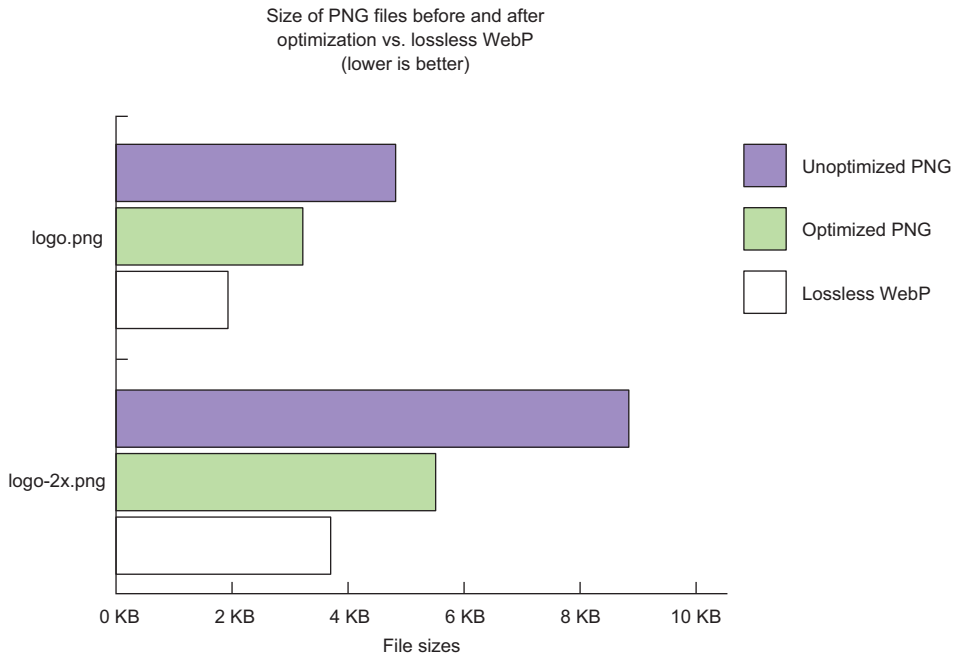


Figure 6.15 A comparison of unoptimized PNGs, optimized PNGs, and lossless WebP images

6.3.3 Supporting browsers that don't support WebP

Although WebP is a great image format that you can start using today, its support isn't as wide as with established image formats. If you have an audience reliant on browsers such as Firefox or Safari, they'll see something similar to figure 6.16.

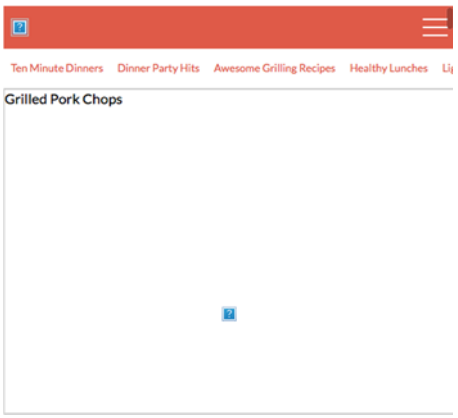


Figure 6.16 Safari failing to display a WebP image

That's no good! You need to specify a fallback that other browsers can handle. That's where the `<picture>` element's ability to fall back to images based on their type comes in handy.

You learned about this method in chapter 5, but here, you'll apply it to the recipe website so that browsers that support WebP can benefit. Those that can't will be able to fall back to a JPEG image. To start, you'll switch to a new branch of code for the website by using `git`:

```
git checkout webp-fallback
```

After the code downloads, open the site in Chrome, and open it again in another browser that doesn't support WebP (such as Safari or Firefox). You'll notice that the images work in Chrome but fail to load in the other browser, as shown in figure 6.16.

At this time, open index.html in your text editor and look for the reference to the logo.webp file. The line will look something like the following code:

```
<img src="optimg/logo.webp"
     srcset="optimg/logo.webp 1x, optimg/logo-2x.webp 2x"
     alt="AllTheFoods" id="logo">
```

You'll take this `<img>` tag and rework it by using the `<picture>` element with `type` attribute fallbacks, as you see in the following listing.

Listing 6.6 Establishing fallbacks with `<picture>`

```
<picture>
    <source srcset="optimg/logo-2x.webp 2x, optimg/logo.webp 1x"
            type="image/webp">
    <source srcset="img/logo-2x.png 2x, img/logo.png 1x" type="image/png">
    <img src="img/logo.png" id="logo">
</picture>
```

Preferred webp image source. Source is a WebP image.

Backup image source pointing to a PNG fallback.

