5 Making images responsive

This chapter covers

- Using CSS media queries to deliver the right background images for a user's device
- Delivering responsive images in HTML using `srcset` and the `<picture>` element
- Using Picturefill to polyfill `srcset` and `<picture>` in certain browsers
- Using SVG images in CSS and HTML

Now that you've learned useful CSS optimization techniques, you can dive into the importance of managing the images on your website.

Images often compose the largest portion of a website's total payload, and that trend shows no sign of changing. Though internet connection speeds are continually improving, many devices are shipping with high DPI displays. In order for images to display optimally on these devices, higher-resolution imagery is required. Because you still need to support devices with less-capable screens, however, you still require lower-resolution images and must pay attention to the way images are delivered to various devices.

When you think about image delivery, the intent isn't only to deliver the best visual experience possible, but also to deliver images that are appropriate for a device's capabilities. Knowing how to properly deliver images ensures that devices with less capability are never burdened with more than what they need, while ensuring that the most capable devices are receiving the best possible experience. Maintaining this balance ensures an appropriate mix of visual appeal and performance.

5.1 Why think about image delivery?

One component of web performance as it pertains to imagery means delivering the right image sizes and types to the devices that can best use them. This section introduces the importance of properly delivering images—in CSS as well as in HTML. When we talk about image delivery, what we're really talking about is serving images *responsively*.

Responsive images matter if you care about the performance of your website. It's important for your CSS to be responsive so that your site is viewable on as many devices as possible. But it's also important for your images to be responsive for these two reasons: scaling and file size.

When images are served with responsiveness in mind, users are getting the best experience that their devices are capable of. For instance, one large image can scale down well for all devices, but it isn't the best choice even if it looks great for all devices. The device has to take an image that's grossly oversized and rescale it to fit the screen. This image will also have a larger file size, thus taking more time to download. This impairs performance.

Instead, it makes more sense to serve images to best fit the device's needs. This entails maintaining multiple sets of images, but the effort is worthwhile because processing and downloading times are minimized. Figure 5.1 illustrates inefficient versus efficient methods of image scaling.

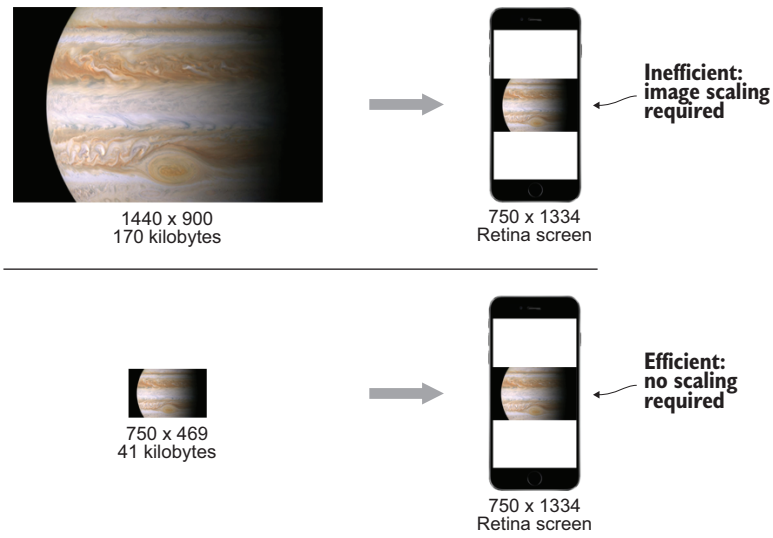


Figure 5.1 Two examples of scaling an image to a mobile phone. At the top, a 170-KB image with a width of 1440 pixels is scaled down to the width of the phone's high DPI display. At the bottom, a 41-KB image with a width of 750 pixels is delivered to the screen without having to be scaled; this process is more efficient.

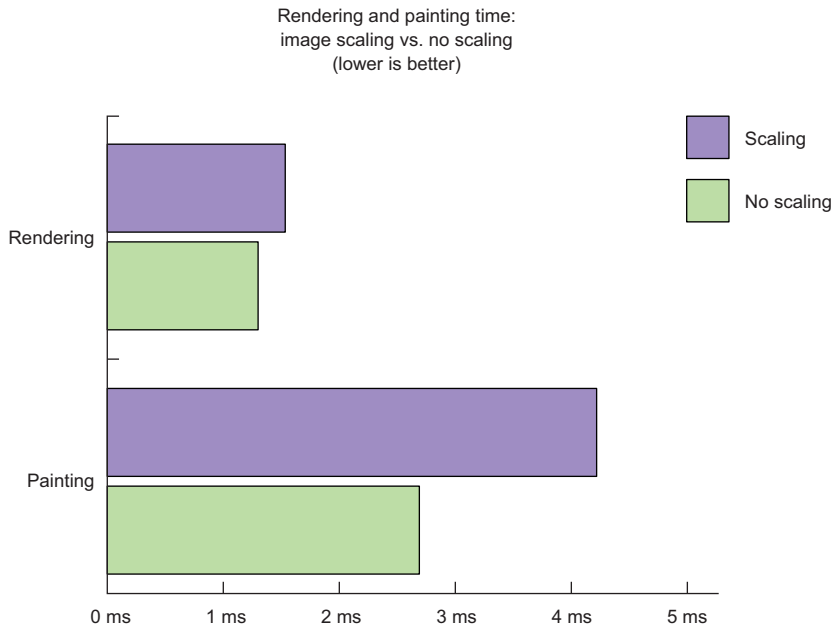


Figure 5.2 A comparison of rendering and painting times for a single image in Chrome. For the scaling scenario, the source image of 1440 x 900 is scaled to fit a container 375 pixels wide. In the no-scaling scenario, an image resized to fit the container is used and triggers no scaling. Rendering and painting times are faster with no scaling.

The performance impacts of responsive imagery must be examined as well. Google Chrome's Timeline tool comes in handy once again in measuring the rendering and painting performance of inefficient versus efficient image scaling. This test was run five times for each scenario and captured two aspects: rendering and painting time. Figure 5.2 shows the results.

In this scenario, I observed a 15% improvement in rendering time and a 36% improvement in painting time. Though this represents an improvement of only a millisecond, it's important to remember that this measurement was taken for a single image. Implementing responsive imagery across an entire website can help make pages feel more responsive to user input by reducing processing time.

Is it possible to deliver perfectly scaled images for all devices? Anything is possible, but such a goal *isn't* practical. The best approach is to define an array of image widths that covers the entire spectrum of your needs, with the understanding that some overlap will exist between images in the array to cover the needs of different devices and display densities. Some scaling is always going to happen. You want to minimize how much of it occurs.

How you use responsive images depends on *where* they're being used. As you well know, images are most often referenced in CSS and HTML. As you work through the

rest of this chapter, you'll learn about the various types of images, their best uses, and ways to use them in a responsive fashion in both CSS and HTML.

5.2 Understanding image types and their applications

Using images on the internet was once simple: a few formats existed, but all were bitmap (also known as *raster*) images. Some images were better suited than others for certain tasks. These rules hold true today, but the landscape has changed, and the playing field is larger than it used to be. In this section, you'll learn about the two major image types: raster and SVG.

5.2.1 Working with raster images

As you likely know, the most common type of image used on the web is a *raster*. These are sometimes called *bitmap images*. Typical examples are JPEG, PNG, and GIF images. These comprise pixels aligned on a two-dimensional grid. Figure 5.3 shows an example of a YouTube favicon enlarged from 16 x 16 pixels to 512 x 512 pixels to illustrate the concept.

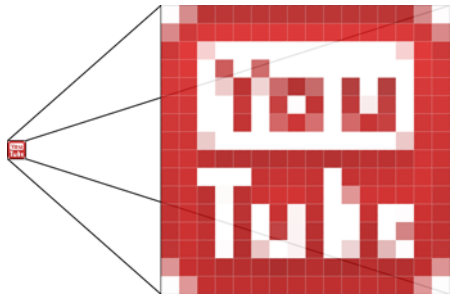


Figure 5.3 A 16 x 16 raster image of a YouTube favicon. On the left is the native size of the image, and on the right is the enlarged version. Each pixel is part of a two-dimensional grid.

