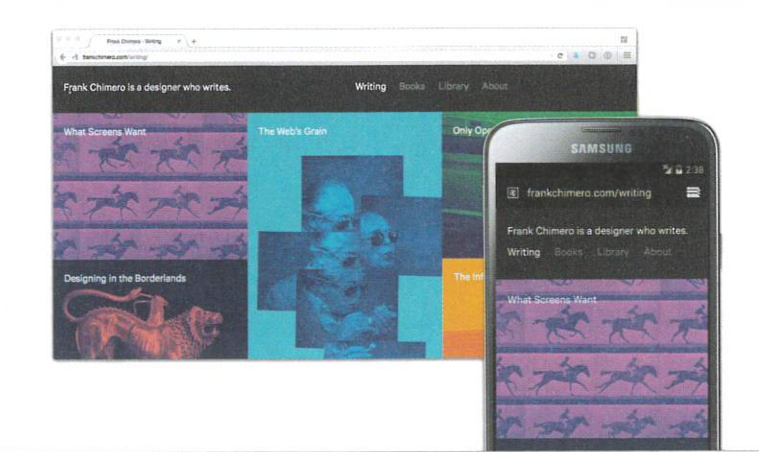

FIG 2.33: Frank Chimero's stunning responsive site, featuring a nav that never quits—or hides (http://bkaprt.com/rdpp/02-27/).

That's not to say responsive navigation systems can't be elegant *and* complex, as the BBC and the *Guardian* have shown. But I suspect that with all the challenges we face on the web, we should constantly search for opportunities to simplify our interfaces. If our responsive navigation can do that, we'll be in a better position to show our users the way.

3 IMAGES AND VIDEOS

> *"So many of the films made today are photographs of people talking."*
> —ALFRED HITCHCOCK (http://bkaprt.com/rdpp/03-01/)

THERE'S BEEN A CONSIDERABLE AMOUNT of writing about how to produce images as flexible as our layouts. In fact, all it takes is a single line of CSS:

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

First discovered by designer Richard Rutter, this single rule says that our images can render at whatever dimensions they want, *as long as* their width never exceeds the width of their containing element. In other words, every image's `max-width` is now set to 100% of its container's `width` (http://bkaprt.com/rdpp/03-02/). If that container gets smaller than the width of the image inside it, our industrious little `img` will resize proportionally, never escaping its flexible column.

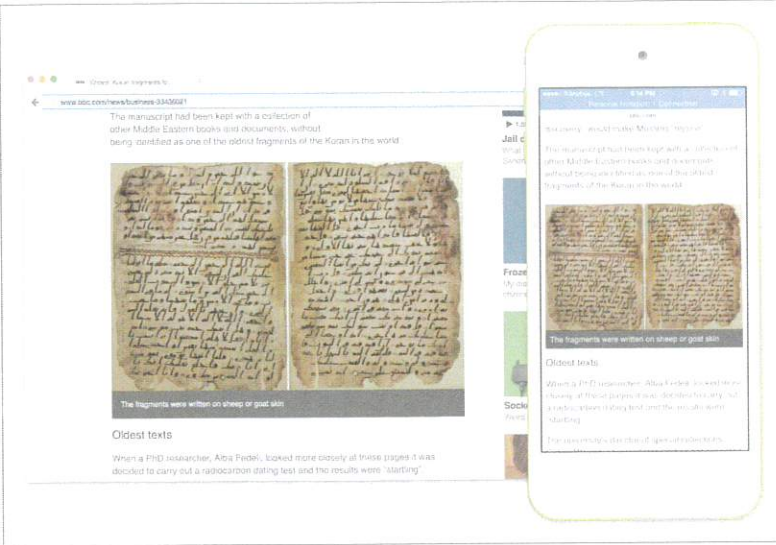

FIG 3.1: A lead image from an article on the BBC News website. As the article's width changes, the image's max-width: 100% allows it to resize proportionally (http://bkaprt.com/rdpp/03-03/).

Alongside fluid grids and media queries, fluid images are one of the three primary ingredients of a responsive layout. And as a result, they're nearly ubiquitous—open any responsive site, like the BBC News' lovely responsive layout, and you'll find images that expand and contract within their containers (**FIG 3.1**).

For me, it's helpful to think of `max-width: 100%` as only part of the story: there are issues of performance, delivery, and design, and we'll cover each in this chapter. In other words, creating *fluid* images is just the first step toward creating more *responsive* images. But before we abandon layout entirely, it's worth mentioning that images aren't the only game in town; after all, our designs need to incorporate other kinds of media, like video. So let's take a moment to make our videos as flexible as our images, and continue from there.

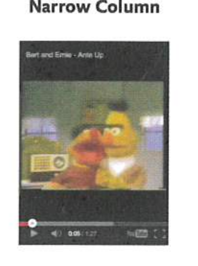

FIG 3.2: Nothing's ever easy on the web: setting our embedded videos to max-width: 100% doesn't *quite* work (http://bkaprt.com/rdpp/03-04/).

TOWARD FLUID VIDEOS

After the past few years of making flexible images, `max-width: 100%` might feel like a natural solution for fluid videos. Unfortunately, it's not quite as easy as that:

```
img,
object,
video {
  max-width: 100%;
}
```

We've extended our CSS slightly, including embedded `objects` and `videos` in our flexible rule. While this works, it doesn't *really* work. If we apply this rule to every video in our responsive layouts, the width of those videos expands and contracts alongside our fluid grid, but the *height* remains fixed (**FIG 3.2**). To see why, let's take a look at the markup behind our movie:

```
<video src="video-main.mp4" height="547"
  width="972"></video>
```

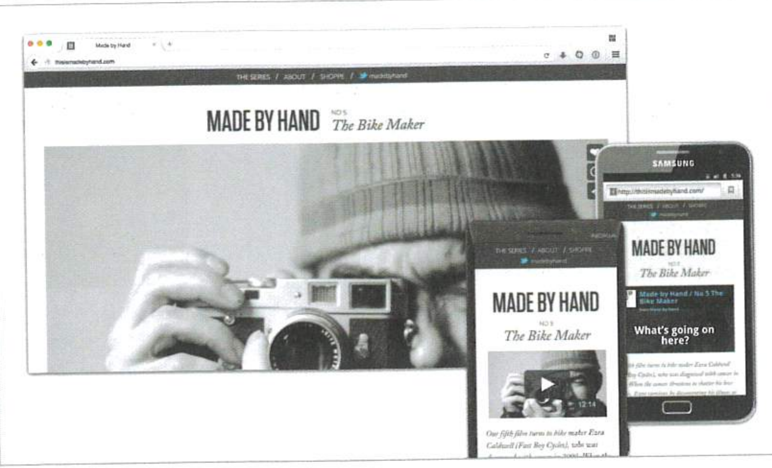

FIG 3.3: Made By Hand, an exquisitely responsive design for an equally moving film series (http://bkaprt.com/rdpp/03-05/).