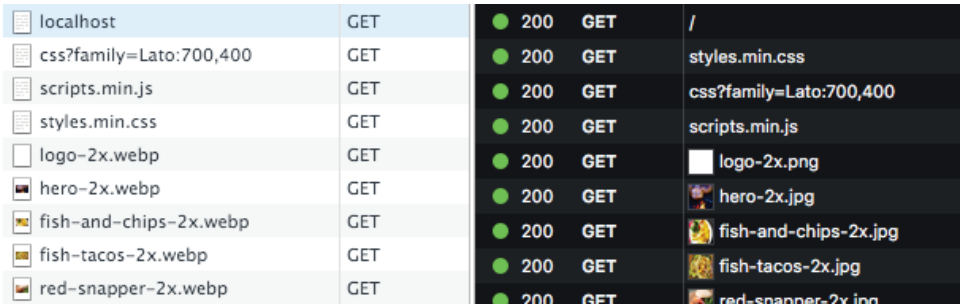
Default image source for browsers that don't support <picture>.

Note that the `<img>` element still retains the `id` attribute value of `logo`. This is nothing more than to make sure that the styling given to the `#logo` selector applies to the image. When styling `<picture>` elements, you should direct styles to the `<img>` tag, because that tag is what `<picture>` assigns the chosen `<source>` element's image to.

Using this pattern, work your way through index.html, and rework the `<img>` tags for the hero and collection gallery images.

> **Want to skip ahead?**
> If you'd like to skip ahead and see the end result, you can do so by entering `git checkout -f webp-picture-fallback`.

After you've worked your way through all the images in the HTML, open the Network tab in the developer tools for each of the browsers, and note what happens when you reload the page. Chrome will download the WebP images, and Firefox will fall back to the image types it supports. Figure 6.17 shows this in action.



Google Chrome loads WebP images

Firefox falls back to JPEGs and PNGs.

Figure 6.17 The network request inspector for two web browsers for our recipe collection page. Chrome (left) can use the WebP images, but Firefox (right) can't, so it falls back to image types it supports.

Here, you're doing two things: you're serving a beneficial image format to a significant slice of users on Chrome, and you're still giving image content to those with other browsers.

Now that you know how to use WebP images, and how to provide fallbacks to browsers that don't support them, you're ready to learn about deferred image loading, also known as *lazy loading*.

6.4 Lazy loading images

Your client appreciates the strides you've made in optimizing the images on the website but they have one last request. The client has noticed that a competitor's website

loads images only when they're in the viewport. Your client is wondering whether that's something you can do, too.

The client doesn't want this functionality just because it's something shiny and new. It's a well-established technique that loads images only when they're needed. When you lazy load images, you're preventing the unnecessary loading of images in situations where users may not even see them. This saves bandwidth and decreases the site's initial load time.

In this section, you'll learn how to write a simple lazy loader in JavaScript, implement it in the client's recipe collection page, and then add basic functionality for browsers without JavaScript support. To get started writing your lazy loader, you'll download a new repository of code via `git`. Run these commands to download the code and start the local web server:

```
git clone https://github.com/webopt/ch6-lazyload.git
cd ch6-lazyload
npm install
node http.js
```

When everything is running, you'll start by defining a pattern for images in your markup. If you get stuck at any point, you can type `git checkout -f lazyload` to view the complete index.html and lazyloader.js files. Let's begin by setting up the markup.

6.4.1 Configuring the markup

Setting up the markup for the lazy loader is the least time-consuming part of the task, but it's crucial. You need a pattern that prevents the browser from loading images by default.

To start, let's look at the page and see what images you should lazy load. You should look at what's above and below the fold on the page, and lazy load those images that fall or *could* fall below the fold. To identify where the fold is for your users, consider using the VisualFold! bookmarklet from chapter 4 at http://jlwagner.net/visualfold. Figure 6.18 shows which images you want to load normally, and has suggestions for what images you *should* lazy load.

When you audit the page, you can immediately see two images that aren't candidates for lazy loading: the logo image and the large main image, known as a *hero image* in marketing parlance. These will be above the fold no matter what. The recipe collection thumbnails in the `.collection` elements, however, are perfect; they're likely to lie underneath the fold on most devices because of the space the large hero image occupies. Even if they *are* above the fold, the lazy loader will grab them on page load when the script initializes and load them anyway. You can choose to tweak which elements to lazy load later, if need be.