Raster images are used to depict many things on the web: logos, icons, photos, and so on. In HTML, they're displayed using `<img>` tags. In CSS, they're often used in the background property, but also find use in other lesser-used properties such as `list-style-image`.

Raster images come in a few formats, and each is best suited to a particular kind of content. In this section, you'll categorize these images by the way they're compressed. Recall that in chapter 1 you used server compression to achieve smaller file sizes for style sheets, scripts, and HTML assets. In this section, you'll learn about compression as it pertains to raster images, which fit into two categories: lossy and lossless.

LOSSY IMAGES

Lossy images use compression algorithms that discard data from an uncompressed image source. The idea behind these image types is that some level of quality loss is acceptable in exchange for smaller file sizes.

A good example of lossy images happens right in your digital camera. When you download photos from a digital camera's storage card, you're typically downloading them as JPEGs. When the camera takes a photo, it stores an uncompressed version of the photograph in memory and uses lossless compression to convert the uncompressed source to a JPEG image.

JPEG images are nearly everywhere on the web. A prolific example of the JPEG format put to use is on the popular photo-sharing website Flickr, shown in figure 5.4.

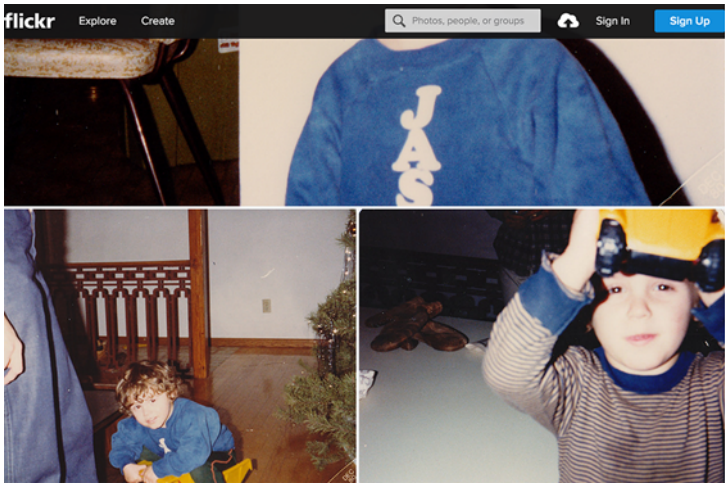
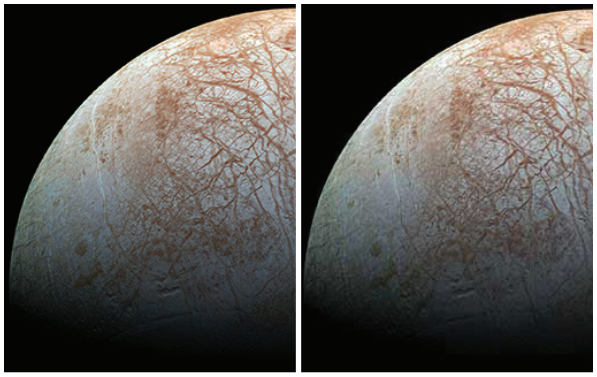


Figure 5.4 JPEG images in use on the popular photo cataloging and sharing site Flickr. Photographic content is best suited to the JPEG format.

A drawback of this format is that extreme compression can be noticeable. These file types are also susceptible to generational loss if they're not saved from an uncompressed source such as a PSD file. Generational loss occurs when an already-compressed file is recompressed, resulting in further visual degradation. In practice, however, the degradation shouldn't be noticeable if compression is used with care, and these image types are saved from an uncompressed source.

Figure 5.5 shows two versions of a photograph. The left image is the uncompressed source, and the right image is a copy after a reasonable amount of JPEG compression has been applied to it.

The visual quality shows subtle signs of degradation from the uncompressed source to the compressed JPEG. Considering that the JPEG is about 96% smaller than its uncompressed TIFF version, this loss in visual quality is an acceptable trade-off. The output quality of the JPEG algorithm is expressed on a scale of 1 to 100, where 1 is the lowest and 100 is the highest.



Uncompressed (269 KB) JPEG (12 KB)

Figure 5.5 A comparison of the same image in uncompressed (TIFF) and compressed (JPEG) formats with their respective file sizes. The JPEG version has some subtle degradation at a quality setting of 30, but is acceptable for this scenario.

Photo credit: NASA Jet Propulsion Laboratory.

JPEG isn't the sole lossy image format used on the web. Other such formats exist, such as Google's new WebP image format (covered in chapter 6). Now let's look at lossless image types.

Lossless images

On the other end of the spectrum are *lossless images*. These image types use compression algorithms that don't discard data from the original image source. A good example of a lossless image in action is the Facebook logo on the desktop version of its site, as shown in figure 5.6.

Figure 5.6 The Facebook logo is a PNG image, which is a lossless image format. PNG images are well suited for lossless formats.

Unlike lossy image formats, lossless formats are perfect when image quality is important. This makes lossless formats a great candidate for content such as icons. Lossless image types usually fall into two categories:

- *8-bit (256-color) images*—These are formats such as the GIF and 8-bit PNG formats and support only 256 colors and 1-bit transparency. Despite their color limitations, they're great candidates for icons and pixel art images that don't require a lot of colors or sophisticated transparency. The 8-bit PNG format tends to be more efficient than GIF images. Unlike GIFs, however, PNGs have no support for animation.
- *Full-color images*—The only formats that support more than 256 colors are the full-color PNG format and the lossless version of the WebP format. Both support full alpha transparency and up to 16.7 million colors. The full-color PNG format enjoys wider support than WebP. These image types are well suited for icons and photos, but their lossless nature means that photographic content is usually best left to the JPEG format unless transparency support is a must.

Results of lossless image formats depend on the subject of the image. Figure 5.7 provides a generalized comparison of lossless image compression methods.

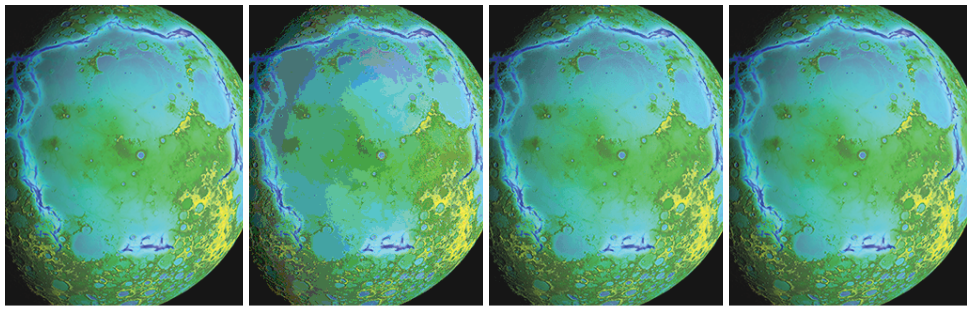


Figure 5.7 A comparison of lossless image-compression methods. The differences between the uncompressed and full-color PNG and WebP versions are imperceptible, whereas the 8-bit lossless image is throttled down to 256 colors.

Photo credit: NASA Jet Propulsion Laboratory.

The various lossless formats have their advantages in certain types of content, and are overall well-suited for line art, iconography, and photography. The trick to finding the right fit requires experimentation, but the basic rules of using these formats are generally simple: Simple images with few colors should use 8-bit lossless formats. Images not suitable for lossy formats and/or in need of full transparency should use the full-color PNG format.

The next section covers an entirely different class of image aside from raster images: scalable vector graphics.