(A quick note: some third-party services might ask you to use an `object` or `iframe` when embedding their video. The following technique will work for those elements as well, but we'll be sticking with `video` for this demo.)

The markup seems pretty straightforward: the `src` of our `video` element points to, *ahem,* a video file (`video-main.mp4`), while the `width` and `height` attributes determine the dimensions at which our movie should render. But with videos, those last two attributes aren't optional—because unlike images, videos and other embedded objects don't have intrinsic dimensions, so we have to specify them in our HTML. And while we can use `max-width: 100%` to override our video's `width`, we can't do the same with `height`: if we used, say, `height: auto`, our videos would collapse to zero pixels in height and be invisible. And darn it, the internet *needs* its cat videos to be visible.

But luckily, there are a whole host of approaches to making videos resize properly, many of which involve a little light JavaScript. For example, take a look at the responsive site for Made By Hand, a beautiful set of short films featuring rather

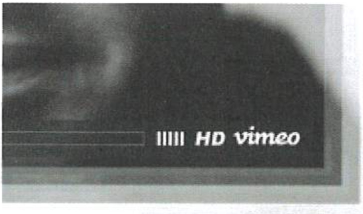

FIG 3.4: Relying on JavaScript for proportional videos is fine, but it's not as smooth as a CSS-only approach—our script causes a slight stutter as our design resizes.

inspirational individuals (**FIG 3.3**). (Responsive or not, the film series is visually stunning and emotionally moving. I highly recommend it.) Since the site's responsive, you can view their videos right in your browser, no matter how wide or small your screen might be. To do so, the site's designers wrote a pinch of JavaScript to measure the video when the page first loads, and store the dimensions for future use. After that, whenever the page resizes itself—or the orientation of a device changes—the video resizes proportionally, using calculations based on those initial measurements.

Many responsive sites have adopted similar JavaScript-enabled tactics. Unfortunately, if you resize your browser while using these sites, you might notice a slight visual stutter. As the design resizes, it often takes a fraction of a second for the video to catch up (**FIG 3.4**). This is partially a performance issue: tying JavaScript to the `resize` event can slow down browsers, or potentially even crash them. But it also underscores the problem in relying on JavaScript for critical parts of our layouts. A significant population of mobile users relies on browsers that offer limited or no JavaScript—and on unstable cellular networks, there's no guarantee our JavaScript will even reach our users.

Thankfully, building completely fluid videos is a solved problem. What's more, it doesn't require a lick of JavaScript. You see, way back in 2009, Thierry Koblentz wrote an article demonstrating how to create videos that resize proportionally in flexible layouts (http://bkaprt.com/rdpp/03-06/). And his approach is, frankly, ingenious.

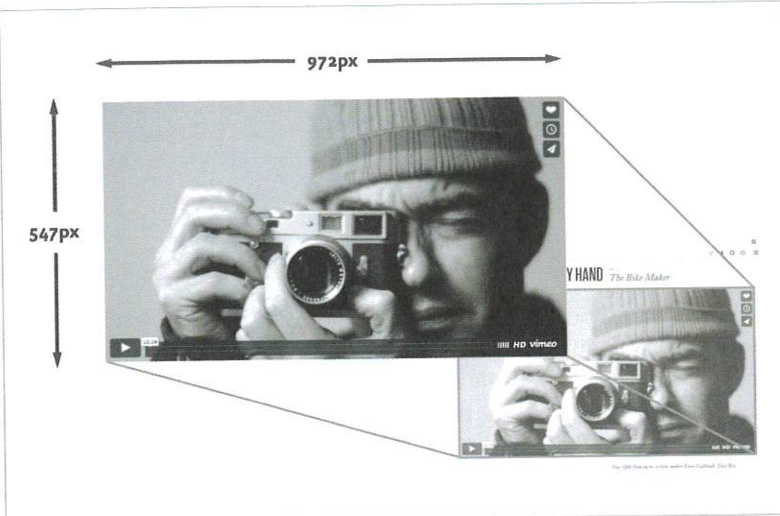

FIG 3.5: Behold a video, embedded in a web page. (I feel like an announcer on one of those wildlife shows.)

Let us pause, if only for a moment, to regard a video on a website. This could be any website, but the homepage for the Made By Hand film series is as good as any (**FIG 3.5**). If we view that page at a viewport width of, say, `1024px`, the video's dimensions are `972×547`—that is, `972px` wide and `547px` tall.

But if we look past the pixels, we're really trying to preserve the relationship between two characteristics of our video—namely, its width and its height. And as it happens, those two measurements have a deep and fundamental connection to each other: the *aspect ratio,* measured from one corner of the video to its diagonal opposite (**FIG 3.6**). Luckily for us, we can calculate that aspect ratio by using a simple formula:

```
height ÷ width = aspect ratio
```

If we plug in the dimensions of our `972×547` video, we're left with the following:

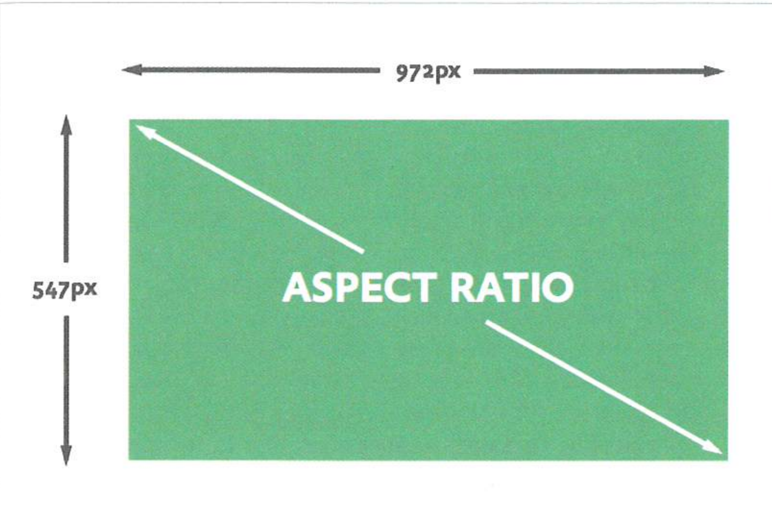

FIG 3.6: The aspect ratio of images and movies describes the relationship between the element's width and its height.

```
547 ÷ 972 = 0.562757202
```

By dividing the height of our video (`547px`) by its width (`972px`), we're left with an aspect ratio of `56.2757202%`. So as the video resizes, the height of the video should remain roughly 56% of the video's width.

We'll come back to that percentage in a bit, so put it in your back pocket for now. (Or your cargo shorts, if that's your metaphorical legwear of choice. No judgment.) With the math out of the way, let's go back to the `video` element in our HTML:

```
<video src="video-main.mp4" height="547"
  width="972"></video>
```

As simple as this markup is, let's make two small adjustments to it:

```
<div class="player">
  <video src="video-main.mp4" height="168"
    width="300"></video>
</div>
```

Not much has changed, but we've sized the video down considerably, setting its `width` and `height` to be small-screen-friendly by default. (After all, there's no need to plop a massive video onto smaller screens, right?) More significantly, we've added a little more markup: namely, there's an element wrapped around our `video` element—we've chosen a `div` with a class of `player` here, but the container could be anything you want.