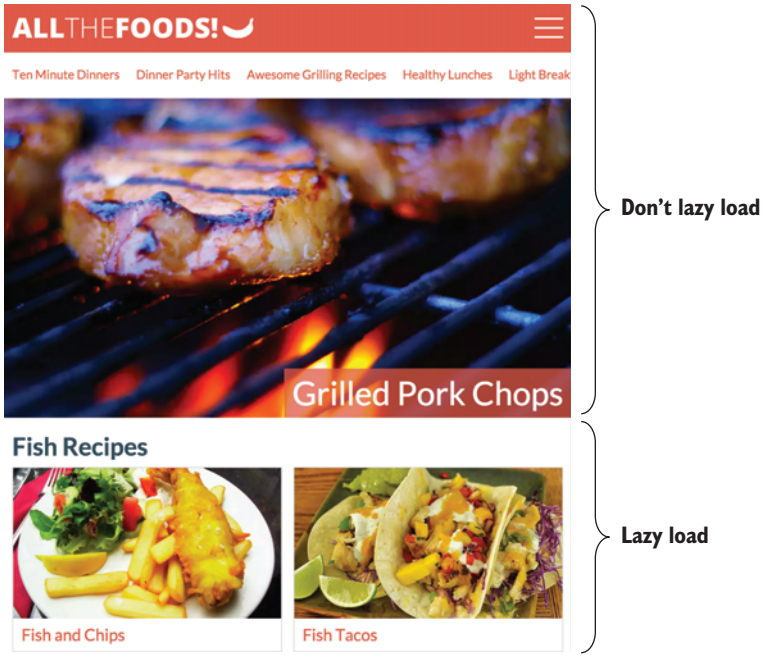


Figure 6.18 An audit of which images make sense to lazy load and which ones don't

Four `.collection` elements are on the page, each with four image thumbnails in `<picture>` elements. One of these elements looks like this:

```
<picture>
    <source srcset="img/fish-and-chips-2x.webp 2x,
                    img/fish-and-chips-1x.webp 1x"
            type="image/webp">
    <source srcset="img/fish-and-chips-2x.jpg 2x,
                    img/fish-and-chips-1x.jpg 1x"
            type="image/jpeg">
    <img src="img/fish-and-chips-1x.jpg" class="recipeImage">
</picture>
```

You need to do two things: move the `srcset` and `src` attributes to data attributes so that the images don't load, and add a class to the `<img>` element that the lazy loader script can attach to. Let's change this markup as shown here.

Listing 6.7 Preparing images for the lazy loading script

```
<picture>
    <source data-srcset="img/fish-and-chips-2x.webp 2x,
                         img/fish-and-chips-1x.webp 1x"
            type="image/webp">
    <source data-srcset="img/fish-and-chips-2x.jpg 2x,
                         img/fish-and-chips-1x.jpg 1x"
```

Points to images to lazy load.

```
            type="image/jpeg">
    <img src="img/blank.png"
         data-src="img/fish-and-chips-1x.jpg"
         class="recipeImage lazy">
</picture>
```

Points to image to lazy load. src points to a placeholder.

Adds the class lazy, which the lazy loader will attach to later.

The changes to the markup are simple. You take all `srcset` and `src` attributes on the `<source>` and `<img>` elements, and change them into `data-srcset` and `data-src` attributes. Storing the image URLs in these placeholder attributes keeps track of the image sources and keeps them from loading until you want them to.

Then, you create a new `src` attribute on the `<img>` tag that points to a 16 x 9 pixel placeholder PNG with a gray background color. This keeps shifting of the layout to a minimum by introducing a placeholder that occupies the same amount of space. The last step is to add the class `lazy` to the `<img>` tag. This is what the lazy loader script will target when it needs to load the image.

From here, you'll want to make the same changes to every other `<picture>` element inside the `.collection` elements. When you're finished, you're ready to write the lazy loading script.

6.4.2 Writing the lazy loader

With the markup pattern defined on your collection page, you can now begin writing the lazy loader script. If you have trepidation, don't worry. I'll explain every step of the way in a series of listings. Let's begin!

LAYING THE FOUNDATIONS

You'll begin by getting the foundations together. This involves creating a closure for your `lazyLoader` object, which isolates the scope of the script from other scripts on the page.

Create a new JavaScript file in the js folder named lazyloader.js. Then enter the contents of the following listing into the file.

Listing 6.8 Beginning the lazy loader

```
(function(window, document){
    var lazyLoader = {
        lazyClass: "lazy",
        images: null,
        processing: false,
        throttle: 200,
        buffer: 50
    }
})(window, document);
```

Start of the closure

lazyLoader object definition

Class name lazy loading images will use in the HTML.

Image element collection

Processing state. Used to throttle executions.

Throttling time in milliseconds

Viewport buffer. Used to load images near the viewport's edge.

End of the closure

Okay, so there's already a quite a bit going on here. You're building an object for the lazy loader that you assign to the `lazyLoader` variable, and encapsulating everything inside a closure. This object will be a collection of properties and functions that facilitate the lazy loading behavior. You'll use the contents of the `lazyClass` property to select all image elements with a class of `lazy`, and store that collection later in the `images` property.

The `processing` property signifies whether the lazy loader is scanning the document for images, which is checked against later to prevent excessive script activity. The `throttle` property is the amount of time in milliseconds that the lazy loader needs to wait before it can scan for images again. The `buffer` property tops it off by specifying the number of pixels beyond the bottom edge of the viewport that will trip the lazy loading behavior for a particular image. Figure 6.19 demonstrates this property's effect.

After you have the skeleton of this structure in place, you can build the methods that initialize and destroy the lazy loader's behavior.