5.2.2 Working with SVG images

Another type of image format used on the web is *Scalable Vector Graphics*, commonly referred to as SVG. These images use a vector artwork format. They're different from raster images in that they can be scaled to any size, because they're composed of mathematically calculated shapes and sizes. Figure 5.8 demonstrates this effect.

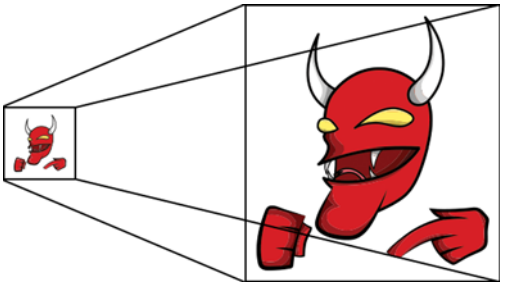


Figure 5.8 A cartoon vector image at different sizes. Notice that the larger version doesn't lose any visual quality as it scales up. This is the primary advantage of vector images over raster images.

Vector images scale so well because of the way they're rendered. Although all device screens are pixel-driven, and thus all display output eventually ends up being represented by pixels, vector images go through a different process than raster images when they're displayed on a screen. They're parsed, their mathematical properties are evaluated, and then they're mapped to the pixel-based display through a process called *rasterization*. This occurs every time the image is scaled, ensuring the best possible visual integrity every time.

If you're familiar with creating vector artwork, you're familiar with programs such as Adobe Illustrator. Although file formats native to these programs are expressed in binary, the SVG format is expressed as XML, which is a text format. The SVG media type `image/svg+xml` reflects its roots in XML. This characteristic allows you to edit SVG files in a text editor, place SVG files inline in HTML, and even to use CSS and media queries in SVG files.

Although SVG has been a W3C standard since 1999, its use in websites has been only relatively recent. The fact that SVG works so flawlessly across different device resolutions and display densities makes it a popular image format.

SVG isn't a silver bullet, however, and it's limited in its applications. SVG images *aren't* suitable for photographs, and are most effective when used to depict logos, iconography, or line art. Images with solid colors and geometric shapes are well served by this flexible image format.

5.2.3 Knowing what image formats to use

With all of the image types available, knowing which type makes the most sense for a certain kind of content can be daunting. Although vector images are a standalone category with SVG as the prevailing format, raster images are classified by the two categories of compression (lossy and lossless) and within them are a slew of formats. Table 5.1 should give you some ideas on the kinds of content that fit best with each image type

In the next section, you'll optimize image delivery in CSS for a website's masthead image.

Table 5.1 You can choose an Image format based on the type of content for your site. Each image format varies in color restrictions, image type, and compression category. (*Full* color indicates a range of 16.7 million or more colors, 24/32-bit.)

Image format	Colors	Image type	Compression	Best fit
PNG	Full	Raster	Lossless	Content that may or may not require a full range of colors. Quality loss is unacceptable, and/or the content requires full transparency. Accommodates any image format, but may not compress as well as JPEG for photographs.
PNG (8-bit)	256	Raster	Lossless	Content that doesn't require a full range of colors but might require single-bit transparency—such as icons and pixel art.
GIF	256	Raster	Lossless	Same as 8-bit PNG with somewhat lower-compression performance, but supports animation.
JPEG	Full	Raster	Lossy	Content that requires a full range of colors and for which quality loss and lack of transparency are acceptable—such as photographs.
SVG	Full	Vector	Uncompressed	Content that may or may not require a full range of colors, and for which quality loss is unacceptable when scaled. Best for line art, diagrams, and other generally nonphotographic content. Requires the least amount of development effort for optimal display on all devices.
WebP (Lossy)	Full	Raster	Lossy	Same as JPEG, but also supports full transparency, with the potential for better compression performance.
WebP (Lossless)	Full	Raster	Lossless	Same as full-color PNG, with the potential for better compression performance.

5.3 Image delivery in CSS

Image delivery in CSS is a great place to start, because it involves CSS properties and features that you're familiar with if you've ever developed responsive websites. The main CSS feature you'll use to properly deliver images is the media query.

This time around, you'll optimize the delivery of a masthead image for Legendary Tones, a website introduced in chapter 1 that publishes articles of interest to guitarists. The images in this site are poorly managed, resulting in low quality on larger screens. You want to deliver higher-quality images to these users, but you want to do so in a way that doesn't overburden them with images that are too large, no matter the device they could using. You also want to ensure that users with high DPI displays are getting the best possible experience on their screens.

You'll start by downloading this website and running it locally. To do this, you need to perform these commands in a folder of your choosing:

```
git clone https://github.com/webopt/ch5-responsive-images.git
cd ch5-responsive-images
npm install
node http.js
```

After the website has been downloaded, you can pull it up on your local machine at http://localhost:8080. It should look something like figure 5.9.

When the website is running on your local machine, you can segment images and form a plan for targeting displays by device width, device DPI, as well as how to use SVGs in CSS.

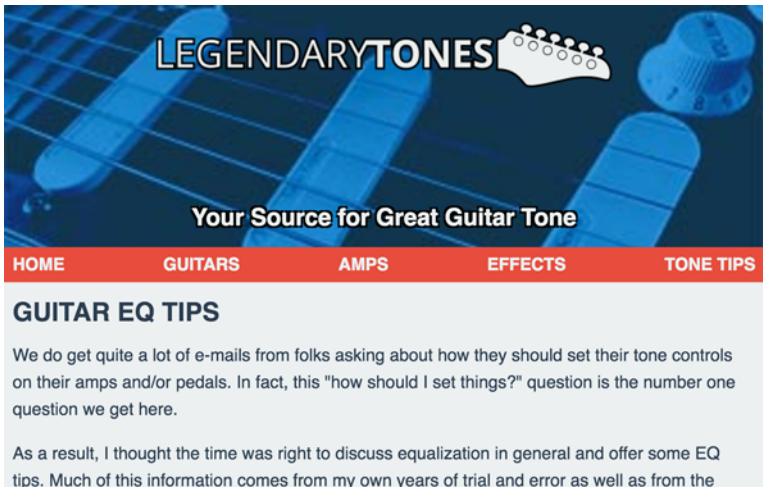


Figure 5.9 The Legendary Tones website as it appears in the browser

5.3.1 Targeting displays in CSS by using media queries

The goal of this section is for you to improve the visual quality of the Legendary Tones website masthead image by using a set of background images supplied in the img folder to create an experience that's optimal for *all* devices.

When implementing responsive images using CSS, the best tool for the job is the media query. With media queries, you can decide at which screen width you should change a `background-image` rule for a particular selector.

On the Legendary Tones website, only one selector has a `background-image` property set, and that's the `#masthead` selector. It sets the styling for the top of the page, which has a large background image, logo, and site tagline. The CSS for this selector is shown here.

Listing 5.1 The #masthead styling for the Legendary Tones website

```
#masthead{
     padding: .5rem 0 0;
     height: 10rem;
     background-size: cover;
     background-image: url("../img/masthead-xxxsmall.jpg");
     background-position: 50% 50%;
     position: relative;
}
```

Ensures the background image covers the entirety of the container no matter its size.

Default mobile-first background image for the masthead

The `#masthead` element spans the width of the browser window and has a `background-image` value of `masthead-xxxsmall.jpg`. If you load this website on large screens, you'll immediately notice how poor the visual quality of the background image is. This is because the default styling is mobile-first, so the smallest image meant for mobile devices is served.

Let's take stock of the images in the img folder which contains an array of masthead background images that you'll plug into media query breakpoints for the `#masthead` selector. Table 5.2 lists these images and the media queries they'll be targeted for.