But once it's combined with the aspect ratio we measured previously, that unassuming container is the key to making our video responsive. Let's begin by applying some styles to the outermost `div`:

```
.player {
  padding-top: 56.2757202%;
}
```

Okay, maybe not so much "styles" as "style": with one rule, we've added a `padding-top` equal to the aspect ratio we calculated earlier. But why, you might ask? Well, according to the CSS specification, percentages on `padding-top` and `padding-bottom` are relative to the *width* of the containing block, not the height (http://bkaprt.com/rdpp/03-07/). As a result, that vertical padding will always be `56.2757202%` of the box's width.

Here's a quick example: I've rooted around in my browser's inspector, and removed the video from the Made By Hand homepage. I also disabled the JavaScript that resized the video, *and* added that `padding-top` to its container. And finally, because I am a very professional web designer, I added a not-at-all-garish background color (FIG 3.7). But as we resize the design, the `padding-top` resizes as well: it's always roughly 56% of the container's width. In other words, our container `div` might be completely empty, but it has an intrinsic aspect ratio. No matter how wide or small that block gets, its height

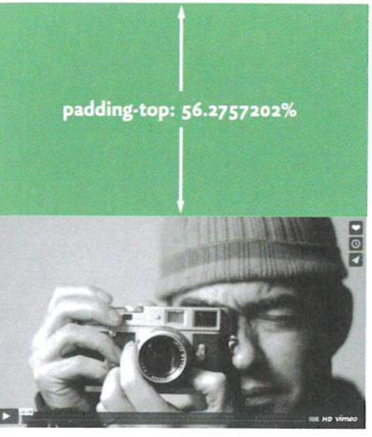

FIG 3.7: By applying the aspect ratio as a percentage-based `padding-top` to our container, we've created an empty "ghost box."

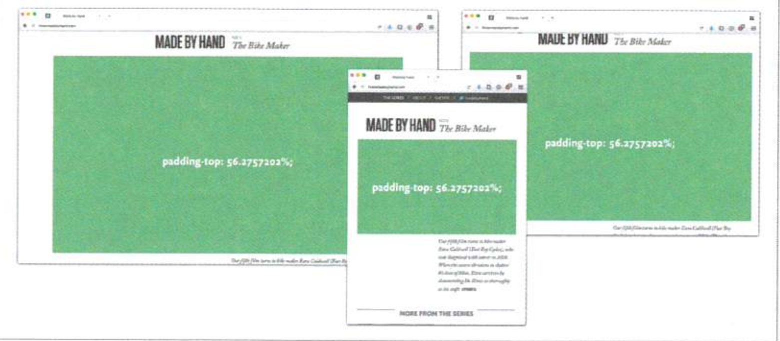

FIG 3.8: As we resize our `padding-top`-enabled box, it maintains the shape and proportion our video needs, without a single scrap of content inside.

will always be `56.2757202%` as tall as its width. The empty area created by our `padding-top` is *aspect ratio-aware* (FIG 3.8).

Pretty darned cool, right? Well, I think it's cool. (I might have just figured out why I'm never invited to any parties.) But it's only the foundation for our flexible video. With that percentage-based `padding-top` in place, we can go back to our CSS and add a few more styles:

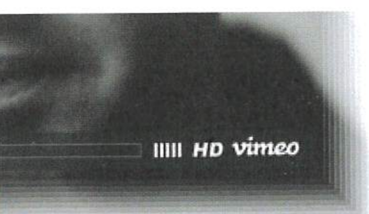

FIG 3.9: With some proportional math and a little extra markup, our video is now resizing responsively—all without a single line of JavaScript.

```
.player {
  position: relative;
  padding-top: 56.2757202%;
}
.player video {
  position: absolute;
  left: 0;
  top: 0;
  height: 100%;
  width: 100%;
}
```

To begin, we've added `position: relative` to the `.player` container. This creates what's known as a *positioning context:* any element absolutely positioned inside the context of that container will now be positioned relative to `.player`, rather than the viewport. And that's what allows the second rule to work: we're positioning the `video` in the top left corner of `.player`. Immediately after that, we're setting the video's `width` and `height` to `100%`, which ensures that they'll be equal to the width and height of their containing element.

If we return to our browser and reinstate the video, we can see the final effect in action (**FIG 3.9**). Remember, that container has an intrinsic aspect ratio: thanks to the percentage-based `padding-top`, the height of our `.player` box will resize proportionally, no matter how wide it becomes. With that in place, we've taken our video and—with some absolute positioning—stretched it across the entirety of our container. And the effect is much, much smoother than if we'd relied on JavaScript.

With nothing more than a little proportional math and an extra container, we've got fluid videos resizing seamlessly within a responsive design.

WORKING WITH FLEXIBLE BACKGROUNDS

`max-width: 100%` is, of course, wonderful—but only for inline images. For flexible background images, we have a number of helpful CSS properties available to us—most notably, `background-size`.

Typically, when we're applying background images to an element, we're asking the browser to render that image at its native resolution. Here's a fairly basic `background` rule:

```
.intro {
  background: url("bg-demo.jpg") no-repeat;
}
```

The browser will apply `bg-demo.jpg` to our `.intro` block, and render that image at its native dimensions. And that's the outcome regardless of whether the image in our rule is four thousand pixels wide or fourteen—if the image happens to be wider or taller than the containing block, the extra pixels won't be displayed.

However, we can override that behavior with the `background-size` property, which allows us to specify the size we'd like our images to display at. We can specify lengths as well, ensuring that our image displays at `250×400`:

```
.intro {
  background: url("bg-demo.jpg") no-repeat;
  background-size: 250px 400px;
}
```

Alternately, if we specify one of the lengths as `auto`, the image will scale proportionally to a specific width or height. For

example, a `background-size` of `250px auto` sets our image's width to `250px` without distorting its aspect ratio:

```
.intro {
  background: url("bg-demo.jpg") no-repeat;
  background-size: 250px auto;
}
```

We can even define our `background-size` in percentages, scaling the image relative to the dimensions of its container. So if we wanted our image's width and height to be 50% of `.intro`'s width and height, our rule would look like this:

```
.intro {
  background: url("bg-demo.jpg") no-repeat;
  background-size: 50% 50%;
}
```