BUILDING THE INITIALIZER AND DESTROYER

That heading seems a little bit like a self-important progressive metal album. The Alpha and The Omega: The Initializer and The Destroyer! All feeble attempts at humor aside, they're important parts of this script. Without them, the script doesn't have an origin from where it can perform its lazy loading goodness. Without a function to destroy the lazy loading behavior, the script will just keep going, uselessly burning up CPU time even after all the images have been loaded.

Continuing on, you'll write two new object properties. These are the `init` and `destroy` properties, each tied to a method that initializes and destroys the lazy

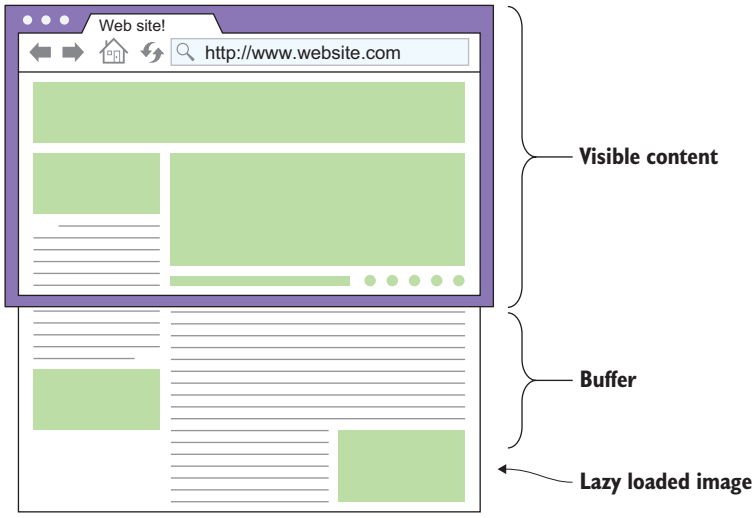
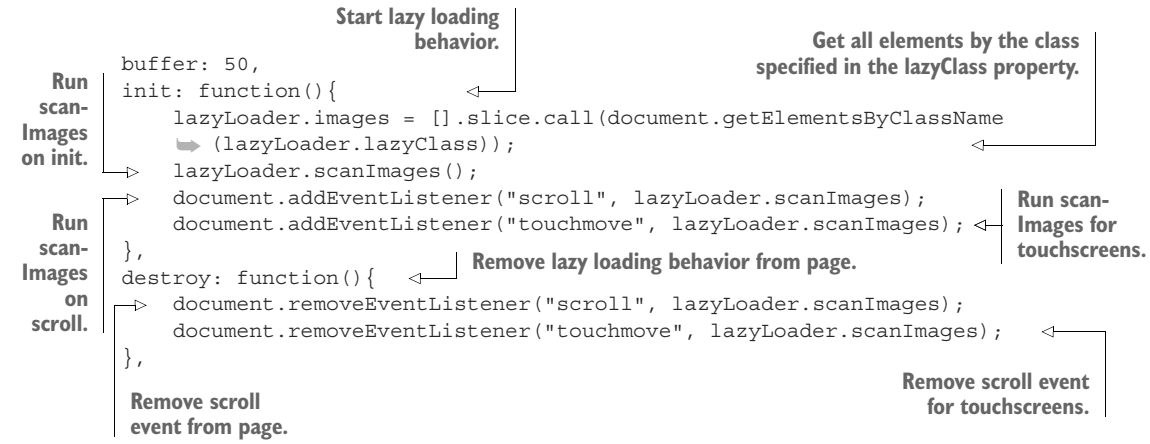


Figure 6.19 The buffer property specifies how far out of the viewport the lazy loader will look for images to load. By extending what the lazy loader looks for beyond the viewport, you can begin loading images as you approach them to give the browser a head start.

loader, respectively. This code is shown next, leaving off from where the `buffer` property was defined.

Listing 6.9 The initializing and destroying functions



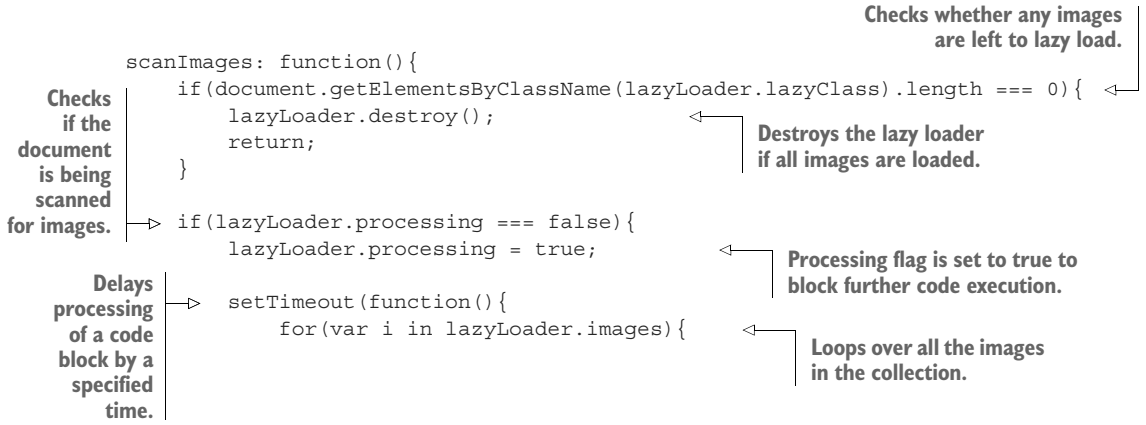

```
buffer: 50,
init: function(){
    lazyLoader.images = [].slice.call(document.getElementsByClassName
    ➥ (lazyLoader.lazyClass));
    lazyLoader.scanImages();
    document.addEventListener("scroll", lazyLoader.scanImages);
    document.addEventListener("touchmove", lazyLoader.scanImages);
},
destroy: function(){
    document.removeEventListener("scroll", lazyLoader.scanImages);
    document.removeEventListener("touchmove", lazyLoader.scanImages);
},
```