Table 5.2 Images, their resolutions, and their target media query breakpoints in the website's CSS

Image name	Image resolution	Media query
masthead-xxxsmall.jpg	320 x 135	None (default image)
masthead-xxsmall.jpg	640 x 269	`(min-width: 30em)`
masthead-xsmall.jpg	768 x 323	`(min-width: 44em)`
masthead-small.jpg	1024 x 430	`(min-width: 56em)`
masthead-medium.jpg	1440 x 604	`(min-width: 77em)`
masthead-large.jpg	1920 x 805	`(min-width: 105em)`
masthead-xlarge.jpg	2560 x 1073	`(min-width: 140em)`
masthead-xxlarge.jpg	3840 x 1609	Not used

As you can see, each image falls into a specific device-resolution range that allows it to be scaled without being stretched to the point of being pixelated. The first image, masthead-xxxsmall.jpg, starts at a width of 320 pixels and stretches up to 479 pixels. At the 480-pixel breakpoint, the 640-pixel version of the image kicks in and scales larger and larger to a width of 703 pixels. At 704 pixels, a new breakpoint kicks in and a higher-resolution image is substituted. This continues until the maximum resolution is hit. Note that the masthead-xxlarge.jpg file isn't used. You'll use it later on this section when you target high DPI screens. For now, you'll ignore it.

Open the styles.css file from the css folder in your text editor. You'll notice that it's a mobile-first example, and that a slew of media queries exist at the bottom of the file. The styles in these media queries serve to change the size of the site logo, the site tagline copy, and the height of the `#masthead` container. The first breakpoint set on line 183 at `480px` (or `30em`, because you're using `ems` instead of pixels) looks like the following listing.

Listing 5.2 Media query breakpoint

```
@media screen and (min-width: 30em){ /* 480px/16px */
     #masthead{
            height: 12rem;
     }

     #logo{
            width: 70%;
     }

     #tagline{
            font-size: 1.25rem;
     }
}
```

Media query for a screen width of 480 pixels

The element you'll be adding a background image to in this and further breakpoints

What you want to do with this code is modify the `#masthead` selector's content to provide a higher-resolution background image than masthead-xxxsmall.jpg when this breakpoint kicks in. So go ahead and change the content of the `#masthead` selector to the following:

```
#masthead{
     height: 12rem;
     background-image: url("../img/masthead-xxsmall.jpg");
}
```

After you add the `background-image` property for the new image, save the document and reload the page. Then observe the improvement in image quality, as illustrated in figure 5.10.

All that's left to do from here is to repeat this process for each breakpoint listed in table 5.2. When you're finished, you should have a masthead with a background

Before adding background image | After adding background image

Figure 5.10 The masthead background image at the 480-pixel (`30em`) breakpoint before (left) and after (right) adding the new background image. Note the improved visual quality in the after image.

image that transitions well from a mobile phone resolution all the way to a large desktop device resolution.

> **Want to skip ahead?**
> Having a bit of trouble? Or want to skip ahead and see how everything works? You can do this by using `git` to switch to the finished state of the project, where you can view the code. Enter `git checkout responsive-images -f` at the command line to skip ahead. Be warned that any changes you have in your local copy may be lost!

In the next section, you'll learn how to target high DPI displays and use media queries to deliver higher-resolution images to those more capable devices.

5.3.2 Targeting high DPI displays with media queries

One aspect of implementing responsive images that you must keep in mind are *high DPI displays*, such as 4K and 5K ultra HD displays. One well-known example of this technology is Apple's Retina Display, but it's certainly not limited to Apple devices. Most devices now ship with screens that can be considered high DPI. A comparison of standard versus high DPI displays can be seen from our masthead in figure 5.11.

High DPI displays deliver enhanced visual experiences but present a new challenge for the developer: delivering images efficiently for them. Figure 5.12 provides a comparison of properly delivering images to these displays.

In a continuation of the efforts you began earlier in this section, you'll implement background images for the `#masthead` element, but this time for high DPI screens. All this requires is to target not only device widths as you did earlier, but also their pixel

Figure 5.11 An enlarged visual representation of graphics on standard displays versus high DPI displays

Standard DPI High DPI

Figure 5.12 A comparison of two versions of a background image on two display types. On the left, a background image intended for use on standard displays appears on a high DPI display. On the right, the proper resolution image is used for the high DPI display, creating a better visual experience.

density with a combination of media queries. Here's an example of a basic high DPI screen media query:

```
@media screen (-webkit-min-device-pixel-ratio: 2),
                  (min-resolution: 192dpi){
     /* Put High DPI Styles Here */
}
```

Here you see two media queries in action: the vendor-prefixed `-webkit-min-device-pixel-ratio` media query is the WebKit implementation for high DPI display support for older browsers, whereas the `min-resolution` media query is used for modern browser support (although newer browsers tend to recognize the vendor-prefixed media query as well).

The `-webkit-min-device-pixel-ratio` media query checks for a simple ratio of pixel density in which 1 equals 96 pixels. In this case, you're making sure that the display has at least 192 DPI of pixel density before you have it download the higher-resolution images. The `min-resolution` media query takes a more straightforward value of `192dpi`.

At this point, you need to assign the proper background image for each new breakpoint. This means taking the background images in table 5.2 and reworking them so that you're adjusting the background images on the `#masthead` element for high DPI screens. Table 5.3 is the result of this effort.

Table 5.3 Background images for the `#masthead` selector in the CSS, their resolution, and the high DPI screen media queries

Image name	Image resolution	Standard DPI media query	High DPI media query
masthead-xxxsmall.jpg	320 x 135	None (default)	Not used
masthead-xxsmall.jpg	640 x 269	`(min-width: 30em)`	`(-webkit-min-device-pixel-ratio:2), (min-resolution: 192dpi)`
masthead-xsmall.jpg	768 x 323	`(min-width: 44em)`	`(-webkit-min-device-pixel-ratio:2), (min-resolution: 192dpi), and (min-width: 30em)`

Table 5.3 Background images for the `#masthead` selector in the CSS, their resolution, and the high DPI screen media queries *(continued)*

Image name	Image resolution	Standard DPI media query	High DPI media query
masthead-small.jpg	1024 x 430	`(min-width: 56em)`	`(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 44em)`
masthead-medium.jpg	1440 x 604	`(min-width: 77em)`	`@media screen (-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 56em)`
masthead-large.jpg	1920 x 805	`(min-width: 105em)`	`@media screen (-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 77em)`
masthead-xlarge.jpg	2560 x 1073	`(min-width: 140em)`	`@media screen (-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 105em)`
masthead-xxlarge.jpg	3840 x 1609	Not Used	`(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 140em)`

If you compare table 5.3 to table 5.2, it looks similar at first, except each image has been shifted up so that higher-resolution images are used for smaller screen widths. You might remember that masthead-xxxsmall.jpg was used as the default background image for the masthead. For higher-resolution screens, this has been bumped up to the next largest image, which is masthead-xxsmall.jpg. For standard DPI displays, the image masthead-xxlarge.jpg was left unused. This image has now been put into play at the largest breakpoint to cover high DPI displays on large-screen devices.

To begin using the higher-resolution background images, find the start of the high DPI media queries in styles.css. This starts on line 280. You'll see a media query that looks like this:

```
@media screen (-webkit-min-device-pixel-ratio: 2),
              (min-resolution: 192dpi){ /* High DPI Default */
    #masthead{
    }
}
```

This media query is similar to your mobile-first styles, except that it defines the default styles for the page on high DPI screens. Within this media query, you'll need to change the content of the `#masthead` selector to include a new `background-image` property:

```
#masthead{
    background-image: url("../img/masthead-xxsmall.jpg");
}
```

When you make this change, the increase in quality will be noticeable, as it is in figure 5.12. After you've made this adjustment, you need only to work through the list of images in table 5.3 and apply them to their respective media queries.

> **Want to skip ahead?**
> If you're having some trouble or want to skip ahead to see how the code works, you can do so by going to your terminal or command line and using `git`. Go to the folder that the website is running in and enter the command `git checkout hi-dpi-images -f`. Be wary that you could lose any changes you have in your local copy!

In the next section, you'll learn about using SVG images in your CSS, and the advantage this vector image format has over raster images when it comes to efficiently delivering high-resolution images to all types of displays.