As fun as `background-size` is, it's worth noting that older versions of Internet Explorer (versions 8 and lower) don't support it. If you're worried about fallbacks for those older browsers, I might suggest a variation on Paul Irish's conditional comments technique (http://bkaprt.com/rdpp/03-08/). In fact, you can see this in the HTML for http://responsivewebdesign.com/:

```
<!DOCTYPE html>
  <!--[if IE]><![endif]-->
  <!--[if lt IE 9]>  <html class="oldie ie">
  <![endif]-->
  <!--[if IE 9]>     <html class="ie ie9">
  <![endif]-->
  <!--[if gt IE 9]>  <html class="ie"><![endif]-->
  <!--[if !IE]><!--> <html> <!—<![endif]-->
```

With those conditional comments in place, older versions of IE will have a class of `oldie` applied to their opening `<html>` tag. As a result, I can apply an acceptable fallback style by starting a second selector with `.oldie`:

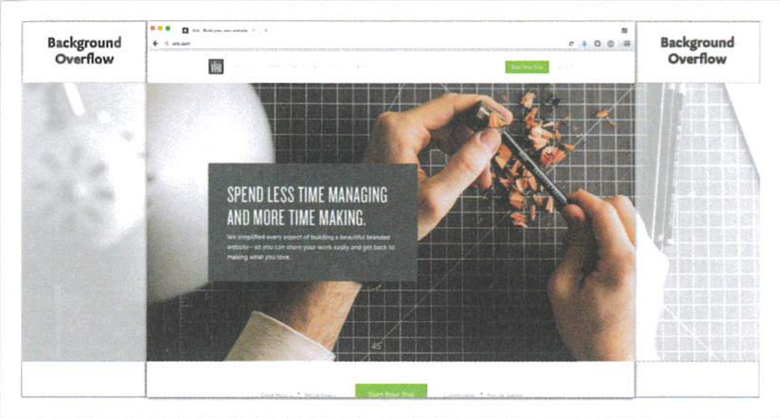

FIG 3.10: By using `background-size: cover`, the lead photo on Virb's responsive homepage proportionally resizes to, well, *cover* its container.

```
.intro {
  background: url("bg-demo.jpg") no-repeat;
  background-size: 50% 50%;
}
.oldie .intro {
  background-image: url("bg-demo-noresize.jpg");
}
```

With our fallbacks sorted, let's take a look at two incredibly useful keywords we can apply to the `background-size` property: `cover` and `contain`. Let's start with `cover`:

```
.intro {
  background: url("bg-demo.jpg") no-repeat;
  background-size: cover;
}
```

The browser will evaluate the width and height of the background image, and find the smaller of the two values. Once that's done, it will scale the image proportionally, ensuring that the smaller dimension—either the width or the height—covers

FIG 3.11: Need your background image to be completely visible *and* flexible? There's a `background-size: contain` for that.

its container. You can see this in action on Virb's responsive homepage (http://bkaprt.com/rdpp/03-09/). The lead image's native dimensions are `1600×600`. Since the height (600 pixels) is smaller than the width (1600 pixels), the image stretches vertically over the height of its container (**FIG 3.10**). No matter how large—or small!—that box becomes, the background scales proportionally to perfectly cover it.

Applying `background-size: contain` will also scale our backgrounds, but the resulting layout is quite different:

```
.intro {
  background: url("bg-demo.jpg") no-repeat;
  background-size: contain;
}
```

Whereas `background-size: cover` may occasionally hide parts of our images from view, `background-size: contain` ensures the entire background is always visible within its container (**FIG 3.11**).

When combined with `background-position`, `background-size` can create some really stunning image treatments. The

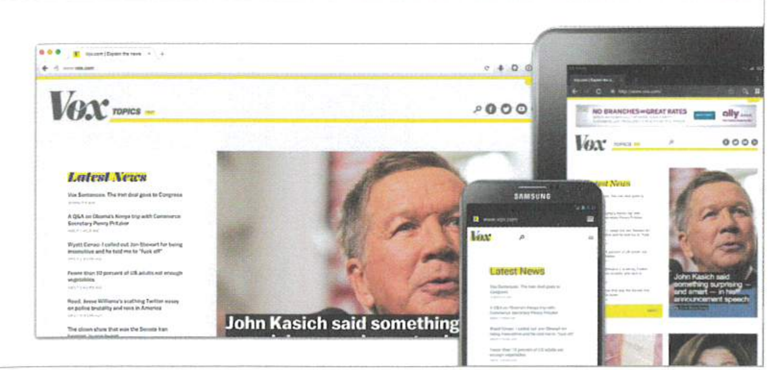

FIG 3.12: Vox's responsive homepage is a stunning combination of background images and lovingly typeset text (http://bkaprt.com/rdpp/03-10/).

homepage of Vox.com combines the two properties beautifully (**FIG 3.12**). Each of the blocks show featured stories and headlines with a flexible background image:

```
.content {
  background: url("beyonce_grammy.jpg") no-repeat;
  background-size: cover;
  background-position: center, center;
  height: 600px;
}
```

Rather than anchoring their images top and left within each block, Vox uses `background-position: center, center` to, well, center them within their containers (**FIG 3.13**). And with that positioning in place, `background-size: cover` ensures that each block is covered by a perfectly centered, flexible background.

In theory, Vox could use media queries to load different images at certain breakpoints, perhaps loading in widescreen-appropriate crops as the viewport expands:

```
.content {
  background: url("beyonce_grammy.jpg") no-repeat;
  background-size: cover;
  background-position: center, center;
  height: 600px;
}
@media screen and (min-width: 39em) {
  .content {
    background-image:
      url("beyonce_grammy-medium.jpg");
  }
}
@media screen and (min-width: 60em) {
  .content {
    background-image:
      url("beyonce_grammy-wide.jpg");
    background-position: 0 0;
  }
}
```

The sky is, as the kids say, the limit.

SCALING RESPONSIBLY: SRCSET AND SIZES

There are some very real downsides to simply scaling or shifting images with a bit of CSS. But thankfully, there are tools to help us tackle them. Let's step through each in turn.

First, CSS-based resizing can often be bad for the weight of our work. As of the middle of 2015, the average weight of a web page was 2.1MB (http://bkaprt.com/rdpp/03-11/), up from a relatively paltry 320KB in 2010 (http://bkaprt.com/rdpp/03-12/). And most of that weight? You guessed it: images. Our beloved JPGs, PNGs, and GIFs comprise more than 60% of that 2.1MB footprint—over 1.2MB per page on average.

Most of that has come about with high-density displays. In 2012, web developer Jason Grigsby found that the Apple.com homepage jumped in size from 500KB to well over 2 MB, simply by upgrading its images to higher-resolution versions that

FIG 3.13: By applying `background-size: cover` to their centered background images, Vox.com can feature content alongside rather evocative—and completely flexible—background images.

would look crisp on high-density screens (http://bkaprt.com/rdpp/03-13/). And Apple's not alone—as our screens have gotten sharper, our images have gotten bigger, bulking up our pages.