With these additions, you're pulling up to the gas pump, so to speak. You're getting the framework laid down for the lazy load behavior to be attached to the appropriate image elements. The next piece of the puzzle is the `scanImages` method.

SCANNING THE DOCUMENT FOR IMAGES

The previous snippet of code refers to the `scanImages` method in many places, but you have yet to write that method. This method is fired from the `scroll` event (and `touchmove` event on mobile devices) and checks whether images with the `lazy` class are within 50 pixels of the viewport's bottom edge. This listing illustrates how to define the `scanImages` method.

Listing 6.10 Defining the `scanImages` method




```
scanImages: function(){
    if(document.getElementsByClassName(lazyLoader.lazyClass).length === 0){
        lazyLoader.destroy();
        return;
    }

    if(lazyLoader.processing === false){
        lazyLoader.processing = true;

        setTimeout(function(){
            for(var i in lazyLoader.images){
```

```
                                                    Checks if element contains
                                                          the lazy class.
Checks if image
element is in the
viewport.
        if(lazyLoader.images[i]
        ➥ .className.indexOf(lazyLoader.lazyClass) !== -1){
            if(lazyLoader.inViewport(lazyLoader.images[i])){
                lazyLoader.loadImage(lazyLoader.images[i]);
            }
        }                                   Passes current element to
    }                                       the loadImage method.
Turns off
processing
flag.
    lazyLoader.processing = false;
  }, lazyLoader.throttle);          Specifies timeout period
}                                   via the throttle property.
},
```

This method first checks whether any more images need to be lazy loaded. If there aren't more, the `destroy` method is run and the lazy loader is finished. If there are more images to process, a `setTimeout` call is run with a `for` loop that uses the `inViewport` method to go through all the images and see whether they're in the viewport. If this method returns `true` for an image, it then loads the image. This behavior is protected from excessive calls from the `scroll` event listener by setting the `processing` property to `true`. From here, you'll need to write the `inViewport` and `loadImage` methods that the `scanImages` references.

WRITING THE CORE LAZY LOADING METHODS

To trigger the lazy loading behavior, you need a cross-browser-compatible method to be able to determine whether a given image element is within the viewport. This is the `inViewport` method.

Listing 6.11 Defining the `inViewport` method

```
                                        inViewport method
                                        definition.
inViewport: function(img){
    var top = ((document.body.scrollTop ||
                document.documentElement.scrollTop) + window.innerHeight) +
                lazyLoader.buffer;
    return img.offsetTop <= top;
},
```

Finds the viewport's position and height and the buffer threshold.

Checks if the given image element is in the viewport.

The `inViewport` method is simple. It checks to see how far the user has scrolled down the page. It does this by conditionally tapping into either the `document.body.scrollTop` or the `document.documentElement.scrollTop` properties. The reason you check either one of these is that IE 9 has a compatibility problem with `document.body.scrollTop` always returning `0`, so you fall back to a similar property using the `||` operator. This value is then added to the height of the window to track the bottom of the user's viewport, and then an additional buffer value is added to trigger loading of images when