5.3.3 Using SVG background images in CSS

Sometimes it may be preferable to use SVGs in background images, depending on the content of the image. As I said earlier in this chapter, if you have an image that contains a lot of line art, SVGs are perfect. When using these image types in CSS, no media queries are required for the image to display flawlessly across all resolutions and display densities.

Find the SVG file in the img folder named masthead.svg. Open styles.css in your text editor and go to line 66 to the `#masthead` selector and change the `background-image` property in this selector to the following:

```
background-image: url("../img/masthead.svg");
```

Then remove all the media queries in the document, starting from line 180, so the background image overrides in those media queries don't override the SVG background you've set. After you make these changes, save and reload the page to see the new background image.

When the new SVG goes into effect, resize the window and observe how the image scales flawlessly at all resolutions. This is the primary advantage of SVG files at work. No media queries required, the image scales properly at all device widths and screen types, high DPI or otherwise.

Again, it's important to remember that this image type isn't suitable for all kinds of content. You'll still be better served by JPEGs and full-color PNG files for photographs. SVGs are best used for content such as logos, line art, and patterns. If in doubt, try SVG to see whether it works. Be sure to take note of the file size and see whether it's efficient for the intended device. If a user can be better served by a smaller raster image, consider switching. Most of the time, the SVG will be the most efficient if the content of the image is well suited to the format.

In the next section, you'll learn about various techniques for implementing responsive images in HTML.

5.4 Image delivery in HTML

Although CSS is useful for managing images in a responsive manner, it doesn't solve the problem of making images responsive when they're referenced from HTML. Since the birth of HTML, the `<img>` tag has been the vehicle for images on the web. Because responsive web design is ubiquitous, it makes sense that there should be a solution to make images responsive when they're used in HTML.

In this section, you'll learn about two approaches to using responsive images in HTML, both useful for different scenarios. These are the `<picture>` element and the `<img>` tag's `srcset` attribute. Although these features are native to HTML, they aren't fully supported in all browsers, so you'll also learn how to polyfill these features in older browsers that don't support them.

Before you do that, however, we have to cover an important CSS rule that should be added to your style sheets.

5.4.1 The universal max-width rule for images

The one rule you should always have in your CSS for any website, responsive or not, is this simple code.

Listing 5.3 The universal `max-width` rule for all `img` elements

```
img{
        max-width: 100%;
}
```

This terse rule does a whole lot of good. It says that any `<img>` element will render to its natural width, *except* when it exceeds its container. If it does, this rule will limit its width to that of its container. Figure 5.13 is an example of this in action.

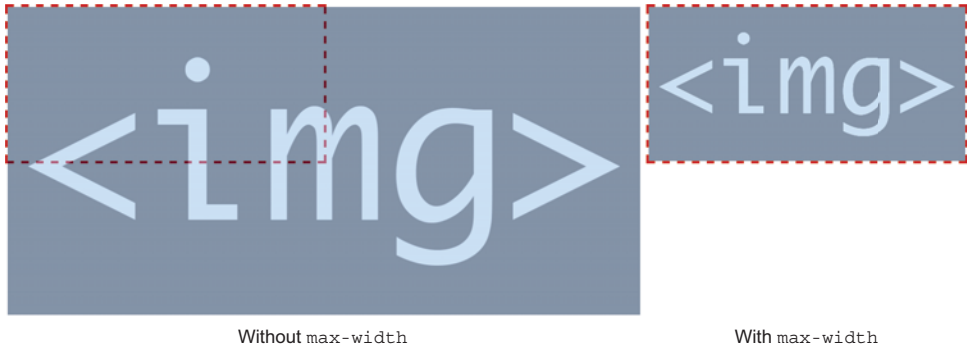


Figure 5.13 A comparison of image behaviors with and without `max-width` restrictions. The example on the left is the default behavior: If the image is larger than its container, it'll exceed the boundaries. On the right is an image with a `max-width` of 100%, which constricts the image to the width of the container.

With this rule in place, you're ensuring that images will behave as they normally do, except when they're larger than their parent containers. With a solid starting point in place, you can move on to using responsive images in HTML in earnest.

5.4.2 Using srcset

One method for displaying responsive images is an HTML5 feature baked into the `<img>` tag, and it goes by the name of `srcset`. This optional attribute for the `<img>` tag doesn't replace the `src` attribute but is used in addition to it.

SPECIFYING IMAGES WITH SRCSET

One example of `srcset` in action follows:

```
<img src="image-small.jpg"
     srcset="image-medium.jpg 640w,
             image-large.jpg 1280w">
```

In this example, the `src` attribute is used for the default image, which in mobile-first sites, should be the smallest in the image set. This acts as a fallback for browsers that don't support `srcset`. The `srcset` attribute here refers to two higher-resolution images (in bold in the preceding example). The format that the `srcset` attribute takes is the image URL, and the width of the image, separated by a space. Image names are in the format you'd normally use in an `<img>` tag's `src` attribute, and the width is denoted using the suffix `w`. For example, an image 512 pixels in width would be denoted as `512w`. Additional images and dimensions can be added. Just separate them with a comma!

The strength of `srcset` is that it doesn't need media queries to work. The browser takes the information it's given and chooses the best image based on the current state of the viewport. This makes `srcset` great for circumstances where all of your images used in a given `<img>` tag have the same treatment but are in different aspect ratios. That's an important bit to remember. If you provide images that aren't in the same aspect ratio as one another, `srcset` *isn't* going to work well and will produce unexpected results. If you need a responsive image approach that allows you to have different treatments at different screen sizes, you can use media queries in CSS, or skip ahead and read about the `<picture>` element.

In this section, you'll continue with your image delivery optimization efforts, and implement `srcset` for an article image on the Legendary Tones website. But first you need to update your working copy of the website to a new branch by using `git`. To do this, run the following command:

```
git checkout srcset -f
```

Next, you can open index.html in your text editor and see on line 26 that a new feature image has been added. If you navigate to the website in your browser, you'll see something similar to figure 5.14.

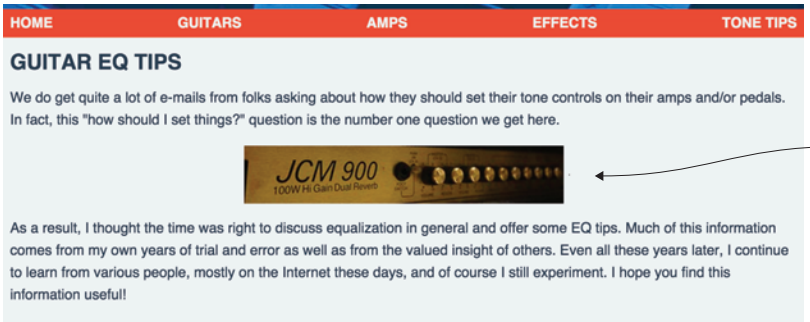


Figure 5.14 The new feature image as it appears on the Legendary Tones website

The annotated feature image displayed in figure 5.14 resides in the img folder, and its filename is amp-xsmall.jpg. The goal of the work on this website at this stage is to get this image to scale to the width of the container while maintaining a reasonable resolution. Before you go about using srcset to achieve what you want, you'll take an inventory of images in the img folder and their widths so you can craft the value for the srcset attribute. This inventory is in table 5.4.

Table 5.4 An inventory of images in the website's img folder and their widths, which will be used for the `srcset` attribute

Image name	Image width (pixels)
amp-xsmall.jpg	`320w` (already referenced in `src` attribute)
amp-small.jpg	`512w`
amp-medium.jpg	`768w`
amp-large.jpg	`1280w`

Using the information in table 5.4, you can construct the content for the `srcset` attribute. In line 26 of index.html, change the `<img>` tag to the following:

```
<img src="img/amp-xsmall.jpg" class="articleImageFull"
     srcset="img/amp-small.jpg 512w,
             img/amp-medium.jpg 768w,
             img/amp-large.jpg 1280w">
```

With this `srcset` value, you'll have a feature image that will span all breakpoints in browsers that support it. The best part is that you don't have to write any media queries to make this work. The browser makes the best choice it can, and does all the leg work for you.