Given this ever-increasing page size, we should attend to the amount of data we're asking our users to download, rather than simply squishing down massive images to fit smaller screens. Now, to be clear: "small screen" does not imply "slow connection." Far from it. In fact, there is *no* correlation between the width of a screen and the amount of bandwidth available to it. My laptop could be tethered to a phone's 3G connection, on a steady ethernet connection, or on a hotel's barely-functioning Wi-Fi network; conversely, my phone could just as easily be on a fast, reliable Wi-Fi network as it could be connected to a flaky cellular signal. Right now, there's simply no way to detect the amount of bandwidth actually available to our users' devices.

To their credit, browser and hardware vendors are working on ways for us to detect a user's connection speed, like the

Network Information API (http://bkaprt.com/rdpp/03-14/), but a standard solution hasn't been established yet. In the short term, I think that uncertainty is actually okay. If anything, it emphasizes the need to reduce the amount of data we're serving to our users, regardless of the size of their screen. Jake Archibald, a developer advocate for Google Chrome, suggests that lower-end networks should be our true priority (http://bkaprt.com/rdpp/03-15/):

> *It's important to focus on 3G load times, because even though we have 4G now, those users are on 3G (or worse) a lot of the time: a quarter of the time in the United States, half the time in large parts of Europe.*

The commercial benefits of lighter pages are legion. GQ (http://bkaprt.com/rdpp/03-16/) recently found that its responsive redesign was entirely too slow—but by reducing page load time by 80%, its number of unique visitors jumped by 80% (http://bkaprt.com/rdpp/03-17/). If we assume *all* our users may have low bandwidth, it can help us lighten our sites and create interfaces that are fast for everyone—whether accessed on mobile, desktop, or something else entirely.

And that brings us back to flexible images. Those massive images can be resized to fit on smaller devices, but even if those tiny screens are on fast connections, they'll be downloading a lot of pixels they won't use. It's *invisible overhead,* and we should reduce it whenever possible. Thankfully, some standards-based tools are emerging to help us tackle this problem, authored by the Responsive Issues Community Group (http://bkaprt.com/rdpp/03-18/). They've worked with browser vendors to produce a number of additions to the HTML specification, specifically some attributes to make our images a bit more intelligent.

To begin, let's look at our dear friend, the humble `img` element:

```
<img src="img/main.jpg" alt="A friendly-looking dog"
  />
```

Nothing too fancy, right? Our `img` element has a `src` that points to the URL of an image file (`img/main.jpg`), accompanied by

some accessible `alt` text to describe the contents of our image (`"A friendly-looking dog"`). And if we've done our job right, `main.jpg` should show up in our browser. But here's the thing: that image file is going to be served to every browser and device that accesses our page, regardless of its network speed, screen density, or viewport size.

To help our image scale more efficiently, we'll add one of the new responsive image tools: namely, the `srcset` attribute.

```
<img srcset="img/main-200.jpg 2x, img/main-300.jpg
  3x" src="img/main.jpg" alt="A friendly-looking
  dog" />
```

...okay, hold on a moment. Our once-pristine `img` element now looks like a Perl script threw up in the middle of our HTML. What, pray, is all that gibberish inside our `srcset` attribute? Commas? `2x`? `3x`? What is happening here?

Thankfully, it's not as bad as it looks. To decipher our `srcset` attribute, let's make it more legible:

```
<img srcset="img/retina.jpg 2x,
             img/retinarok.jpg 3x"
  src="img/normal.jpg" alt="A friendly-looking dog"
  />
```

That's a bit better. What we've done is create three different versions of our image, each identical, but with different *pixel densities:* their dimensions are the same, but they're tailored to be viewed on increasingly higher-resolution displays. And inside our `srcset`, we're simply spelling out the path to each image, and then describing its ideal pixel density—`2x`, `3x`, and so on.

With those images and resolutions spelled out, our `img` tag is no longer loading one image for all screens: instead, our `srcset` is filled with *options* of multiple images that could be loaded, depending on which is best for the user. Armed with that information, the browser can select the image best suited to the density of the display. That prevents us from saddling

main-large.jpg

main-medium.jpg

main-small.jpg

FIG 3.14: Our three new images. Each of them is identical, except for their dimensions: they've simply been scaled down.

lower-resolution screens with incredibly complex images, and conserves some bandwidth in the process.

Neat, right? Unfortunately, those `x` descriptors are intended for *fixed-width* images, which are so, like, 1999. But all is not lost: we can use `srcset` to negotiate images based on the width available to them in the layout. Let's look at another `img`:

```
<img srcset="img/main-large.jpg 1440w,
  img/main-medium.jpg 720w, img/main-small.jpg 360w"
  src="img/main-medium.jpg" alt="A friendly-looking
  dog" />
```

And yep, once again, that looks a little terrifying. Sorry about that! Once you've climbed down from the flagpole, we can add a few well-placed line breaks, and produce something that looks a bit more sane:


```
<img srcset="img/main-large.jpg 1440w,
             img/main-medium.jpg 720w,
             img/main-small.jpg 360w"
    src="img/main-small.jpg" alt="A friendly-looking
    dog" />
```

As before, we've created three different versions of our image—but this time, they're each identical except for their scale: they only vary in size (**FIG 3.14**). Similarly, our `srcset` spells out the paths to each image, separated by commas. But this time, instead of using `2x` or `3x` modifiers to describe the density of the image, we're describing each image's width in pixels, followed by a `w`. Since `main-large.jpg` is `1440px` wide, the `1440w` allows us to tell the browser its native width. The same is true for our `720px`-wide `main-medium.jpg` and our `360px`-wide `main-small.jpg`—each is described as having widths of `720w` and `360w`, respectively.

(Quick aside: sharp readers will note that in both of our code snippets, there's still a `src` on our `img` elements. Strictly speaking, `src` is required by the responsive images specification—your images *must* have `src` attributes, even if you're using `srcset` (http://bkaprt.com/rdpp/03-19/). This might seem redundant, but is actually a boon for backwards compatibility. If a browser doesn't understand `srcset`, it'll still download an image.)

After specifying three different widths for the lead image, you might be wondering how we decide which image the browser loads. And that's a perfectly reasonable thing to wonder! But here's the thing: *we don't*. If you read the specification, there's nothing telling browsers how to "pick" the best option from `srcset` (http://bkaprt.com/rdpp/03-20/). It's up to the browser to choose the best image—not us.

...okay, I know how that sounds. Maybe you're mildly panicking. Maybe you're more-than-mildly panicking! After all, we're the designers! We should have the final say in which images our users see, right?

But really: don't worry. This lack of control is actually a good thing. Consider that these images aren't just chosen for which has the best "fit" for our layout: an image from `srcset`

could be selected to match the speed of the user's network, the resolution of her display—or, or, *or*. There are countless factors that determine which image is the best pick. And while some things in our responsive images toolkit allow us a higher degree of control, determining the best *resolution* for our image is best left to the browser. It'll keep our markup lighter, and our users happier.