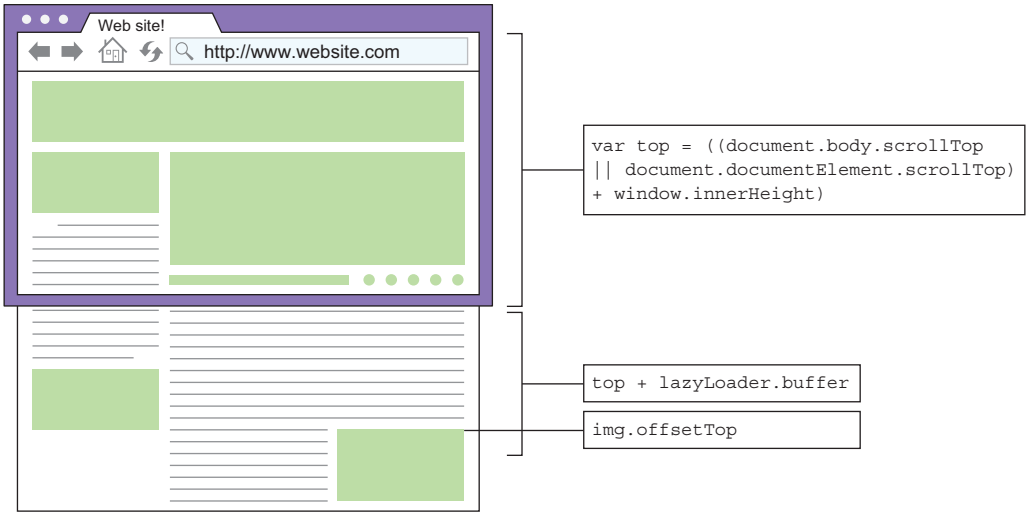


Figure 6.20 The position calculations of the `inViewport` method, and how they relate to the viewport and the targeted image element. In this case, the calculation of the viewport height plus the amount of buffer space given exceeds the top boundary of the image element, resulting in a return value of `true`.

they're near, but not quite in, the viewport. Figure 6.20 shows how these calculations relate to the browser viewport and the image element passed to `inViewport`.

From here, you continue on to write the central piece of the program that drives the lazy loading behavior itself: the `loadImage` method, shown next.

Listing 6.12 Defining the `loadImage` method

```
loadImage: function(img){
    if(img.parentNode.tagName === "PICTURE"){
        var sourceEl = img.parentNode.getElementsByTagName("source");

        for(var i = 0; i < sourceEl.length; i++){
            var sourceSrcset = sourceEl[i].getAttribute("data-srcset");

            if(sourceSrcset !== null){
                sourceEl[i].setAttribute("srcset", sourceSrcset);
                sourceEl[i].removeAttribute("data-srcset");
            }
        }
    }

    var imgSrc = img.getAttribute("data-src"),
        imgSrcset = img.getAttribute("data-srcset");

    if(imgSrc !== null){
```

loadImage definition for the method

Checks if the image element's parent is a <picture> element.

Grabs nearby <source> elements.

Loops through <source> elements in the <picture> element.

Gets data-srcset attribute to load the image.

Checks if srcset exists on the <source> element.

Sets srcset and remove data-srcset.

data-src and data-srcset grabbed from <img>.

Checks if data-src exists on <img>.

```
        img.setAttribute("src", imgSrc);
        img.removeAttribute("data-src");
    }

    if(imgSrcset !== null){
        img.setAttribute("srcset", imgSrcset);
        img.removeAttribute("data-srcset");
    }

    lazyLoader.removeClass(img, lazyLoader.lazyClass);
},
```

Checks if a data-srcset exists on the <img> element.

src set to the value of data-src on <img>.

srcset replaced with content of data-srcset.

lazy class is removed from <img> element.

The `loadImage` method first checks whether it's a direct child of a `<picture>` element. If this is true, the neighboring `<source>` elements are scanned, and their `data-src` and `data-srcset` attributes are flipped to `src` and `srcset` attributes. After this completes, the `<img>` element's `data-src` and `data-srcset` attributes are processed and flipped to `src` and `srcset` attributes. The browser then sends a request to the web server for those assets. The way this function is written allows the lazy loader to work on images specified by the `<picture>` element, as well as standard `<img>` with optional support for `srcset`. After the attributes are changed and the image loading starts, the `lazy` class is removed using a new method that you must define, called the `removeClass` method, which you can see in the next listing.

Listing 6.13 Defining the `removeClass` property

```
removeClass: function(img, className){
    var classArr = img.className.split(" ");

    for(var i = 0; i < classArr.length; i++){
        if(classArr[i] === className){
            classArr.splice(i, 1);
        }
    }

    img.className = classArr.toString().replace(",", " ");
}
```

className string is converted to an array.

Defines the removeClass method.

Array is looped over.

If this array element is the "lazy" class . . .

. . . the array item is removed.

Array converted into string and reassigned.

This method converts the image element's `className` string into an array and iterates over it. If the `lazy` class is found, it's removed, and the array is converted back into a string and assigned back to the image element's `className` property.

TURNING THE KEY AND RUNNING THE SCRIPT

Now that everything in the object is defined, you can fire the `init` method inside an onreadystate change event after the `lazyLoader` object, like so:

```
document.onreadystatechange = lazyLoader.init;
```

This event waits for the DOM to load, and after it does, the lazy loading behavior is attached to the specified image elements. All that's left to do is to load the script by placing a `<script>` tag referencing lazyloader.js after the reference for scripts.min.js:

```
<script src="js/lazyloader.js"></script>
```

After the script is loaded, and assuming no syntax errors exist, you should be able to scroll down the page and see images load as they scroll into view. To see how the lazy loader works, try opening the network tab of the browser you're using and reload the page. Wait for the page to load and scroll the page. You should see new network requests appear in the waterfall graph as images lazy load. Figure 6.21 shows this behavior.