Where optimization is concerned, `srcset` is efficient. This is because the browser downloads only what's needed for the best visual quality. In this example, if you load the page at a large screen size, the browser will load amp-large.jpg, but if you scale

down, the browser won't request amp-medium.jpg, amp-small.jpg, and so forth. The browser will adapt the image that has already been loaded. This prevents unnecessary fetching of image assets. Optimization win!

If you load the page at a smaller screen size and scale upward, the browser will download what it needs to ensure good image quality. So you get to have your cake *and* eat it too. Bottom line: `srcset` grabs only what it needs *when* it needs it.

GETTING MORE GRANULAR WITH SIZES

You may need more flexibility than what `srcset` alone provides. Maybe you need an image to change sizes depending on the screen's width. This is where the `sizes` attribute comes in.

Like `srcset`, the `sizes` attribute is used in the `<img>` tag. It accepts a set of media queries and widths. The media query, like a typical CSS media query, defines a point at which an image should change. The width that comes after it sets how wide the image should appear when that media query takes place. Multiple pairs of these media queries and image sizes can be separated by commas. An example is shown here:

```
<img src="image-small.jpg"
     srcset="image-medium.jpg 640w, image-large.jpg 1280w"
     sizes="(min-width: 704px) 50vw, 100vw">
```

The content of the `sizes` attribute in this example does two things. On screens `704px` and wider, the image is instructed to take up 50% of the width of the viewport. To tell the browser this, you use viewport width (`vw`) units, which are a percentage of the viewport's current width. The next comma-separated rule after this, without the media query, is the default width of the image. If none of the media queries match, the image will be instructed to fill the viewport. Because of the `max-width: 100%` rule on all images (covered earlier in this section), the image will never exceed the width of the its container. To try the `sizes` attribute out for yourself, change the `<img>` tag on line 26 of index.html to the following:

```
<img src="img/amp-xsmall.jpg" class="articleImageFull"
     srcset="img/amp-small.jpg 512w,
             img/amp-medium.jpg 768w,
             img/amp-large.jpg 1280w"
     sizes="(min-width: 704px) 50vw, (min-width: 480px) 75vw, 100vw">
```

With the `sizes` attribute added to this `<img>` tag, the image behavior is affected in different breakpoints, as shown in figure 5.15.

After you make this change, resize the browser window and see how the image adapts to the viewport as you progress through the media queries. As with any responsive image approach, tweaking yields the best results. One rule that's helpful to follow in using `sizes` is that your media queries should be congruent with what you're using in your CSS. You *can* deviate, but always be sure to test, test, test!

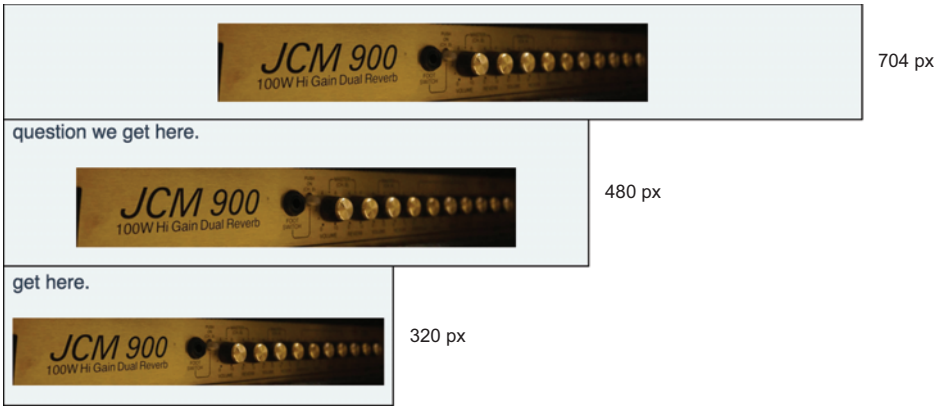


Figure 5.15 The effect of the `sizes` attribute on the article image in Google Chrome. On the 704 px breakpoint, the image takes up 50% of the viewport, at the 480 px breakpoint the image takes up 75%, and the default image behavior below 480 px is to occupy the entire viewport.

Most of the time, `srcset` (and maybe `sizes`) ought to be fine, but in some cases you may need a responsive image approach that allows you to have different treatments at varying screen sizes. This is where the `<picture>` element comes in handy.

5.4.3 Using the <picture> element

`srcset` is capable, but it falls short when you need art direction for your images. *Art direction* is a technique that applies to responsive images, and refers to the practice of giving an image a different cropping and focal point for different screens. This is done when an image for a larger screen isn't optimal for smaller screens, for example. Figure 5.16 is an example of art direction in action, with my neurotic cat as the subject of the image set.

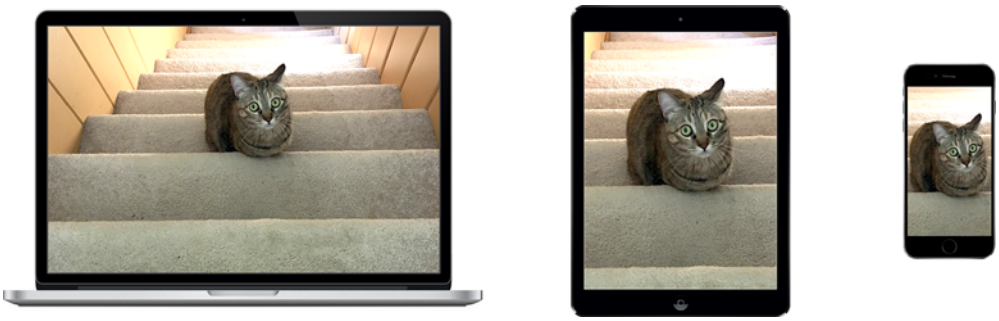


Figure 5.16 An example of art direction across a trio of images. In the largest version, the subject has more context and surrounding details, because larger screens can accommodate more. As the screen width decreases, the image is cropped differently so the subject is still visible on smaller screens.

The `srcset` feature as used on the `<img>` tag isn't a good fit for images that need different treatments in different breakpoints, because it requires that a set of images maintain the same aspect ratio. The `<picture>` element has no such requirement, and will display any image at any defined transition point.

Before you embark on learning how to use the `<picture>` element, you need to switch to a new branch of the website's code by using `git`. If you have work that you'd like to keep, save it before switching. Then run this command:

```
git checkout picture -f
```

This switches you over to a new branch of code, where you'll get to experiment with the `<picture>` element. With the new code in place, you can start using the `<picture>` element to give an article image varying treatment across different devices.

USING ART-DIRECTED IMAGES ON THE LEGENDARY TONES WEBSITE

Reload the Legendary Tones website in your browser and scroll down. You'll see a new article image of a guitar amplifier, as shown in figure 5.17. On screens smaller than 704 pixels, the image sits between the paragraphs and is centered in the viewport. On screens 704 pixels and wider, the image floats to the right, and the text wraps around it.

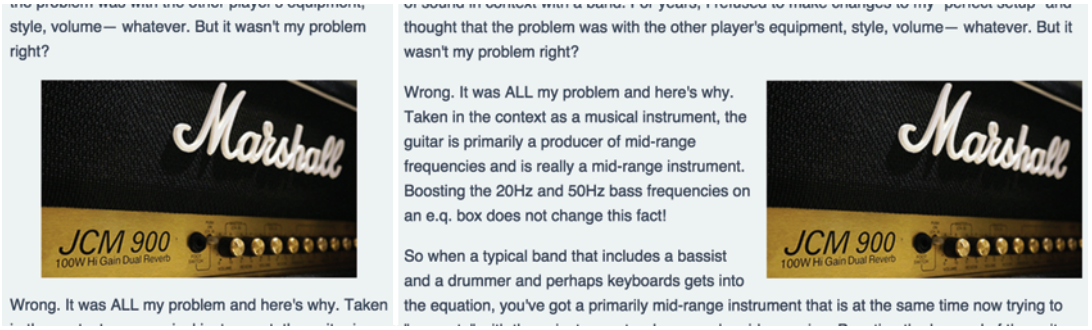


Figure 5.17 Image behaviors on the Legendary Tones website. On small screens (left), the image centers in the viewport and breaks between paragraphs. On large screens (right), the image floats to the right, and the text wraps around it.