While we can't choose the best option out of `srcset`, we can help the browser make a more intelligent selection. To do that, we'll add a `sizes` attribute:

```
<img srcset="img/main-large.jpg 1440w,
             img/main-medium.jpg 720w,
             img/main-small.jpg 360w"
  sizes="(min-width: 50em) 250px,
         (min-width: 35em) 33vw,
         100vw"
  src="img/main-medium.jpg" alt="A friendly-looking
    dog" />
```

So again, I realize that `(min-width: 50em) 250px, (min-width: 35em) 33vw, 100vw` looks a lot like unadulterated, robot-generated gobbledygook. But as with `srcset`, we're creating a list of items our browser, each separated by a comma. Basically, each entry in our list describes the physical width of our image at different points in our responsive design—that is to say, the *size* it will occupy in the layout.

Let's walk through our `sizes` attribute, and see if we can't decipher it:

1. `(min-width: 50em) 250px` looks a little like a media query, doesn't it? In fact, that's basically what it is: we're telling the browser that if the viewport has a minimum width of `50em`, the image will be `250px` wide.
2. `(min-width: 35em) 33vw` works basically the same way: if the viewport has a minimum width of `35em`, the image will be `33vw` in width. But what's a `vw`, you ask? Well, a `vw` is just another unit of length in CSS, equal to 1% of the viewport's width. So `33vw` is another way of saying the image will occupy 33% of the width of the viewport.
3. `100vw` is the default value for the `sizes` attribute. It means the image will occupy the full width of the viewport. So if the condition in our first `sizes` entry isn't met—that is, if our viewport is below that `min-width: 40em` threshold—then our image will be sized at 100% of the viewport's width.

The sizes listed in our, um, `sizes` attribute don't need to perfectly match the image's size at each breakpoint. What we're trying to capture is an *approximation* of the image's width as the layout changes. Once it's armed with information about how an image will be laid out, the browser can intelligently select an image from our `srcset` list and pick the best possible option to load.

(Quick tip: if you're a fan of valid HTML, the `sizes` attribute is actually required by the specification. In other words, if you use `srcset`, it should be accompanied by a corresponding `sizes`. At the time of this writing, `srcset` will still *work* without `sizes`, but it'll make your markup invalid. So tread carefully.)

A note about support, before we continue:

- A number of non-desktop browsers offer fair support for `srcset` and `sizes`. On both iOS and Mac OS X, Safari has partial support for the two attributes. It supports resolution switching with the `x` descriptor, but not with `w`. Newer default browsers for Android—including Chrome and the default Android browser—support the attributes handily.

  However, it's not all good news: neither Android's default browser (as of Android 4.4.4) nor Opera Mini (as of version 8, at least) support `srcset` or `sizes`. Nor do they natively support any other part of the responsive images specification—and that's unfortunate for us, as both of these browsers are massively popular.
- In happier news, support for `srcset` and `sizes` is fairly robust among most modern desktop browsers. Chrome, for example, has supported `srcset` and `sizes` since version 38, while Opera has supported it since version 26. At the time of this writing, Firefox is still finalizing its implementation of

`srcset` and `sizes`, but allows you to activate the attributes if you root around in the developer preferences. And while Internet Explorer hasn't shipped a working version, it is actively working on support for the new attributes (http://bkaprt.com/rdpp/03-21/).

Overall, the support for responsive images is impressive, but still in its infancy—a significant number of browsers don't yet support `srcset` or `sizes` natively. However, you can use a JavaScript library like Picturefill (http://bkaprt.com/rdpp/03-22/) to patch responsive image support into older browsers. Simply download Picturefill into your project, and include the following in the `head` of your document:

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

And with that, you'll have responsive images working seamlessly in your flexible layouts, *responsibly* resizing with a little help from `srcset` and `sizes`.

But watch your step, dear reader: while `srcset` and `sizes` can make CSS-resized images a little more weight-conscious, sometimes scaling an image with CSS isn't ideal. In some cases, flexible images can harm more than they help.

When resizing brings regret

...okay, I apologize for the dire note. There's nothing *wrong* with a little `max-width: 100%` to make your `img` elements more flexible—and what's more, it'll work perfectly for most of your images. But sometimes, simply scaling images up or down can reduce their clarity.

Here's a quick example: open "What is Paul Krugman Afraid Of?" (http://bkaprt.com/rdpp/03-23/) on a reasonably large screen, say, a laptop or a tablet. Throughout the interview, you'll notice several photos with text overlaid on them (**FIG 3.15**). Now, there are a number of potential accessibility issues

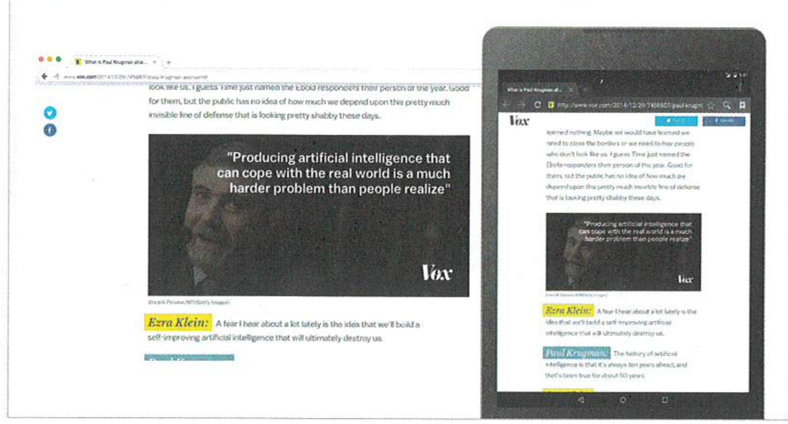

FIG 3.15: Vox features quite lovely visual pull quotes, with text overlaid atop images.

in typesetting pull quotes in images—they'd need `alt` attributes to be accessible to non-sighted readers—but let's put those aside for a moment, and focus on the responsive layout alone.

Since the article's layout is responsive, Vox used `max-width: 100%` to ensure that as their flexible grid reshapes itself, the images never break out of their containing elements. Because of that, opening the same article on a smaller screen resizes the images, but the pull quotes are considerably less legible than on wider screens (**FIG 3.16**).

This is equally true on complex images and charts, if not more so. For example, take the map near the top of this page on Columbia's School of Engineering site (http://bkaprt.com/rdpp/03-24/; **FIG 3.17**). In addition to the colored blocks representing energy consumption across New York City neighborhoods, there's the title of the graph, a legend to help you decipher the map, pie charts for specific land areas, and so on. The image is, in other words, incredibly dense. So while it could be resized, all of those finer details would be lost, and the meaning of the image would degrade.