With the lazy loader written and finished, one last piece of the puzzle remains: providing a fallback to users who have JavaScript turned off.

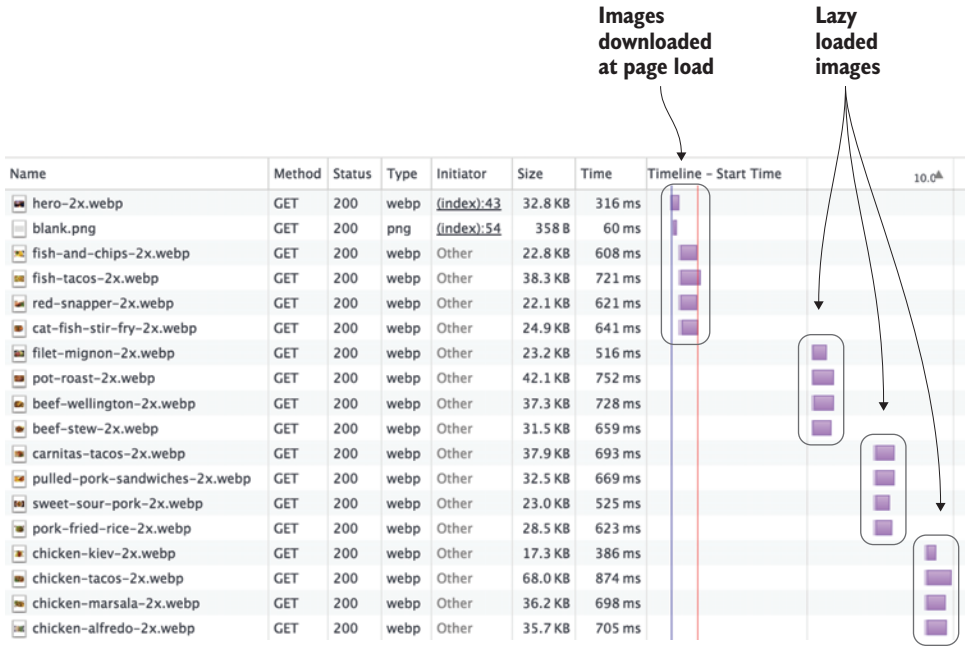


Figure 6.21 The network waterfall graph showing lazy loaded images

6.4.3 Accommodating users without JavaScript

As small of a segment as they may seem, there are users who have JavaScript turned off or otherwise don't have it available. With the lazy loader script in place, those users won't see anything other than the image placeholders. Figure 6.22 shows this effect.

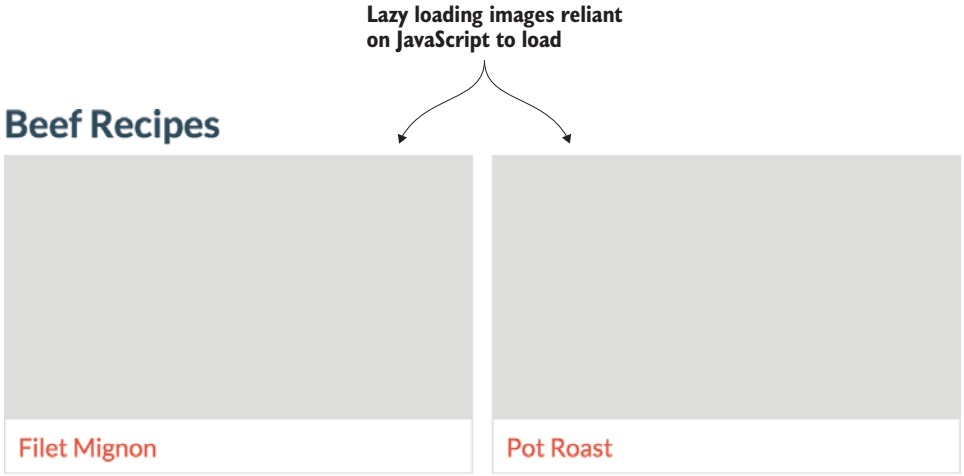


Figure 6.22 The effect of lazy loading a script on browsers with JavaScript turned off. The images never load because the JavaScript never runs.

This isn't acceptable, even if it affects such a small segment of users. The good news is that the fix is easy, thanks to your friend `<noscript>`. You can modify your markup by adding a `<noscript>` tag with the image sources set explicitly in the `src` and `srcset` attributes, like so:

```
<noscript>
    <picture>
        <source srcset="img/fish-and-chips-2x.webp 2x,
                        img/fish-and-chips-1x.webp 1x"
                type="image/webp">
        <source srcset="img/fish-and-chips-2x.jpg 2x,
                        img/fish-and-chips-1x.jpg 1x"
                type="image/jpeg">
        <img src="img/fish-and-chips-1x.jpg" class="recipeImage">
    </picture>
</noscript>
```

You'll note that the contents of the `<noscript>` tag are the `<picture>` elements as they were before you modified them for the lazy loader. When you add this code, you should add it immediately after the `<picture>` element that's lazy loaded. Try turning off JavaScript in your browser and then reload the page. You'll see something that looks like figure 6.23.