The image in figure 5.17 is set in a `<picture>` element, which you can find on line 30 of index.html.

Listing 5.4 The `<picture>` element on the Legendary Tones website

```
<picture>                                  <-- The <picture> tag.
    <img src="img/amp-small.jpg">          <-- Displays if <picture> isn't supported.
</picture>
```

What you want to add to this setup are higher-resolution images for displays that can use them, and provide an alternate cropping of the image for smaller devices. To accomplish this, you add a few `<source>` tags to the `<picture>`, thereby defining more images for the browser to use. The code added to the `<picture>` element is annotated next.

Listing 5.5 Adding new image treatments for different devices via `<picture>`

```
<picture>
    <source media="(min-width: 704px)"
            srcset="img/amp-medium.jpg 384w" sizes="33.3vw">
    <source srcset="img/amp-cropped-small.jpg 320w" sizes="75vw">
    <img src="img/amp-small.jpg">
</picture>
```

If <picture> isn't supported by the browser, <img> will still work.

To achieve your goal, you add two `<source>` tags for different images. The first `<source>` tag contains a `media` attribute that takes effect when the screen is `704px` or wider. When this condition is met, the `srcset` attribute will provide an image `384px` in width, and render it at a third of the viewport size.

When the screen width is less than `704px`, the second `<source>` tag kicks in. This `<source>` has an `srcset` attribute to bring in a different image treatment `320px` in width, and sizes it to 75% of the viewport width. Figure 5.18 shows the effect of your new code.

The power of the `<picture>` element isn't necessarily in itself. It's merely a container for other elements that dictate responsive image behavior, namely the `<source>` and `<img>` elements. The `<source>` elements are where the image configuration is done, and the `<img>` tag is a fallback for browsers that don't support the `<picture>` element. Though the `<img>` tag provides the fallback behavior, it's also necessary for the `<picture>` element to work, so it should never be omitted.

What you've done works well enough for some lower DPI displays, but you should pump this up a bit so that high DPI displays can benefit from higher-quality images.

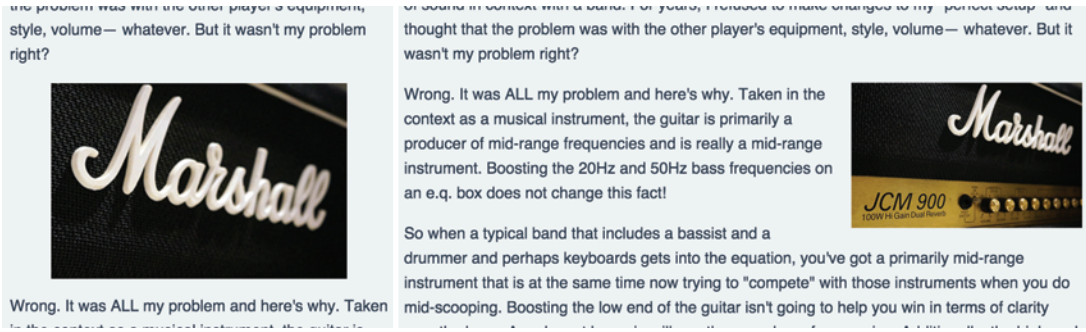


Figure 5.18 Image behaviors of the website after modifications to the `<picture>` element. Note that small screens (left) offer a different treatment of the image based on the screen resolution.

TARGETING HIGH DPI DISPLAYS

You can target high DPI displays with the `<picture>` element quite easily. Doing so requires a bit more tweaking of the `srcset` attributes on the `<source>` tags. For this website, you can modify the contents of the `<picture>` element to provide a better experience for these displays. Modifications are in bold in the following listing.

Listing 5.6 Adding images for high-DPI displays by using `<picture>`

```
<picture>
    <source media="(min-width: 704px)"
            srcset="img/amp-medium.jpg 384w,
                    img/amp-large.jpg 512w"
            sizes="33.3vw">
    <source srcset="img/amp-cropped-small.jpg 1x,
                    img/amp-cropped-medium.jpg 2x"
            sizes="75vw">
    <img src="img/amp-small.jpg">
</picture>
```

A ratio of 1x is assigned to amp-cropped-small.jpg.

A higher-resolution image (amp-large.jpg) is added.

amp-cropped-medium.jpg is added with a ratio of 2x.

These small tweaks do two things: on larger screens, the browser will choose between an image `384px` or `512px` in width, and on smaller screens, the browser will choose between an image appropriate for low-DPI displays (amp-cropped-small.jpg) and an image appropriate for high-DPI displays (amp-cropped-medium.jpg).

In order to signal to the browser which image should be for which type of display, an `x` value is used in place of a width value in the `srcset` attribute. Think of it as a simple multiplier. `1x` marks images appropriate for standard DPI screens, and `2x` or higher signals images that are appropriate for higher DPI screens. If you wanted to, you could even use multipliers of `3x` or higher, since 5K displays are out in the wild now.

Next, you'll use the `<picture>` element to specify fallbacks for different file types, and see how this can be used to take advantage of new image formats while maintaining compatibility with browsers that don't support them.

USING THE TYPE ATTRIBUTE FOR FALLBACK IMAGES

The `<picture>` element also has the ability to reference a series of fallback images based on their type. This is useful when you want to take advantage of any up-and-coming image formats, but want to ensure that less-capable browsers will still be able to use common formats.

You can see a good example of this feature in action with Google's WebP format. WebP is a capable format that, depending on the image content, offers lower file sizes than equivalent formats.

To create a series of fallbacks in the `<picture>` element, you use the `type` attribute on `<source>` elements. `type` accepts an image's file type as an argument for an image specified in the `srcset` attribute.

Continuing on, you'll use the `type` attribute within `<picture>` to establish a preference for a WebP image, and fall back to a JPEG image for browsers that don't support

WebP. In your text editor, go to line 30 and change the content of the <picture> element to the following:

```
<picture>
    <source srcset="img/amp-small.webp" type="image/webp">
    <img src="img/amp-small.jpg">
</picture>
```

With this, you get the best of both worlds: browsers that can handle WebP get WebP, and browsers that can't will fall back to JPEG. Furthermore, because the <img> tag in the <picture> element is the fallback, it'll work in browsers that don't support <picture> itself, making this method a great way to use any kind of format without having to worry about incompatibility.

Now that you've explored the usefulness of the <picture> element, you'll cover how to polyfill the <picture> element and the srcset attribute for older browsers by using the Picturefill library.

5.4.4 Polyfilling support with Picturefill

Although srcset and <picture> are both useful, their browser support isn't universal. Thankfully, you can take advantage of these markup patterns in browsers that don't support them by using a small 11 KB script called *Picturefill*.

Using Picturefill

Like any good polyfill, the strength of Picturefill is in its transparency. You get to use new browser features the way they were intended to be used, and all browsers will play nice. Browsers that support <picture> and srcset will use the native implementation, and other less capable browsers will use Picturefill.

Download Picturefill from https://scottjehl.github.io/picturefill and place it into your project. To see how Picturefill is used firsthand, you can switch to a new branch that already has it in place by entering this command:

```
git checkout picturefill -f
```

After the branch has loaded, open index.html in your text editor and take a peek at lines 7 and 8. You'll see these two <script> blocks in the <head>:

```
<script>document.createElement("picture");</script>
<script src="js/picturefill.min.js" async></script>
```

The first <script> block is for browsers that don't recognize the <picture> element, and prevents problems from occurring if the browser parses them in the HTML before Picturefill has finished loading. The second block then loads the Picturefill library, but does so without blocking page rendering with the async attribute (async is covered in detail in chapter 8).