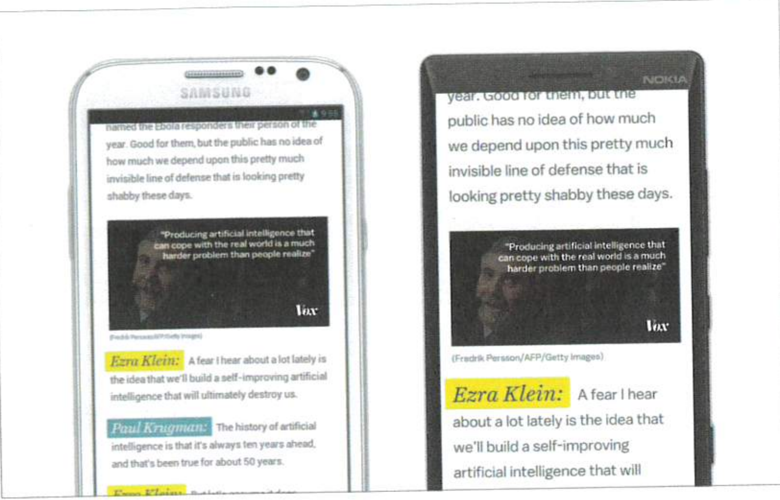

FIG 3.16: On smaller screens, simply resizing the images makes their text hard to read. Flexible, but frustrating.

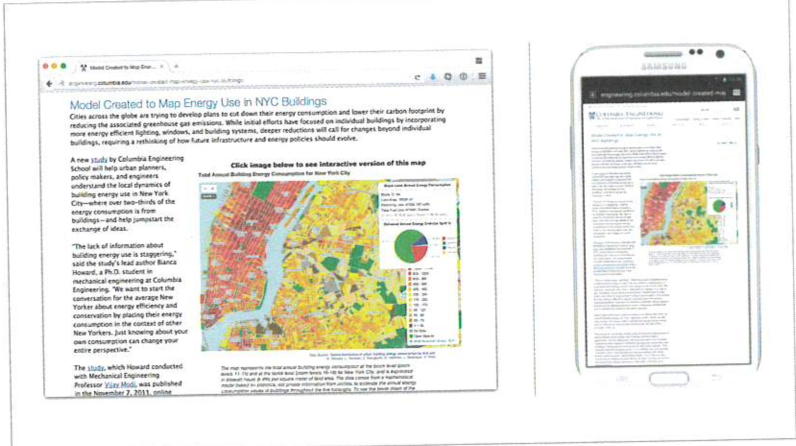

FIG 3.17: A useful and interactive map of New York. Is there a way to effectively resize something this dense?

The form, frame, and shape of our images

The problem with CSS-based resizing is that it's *content-blind:* it focuses on the shape of the container that holds the image, not the image itself. And sometimes, if we're not paying attention, those images can be resized past the point of usefulness.

This isn't a new problem. Long before the advent of the web, photographers and graphic designers have been concerned with resizing their work effectively, and how to preserve the integrity of their artwork across differently-sized media. In the middle of the twentieth century, the Swiss designer Karl Gerstner applied a systematic approach to the problem, demonstrating how a design system could be used to adapt a wordmark so that it doesn't just fit in different paper formats, it *thrives* (**FIG 3.18**).

More recently, designer Raymond Brigleb charted the rise of the cassette tape's popularity in the '80s and the challenge it posed for the designers of LP album covers. Constrained by the cassette's smaller size and unforgivingly weird aspect ratio, designers changed the layout, size, and position of key elements to preserve the message they wanted to convey (**FIG 3.19**).

Even the process of cropping a photograph relies on understanding the contents of the picture, not simply its dimensions. A photographer identifies the primary subject of a photograph—the focal point—and trims away the inessential parts of the picture. Various crops of a photograph may differ greatly in the amount of detail shown, but the subject is usually consistent. As different as they might look, all of the crops are, in essence, the same photo (**FIG 3.20**).

In looking at the problem of making our images not just resize, but *respond,* there have been some attempts to automate intelligent image cropping. For example, Adam Bradley built a framework that allowed designers to apply CSS classes to an image's container that would, in turn, preserve the focal point as the image scaled up or down (http://bkaprt.com/rdpp/03-25/).

So it's absolutely possible—and often ideal—to simply resize your images with a mixture of `max-width: 100%`, `srcset`, and `sizes`. But it's worth remembering that the images inside our documents are actually documents themselves. After all,

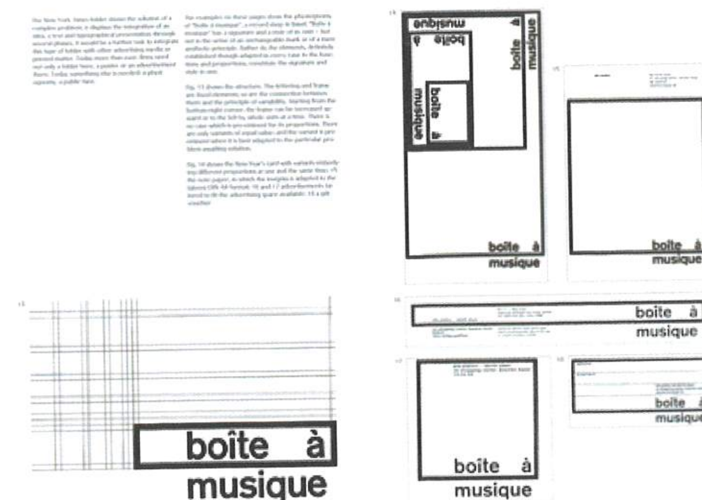

FIG 3.18: In his book *Designing Programmes,* Karl Gerstner demonstrated how a well-thought-out design system could maintain a logo's integrity on any number of printed formats, from full-sized advertisements to handheld gift cards (http://bkaprt.com/rdpp/03-26/).

they're there to convey information to our readers, so we should ensure the message survives at any scale.

FINER-GRAINED CONTROL: PICTURE AND SOURCE

In other words, you might come across situations when images shouldn't be resized, but *replaced*—swapped for alternate files optimized for different breakpoints, ensuring that they maintain clarity even as their containers expand and contract. When that happens, you'll want to specify a completely different image to load.

FIG 3.19: In reviewing how album cover art had to adapt across LPs and cassettes, Raymond Brigleb demonstrates the need for responsive images (http://bkaprt.com/rdpp/03-27/). (And suggests, I think, that our problems on the web aren't entirely new.)

FIG 3.20: While the dimensions of a photograph may change from crop to crop, the focal point remains intact. Photograph by Tim Evanson (http://bkaprt.com/rdpp/03-28/).

Of course, that's just for background images. Inline images, such as those specified by our industrious `img` element, need some extra help. And that's where the new `picture` element comes in. As it happens, Shopify's responsive site has a great example of `picture` in action. Near the top of their homepage is a photo of a Shopify customer, which is repositioned at different breakpoints (**FIG 3.21**). But if you look under the hood, you'll see that it's not one photo, but three—each sized and cropped slightly differently from the others (**FIG 3.22**). And if

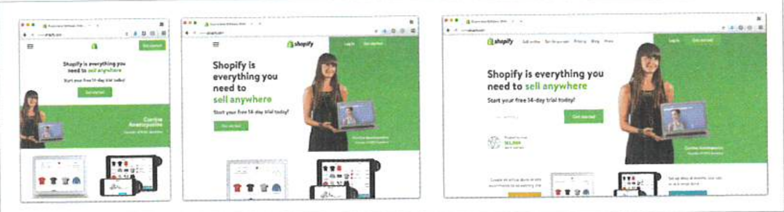

FIG 3.21: Follow the photo: the lead picture on Shopify's responsive homepage is repositioned at different breakpoints (http://bkaprt.com/rdpp/03-29/).