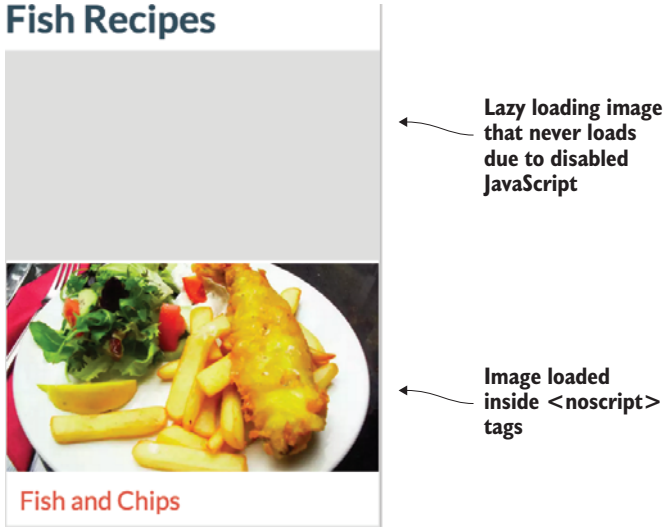


Figure 6.23 The `<noscript>` tag at work. Both the image placeholder and the image loaded in the `<noscript>` tag are visible because the image placeholder is never hidden when JavaScript is disabled.

Both the lazy loader image placeholder and the image loaded via the `<noscript>` tag are visible. This isn't going to fly with your client, so you need to make sure that the image placeholder is hidden when JavaScript is turned off. The solution to this is simple. First, add a class of `no-js` to the `<html>` element:

```
<html class="no-js">
```

You'll use this class to target the lazy loaded images in the markup with CSS. To do this, you add a simple rule at the end of global_small.less in the less/components folder:

```
.no-js .lazy{
    display: none;
}
```

After adding this, recompile the LESS files via `less.sh` (or `less.bat` for Windows users).

So what's going on here? By adding a `no-js` class to the `<html>` tag, and adding a style that hides the `.lazy` elements in the DOM, you're ensuring that you don't see both the image placeholder and the recipe image when JavaScript is turned off.

But this means that those elements will be hidden for browsers that *do* have JavaScript enabled! To fix this, you add a small bit of inline script that removes the `no-js` class from the `<html>` tag when JavaScript is available. Open index.html in your text editor, and add this one-liner script right before the closing `</head>` tag:

```
<script>document.getElementsByTagName("html")[0].className="";</script>
```

The end result is that browsers with JavaScript available will benefit from lazy loading, and those that have JavaScript disabled will still display the images. Those users won't benefit from the lazy loading script, but they'll receive an acceptable experience.

> **A note on removing HTML element classes**
>
> The preceding method employs a scorched-earth policy in removing the `no-js` class by emptying the entire `<html> class` attribute. If you have other classes you need to preserve (Modernizr classes, for example), you can use a jQuery function such as `removeClass()` to selectively remove the `no-js` class. Better yet, consider using the native `classList` method, which is covered in chapter 8.

With your client happy, let's summarize what you've learned and get ready to move on to the next chapter.

6.5 Summary

In this chapter, you learned the following image-optimization techniques and their performance benefits:

- Image sprites are a method of concatenating multiple images into a single file, which can save on HTTP requests. SVG sprites can easily be generated from a set of individual SVG images by using the `svg-sprite` Node utility.
- Not all browsers support SVG images. If you have a segment of users in your audience who can't use SVG, you can provide a PNG fallback by using the Grumpicon online utility at www.grumpicon.com.
- Images can account for a large portion of the data that your users download when they visit your site. You can reduce the size of your site's images via the `imagemin` Node utility, along with `imagemin` plugins specific to various image formats.
- If you have a large segment of users using Chrome or Chrome-derived browsers, you can serve even smaller images with equivalent visual quality by using Google's WebP image format. Using the `imagemin-webp` plugin in Node, you can even generate lossy WebP versions in place of JPEGs, and lossless WebP versions in place of PNGs.
- Not everyone uses Chrome, so you can't just slap WebP images up on your site and expect them to work for everyone. Using the `<picture>` element in conjunction with the `<source>` element's `type` attribute, you can specify fallbacks to established image types for browsers that don't support WebP.
- Deferred loading of images, or lazy loading, is a great way to reduce the initial load time of your site. This technique also saves bandwidth by avoiding loading images that your visitors may never see. By writing your own lazy loading script, you can implement this behavior on your own site. You also can accommodate users with JavaScript turned off by coming up with a solution that uses the `<noscript>` tag.

With these techniques under your belt, you'll be able to ensure that your websites accommodate rich visual content without sacrificing the speed and compatibility your site visitors demand. You're now ready to move on to the world of fonts, and learn how to optimize their delivery to the user!