Believe it or not, this is all it takes for Picturefill to work. After the script is loaded, browsers that didn't support these new image-delivery features in HTML will now support them just fine.

Unfortunately, this approach doesn't prevent browsers that support `<picture>` and `srcset` from unnecessarily downloading Picturefill. Next, you'll see how to use Modernizr to check for `<picture>` and `srcset` support, and load Picturefill for *only* the browsers that need it.

CONDITIONALLY LOADING PICTUREFILL WITH MODERNIZR

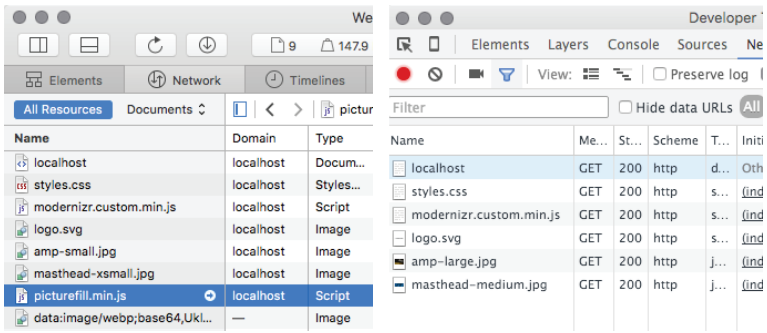
Modernizr (http://modernizr.com) is a robust feature-detection library that provides a simple way to detect browser support for an array of features. In this section, I've created a 1.8 KB custom Modernizr build that contains only feature-detection code for `<picture>` and `srcset`.

You'll use Modernizr to avoid loading the 11 KB Picturefill library in modern browsers, by first checking to see whether the browser needs it. If tests for either feature fail, you load Picturefill. If they both succeed, you don't load Picturefill, and save those browsers the hassle of downloading unnecessary code.

To start, you remove line 8 of index.html, which is the `<script>` block that loads picturefill.min.js. You then add the following code just before the closing `</body>` tag:

```
<script src="js/modernizr.custom.min.js"></script>
<script>
    if(Modernizr.srcset === false || Modernizr.picture === false){
        var picturefill = document.createElement("script");
        picturefill.src = "js/picturefill.min.js";
        document.body.appendChild(picturefill);
    }
</script>
```

The preceding code begins by including the custom Modernizr build. Then, in a separate `<script>` tag, you write a bit of code that checks for `srcset` and `<picture>` support in the `Modernizr` object. If *either* of these checks fails, you create another `<script>` tag, set its `src` attribute to point to Picturefill, and inject it into the DOM. When you examine the network tab in the developer tools in different browsers, you can see that the less-capable browser downloads picturefill.min.js, whereas the modern browser doesn't load it. This is seen in figure 5.19.



Safari Chrome

Figure 5.19 Conditional loading of Picturefill as seen in two browsers' network request inspectors. On the left is a version of Safari that doesn't support the `<picture>` or `srcset` features and therefore loads Picturefill. On the right is Chrome, which fully supports these features and therefore skips loading Picturefill.

With this little bit of code, you're saving those users with better browsers the trouble of having to download Picturefill if they don't need it. Fewer requests and less code mean faster load times for users who can support these newer features.

The next section covers how to use SVG in HTML, and the inherent flexibility of the format when it comes to responsive imagery.

5.4.5 Using SVG in HTML

Similar to the way SVG behaves when used in CSS, SVG in HTML is the best choice when it comes to responsive imagery, assuming that what you want to depict translates well to the format.

> **Warning: An HTTP/2 antipattern is discussed in this section!**
> This section briefly discusses inlining SVGs, which is an appropriate optimization technique for sites hosted on HTTP/1 servers, but should be avoided on HTTP/2. Always ensure that the optimization techniques you choose are appropriate for the protocol version your site runs on.

As with SVG in CSS, the advantages for using SVG in HTML are in the format's flexibility. If you have image content that comports well to the format, you should seriously consider using it, because it'll save you the trouble of having to configure multiple image sources for optimal display across different devices. It's a one-size-fits-all format.

You know how to use the `<img>` tag, so you'll feel right at home using SVG in HTML. You have two nearly universally supported options when it comes to using this image format in HTML, and both methods provide the same benefits that you've seen when placing SVGs into your CSS, in terms of visual quality and ease of use in responsive web pages. The two options are:

- *Use the* `<img>` *tag*—This is the most simple method, and you can try it for yourself in index.html. The logo in the masthead points to a PNG version:

  ```
  <img src="img/logo.png" alt="Legendary Tones" id="logo">.
  ```

  An SVG version of the logo exists alongside logo.png in the img folder, named logo.svg. To use it, point to `img/logo.svg` in the `<img>` tag's `src` attribute:

  ```
  <img src="img/logo.svg" alt="Legendary Tones" id="logo">.
  ```

  When you make this change and reload the page, you'll see the SVG version of the logo in use. When you use an SVG file in an HTML `<img>` tag, there's rarely any reason to use it in a `<picture>` element or with `srcset`, unless you're using it in a `<picture>` element as a part of a series of image fallbacks.
- *Inline the SVG file*—SVGs are XML, which is syntactically similar to and compatible with HTML. Therefore, it's possible to copy and paste SVG files directly into HTML.

Option 2 has a pro and a con. The pro is that inlining an SVG *could* help reduce page-load time by removing an HTTP request, provided that your site isn't hosted on an HTTP/2-enabled server. The downside is that this also makes the resource less cacheable from page to page. Weigh the benefits to see which approach is better. You'll try inlining the contents of logo.svg into index.html as an example. To get started, switch to the `inline-svg` branch with this command:

```
git checkout -f inline-svg
```

Inlining an SVG image is easy. For this website, all you need to do is open logo.svg in your text editor from the img folder, copy the contents of the file to your clipboard, and replace the logo's `<img>` tag with the SVG file contents. Ensure that the *only* thing you're copying are the `<svg>` tag and its contents. Leave out anything else, such as the `<?xml>` header at the top of the file. The final result should look like the following listing, only without truncation.

Listing 5.7 Inlined SVG in HTML

```
<section id="masthead">
    <svg id="logo" xmlns=http://www.w3.org/2000/svg
         viewBox="0 0 216.7 34">...</svg> /
    <h2 id="tagline">Your Source for Great Guitar Tone</h2>
</section>
```

The inlined contents of logo.svg (truncated for brevity)

This scenario isn't the most optimal but illustrates the concept. A good scenario for inlining an SVG is a resource that appears on one page as part of some content. Vectorized infographics are a potential use case for inlining SVG, for example.

Even so, it's crucial to remember that the typical bottleneck of an internet connection is its latency. Inlining reduces load time by spreading latency across fewer requests. Still, effective caching is important too. Weigh your options and see what makes sense for your site.

5.5 Summary

In this chapter, you learned the importance of delivering images to devices that can best use them. As a part of this concept, you learned about these smaller, related topics:

- Delivering responsive images in CSS with media queries, and how supplying the correct image sources to the proper devices can positively impact loading and processing time
- Delivering responsive images in HTML by using the `srcset` attribute and the `<picture>` element
- Providing polyfill support for the `<picture>` element and `srcset` attribute for older browsers in an optimal fashion
- Using SVG images in both CSS and HTML, and the convenience and flexibility inherent to the format when it comes to optimal display on all devices

With these concepts under your belt, you're optimizing your projects so that your users are downloading only the image content that they need, while ensuring that they're receiving the best possible experience. Now it's time to learn image-optimization concepts and methods, such as image spriting, new image compression methods, and using the WebP format.