FIG 3.22: It might look like one image, but it's actually *three:* each photo features the same subject, but with a slightly different crop.

you look at the page's source, you'll see our first example of the `picture` element:

```
<picture>
  <source
    srcset="homepage-person@desktop.png"
    media="(min-width: 990px)">
  <source
    srcset="homepage-person@tablet.png"
    media="(min-width: 750px)">
  <img
    srcset="homepage-person@mobile.png"
    alt="A featured Shopify Merchant">
</picture>
```


I've simplified their markup slightly, but the structure's the same. As you can see, a `picture` element contains any number of `source` elements, and exactly one `img`. On each `source`, there's a media query inside the oh-so-aptly named `media` attribute. The browser loops through each of the `source` elements until it finds one whose media query matches the conditions in the browser. Upon finding a match, it will send that `source`'s `srcset` to the `img` element and load it.

And that relationship between the `source` and the `img` is actually quite important: the matching `source` is never rendered by the browser. In fact, neither is the `picture` element: the `srcset` of the relevant `source` element is sent to the innermost `img`, and *that's* what gets displayed. So on widescreen displays, the `source` with `(min-width: 990px)` will send the largest version of Shopify's lead photo to the `img`; on midsize breakpoints, `homepage-person@tablet.png` will get rendered, thanks to the `(min-width: 750px)` query. And finally, if *none* of the media queries match, the browser will just load the `img`.

Instead of using `srcset` and `sizes` to load bigger and smaller versions of the same image, `picture` allows us to tailor our image content to fit specific viewports. In the language of the responsive images specification, this is referred to as *art direction*. Rather than simply resizing the image, we're cropping or otherwise optimizing it to fit a specific breakpoint. In doing so, we're ensuring that it still conveys its meaning, even though the details inside the image may change.

But swapping in different crops of images isn't all the `picture` element can do. In fact, take a quick look at http://responsivewebdesign.com/workshop. Halfway down the page, you'll see a list of logos (**FIG 3.23**). If you peek under the hood, each of those logos look something like this:

```
<picture>
  <source srcset="/logos/cibc.svg"
    type="image/svg+xml" />
  <img src="/logos/cibc.png" alt="CIBC" />
</picture>
```

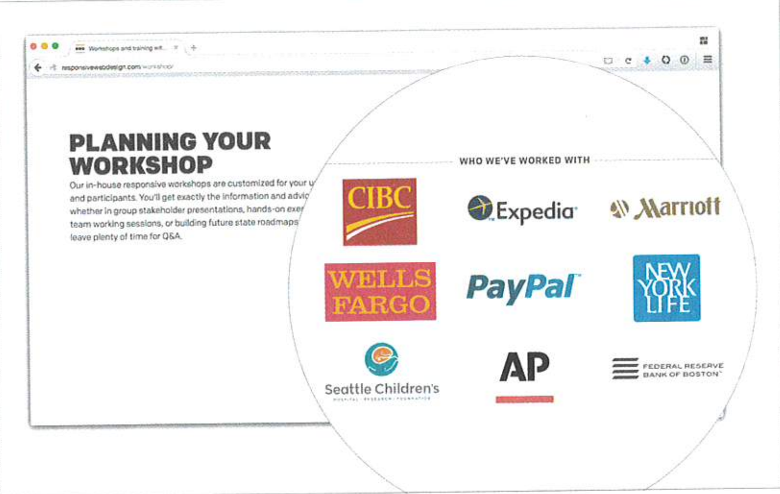

FIG 3.23: A little list of logos, powered by picture.

You might have noticed our `picture` element is completely lacking in media queries. Instead, there's a single `type` attribute on the `source`, indicating that the image it references is actually a vector-based SVG file (`image/svg+xml`). So instead of using media queries to select an image, our browser is actually checking our `source`s to see if it supports their individual `type`s. In this particular case, we're looking for SVG support: if a browser supports that `image/svg+xml` format, then it'll load the vector-based version of the image; but if it doesn't, it'll just load the PNG specified in our `img`.

In theory, we could extend this further, enhancing our `srcset` with some of those resolution-sensitive `w` or `x` flags we discussed earlier. This `type`-based switching allows us to use the `picture` element to ask a browser which file formats it supports. And it gets considerably more powerful when coupled with media queries, as you'll see if you look at the logo at the top of the Responsive Web Design site (http://responsivewebdesign.com/workshop):


```
<picture>
  <source
    media="(min-width: 50em)"
    type="image/svg+xml"
    srcset="/img/logo-rwd-sq.svg" />
  <source
    media="(min-width: 50em)"
    srcset="/img/logo-rwd-sq.png" />
  <source
    media="(min-width: 39em)"
    type="image/svg+xml"
    srcset="/img/logo-rwd.svg" />
  <source
    media="(min-width: 39em)"
    srcset="/img/logo-rwd.png" />
  <source
    type="image/svg+xml"
    srcset="/img/logo-rwd-sq.svg" />
  <img src="/img/logo-rwd-sq.png" alt="Responsive
    Web Design" />
</picture>
```

Here we're combining media queries on each `source` with a `type` attribute, allowing us to query not just the width of the viewport, but whether or not the browser *also* supports SVG (`type="image/svg+xml"`). And we're doing so at multiple breakpoints. At the widest (`(min-width: 50em)`) and smallest ends of the masthead's layout, we're looking to load a two-line version of the image, either as a SVG (`logo-rwd.svg`) or PNG (`logo-rwd.svg`). But at the middle breakpoint (`(min-width: 39em)`), the wordmark's laid out in a single line; and once again, we're using `type`-based switching to test for SVG support.

All that extra code might look complex, but the process is still the same: our browser is going to start at the top of our `source`s, and work its way down, searching for a `source` whose media query matches the viewport *and* whose `type` attribute matches the image formats supported by the browser. Once it finds a match, it'll send that `srcset` to the `img` to be rendered; if there aren't any matches, then it'll just load our `img`.

DESIGNER, FRAME THYSELF

We've looked at an incredibly broad array of techniques in this chapter. But in many ways, we're being asked to balance our designerly need for control—using `background-position` and `background-size` in our CSS, or `picture` in our markup—with the browser's ability to solve some of these image problems for us with `srcset` and `sizes`. More than any coding technique, that feels like the biggest challenge: to reframe the discussion to focus not on a specific technology, but on relinquishing perfect control over the experience.